# bitstring

*A Python module to help you manage your bits*

by Scott Griffiths

version 3.1.6

July 8, 2019

github.com/scott-griffiths/bitstring

# Contents

I User Manual 1

1 Walkthrough 3
- 1.1 A Brief Introduction . . . 3
  - 1.1.1 Prerequisites . . . 3
  - 1.1.2 Getting started . . . 3
  - 1.1.3 Modifying bitstrings . . . 5
  - 1.1.4 Finding and Replacing . . . 5
  - 1.1.5 Constructing a bitstring . . . 6
  - 1.1.6 Parsing bitstreams . . . 7
- 1.2 Worked examples . . . 7
  - 1.2.1 Hamming distance . . . 8
  - 1.2.2 Sieve of Eratosthenes . . . 8

2 Introduction 9
- 2.1 Getting Started . . . 10

3 Creation 11
- 3.1 The bitstring classes . . . 11
- 3.2 Using the constructor . . . 12
  - 3.2.1 From a hexadecimal string . . . 12
  - 3.2.2 From a binary string . . . 12
  - 3.2.3 From an octal string . . . 13
  - 3.2.4 From an integer . . . 13
  - 3.2.5 Big and little-endian integers . . . 13
  - 3.2.6 From a floating point number . . . 14
  - 3.2.7 Exponential-Golomb codes . . . 14
  - 3.2.8 From raw byte data . . . 14
  - 3.2.9 From a file . . . 15
- 3.3 The auto initialiser . . . 15

4 Packing 17
- 4.1 Compact format strings . . . 19

5 Interpreting Bitstrings 21
- 5.1 bin . . . 21
- 5.2 hex . . . 21
- 5.3 oct . . . 22
- 5.4 uint / uintbe / uintle / uintne . . . 22
- 5.5 int / intbe / intle / intne . . . 22
- 5.6 float / floatbe / floatle / floatne . . . 23
- 5.7 bytes . . . 23

5.8 ue . . . 23
5.9 se . . . 23
5.10 uie / sie . . . 24

**6 Slicing, Dicing and Splicing 25**
6.1 Slicing . . . 25
6.1.1 Stepping in slices . . . 26
6.2 Joining . . . 26
6.3 Truncating, inserting, deleting and overwriting . . . 27
6.3.1 Deleting and truncating . . . 27
6.3.2 `insert` . . . 27
6.3.3 `overwrite` . . . 27
6.4 The bitstring as a list . . . 27
6.5 Splitting . . . 28
6.5.1 `split` . . . 28
6.5.2 `cut` . . . 28

**7 Reading, Parsing and Unpacking 29**
7.1 Reading and parsing . . . 29
7.1.1 `read / readlist` . . . 29
7.1.2 Reading using format strings . . . 30
7.1.3 Peeking . . . 31
7.2 Unpacking . . . 31
7.3 Seeking . . . 31
7.4 Finding and replacing . . . 32
7.4.1 `find / rfind` . . . 32
7.4.2 `findall` . . . 32
7.4.3 `replace` . . . 32
7.5 Working with byte aligned data . . . 33

**8 Miscellany 35**
8.1 Other Functions . . . 35
8.1.1 `bytealign` . . . 35
8.1.2 `reverse` . . . 35
8.1.3 `tobytes` . . . 35
8.1.4 `tofile` . . . 36
8.1.5 `startswith / endswith` . . . 36
8.1.6 `ror / rol` . . . 36
8.2 Special Methods . . . 36
8.2.1 `__len__` . . . 36
8.2.2 `__str__ / __repr__` . . . 37
8.2.3 `__eq__ / __ne__` . . . 37
8.2.4 `__invert__` . . . 37
8.2.5 `__lshift__ / __rshift__ / __ilshift__ / __irshift__` . . . 37
8.2.6 `__mul__ / __imul__ / __rmul__` . . . 38
8.2.7 `__copy__` . . . 38
8.2.8 `__and__ / __or__ / __xor__ / __iand__ / __ior__ / __ixor__` . . . 38

**II Reference 39**

**9 Quick Reference 41**
9.1 Bits . . . 41
9.1.1 Methods . . . 41
9.1.2 Special methods . . . 42
9.1.3 Properties . . . 42
9.2 BitArray . . . 42
9.2.1 Additional methods . . . 42
9.2.2 Additional special methods . . . 43

9.2.3 Attributes . . . . . 43
9.3 ConstBitStream . . . . . 43
9.3.1 Additional methods . . . . . 43
9.3.2 Additional attributes . . . . . 43
9.4 BitStream . . . . . 43

**10 The bitstring module** **45**
10.1 The auto initialiser . . . . . 45
10.2 Compact format strings . . . . . 46
10.3 Class properties . . . . . 46

**11 The Bits class** **47**

**12 The BitArray class** **57**

**13 The ConstBitStream class** **63**

**14 The BitStream class** **67**

**15 Functions** **69**

**16 Exceptions** **71**

**III Appendices** **73**

**17 Examples** **77**
17.1 Creation . . . . . 77
17.2 Manipulation . . . . . 77
17.3 Parsing . . . . . 78
17.4 Sieve of Eratosthenes . . . . . 78

**18 Exponential-Golomb Codes** **79**
18.1 Interleaved exponential-Golomb codes . . . . . 80

**19 Optimisation Techniques** **81**
19.1 Use combined read and interpretation . . . . . 81
19.2 Choose the simplest class you can . . . . . 81
19.3 Use dedicated functions for bit setting and checking . . . . . 82

**20 Release Notes** **83**
20.1 Full Version History . . . . . 83
20.1.1 July 9th 2019: version 3.1.6 released . . . . . 83
20.1.2 May 17th 2016: version 3.1.5 released . . . . . 83
20.1.3 March 19th 2016: version 3.1.4 released . . . . . 83
20.1.4 March 4th 2014: version 3.1.3 released . . . . . 83
20.1.5 April 18th 2013: version 3.1.2 released . . . . . 84
20.1.6 March 21st 2013: version 3.1.1 released . . . . . 84
20.1.7 February 26th 2013: version 3.1.0 released . . . . . 84
20.1.8 November 21st 2011: version 3.0.0 released . . . . . 84
20.1.9 Backwardly incompatible changes . . . . . 84
20.1.10 New features . . . . . 85
20.1.11 June 18th 2011: version 2.2.0 released . . . . . 85
20.1.12 February 23rd 2011: version 2.1.1 released . . . . . 86
20.1.13 January 23rd 2011: version 2.1.0 released . . . . . 86
20.1.14 New class hierarchy introduced with simpler classes . . . . . 86
20.1.15 July 26th 2010: version 2.0.3 released . . . . . 87
20.1.16 July 25th 2010: version 2.0.2 released . . . . . 87
20.1.17 The backwardly incompatible changes are: . . . . . 88
20.1.18 The new features in this release are: . . . . . 91

20.1.19 March 18th 2010: version 1.3.0 for Python 2.6 and 3.x released . . . . . . . . . . . . . . 92
20.1.20 New features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
20.1.21 January 19th 2010: version 1.2.0 for Python 2.6 and 3.x released . . . . . . . . . . . . 94
20.1.22 New ‘Bits’ class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
20.1.23 December 22nd 2009: version 1.1.3 for Python 2.6 and 3.x released . . . . . . . . . . . 95
20.1.24 December 18th 2009: version 1.1.2 for Python 2.6 and 3.x released . . . . . . . . . . . . 95
20.1.25 November 24th 2009: version 1.1.0 for Python 2.6 and 3.x released . . . . . . . . . . . 95
20.1.26 New features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
20.1.27 October 10th 2009: version 1.0.1 for Python 3.x released . . . . . . . . . . . . . . . . . 96
20.1.28 October 9th 2009: version 1.0.0 for Python 2.x released . . . . . . . . . . . . . . . . . . 97
20.1.29 API Changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
20.1.30 New features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
20.1.31 September 11th 2009: version 0.5.2 for Python 2.x released . . . . . . . . . . . . . . . . 98
20.1.32 August 26th 2009: version 0.5.1 for Python 2.x released . . . . . . . . . . . . . . . . . 99
20.1.33 July 19th 2009: version 0.5.0 for Python 2.x released . . . . . . . . . . . . . . . . . . . 100
20.1.34 June 15th 2009: version 0.4.3 for Python 2.x released . . . . . . . . . . . . . . . . . . . 102
20.1.35 May 25th 2009: version 0.4.2 for Python 2.x released . . . . . . . . . . . . . . . . . . . 103
20.1.36 April 23rd 2009: Python 3 only version 0.4.1 released . . . . . . . . . . . . . . . . . . . 104
20.1.37 April 11th 2009: version 0.4.0 released . . . . . . . . . . . . . . . . . . . . . . . . . 104
20.1.38 March 11th 2009: version 0.3.2 released . . . . . . . . . . . . . . . . . . . . . . . . . 105
20.1.39 February 26th 2009: version 0.3.1 released . . . . . . . . . . . . . . . . . . . . . . . . 105
20.1.40 February 15th 2009: version 0.3.0 released . . . . . . . . . . . . . . . . . . . . . . . . 106
20.1.41 January 13th 2009: version 0.2.0 released . . . . . . . . . . . . . . . . . . . . . . . . 107
20.1.42 December 29th 2008: version 0.1.0 released . . . . . . . . . . . . . . . . . . . . . . . 107

**Python Module Index** **109**

**Index** **111**

# Part I

# User Manual

## CHAPTER 1

# Walkthrough

## 1.1 A Brief Introduction

The aim of the `bitstring` module is make dealing with binary data in Python as easy as possible. In this section I will take you through some of the features of the module to help you get started using it.

Only a few of the module's features will be covered in this walkthrough; the *User Manual* and *Reference* provide a more thorough guide. The whole of this section can be safely skipped or skimmed over if you prefer to start with the manual. If however you'd like a gentler introduction then you might like to follow along the examples with a Python interpreter.

### 1.1.1 Prerequisites

- Python 2.7 or 3.x.
- An installed bitstring module.
- A rudimentary knowledge of binary concepts.
- A little free time.

If you haven't yet downloaded and installed `bitstring` then please do so (it should be as easy as typing "`pip install bitstring`"). I'll be going through some examples using the interactive Python interpreter, so feel free to start up a Python session and follow along.

### 1.1.2 Getting started

```
>>> from bitstring import BitArray, BitStream
```

First things first, we're going to be typing 'bitstring' a lot, so importing directly saves us a lot of `bitstring.BitStream` nonsense. The classes we have imported are `BitArray` which is just a container for our binary data and `BitStream` which adds a bit position and reading methods to treat the data as a stream. There are also immutable versions of both these classes that we won't be using here.

We can now create a couple of bitstrings:

```
>>> a = BitArray('0xff01')
>>> b = BitArray('0b110')
```

The first of these we made from the hexadecimal string `0xff01` - the `0x` prefix makes it hexadecimal just as `0b` means binary and `0o` means octal. Each hex digit represents four bits, so we have a bitstring of length 16 bits.

The second was created from a binary string. In this case it is just three bits long. Don't worry about it not being a whole number of bytes long, that's all been taken care of internally.

---

**Note:** Be sure to remember the quotes around the hex and binary strings. If you forget them you would just have an ordinary Python integer, which would instead create a bitstring of that many '0' bits. For example `0xff01` is the same as the base-10 number 65281, so `BitArray(0xff01)` would consist of 65281 zero bits!

---

There are lots of things we can do with our new bitstrings, the simplest of which is just to print them:

```
>>> print(a)
0xff01
>>> print(b)
0b110
```

Now you would be forgiven for thinking that the strings that we used to create the two bitstrings had just been stored to be given back when printed, but that's not the case. Every bitstring should be considered just as a sequence of bits. As we'll see there are lots of ways to create and manipulate them, but they have no memory of how they were created. When they are printed they just pick the simplest hex or binary representation of themselves. If you prefer you can pick the representation that you want:

```
>>> a.bin
'1111111100000001'
>>> b.oct
'6'
>>> b.int
-2
>>> a.bytes
'\xff\x01'
```

There are a few things to note here:

- To get the different interpretations of the binary data we use properties such as *`bin`*, *`hex`*, *`oct`*, *`int`* and *`bytes`*. You can probably guess what these all mean, but you don't need to know quite yet. The properties are calculated when you ask for them rather than being stored as part of the object itself.
- The *`bytes`* property returns a `bytes` object. This is slightly different in Python 2.7 to Python 3 - in Python 3 you would get `b'\xff\x01'` returned instead.

Great - let's try some more:

```
>>> b.hex
bitstring.InterpretError: Cannot convert to hex unambiguously - not multiple of 4␣
↪bits.
```

Oh dear. The problem we have here is that `b` is 3 bits long, whereas each hex digit represents 4 bits. This means that there is no unambiguous way to represent it in hexadecimal. There are similar restrictions on other interpretations (octal must be a multiple of 3 bits, bytes a multiple of 8 bits etc.)

An exception is raised rather than trying to guess the best hex representation as there are a multitude of ways to convert to hex. I occasionally get asked why it doesn't just do the 'obvious' conversion, which is invariably what that person expects from his own field of work. This could be truncating bits at the start or end, or padding at the start or end with either zeros or ones. Rather than try to guess what is meant we just raise an exception - if you want a particular behaviour then write it explicitly:

```
>>> (b + [0]).hex
'c'
>>> ([0] + b).hex
'6'
```

Here we've added a zero bit first to the end and then to the start. Don't worry too much about how it all works, but just to give you a taster the zero bit `[0]` could also have been written as `BitArray([0])`, `BitArray([0])`, `BitArray('0b0')`, `BitArray(bin='0')`, `'0b0'` or just `1` (this final method isn't a typo, it means construct a bitstring of length one, with all the bits initialised to zero - it does look a bit confusing though which is why I prefer [0] and [1] to represent single bits). Take a look at *The auto initialiser* for more details.

### 1.1.3 Modifying bitstrings

A `BitArray` can be treated just like a list of bits. You can slice it, delete sections, insert new bits and more using standard index notation:

```
>>> print(a[3:9])
0b111110
>>> del a[-6:]
>>> print(a)
0b1111111100
```

The slicing works just as it does for other containers, so the deletion above removes the final six bits.

If you ask for a single item, rather than a slice, a boolean is returned. Naturally enough `1` bits are `True` whereas `0` bits are `False`.

```
>>> a[0]
True
>>> a[-1]
False
```

To join together bitstrings you can use a variety of methods, including `append`, `prepend`, `insert`, and plain `+` or `+=` operations:

```
>>> a.prepend('0b01')
>>> a.append('0o7')
>>> a += '0x06'
```

Here we first put two bits at the start of `a`, then three bits on the end (a single octal digit) and finally another byte (two hex digits) on the end.

Note how we are just using ordinary strings to specify the new bitstrings we are adding. These get converted automatically to the right sequence of bits.

**Note:** The length in bits of bitstrings specified with strings depends on the number of characters, including leading zeros. So each hex character is four bits, each octal character three bits and each binary character one bit.

### 1.1.4 Finding and Replacing

A `find` is provided to search for bit patterns within a bitstring. You can choose whether to search only on byte boundaries or at any bit position:

```
>>> a = BitArray('0xa9f')
>>> a.find('0x4f')
(3,)
```

Here we have found the `0x4f` byte in our bitstring, though it wasn't obvious from the hexadecimal as it was at bit position 3. To see this clearer consider this equality:

```
>>> a == '0b101, 0x4f, 0b1'
True
```

in which we've broken the bitstring into three parts to show the found byte. This also illustrates using commas to join bitstring sections.

### 1.1.5 Constructing a bitstring

Let's say you have a specification for a binary file type (or maybe a packet specification etc.) and you want to create a bitstring quickly and easily in Python. For this example I'm going to use a header from the MPEG-2 video standard. Here's how the header is described in the standard:

| sequence_header() | No. of bits | Mnemonic |
|---|---|---|
| sequence_header_code | 32 | bslbf |
| horizontal_size_value | 12 | uimsbf |
| vertical_size_value | 12 | uimsbf |
| aspect_ratio_information | 4 | uimsbf |
| frame_rate_code | 4 | uimsbf |
| bit_rate_value | 18 | uimsbf |
| marker_bit | 1 | bslbf |
| vbv_buffer_size_value | 10 | uimsbf |
| constrained_parameters_flag | 1 | bslbf |
| load_intra_quantiser_matrix | 1 | uimsbf |
| if (load_intra_quantiser_matrix) | | |
| { intra_quantiser_matrix[64] } | 8*64 | uimsbf |
| load_non_intra_quantiser_matrix | 1 | uimsbf |
| if (load_non_intra_quantiser_matrix) | | |
| { non_intra_quantiser_matrix[64] } | 8*64 | uimsbf |
| next_start_code() | | |

The mnemonics mean things like uimsbf = 'Unsigned integer, most significant bit first'.

So to create a sequence_header for your particular stream with width of 352 and height of 288 you could start like this:

```
s = BitArray()
s.append('0x000001b3')  # the sequence_header_code
s.append('uint:12=352') # 12 bit unsigned integer
s.append('uint:12=288')
...
```

which is fine, but if you wanted to be a bit more concise you could just write

```
s = BitArray('0x000001b3, uint:12=352, uint:12=288')
```

This is better, but it might not be a good idea to have the width and height hard-wired in like that. We can make it more flexible by using a format string and the *pack* function:

```
width, height = 352, 288
s = bitstring.pack('0x000001b3, 2*uint:12', width, height)
```

where we have also used `2*uint:12` as shorthand for `uint:12, uint:12`.

The *pack* function can also take a dictionary as a parameter which can replace the tokens in the format string. For example:

```
fmt = 'sequence_header_code,
       uint:12=horizontal_size_value,
       uint:12=vertical_size_value,
       uint:4=aspect_ratio_information,
       ...
       '
d = {'sequence_header_code': '0x000001b3',
     'horizontal_size_value': 352,
     'vertical_size_value': 288,
     'aspect_ratio_information': 1,
     ...
    }

s = bitstring.pack(fmt, **d)
```

### 1.1.6 Parsing bitstreams

You might have noticed that *pack* returned a *BitStream* rather than a *BitArray*. This isn't a problem as the *BitStream* class just adds a few stream-like qualities to *BitArray* which we'll take a quick look at here.

First, let's look at the stream we've just created:

```
>>> s
BitStream('0x000001b31601201')
```

The stream-ness of this object is via its bit position, and various reading and peeking methods. First let's try a read or two, and see how this affects the bit position:

```
>>> s.pos
0
>>> s.read(24)
BitStream('0x000001')
>>> s.pos
24
>>> s.read('hex:8')
'b3'
>>> s.pos
32
```

First we read 24 bits, which returned a new *BitStream* object, then we used a format string to read 8 bits interpreted as a hexadecimal string. We know that the next two sets of 12 bits were created from integers, so to read them back we can say

```
>>> s.readlist('2*uint:12')
[352, 288]
```

If you don't want to use a bitstream then you can always use *unpack*. This takes much the same form as *readlist* except it just unpacks from the start of the bitstring. For example:

```
>>> s.unpack('bytes:4, 2*uint:12, uint:4')
['\x00\x00\x01\xb3', 352, 288, 1]
```

## 1.2 Worked examples

Below are a few examples of using the bitstring module, as I always find that a good example can help more than a lengthy reference manual.

### 1.2.1 Hamming distance

The Hamming distance between two bitstrings is the number of bit positions in which the two bitstrings differ. So for example the distance between 0b00110 and 0b01100 is 2 as the second and fourth bits are different.

Write a function that calculates the Hamming weight of two bitstrings.

```
def hamming_weight(a, b):
    return (a^b).count(True)
```

Er, that's it. The `^` is a bit-wise exclusive or, which means that the bits in `a^b` are only set if they differ in `a` and `b`. The *`count`* method just counts the number of 1 (or True) bits.

```
>>> a = Bits('0b00110')
>>> hamming_weight(a, '0b01100')
2
```

### 1.2.2 Sieve of Eratosthenes

The sieve of Eratosthenes is an ancient (and very inefficient) method of finding prime numbers. The algorithm starts with the number 2 (which is prime) and marks all of its multiples as not prime, it then continues with the next unmarked integer (which will also be prime) and marks all of its multiples as not prime.

So to print all primes under a million you could write:

```
from bitstring import BitArray
# create a BitArray with a million zero bits.
# The bits will be set to indicate that the bit position isn't prime.
has_factors = BitArray(1000000)
for i in xrange(2, 1000000):
    if not has_factors[i]:
        print(i)
        # Set all multiples of our prime to 1.
        has_factors.set(True, xrange(i*2, 1000000, i))
```

I'll leave optimising the algorithm as an exercise for the reader, but it illustrates both bit checking and setting. One reason you might want to use a bitstring for this purpose (instead of a plain list for example) is that the million bits only take up a million bits in memory, whereas for a list of integers it would be much more. Try asking for a billion elements in a list - unless you've got some really nice hardware it will fail, whereas a billion element bitstring only takes 125MB.

### CHAPTER 2

# 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 four classes, *`BitStream`*, *`BitArray`*, *`ConstBitStream`* 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. I shall use the general term 'bitstring' when referring generically to any of the classes, and use the class names for parts that apply to only one or another.

*`BitArray`* objects can be sliced, joined, reversed, inserted into, overwritten, packed, unpacked etc. with simple functions or slice notation. *`BitStream`* objects 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 the *Reference* section, 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 BitArray
```

Just some of the ways to create bitstrings:

```
# from a binary string
a = BitArray('0b001')
# from a hexadecimal string
b = BitArray('0xff470001')
# straight from a file
c = BitArray(filename='somefile.ext')
# from an integer
d = BitArray(int=540, length=11)
# using a format string
d = BitArray('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)
'9249cdcd'
>>> print(e.int)
-1840656947
>>> print(e.uint)
2454310349
```

## 2.1 Getting Started

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

```
pip install bitstring
```

or similar.

If you want an earlier version, or need other files in the full package, you can download it from the project's website.

If you then extract the contents of the zip file you should find files organised in these directories

- `bitstring/` : The bitstring module files.
- `test/` : Unit tests for the module, plus some example files for testing purposes.
- `doc/` : This manual as a PDF and as HTML.

If you downloaded the source and want to install, run:

```
python setup.py install
```

You might need to add a 'sudo' to the start of that command, depending on your system. This will copy the source files to your Python installation's `site-packages` directory.

The module comes with comprehensive unit tests. To run them yourself use your favourite unit test running method, mine is:

```
python -m unittest discover
```

which when run in the *test* folder should run all the tests (almost 500) and say OK. If tests fail then either your version of Python isn't supported (you need Python 2.7 or 3.x) or something unexpected has happened - in which case please tell me about it.

# CHAPTER 3

# Creation

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.

## 3.1 The bitstring classes

Four classes are provided by the bitstring module: `BitStream` and `BitArray` together with their immutable versions `ConstBitStream` and `Bits`:

- `Bits (object)`: This is the most basic class. It is immutable and so its contents can't be changed after creation.
- `BitArray (Bits)`: This adds mutating methods to its base class.
- `ConstBitStream (Bits)`: This adds methods and properties to allow the bits to be treated as a stream of bits, with a bit position and reading/parsing methods.
- `BitStream (BitArray, ConstBitStream)`: This is the most versatile class, having both the bitstream methods and the mutating methods.

Before version 3.0 `Bits` was known as `ConstBitArray`. The old name is still available for backward compatibility.

The term 'bitstring' is used in this manual to refer generically to any of these classes.

Most of the examples in this manual use the `BitArray` class, with `BitStream` used when necessary. For most uses the non-const classes are 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 `BitArray` or `BitStream`. Truncating, replacing, inserting, appending etc. are not available for the const classes.
- 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` or a `ConstBitStream`. As `BitArray` and `BitStream` objects are mutable they do not support hashing and so cannot be used in these ways.
- If you are creating directly from a file then a `BitArray` or `BitStream` will read the file into memory whereas a `Bits` or `ConstBitStream` will not, so using the const classes allows extremely large files to be examined.

- If you don't need the extra functionality of a particular class then the simpler ones might be faster and more memory efficient. The fastest and most memory efficient class is *`Bits`*.

The *`Bits`* class is the base class of the other three class. This means that `isinstance(s, Bits)` will be true if `s` is an instance of any of the four classes.

## 3.2 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, bytearray, array, bytes or another bitstring.
- `bytes` : A `bytes` object (a `str` in Python 2), 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.
- `sie`, `uie` : Signed or unsigned interleaved exponential-Golomb coded integers.
- `bool` : A boolean (i.e. True or False).
- `filename` : Directly from a file, without reading into memory.

### 3.2.1 From a hexadecimal string

```
>>> c = BitArray(hex='0x000001b3')
>>> c.hex
'000001b3'
```

The initial `0x` or `0X` is optional. 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 = BitArray('0x000001b3')
```

### 3.2.2 From a binary string

```
>>> d = BitArray(bin='0011 00')
>>> d.bin
'001100'
```

An initial `0b` or `0B` is optional and whitespace will be ignored.

As with `hex`, the `auto` initialiser will work if the binary string is prefixed by `0b`:

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

### 3.2.3 From an octal string

```
>>> o = BitArray(oct='34100')
>>> o.oct
'34100'
```

An initial `0o` or `0O` is optional, but `0o` (a zero and lower-case 'o') is preferred as it is slightly more readable.

As with `hex` and `bin`, the `auto` initialiser will work if the octal string is prefixed by `0o`:

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

### 3.2.4 From an integer

```
>>> e = BitArray(uint=45, length=12)
>>> f = BitArray(int=-1, length=7)
>>> e.bin
'000000101101'
>>> f.bin
'1111111'
```

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 = BitArray('uint:12=45')
>>> f = BitArray('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.

### 3.2.5 Big and little-endian integers

```
>>> big_endian = BitArray(uintbe=1, length=16)
>>> little_endian = BitArray(uintle=1, length=16)
>>> native_endian = BitArray(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
'0001'
>>> little_endian.hex
'0100'
```

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`).

### 3.2.6 From a floating point number

```
>>> f1 = BitArray(float=10.3, length=32)
>>> f2 = BitArray('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 = BitArray(floatle=0.0, length=64)
>>> native_endian = BitArray('floatne:32=-6.3')
```

### 3.2.7 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. Interleaved exponential-Golomb codes are also supported via `uie` and `sie`:

```
>>> g = BitArray(ue=12)
>>> h = BitArray(se=-402)
>>> g.bin
'0001101'
>>> h.bin
'0000000001100100101'
```

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 = BitArray('ue=12')
>>> h = BitArray('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.

### 3.2.8 From raw byte data

Using the length and offset parameters to specify the length in bits and an offset at the start to be ignored is particularly useful when initialising from raw data or from a file.

```
a = BitArray(bytes=b'\x00\x01\x02\xff', length=28, offset=1)
b = BitArray(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.

You can also use a `bytearray` object, either explicitly with a `bytes=some_bytearray` keyword or via the `auto` initialiser:

```
c = BitArray(a_bytearray_object)
```

If you are using Python 3.x you can use this trick with `bytes` objects too. This should be used with caution as in Python 2.7 it will instead be interpreted as a string (it's not possible to distinguish between `str` and `bytes` in Python 2) and so your code won't work the same between Python versions.

```
d = BitArray(b'\x23g$5')    # Use with caution! Only works correctly in Python 3.
```

### 3.2.9 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 = Bits(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.

Note that we created a *`Bits`* here rather than a *`BitArray`*, as they have quite different behaviour in this case. The immutable *`Bits`* will never read the file into memory (except as needed by other operations), whereas if we had created a *`BitArray`* then the whole of the file would immediately have been read into memory. This is because in creating a *`BitArray`* you are implicitly saying that you want to modify it, and so it needs to be in memory.

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

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

## 3.3 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, an iterable, another bitstring, an integer, a bytearray 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, (non string) iterables are interpreted as boolean arrays and file objects acts a source of binary data. An `array` object will be converted into its constituent bytes. Finally you can use an integer to create a zeroed bitstring of that number of bits.

```
>>> fromhex = BitArray('0x01ffc9')
>>> frombin = BitArray('0b01')
>>> fromoct = BitArray('0o7550')
>>> fromint = BitArray('int:32=10')
>>> fromfloat = BitArray('float:64=0.2')
>>> acopy = BitArray(fromoct)
>>> fromlist = BitArray([1, 0, 0])
>>> f = open('somefile', 'rb')
>>> fromfile = BitArray(f)
>>> zeroed = BitArray(1000)
>>> frombytes = BitArray(bytearray(b'xyz'))
>>> fromarray = BitArray(array.array('h', [3, 17, 10]))
```

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

```
>>> immutable = Bits('0xabc')
>>> mutable = BitArray(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
'7550'
>>> fromoct.hex
'f68'
>>> fromoct.bin
'111101101000'
>>> fromoct.uint
3994
>>> fromoct.int
-152

>>> BitArray('0o7777') == '0xfff'
True
>>> BitArray('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 = BitArray('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.

# CHAPTER 4

# Packing

Another method of creating *`BitStream`* 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 = BitStream('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. |
| `bool[:1]` | single bit as a boolean (True or False). |
| `ue` | an unsigned integer as an exponential-Golomb code. |
| `se` | a signed integer as an exponential-Golomb code. |
| `uie` | an unsigned integer as an interleaved exponential-Golomb code. |
| `sie` | a signed integer as an interleaved 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`. |
| `uie=m` | interleaved exponential-Golomb code for unsigned integer `m`. |
| `sie=m` | interleaved exponential-Golomb code for signed integer `m`. |
| `bool=b` | a single bit, either True or False. |
| `pad:n` | `n` zero bits (for use as padding). |

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

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

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

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

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

```
>>> s = bitstring.pack('ue, oct:3, hex:8, uint:14', 3, '0o7', '0xff', 90)
>>> s.unpack('ue, oct:3, hex:8, uint:14')
[3, '7', 'ff', 90]
```

## 4.1 Compact format strings

Another option when using *pack*, as well as other methods such as *read* and *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 *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.

CHAPTER 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 = BitArray('0x123')
>>> b = BitArray('0b111')
```

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

```
>>> a.bin
'000100100011'
>>> b.bin
'111'
```

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

## 5.2 hex

For whole-byte bitstrings the most natural interpretation is often as hexadecimal, with each byte represented by two hex digits.

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

```
>>> a.hex
'123'
>>> b.hex
ValueError: Cannot convert to hex unambiguously - not multiple of 4 bits.
```

## 5.3 oct

For an octal interpretation use the `oct` property.

If the bitstring does not have a length that is a multiple of three then an `InterpretError` exception will be raised.

```
>>> a.oct
'0443'
>>> b.oct
'7'
>>> (b + '0b0').oct
ValueError: Cannot convert to octal unambiguously - not multiple of 3 bits.
```

## 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 = BitArray('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
```

## 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.

## 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.

## 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 2.6/2.7).

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 necessary.

```
>>> open('somefile', 'wb').write(a.tobytes())
>>> open('anotherfile', 'wb').write(('0x0'+a).bytes)
>>> a1 = BitArray(filename='somefile')
>>> a1.hex
'1230'
>>> a2 = BitArray(filename='anotherfile')
>>> a2.hex
'0123'
```

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.

## 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 an `InterpretError` 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 = BitArray(ue=12)
>>> s.bin
'0001101'
>>> s.append(BitArray(ue=3))
>>> print(s.readlist('2*ue'))
[12, 3]
```

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

## 5.10 uie / sie

A slightly different type, interleaved exponential-Golomb codes are also supported. The principles are the same as with `ue` and `se` - see *Exponential-Golomb Codes* for detail of the differences.

## CHAPTER 6

# 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.

## 6.1 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 = BitArray('0b00011110')
>>> b = a[3:7]
>>> print(a, b)
0x1e 0xf
```

For single bit indices (as opposed to slices) a boolean is returned; that is `True` for '1' bits and `False` for '0' bits:

```
>>> a[0]
False
>>> a[4]
True
```

If you want a single bit as a new bitstring then use a one-bit slice instead:

```
>>> a[0:1]
BitArray('0b0')
```

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

```
>>> a = BitArray('0b00011110')
>>> print(a[:5])         # first 5 bits
0b00011
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

### 6.1.1 Stepping in slices

The step parameter (also known as the stride) can be used in slices and has the same meaning as in the built-in containers:

```
>>> s = BitArray(16)
>>> s[::2] = [1]*8
>>> s.bin
'1010101010101010'
>>> del s[8::2]
>>> s.bin
'101010100000'
>>> s[::3].bin
'1010'
```

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).

## 6.2 Joining

To join together a couple of bitstring objects use the + or += operators, or the *`append`* and *`prepend`* methods.

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

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

If you want to join a large number of bitstrings then the method *`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 = [BitArray(uint=n, length=12) for n in xrange(1000)]
s = BitArray('0b1111').join(bslist)
```

## 6.3 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.

### 6.3.1 Deleting and truncating

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

```
>>> a = BitArray('0b00011000')
>>> del a[3:5]                # remove 2 bits at pos 3
>>> a.bin
‘000000’
>>> b = BitArray('0x112233445566')
>>> del b[24:40]
>>> b.hex
'11223366'
```

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

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

### 6.3.2 `insert`

As you might expect, *`insert`* takes one *`BitArray`* and inserts it into another. A bit position must be specified for *`BitArray`* and *`Bits`*, but for BitStreams if not present then the current *`pos`* is used.

```
>>> a = BitArray('0x00112233')
>>> a.insert('0xffff', 16)
>>> a.hex
'0011ffff2233'
```

### 6.3.3 `overwrite`

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

```
>>> a = BitStream('0x00112233')
>>> a.pos = 4
>>> a.overwrite('0b1111')         # Uses current pos as default
>>> a.hex
'0f112233'
```

## 6.4 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` |

## 6.5 Splitting

### 6.5.1 `split`

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

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

470000
4711
472222
```

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

### 6.5.2 `cut`

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

```
>>> a = BitArray('0x47001243')
>>> for byte in a.cut(8):
...     print(byte.hex)
47
00
12
43
```

CHAPTER 7

# Reading, Parsing and Unpacking

## 7.1 Reading and parsing

The `BitStream` and `ConstBitStream` classes contain number of methods for reading the bitstring as if it were a file or stream. Depending on how it was constructed the bitstream might actually be contained in a file rather than stored in memory, but these methods work for either case.

In order to behave like a file or stream, every bitstream 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 bitstream, a position from which all reads will fail as it is past the last bit. Note that the `pos` property isn't considered a part of the bitstream's identity; this allows it to vary for immutable `ConstBitStream` objects and means that it doesn't affect equality or hash values.

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 `ByteAlignError` exception will be raised.

### 7.1.1 `read / readlist`

For simple reading of a number of bits you can use `read` with an integer argument. A new bitstring object gets returned, which can be interpreted using one of its properties or used for further reads. 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 = ConstBitStream(filename='test/test.m1v')
>>> print(s.pos)
0
>>> start_code = s.read(32).hex
>>> width = s.read(12).uint
>>> height = s.read(12).uint
>>> print(start_code, width, height, s.pos)
000001b3 352 288 56
>>> s.pos += 37
>>> flags = s.read(2)
>>> constrained_parameters_flag = flags.read(1)
>>> load_intra_quantiser_matrix = flags.read(1)
```

(continues on next page)

(continued from previous page)

```
>>> print(s.pos, flags.pos)
95 2
```

If you want to read multiple items in one go you can use *`readlist`*. This can take an iterable of bit lengths and return a list of bitstring objects. So for example instead of writing:

```
a = s.read(32)
b = s.read(8)
c = s.read(24)
```

you can equivalently use just:

```
a, b, c = s.readlist([32, 8, 24])
```

### 7.1.2 Reading using format strings

The *`read`* / *`readlist`* methods can also take a format string similar to that used in the auto initialiser. Only one token should be provided to *`read`* and a single value is returned. To read multiple tokens use *`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. |
| `uie` | next bits as an interleaved unsigned exponential-Golomb code. |
| `sie` | next bits as an interleaved signed exponential-Golomb code. |
| `bool[:1]` | next bit as a boolean (True or False). |
| `pad:n` | next `n` bits will be ignored (padding). |

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 *`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, everything_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`, `se`, `uie` 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).

The `pad` token is a special case in that it just causes bits to be skipped over without anything being returned. This can be useful for example if parts of a binary format are uninteresting:

```
a, b = s.readlist('pad:12, uint:4, pad:4, uint:8')
```

### 7.1.3 Peeking

In addition to the read methods there are matching peek methods. These are identical to the read except that they do not advance the position in the bitstring to after the read elements.

```
s = ConstBitStream('0x4732aa34')
if s.peek(8) == '0x47':
    t = s.read(16)          # t is first 2 bytes '0x4732'
else:
    s.find('0x47')
```

## 7.2 Unpacking

The *`unpack`* method works in a very similar way to *`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')
```

## 7.3 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 *`ByteAlignError`* exception is raised.

For example:

```
>>> s = BitStream('0x123456')
>>> s.pos
0
>>> s.bytepos += 2
```

(continues on next page)

(continued from previous page)

```
>>> s.pos                    # note pos verses bytepos
16
>>> s.pos += 4
>>> print(s.read('bin:4'))   # the final nibble '0x6'
0110
```

## 7.4 Finding and replacing

### 7.4.1 `find / rfind`

To search for a sub-string use the *`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 a tuple containing that position, otherwise it will return an empty tuple. 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 = ConstBitStream('0x00123400001234')
>>> found = s.find('0x1234', bytealigned=True)
>>> print(found, s.bytepos)
(8,) 1
>>> found = s.find('0xff', bytealigned=True)
>>> print(found, s.bytepos)
() 1
```

The reason for returning the bit position in a tuple is so that the return value is `True` in a boolean sense if the sub-string is found, and `False` if it is not (if just the bit position were returned there would be a problem with finding at position 0). The effect is that you can use `if s.find(...):` and have it behave as you'd expect.

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

```
>>> t = BitArray('0x0f231443e8')
>>> found = t.rfind('0xf')           # Search all bit positions in reverse
>>> print(found)
(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 *`rfind`* will start at `end` and end at `start` (so you always need `start` < `end`).

### 7.4.2 `findall`

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

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

### 7.4.3 `replace`

To replace all occurrences of one *`BitArray`* with another use *`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`* or *`ConstBitStream`* classes.

```
>>> s = BitArray('0b110000110110')
>>> s.replace('0b110', '0b1111')
3             # The number of replacements made
>>> s.bin
'111100011111111'
```

## 7.5 Working with byte aligned data

The emphasis with the bitstring module is always towards not worrying if things are a whole number of bytes long or are aligned on byte boundaries. Internally the module has to worry about this quite a lot, but the user shouldn't have to care. To this end methods such as *`find`*, *`findall`*, *`split`* and *`replace`* by default aren't concerned with looking for things only on byte boundaries and provide a parameter `bytealigned` which can be set to `True` to change this behaviour.

This works fine, but it's not uncommon to be working only with whole-byte data and all the `bytealigned=True` can get a bit repetitive. To solve this it is possible to change the default throughout the module by setting `bitstring.bytealigned`. For example:

```
>>> s = BitArray('0xabbb')
>>> s.find('0xbb')                   # look for the byte 0xbb
(4,)                                 # found, but not on byte boundary
>>> s.find('0xbb', bytealigned=True) # try again...
(8,)                                 # not found on any byte boundaries
>>> bitstring.bytealigned = True     # change the default behaviour
>>> s.find('0xbb')
(8,)                                 # now only finds byte aligned
```

# CHAPTER 8

# Miscellany

## 8.1 Other Functions

### 8.1.1 `bytealign`

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

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

### 8.1.2 `reverse`

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

```
>>> a = BitArray('0b000001101')
>>> a.reverse()
>>> a.bin
'101100000'
>>> a.reverse(0, 4)
>>> a.bin
'110100000'
```

### 8.1.3 `tobytes`

Returns the byte data contained in the bitstring as a `bytes` object (equivalent to a `str` if you're using Python 2.7). 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.

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

### 8.1.4 `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')
>>> BitArray('0xffee3241fed').tofile(f)
```

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

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

### 8.1.6 `ror / rol`

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

```
>>> s = BitArray('0x00001')
>>> s.rol(6)
>>> s.hex
'00040'
```

## 8.2 Special Methods

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

### 8.2.1 `__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 with 32-bit Python).

There's not much more to say really, except to emphasise that it is always in bits and never bytes.

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

### 8.2.2 `__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 *__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 = BitArray('0b1111 111')
>>> print(a)
0b1111111
>>> a
BitArray('0b1111111')
>>> a += '0b1'
>>> print(a)
0xff
>>> print(a.bin)
11111111
```

### 8.2.3 `__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).

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

### 8.2.4 `__invert__`

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

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

### 8.2.5 `__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 = BitArray('0b10011001')
>>> b = a << 2
>>> print(b)
0b01100100
>>> a >>= 2
```

(continues on next page)

(continued from previous page)

```
>>> print(a)
0b00100110
```

### 8.2.6 `__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 = BitArray('0b10')*8
>>> print(a.bin)
1010101010101010
```

### 8.2.7 `__copy__`

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

```
>>> import copy
>>> a = Bits('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 = BitArray(a)` is another option, but `b = a[:]` may be more familiar to some.

### 8.2.8 `__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 = BitArray('0b00001111')
>>> b = BitArray('0b01010101')
>>> print((a&b).bin)
00000101
>>> print((a|b).bin)
01011111
>>> print((a^b).bin)
01011010
>>> b &= '0x1f'
>>> print(b.bin)
00010101
```

# Part II

# Reference

# CHAPTER 9

# Quick Reference

This section lists the bitstring module's classes together with all their methods and attributes. The next section goes into full detail with examples.

## 9.1 Bits

`Bits(object)`

A `Bits` is the most basic class. It is immutable, so once created its value cannot change. It is a base class for all the other classes in the *bitstring* module.

### 9.1.1 Methods

- `all` – Check if all specified bits are set to 1 or 0.
- `any` – Check if any of specified bits are set to 1 or 0.
- `count` – Count the number of bits set to 1 or 0.
- `cut` – Create generator of constant sized chunks.
- `endswith` – Return whether the bitstring ends with a sub-bitstring.
- `find` – Find a sub-bitstring in the current bitstring.
- `findall` – Find all occurrences of a sub-bitstring in the current bitstring.
- `join` – Join bitstrings together using current bitstring.
- `rfind` – Seek backwards to find a sub-bitstring.
- `split` – Create generator of chunks split by a delimiter.
- `startswith` – Return whether the bitstring starts with a sub-bitstring.
- `tobytes` – Return bitstring as bytes, padding if needed.
- `tofile` – Write bitstring to file, padding if needed.
- `unpack` – Interpret bits using format string.

### 9.1.2 Special methods

Also available are the operators `[]`, `==`, `!=`, `+`, `*`, `~`, `<<`, `>>`, `&`, `|` and `^`.

### 9.1.3 Properties

- `bin` – The bitstring as a binary string.
- `bool` – For single bit bitstrings, interpret as True or False.
- `bytes` – The bitstring as a bytes object.
- `float` – Interpret as a floating point number.
- `floatbe` – Interpret as a big-endian floating point number.
- `floatle` – Interpret as a little-endian floating point number.
- `floatne` – Interpret as a native-endian floating point number.
- `hex` – The bitstring as a hexadecimal string.
- `int` – Interpret as a two's complement signed integer.
- `intbe` – Interpret as a big-endian signed integer.
- `intle` – Interpret as a little-endian signed integer.
- `intne` – Interpret as a native-endian signed integer.
- `len` – Length of the bitstring in bits.
- `oct` – The bitstring as an octal string.
- `se` – Interpret as a signed exponential-Golomb code.
- `ue` – Interpret as an unsigned exponential-Golomb code.
- `sie` – Interpret as a signed interleaved exponential-Golomb code.
- `uie` – Interpret as an unsigned interleaved exponential-Golomb code.
- `uint` – Interpret as a two's complement unsigned integer.
- `uintbe` – Interpret as a big-endian unsigned integer.
- `uintle` – Interpret as a little-endian unsigned integer.
- `uintne` – Interpret as a native-endian unsigned integer.

## 9.2 BitArray

`BitArray(Bits)`

This class adds mutating methods to *Bits*.

### 9.2.1 Additional methods

- `append` – Append a bitstring.
- `byteswap` – Change byte endianness in-place.
- `clear` – Remove all bits from the bitstring.
- `copy` – Return a copy of the bitstring.
- `insert` – Insert a bitstring.

- `invert` – Flip bit(s) between one and zero.
- `overwrite` – Overwrite a section with a new bitstring.
- `prepend` – Prepend a bitstring.
- `replace` – Replace occurrences of one bitstring with another.
- `reverse` – Reverse bits in-place.
- `rol` – Rotate bits to the left.
- `ror` – Rotate bits to the right.
- `set` – Set bit(s) to 1 or 0.

### 9.2.2 Additional special methods

Mutating operators are available: `[]`, `<<=`, `>>=`, `*=`, `&=`, `|=` and `^=`.

### 9.2.3 Attributes

The same as `Bits`, except that they are all (with the exception of `len`) writable as well as readable.

## 9.3 ConstBitStream

`ConstBitStream(Bits)`

This class, previously known as just `Bits` (which is an alias for backward-compatibility), adds a bit position and methods to read and navigate in the bitstream.

### 9.3.1 Additional methods

- `bytealign` – Align to next byte boundary.
- `peek` – Peek at and interpret next bits as a single item.
- `peeklist` – Peek at and interpret next bits as a list of items.
- `read` – Read and interpret next bits as a single item.
- `readlist` – Read and interpret next bits as a list of items.
- `readto` – Read up to and including next occurrence of a bitstring.

### 9.3.2 Additional attributes

- `bytepos` – The current byte position in the bitstring.
- `pos` – The current bit position in the bitstring.

## 9.4 BitStream

`BitStream(BitArray, ConstBitStream)`

This class, also known as `BitString`, contains all of the 'stream' elements of `ConstBitStream` and adds all of the mutating methods of `BitArray`.

CHAPTER 10

# The bitstring module

The bitstring module provides four classes, `Bits`, `BitArray`, `ConstBitStream` and `BitStream`. `Bits` is the simplest, and represents an immutable sequence of bits, while `BitArray` adds various methods that modify the contents (these classes are intended to loosely mirror `bytes` and `bytearray` in Python 3). The 'Stream' classes have additional methods to treat the bits as a file or stream.

If you need to change the contents of a bitstring after creation then you must use either the `BitArray` or `BitStream` classes. If you need to use bitstrings as keys in a dictionary or members of a set then you must use either a `Bits` or a `ConstBitStream`. In this section the generic term 'bitstring' is used to refer to an object of any of these classes.

Note that for the bitstream classes the bit position within the bitstream (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 `ConstBitStream` object.

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

## 10.1 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=`, `uie=`, `se=` or `sie=` implies an exponential-Golomb coded integer. e.g. `ue=12`, `sie=-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 `bytearray` object.
- An `array` object. This is used after being converted to it's constituent byte data via its `tostring` method.
- In Python 3 only, a `bytes` object. Note this won't work for Python 2.7 as `bytes` is just a synonym for `str`.

## 10.2 Compact format strings

For the *`read`*, *`unpack`*, *`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*.

## 10.3 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. For the mutable *`BitStream`* and *`BitArray`* objects the properties are all read and write (with the exception of the *`len`*), whereas for immutable objects the only write enabled properties are for the position in the bitstream (*`pos`*/*`bitpos`* and *`bytepos`*).

## CHAPTER 11

# The Bits class

**class** `bitstring.`**`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`, `sie`, `uie`, `float`, `floatbe`, `floatle`, `floatne`, `bool` 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 *`BitArray`*, *`BitStream`* and *`ConstBitStream`*.

`offset` is available when using the `bytes` or `filename` initialisers. 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 must also be specified for the float initialisers (the only valid values are 32 and 64). It is optional for the `bytes` and `filename` initialisers and 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=b'\x93@', length=12)
>>> s1 == s2 == s3 == s4 == s5 == s6
True
```

For information on the use of `auto` see *The auto initialiser*.

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

**`all`**`(`*value*[, *pos*]`)`

Returns `True` if all of the specified bits are all set to *value*, otherwise returns `False`.

If *value* is `True` then `1` bits are checked for, otherwise `0` bits are checked for.

*pos* should be 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`. It defaults to the whole bitstring.

```
>>> s = Bits('int:15=-1')
>>> s.all(True, [3, 4, 12, 13])
True
>>> s.all(1)
True
```

**any** (*value*[, *pos* ])

Returns `True` if any of the specified bits are set to *value*, otherwise returns `False`.

If *value* is `True` then `1` bits are checked for, otherwise `0` bits are checked for.

*pos* should be 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`. It defaults to the whole bitstring.

```
>>> s = Bits('0b11011100')
>>> s.any(False, range(6))
True
>>> s.any(1)
True
```

**count** (*value*)

Returns the number of bits set to *value*.

*value* can be `True` or `False` or anything that can be cast to a bool, so you could equally use `1` or `0`.

```
>>> s = BitString(1000000)
>>> s.set(1, [4, 44, 444444])
>>> s.count(1)
3
>>> s.count(False)
999997
```

**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* ])

Searches for *bs* in the current bitstring and sets `pos` to the start of *bs* and returns it in a tuple if found, otherwise it returns an empty tuple.

The reason for returning the bit position in a tuple is so that it evaluates as True even if the bit position is zero. This allows constructs such as `if s.find('0xb3'):` to work as expected.

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)
(16,)
```

**findall** (*bs*[, *start*, *end*, *count*, *bytealigned* ])

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* parameter limits the number of items that will be found - the default is to find all occurrences.

```
>>> 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)
010101010
```

**rfind** (*bs*[, *start*, *end*, *bytealigned* ])

Searches backwards for *bs* in the current bitstring and sets `pos` to the start of *bs* and returns it in a tuple if found, otherwise it returns an empty tuple.

The reason for returning the bit position in a tuple is so that it evaluates as True even if the bit position is zero. This allows constructs such as `if s.rfind('0xb3'):` to work as expected.

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

**split** (*delimiter*[, *start*, *end*, *count*, *bytealigned* ])

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')]
['', '01000', '01001000', '0100011']
```

**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 2.7).

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

This 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=b'hello')
>>> s += '0b01'
>>> s.tobytes()
b'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**(*fmt*, ***kwargs*)

Interprets the whole bitstring according to the *fmt* string or iterable 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.

*fmt* is an iterable or a string with comma separated tokens that describe how to interpret the next bits in the bitstring. See the entry for `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.

**bin**

Property for the representation of the bitstring as a binary string.

**bool**

Property for representing the bitstring as a boolean (`True` or `False`).

If the bitstring is not a single bit then the getter will raise an *`InterpretError`*.

**bytes**

Property representing the underlying byte data that contains the bitstring.

When used as a getter the bitstring must be a whole number of byte long or a *`InterpretError`* 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 = Bits('0x12345678')
>>> s.bytes
b'\x124Vx'
```

**hex**

Property representing the hexadecimal value of the bitstring.

If the bitstring is not a multiple of four bits long then getting its hex value will raise an *`InterpretError`*.

```
>>> s = Bits(bin='1111 0000')
>>> s.hex
'f0'
```

**`int`**

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

**`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 an *`InterpretError`* is 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.

**`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.

**`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 an *`InterpretError`* 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.

If the bitstring is not a multiple of three bits long then getting its octal value will raise a *`InterpretError`*.

```
>>> s = BitString('0b111101101')
>>> s.oct
'755'
>>> s.oct = '01234567'
>>> s.oct
'01234567'
```

**`se`**

Property for the signed exponential-Golomb code representation of the bitstring.

When used as a getter an *`InterpretError`* will be raised if the bitstring is not a single code.

```
>>> s = BitString(se=-40)
>>> s.bin
0000001010001
>>> s += '0b1'
>>> s.se
Error: BitString is not a single exponential-Golomb code.
```

**`ue`**

Property for the unsigned exponential-Golomb code representation of the bitstring.

When used as a getter an *`InterpretError`* will be raised if the bitstring is not a single code.

**`sie`**

Property for the signed interleaved exponential-Golomb code representation of the bitstring.

When used as a getter an *`InterpretError`* will be raised if the bitstring is not a single code.

**`uie`**

Property for the unsigned interleaved exponential-Golomb code representation of the bitstring.

When used as a getter an *`InterpretError`* will be raised if the bitstring is not a single code.

**`uint`**

Property for the unsigned base-2 integer representation of the bitstring.

**`uintbe`**

Property for the byte-wise big-endian unsigned base-2 integer representation of the bitstring.

**`uintle`**

Property for the byte-wise little-endian unsigned base-2 integer representation of the bitstring.

**`uintne`**

Property for the byte-wise native-endian unsigned base-2 integer representation of the bitstring.

**`__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
```

**`__bool__`**()

`if s:`

Returns `True` if at least one bit is set to 1, otherwise returns `False`.

This special method is used in Python 3 only; for Python 2.7 the equivalent is called `__nonzero__`, but the details are exactly the same.

```
>>> bool(Bits())
False
>>> bool(Bits('0b0000010000'))
True
>>> bool(Bits('0b0000000000'))
False
```

**__contains__**(*bs*)

`bs in s`

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

Similar to using *find*, except that you are only told if it is found, and not 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.

```
>>> s = Bits('0x0123456')
>>> s[4:8]
Bits('0x1')
>>> s[1::8] # 1st, 9th, 17th and 25th bits
Bits('0x3')
```

If a single element is asked for then either `True` or `False` will be returned.

```
>>> s[0]
False
```

(continues on next page)

(continued from previous page)

```
>>> s[-1]
True
```

**__hash__**()

`hash(s)`

Returns an integer hash of the `Bits`.

This method is not available for the `BitArray` or `BitStream` classes, 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 an `Error` will be raised.

```
>>> s = ConstBitStream('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`.

**`__nonzero__`**()

See `__bool__`.

**`__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 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
```

## CHAPTER 12

# The BitArray class

**class** `bitstring.`**`BitArray`**(*[auto, length, offset, **kwargs]*)

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

A `BitArray` 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 `BitArray` to the end of the current `BitArray`.

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

**`byteswap`**(*[fmt, start, end, repeat=True]*)

Change the endianness of the `BitArray` in-place according to *fmt*. Return the number of swaps done.

The *fmt* 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 defaults to 0, which means reverse as many bytes as possible. The *fmt* gives a pattern of byte sizes to use to swap the endianness of the `BitArray`. 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 `BitArray`). If *repeat* is `True` then the byte swapping pattern given by the *fmt* is repeated in its entirety as many times as possible.

```
>>> s = BitArray('0x00112233445566')
>>> s.byteswap(2)
3
>>> s
BitArray('0x11003322554466')
>>> s.byteswap('h')
3
>>> s
BitArray('0x00112233445566')
```

(continues on next page)

(continued from previous page)

```
>>> s.byteswap([2, 5])
1
>>> s
BitArray('0x11006655443322')
```

It can also be used to swap the endianness of the whole `BitArray`.

```
>>> s = BitArray('uintle:32=1234')
>>> s.byteswap()
>>> print(s.uintbe)
1234
```

**clear**()

Removes all bits from the bitstring.

`s.clear()` is equivalent to `del s[:]` and simply makes the bitstring empty.

**copy**()

Returns a copy of the bitstring.

`s.copy()` is equivalent to the shallow copy `s[:]` and creates a new copy of the bitstring in memory.

**insert**(*bs*, *pos*)

Inserts *bs* at *pos*.

When used with the `BitStream` class the *pos* is optional, and if not present the current bit position will be used. After insertion the property `pos` will be immediately after the inserted bitstring.

```
>>> s = BitStream('0xccee')
>>> s.insert('0xd', 8)
>>> s
BitStream('0xccdee')
>>> s.insert('0x00')
>>> s
BitStream('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`. The default is to invert the entire `BitArray`.

```
>>> s = BitArray('0b111001')
>>> s.invert(0)
>>> s.bin
'011001'
>>> s.invert([-2, -1])
>>> s.bin
'011010'
>>> s.invert()
>>> s.bin
'100101'
```

**overwrite**(*bs*, *pos*)

Replaces the contents of the current `BitArray` with *bs* at *pos*.

When used with the `BitStream` class the *pos* is optional, and if not present the current bit position will be used. After insertion the property `pos` will be immediately after the overwritten bitstring.

```
>>> s = BitArray(length=10)
>>> s.overwrite('0b111', 3)
>>> s
```

(continues on next page)

(continued from previous page)

```
BitArray('0b0001110000')
>>> s.pos
6
```

**prepend** (*bs*)

Inserts *bs* at the beginning of the current *BitArray*.

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

**replace** (*old*, *new*[, *start*, *end*, *count*, *bytealigned* ])

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 = BitArray('0b0011001')
>>> s.replace('0b1', '0xf')
3
>>> print(s.bin)
0011111111001111
>>> s.replace('0b1', '', count=6)
6
>>> print(s.bin)
0011001111
```

**reverse** ([*start*, *end* ])

Reverses bits in the *BitArray* in-place.

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

```
>>> a = BitArray('0b10111')
>>> a.reverse()
>>> a.bin
'11101'
```

**rol** (*bits*[, *start*, *end* ])

Rotates the contents of the *BitArray* 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 = BitArray('0b01000001')
>>> s.rol(2)
>>> s.bin
'00000101'
```

**ror** (*bits*[, *start*, *end* ])

Rotates the contents of the *BitArray* 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** (*value*[, *pos* ])

Sets one or many bits to either `1` (if *value* is `True`) or `0` (if *value* isn't `True`). *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`. The default is to set every bit in the *BitArray*.

Using `s.set(True, x)` can be more efficient than other equivalent methods such as `s[x] = 1`, `s[x] = "0b1"` or `s.overwrite('0b1', x)`, especially if many bits are being set.

```
>>> s = BitArray('0x0000')
>>> s.set(True, -1)
>>> print(s)
0x0001
>>> s.set(1, (0, 4, 5, 7, 9))
>>> s.bin
'1000110101000001'
>>> s.set(0)
>>> s.bin
'0000000000000000'
```

**bin**
Writable version of *`Bits.bin`*.

**bool**
Writable version of *`Bits.bool`*.

**bytes**
Writable version of *`Bits.bytes`*.

**hex**
Writable version of *`Bits.hex`*.

**int**
Writable version of *`Bits.int`*.

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

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

**intbe**
Writable version of *`Bits.intbe`*.

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

**intle**
Writable version of *`Bits.intle`*.

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

**intne**
Writable version of *`Bits.intne`*.

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

**float**

**floatbe**
Writable version of *`Bits.float`*.

**floatle**
Writable version of *`Bits.floatle`*.

**floatne**
Writable version of *`Bits.floatne`*.

**oct**
Writable version of *Bits.oct*.

**se**
Writable version of *Bits.se*.

**ue**
Writable version of *Bits.uie*.

**sie**
Writable version of *Bits.sie*.

**uie**
Writable version of *Bits.ue*.

**uint**
Writable version of *Bits.uint*.

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

**uintbe**
Writable version of *Bits.uintbe*.

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

**uintle**
Writable version of *Bits.uintle*.

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

**uintne**
Writable version of *Bits.uintne*.

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

**__delitem__**(*key*)
`del s[start:end:step]`

Deletes the slice specified.

**__iadd__**(*bs*)
`s1 += s2`

Appends *bs* to the current bitstring.

Note that for *BitArray* 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 *__add__*).

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

**__iand__**(*bs*)
`s &= bs`

In-place bit-wise AND between two bitstrings. If the two bitstrings are not the same length then a `ValueError` is raised.

**__ilshift__**(*n*)
`s <<= n`

Shifts the bits in-place *n* bits to the left. The *n* right-most bits will become zeros and bits shifted off the left will be lost.

**__imul__**(*n*)

`s *= n`

In-place concatenation of *n* copies of the current bitstring.

```
>>> s = BitArray('0xbad')
>>> s *= 3
>>> s.hex
'badbadbad'
```

**__ior__**(*bs*)

`s |= bs`

In-place bit-wise OR between two bitstrings. If the two bitstrings are not the same length then a `ValueError` is raised.

**__irshift__**(*n*)

`s >>= n`

Shifts the bits in-place *n* bits to the right. The *n* left-most bits will become zeros and bits shifted off the right will be lost.

**__ixor__**(*bs*)

`s ^= bs`

In-place bit-wise XOR between two bitstrings. If the two bitstrings are not the same length then a `ValueError` is raised.

**__setitem__**(*key*, *value*)

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

Replaces the slice specified with a new value.

```
>>> s = BitArray('0x00000000')
>>> s[::8] = '0xf'
>>> print(s)
0x80808080
>>> s[-12:] = '0xf'
>>> print(s)
0x80808f
```

## CHAPTER 13

# The ConstBitStream class

**class** bitstring.**ConstBitStream**([*auto*, *length*, *offset*, ***kwargs*])

The `Bits` class is the base class for `ConstBitStream` and so all of its methods are also available for `ConstBitStream` objects. The initialiser is also the same as for `Bits` and so won't be repeated here.

A `ConstBitStream` is a `Bits` with added methods and properties that allow it to be parsed as a stream of bits.

**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 = ConstBitStream('0xabcdef')
>>> s.pos += 3
>>> s.bytealign()
5
>>> s.pos
8
```

**peek**(*fmt*)

Reads from the current bit position `pos` in the bitstring according to the *fmt* string or integer and returns the result.

The bit position is unchanged.

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

```
>>> s = ConstBitStream('0x123456')
>>> s.peek(16)
ConstBitStream('0x1234')
>>> s.peek('hex:8')
'12'
```

**peeklist**(*fmt*, ***kwargs*)

Reads from current bit position `pos` in the bitstring according to the *fmt* string or iterable 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 *read* and *readlist* for more information.

**read** (*fmt*)

Reads from current bit position *pos* in the bitstring according the format string and returns a single result. If not enough bits are available then a *ReadError* is raised.

*fmt* is either a token string that describes how to interpret the next bits in the bitstring or an integer. If it's an integer then that number of bits will be read, and returned as a new bitstring. Otherwise 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. |
| `uie` | next bits as an interleaved unsigned exp-Golomb. |
| `sie` | next bits as an interleaved signed exp-Golomb. |
| `bits:n` | `n` bits as a new bitstring. |
| `bytes:n` | `n` bytes as `bytes` object. |
| `bool[:1]` | next bit as a boolean (True or False). |
| `pad:n` | next `n` bits will be skipped. |

For example:

```
>>> s = ConstBitStream('0x23ef55302')
>>> s.read('hex:12')
'23e'
>>> s.read('bin:4')
'1111'
>>> s.read('uint:5')
10
>>> s.read('bits:4')
ConstBitStream('0xa')
```

The *read* method is useful for reading exponential-Golomb codes.

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

The `pad` token is not very useful when used in *read* as it just skips a number of bits and returns `None`. However when used within *readlist* or *unpack* it allows unimportant part of the bitstring to be simply ignored.

**readlist** (*fmt*, ***kwargs*)

Reads from current bit position *pos* in the bitstring according to the *fmt* string or iterable and returns a list of results. If not enough bits are available then a *`ReadError`* is raised.

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 *`read`* for information on the format strings.

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

```
>>> s = ConstBitStream('0x43fe01ff21')
>>> s.readlist('hex:8, uint:6')
['43', 63]
>>> s.readlist(['bin:3', 'intle:16'])
['100', -509]
>>> s.pos = 0
>>> s.readlist('hex:b, uint:d', b=8, d=6)
['43', 63]
```

**`readto`**(*bs*, *bytealigned*)

Reads up to and including the next occurrence of the bitstring *bs* and returns the results. If *bytealigned* is *True* it will look for the bitstring starting only at whole-byte positions.

Raises a *`ReadError`* if *bs* is not found, and `ValueError` if *bs* is empty.

```
>>> s = ConstBitStream('0x47000102034704050647')
>>> s.readto('0x47', bytealigned=True)
BitStream('0x47')
>>> s.readto('0x47', bytealigned=True)
BitStream('0x0001020347')
>>> s.readto('0x47', bytealigned=True)
BitStream('0x04050647')
```

**`bytepos`**

Property for setting and getting the current byte position in the bitstring.

When used as a getter will raise a *`ByteAlignError`* if the current position in not byte aligned.

**`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
```

# CHAPTER 14

# The BitStream class

**class** bitstring.**BitStream**([*auto*, *length*, *offset*, ***kwargs*])

Both the *BitArray* and the *ConstBitStream* classes are base classes for *BitStream* and so all of their methods are also available for *BitStream* objects. The initialiser is also the same as for *Bits* and so won't be repeated here.

A *BitStream* is a mutable container of bits with methods and properties that allow it to be parsed as a stream of bits. There are no additional methods or properties in this class - see its base classes (*Bits*, *BitArray* and *ConstBitStream*) for details.

## CHAPTER 15

# Functions

`bitstring.`**`pack`** (*format*[, **values*, ***kwargs* ])

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

**Parameters**

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

**Return type** *BitStream*

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

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

They can also have the value missing, in which case the values 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 16

# Exceptions

**exception** `bitstring.`**`Error`**(*Exception*)
Base class for all module exceptions.

**exception** `bitstring.`**`InterpretError`**(*Error*, *ValueError*)
Inappropriate interpretation of binary data. For example using the 'bytes' property on a bitstring that isn't a whole number of bytes long.

**exception** `bitstring.`**`ByteAlignError`**(*Error*)
Whole-byte position or length needed.

**exception** `bitstring.`**`CreationError`**(*Error*, *ValueError*)
Inappropriate argument during bitstring creation.

**exception** `bitstring.`**`ReadError`**(*Error*, *IndexError*)
Reading or peeking past the end of a bitstring.

# 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 17

# Examples

## 17.1 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 = BitArray('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('2*bin:4')
```

## 17.2 Manipulation

```
s = BitArray('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', BitArray(bytes='hello'))

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

## 17.3 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('4*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 = BitStream('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]
```

## 17.4 Sieve of Eratosthenes

This classic (though inefficient) method of calculating prime numbers uses a bitstring to store whether each bit position represents a prime number. This takes much less memory than an ordinary array.

```
def prime_sieve(top=1000000):
    b = BitArray(top) # bitstring of '0' bits
    for i in xrange(2, top):
        if not b[i]:
            yield i
            # i is prime, so set all its multiples to '1'.
            b.set(True, xrange(i*i, top, i))
```

# CHAPTER 18

# Exponential-Golomb Codes

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

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

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

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

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

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

`001001101101101011000100100101`

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

```
a = BitStream().join([BitArray(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. There are other ways to map the bitstrings to integers:

## 18.1 Interleaved exponential-Golomb codes

This type of code is used in the Dirac video standard, and is represented by the attributes `uie` and `sie`. For the interleaved codes the pattern is very similar to before for the unsigned case:

| Bit pattern | Unsigned |
|---|---|
| `1` | 0 |
| `001` | 1 |
| `011` | 2 |
| `00001` | 3 |
| `00011` | 4 |
| `01001` | 5 |
| `01011` | 6 |
| `0000001` | 7 |
| `0000011` | 8 |
| `0001001` | 9 |
| `...` | ... |

For the signed code it looks a little different:

| Bit pattern | Signed |
|---|---|
| `1` | 0 |
| `0010` | 1 |
| `0011` | -1 |
| `0110` | 2 |
| `0111` | -2 |
| `000010` | 3 |
| `000011` | -3 |
| `000110` | 4 |
| `000111` | -4 |
| `010010` | 5 |
| `010011` | -5 |
| `...` | ... |

I’m sure you can work out the pattern yourself from here!

# CHAPTER 19

# Optimisation Techniques

The *bitstring* 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.

## 19.1 Use combined read and interpretation

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

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

This works fine, but is not very quick. The problem is that the call to `read` 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 it is better to use a string parameter 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 probably not as fast as the combined call:

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

## 19.2 Choose the simplest class you can

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

The speed difference between the classes is noticeable, and there are also memory usage optimisations that are made if objects are known to be immutable.

You should also prefer *ConstBitStream* to *BitStream* if you won't need to modify any bits.

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 *`BitArray`* then the addition would have been done in-place, and have been much more efficient.

## 19.3 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 = BitArray(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. You could make it slightly quicker by using `s[p] = True`, but it is much faster (and I mean at least an order of magnitude) to use the *`set`* method:

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

As well as *`set`* and *`invert`* there are also checking methods *`all`* and *`any`*. So rather than using

```
if s[100] and s[200]:
    do_something()
```

it's better to say

```
if s.all(True, (100, 200)):
    do_something()
```

CHAPTER 20

# Release Notes

## 20.1 Full Version History

### 20.1.1 July 9th 2019: version 3.1.6 released

A long overdue maintenace release with some fixes.

- Fixed immutability bug. Bug 176.
- Fixed failure of __*contains*__ in some circumstances. Bug 180.
- Better handling of open files. Bug 186.
- Better Python 2/3 check.
- Making unit tests easier to run.
- Allowing length of 1 to be specified for bools. (Thanks to LemonPi)

### 20.1.2 May 17th 2016: version 3.1.5 released

- Support initialisation from an array.
- Added a separate LICENSE file.

### 20.1.3 March 19th 2016: version 3.1.4 released

This is another bug fix release.

- Fix for bitstring types when created directly from other bitstring types.
- Updating contact, website details.

### 20.1.4 March 4th 2014: version 3.1.3 released

This is another bug fix release.

- Fix for problem with prepend for bitstrings with byte offsets in their data store.

### 20.1.5 April 18th 2013: version 3.1.2 released

This is another bug fix release.

- Fix for problem where unpacking bytes would by eight times too long

### 20.1.6 March 21st 2013: version 3.1.1 released

This is a bug fix release.

- Fix for problem where concatenating bitstrings sometimes modified method's arguments

### 20.1.7 February 26th 2013: version 3.1.0 released

This is a minor release with a couple of new features and some bug fixes.

#### New 'pad' token

This token can be used in reads and when packing/unpacking to indicate that you don't care about the contents of these bits. Any padding bits will just be skipped over when reading/unpacking or zero-filled when packing.

```
>>> a, b = s.readlist('pad:5, uint:3, pad:1, uint:3')
```

Here only two items are returned in the list - the padding bits are ignored.

#### New clear and copy convenience methods

These methods have been introduced in Python 3.3 for lists and bytearrays, as more obvious ways of clearing and copying, and we mirror that change here.

`t = s.copy()` is equivalent to `t = s[:]`, and `s.clear()` is equivalent to `del s[:]`.

#### Other changes

- Some bug fixes.

### 20.1.8 November 21st 2011: version 3.0.0 released

This is a major release which breaks backward compatibility in a few places.

### 20.1.9 Backwardly incompatible changes

#### Hex, oct and bin properties don't have leading 0x, 0o and 0b

If you ask for the hex, octal or binary representations of a bitstring then they will no longer be prefixed with `0x`, `0o` or `0b`. This was done as it was noticed that the first thing a lot of user code does after getting these representations was to cut off the first two characters before further processing.

```
>>> a = BitArray('0x123')
>>> a.hex, a.oct, a.bin
('123', '0443', '000100100011')
```

Previously this would have returned `('0x123', '0o0443', '0b000100100011')`

This change might require some recoding, but it should all be simplifications.

### ConstBitArray renamed to Bits

Previously Bits was an alias for ConstBitStream (for backward compatibility). This has now changed so that Bits and BitArray loosely correspond to the built-in types bytes and bytearray.

If you were using streaming/reading methods on a Bits object then you will have to change it to a ConstBitStream.

The ConstBitArray name is kept as an alias for Bits.

### Stepping in slices has conventional meaning

The step parameter in `__getitem__`, `__setitem__` and `__delitem__` used to act as a multiplier for the start and stop parameters. No one seemed to use it though and so it has now reverted to the conventional meaning for containers.

If you are using step then recoding is simple: `s[a:b:c]` becomes `s[a*c:b*c]`.

Some examples of the new usage:

```
>>> s = BitArray('0x0000')
s[::4] = [1, 1, 1, 1]
>>> s.hex
'8888'
>>> del s[8::2]
>>> s.hex
'880'
```

## 20.1.10 New features

### New readto method

This method is a mix between a find and a read - it searches for a bitstring and then reads up to and including it. For example:

```
>>> s = ConstBitStream('0x47000102034704050647')
>>> s.readto('0x47', bytealigned=True)
BitStream('0x47')
>>> s.readto('0x47', bytealigned=True)
BitStream('0x0001020347')
>>> s.readto('0x47', bytealigned=True)
BitStream('0x04050647')
```

### pack function accepts an iterable as its format

Previously only a string was accepted as the format in the pack function. This was an oversight as it broke the symmetry between pack and unpack. Now you can use formats like this:

```
fmt = ['hex:8', 'bin:3']
a = pack(fmt, '47', '001')
a.unpack(fmt)
```

## 20.1.11 June 18th 2011: version 2.2.0 released

This is a minor upgrade with a couple of new features.

### New interleaved exponential-Golomb interpretations

New bit interpretations for interleaved exponential-Golomb (as used in the Dirac video codec) are supplied via `uie` and `sie`:

```
>>> s = BitArray(uie=41)
>>> s.uie
41
>>> s.bin
'0b00010001001'
```

These are pretty similar to the non-interleaved versions - see the manual for more details. Credit goes to Paul Sargent for the patch.

### New package-level bytealigned variable

A number of methods take a `bytealigned` parameter to indicate that they should only work on byte boundaries (e.g. `find`, `replace`, `split`). Previously this parameter defaulted to `False`. Instead it now defaults to `bitstring.bytealigned`, which itself defaults to `False`, but can be changed to modify the default behaviour of the methods. For example:

```
>>> a = BitArray('0x00 ff 0f ff')
>>> a.find('0x0f')
(4,)    # found first not on a byte boundary
>>> a.find('0x0f', bytealigned=True)
(16,)   # forced looking only on byte boundaries
>>> bitstring.bytealigned = True  # Change default behaviour
>>> a.find('0x0f')
(16,)
>>> a.find('0x0f', bytealigned=False)
(4,)
```

If you're only working with bytes then this can help avoid some errors and save some typing!

### Other changes

- Fix for Python 3.2, correcting for a change to the binascii module.
- Fix for bool initialisation from `0` or `1`.
- Efficiency improvements, including interning strategy.

## 20.1.12 February 23rd 2011: version 2.1.1 released

This is a release to fix a couple of bugs that were introduced in 2.1.0.

- Bug fix: Reading using the 'bytes' token had been broken (Issue 102).
- Fixed problem using some methods on `ConstBitArray` objects.
- Better exception handling for tokens missing values.
- Some performance improvements.

## 20.1.13 January 23rd 2011: version 2.1.0 released

## 20.1.14 New class hierarchy introduced with simpler classes

Previously there were just two classes, the immutable `Bits` which was the base class for the mutable `BitString` class. Both of these classes have the concept of a bit position, from which reads etc. take place so that the bitstring could be treated as if it were a file or stream.

Two simpler classes have now been added which are purely bit containers and don't have a bit position. These are called `ConstBitArray` and `BitArray`. As you can guess the former is an immutable version of the latter.

The other classes have also been renamed to better reflect their capabilities. Instead of `BitString` you should use `BitStream`, and instead of `Bits` you can use `ConstBitStream`. The old names are kept as aliases for backward compatibility.

The classes hierarchy is:

```
    ConstBitArray
       /    \
      /      \
BitArray   ConstBitStream (formerly Bits)
      \      /
       \    /
      BitStream (formerly BitString)
```

### Other changes

A lot of internal reorganisation has taken place since the previous version, most of which won't be noticed by the end user. Some things you might see are:

- New package structure. Previous versions have been a single file for the module and another for the unit tests. The module is now split into many more files so it can't be used just by copying bitstring.py any more.
- To run the unit tests there is now a script called `runtests.py` in the `test` directory.
- File based bitstring are now implemented in terms of an mmap. This should be just an implementation detail, but unfortunately for 32-bit versions of Python this creates a limit of 4GB on the files that can be used. The work around is either to get a 64-bit Python, or just stick with version 2.0.
- The `ConstBitArray` and `ConstBitStream` classes no longer copy byte data when a slice or a read takes place, they just take a reference. This is mostly a very nice optimisation, but there are occasions where it could have an adverse effect. For example if a very large bitstring is created, a small slice taken and the original deleted. The byte data from the large bitstring would still be retained in memory.
- Optimisations. Once again this version should be faster than the last. The module is still pure Python but some of the reorganisation was to make it more feasible to put some of the code into Cython or similar, so hopefully more speed will be on the way.

## 20.1.15 July 26th 2010: version 2.0.3 released

1. **Bug fix**: Using peek and read for a single bit now returns a new bitstring as was intended, rather than the old behaviour of returning a bool.
2. Removed HTML docs from source archive - better to use the online version.

## 20.1.16 July 25th 2010: version 2.0.2 released

This is a major release, with a number of backwardly incompatible changes. The main change is the removal of many methods, all of which have simple alternatives. Other changes are quite minor but may need some recoding.

There are a few new features, most of which have been made to help the stream-lining of the API. As always there are performance improvements and some API changes were made purely with future performance in mind.

## 20.1.17 The backwardly incompatible changes are:

### Methods removed

About half of the class methods have been removed from the API. They all have simple alternatives, so what remains is more powerful and easier to remember. The removed methods are listed here on the left, with their equivalent replacements on the right:

```
s.advancebit()              ->   s.pos += 1
s.advancebits(bits)         ->   s.pos += bits
s.advancebyte()             ->   s.pos += 8
s.advancebytes(bytes)       ->   s.pos += 8*bytes
s.allunset([a, b])          ->   s.all(False, [a, b])
s.anyunset([a, b])          ->   s.any(False, [a, b])
s.delete(bits, pos)         ->   del s[pos:pos+bits]
s.peekbit()                 ->   s.peek(1)
s.peekbitlist(a, b)         ->   s.peeklist([a, b])
s.peekbits(bits)            ->   s.peek(bits)
s.peekbyte()                ->   s.peek(8)
s.peekbytelist(a, b)        ->   s.peeklist([8*a, 8*b])
s.peekbytes(bytes)          ->   s.peek(8*bytes)
s.readbit()                 ->   s.read(1)
s.readbitlist(a, b)         ->   s.readlist([a, b])
s.readbits(bits)            ->   s.read(bits)
s.readbyte()                ->   s.read(8)
s.readbytelist(a, b)        ->   s.readlist([8*a, 8*b])
s.readbytes(bytes)          ->   s.read(8*bytes)
s.retreatbit()              ->   s.pos -= 1
s.retreatbits(bits)         ->   s.pos -= bits
s.retreatbyte()             ->   s.pos -= 8
s.retreatbytes(bytes)       ->   s.pos -= 8*bytes
s.reversebytes(start, end)  ->   s.byteswap(0, start, end)
s.seek(pos)                 ->   s.pos = pos
s.seekbyte(bytepos)         ->   s.bytepos = bytepos
s.slice(start, end, step)   ->   s[start:end:step]
s.tell()                    ->   s.pos
s.tellbyte()                ->   s.bytepos
s.truncateend(bits)         ->   del s[-bits:]
s.truncatestart(bits)       ->   del s[:bits]
s.unset([a, b])             ->   s.set(False, [a, b])
```

Many of these methods have been deprecated for the last few releases, but there are some new removals too. Any recoding needed should be quite straightforward, so while I apologise for the hassle, I had to take the opportunity to streamline and rationalise what was becoming a bit of an overblown API.

### set / unset methods combined

The `set`/`unset` methods have been combined in a single method, which now takes a boolean as its first argument:

```
s.set([a, b])               ->   s.set(1, [a, b])
s.unset([a, b])             ->   s.set(0, [a, b])
s.allset([a, b])            ->   s.all(1, [a, b])
s.allunset([a, b])          ->   s.all(0, [a, b])
s.anyset([a, b])            ->   s.any(1, [a, b])
s.anyunset([a, b])          ->   s.any(0, [a, b])
```

### all / any only accept iterables

The `all` and `any` methods (previously called `allset`, `allunset`, `anyset` and `anyunset`) no longer accept a single bit position. The recommended way of testing a single bit is just to index it, for example instead of:

```
>>> if s.all(True, i):
```

just use

```
>>> if s[i]:
```

If you really want to you can of course use an iterable with a single element, such as `s.any(False, [i])`, but it's clearer just to write `not s[i]`.

### Exception raised on reading off end of bitstring

If a read or peek goes beyond the end of the bitstring then a `ReadError` will be raised. The previous behaviour was that the rest of the bitstring would be returned and no exception raised.

### `BitStringError` renamed to `Error`

The base class for errors in the bitstring module is now just `Error`, so it will likely appears in your code as `bitstring.Error` instead of the rather repetitive `bitstring.BitStringError`.

### Single bit slices and reads return a bool

A single index slice (such as `s[5]`) will now return a bool (i.e. `True` or `False`) rather than a single bit bitstring. This is partly to reflect the style of the `bytearray` type, which returns an integer for single items, but mostly to avoid common errors like:

```
>>> if s[0]:
...     do_something()
```

While the intent of this code snippet is quite clear (i.e. do_something if the first bit of s is set) under the old rules `s[0]` would be true as long as `s` wasn't empty. That's because any one-bit bitstring was true as it was a non-empty container. Under the new rule `s[0]` is `True` if `s` starts with a `1` bit and `False` if `s` starts with a `0` bit.

The change does not affect reads and peeks, so `s.peek(1)` will still return a single bit bitstring, which leads on to the next item…

### Empty bitstrings or bitstrings with only zero bits are considered False

Previously a bitstring was False if it had no elements, otherwise it was True. This is standard behaviour for containers, but wasn't very useful for a container of just 0s and 1s. The new behaviour means that the bitstring is False if it has no 1 bits. This means that code like this:

```
>>> if s.peek(1):
...     do_something()
```

should work as you'd expect. It also means that `Bits(1000)`, `Bits(0x00)` and `Bits('uint:12=0')` are all also `False`. If you need to check for the emptiness of a bitstring then instead check the `len` property:

```
if s                -> if s.len
if not s            -> if not s.len
```

### Length and offset disallowed for some initialisers

Previously you could create bitstring using expressions like:

```
>>> s = Bits(hex='0xabcde', offset=4, length=13)
```

This has now been disallowed, and the offset and length parameters may only be used when initialising with bytes or a file. To replace the old behaviour you could instead use

```
>>> s = Bits(hex='0xabcde')[4:17]
```

### Renamed `format` parameter `fmt`

Methods with a `format` parameter have had it renamed to `fmt`, to prevent hiding the built-in `format`. Affects methods `unpack`, `read`, `peek`, `readlist`, `peeklist` and `byteswap` and the `pack` function.

### Iterables instead of `*` format accepted for some methods

This means that for the affected methods (`unpack`, `readlist` and `peeklist`) you will need to use an iterable to specify multiple items. This is easier to show than to describe, so instead of

```
>>> a, b, c, d = s.readlist('uint:12', 'hex:4', 'bin:7')
```

you would instead write

```
>>> a, b, c, d = s.readlist(['uint:12', 'hex:4', 'bin:7'])
```

Note that you could still use the single string `'uint:12, hex:4, bin:7'` if you preferred.

### Bool auto-initialisation removed

You can no longer use `True` and `False` to initialise single bit bitstrings. The reasoning behind this is that as bool is a subclass of int, it really is bad practice to have `Bits(False)` be different to `Bits(0)` and to have `Bits(True)` different to `Bits(1)`.

If you have used bool auto-initialisation then you will have to be careful to replace it as the bools will now be interpreted as ints, so `Bits(False)` will be empty (a bitstring of length 0), and `Bits(True)` will be a single zero bit (a bitstring of length 1). Sorry for the confusion, but I think this will prevent bigger problems in the future.

There are a few alternatives for creating a single bit bitstring. My favourite is to use a list with a single item:

```
Bits(False)            ->    Bits([0])
Bits(True)             ->    Bits([1])
```

### New creation from file strategy

Previously if you created a bitstring from a file, either by auto-initialising with a file object or using the filename parameter, the file would not be read into memory unless you tried to modify it, at which point the whole file would be read.

The new behaviour depends on whether you create a Bits or a BitString from the file. If you create a Bits (which is immutable) then the file will never be read into memory. This allows very large files to be opened for examination even if they could never fit in memory.

If however you create a BitString, the whole of the referenced file will be read to store in memory. If the file is very big this could take a long time, or fail, but the idea is that in saying you want the mutable BitString you are implicitly saying that you want to make changes and so (for now) we need to load it into memory.

The new strategy is a bit more predictable in terms of performance than the old. The main point to remember is that if you want to open a file and don't plan to alter the bitstring then use the Bits class rather than BitString.

Just to be clear, in neither case will the contents of the file ever be changed - if you want to output the modified BitString then use the `tofile` method, for example.

#### find and rfind return a tuple instead of a bool

If a find is unsuccessful then an empty tuple is returned (which is False in a boolean sense) otherwise a single item tuple with the bit position is returned (which is True in a boolean sense). You shouldn't need to recode unless you explicitly compared the result of a find to True or False, for example this snippet doesn't need to be altered:

```
>>> if s.find('0x23'):
...     print(s.bitpos)
```

but you could now instead use

```
>>> found = s.find('0x23')
>>> if found:
...     print(found[0])
```

The reason for returning the bit position in a tuple is so that finding at position zero can still be True - it's the tuple `(0,)` - whereas not found can be False - the empty tuple `()`.

### 20.1.18 The new features in this release are:

#### New count method

This method just counts the number of 1 or 0 bits in the bitstring.

```
>>> s = Bits('0x31fff4')
>>> s.count(1)
16
```

#### `read` and `peek` methods accept integers

The `read`, `readlist`, `peek` and `peeklist` methods now accept integers as parameters to mean "read this many bits and return a bitstring". This has allowed a number of methods to be removed from this release, so for example instead of:

```
>>> a, b, c = s.readbits(5, 6, 7)
>>> if s.peekbit():
...     do_something()
```

you should write:

```
>>> a, b, c = s.readlist([5, 6, 7])
>>> if s.peek(1):
...     do_something()
```

#### `byteswap` used to reverse all bytes

The `byteswap` method now allows a format specifier of 0 (the default) to signify that all of the whole bytes should be reversed. This means that calling just `byteswap()` is almost equivalent to the now removed `bytereverse()` method (a small difference is that byteswap won't raise an exception if the bitstring isn't a whole number of bytes long).

#### Auto initialise with bytearray or (for Python 3 only) bytes

So rather than writing:

```
>>> a = Bits(bytes=some_bytearray)
```

you can just write

```
>>> a = Bits(some_bytearray)
```

This also works for the bytes type, but only if you're using Python 3. For Python 2.7 it's not possible to distinguish between a bytes object and a str. For this reason this method should be used with some caution as it will make you code behave differently with the different major Python versions.

```
>>> b = Bits(b'abcd\x23\x00') # Only Python 3!
```

### `set`, `invert`, `all` and `any` default to whole bitstring

This means that you can for example write:

```
>>> a = BitString(100)        # 100 zero bits
>>> a.set(1)                  # set all bits to 1
>>> a.all(1)                  # are all bits set to 1?
True
>>> a.any(0)                  # are any set to 0?
False
>>> a.invert()                # invert every bit
```

### New exception types

As well as renaming `BitStringError` to just `Error` there are also new exceptions which use `Error` as a base class.

These can be caught in preference to `Error` if you need finer control. The new exceptions sometimes also derive from built-in exceptions:

3. `ByteAlignError(Error)` - whole byte position or length needed.
4. `ReadError(Error, IndexError)` - reading or peeking off the end of the bitstring.
5. `CreationError(Error, ValueError)` - inappropriate argument during bitstring creation.
6. `InterpretError(Error, ValueError)` - inappropriate interpretation of binary data.

## 20.1.19 March 18th 2010: version 1.3.0 for Python 2.6 and 3.x released

## 20.1.20 New features

### byteswap method for changing endianness

Changes the endianness in-place according to a format string or integer(s) giving the byte pattern. See the manual for details.

```
>>> s = BitString('0x00112233445566')
>>> s.byteswap(2)
3
>>> s
BitString('0x11003322554466')
>>> s.byteswap('h')
3
>>> s
BitString('0x00112233445566')
```

(continues on next page)

(continued from previous page)

```
>>> s.byteswap([2, 5])
1
>>> s
BitString('0x11006655443322')
```

### Multiplicative factors in bitstring creation and reading

For example:

```
>>> s = Bits('100*0x123')
```

### Token grouping using parenthesis

For example:

```
>>> s = Bits('3*(uint:6=3, 0b1)')
```

### Negative slice indices allowed

The start and end parameters of many methods may now be negative, with the same meaning as for negative slice indices. Affects all methods with these parameters.

### Sequence ABCs used

The Bits class now derives from `collections.Sequence`, while the BitString class derives from `collections.MutableSequence`.

### Keywords allowed in `readlist`, `peeklist` and `unpack`

Keywords for token lengths are now permitted when reading. So for example, you can write

```
>>> s = bitstring.pack('4*(uint:n)', 2, 3, 4, 5, n=7)
>>> s.unpack('4*(uint:n)', n=7)
[2, 3, 4, 5]
```

### start and end parameters added to `rol` and `ror`

### join function accepts other iterables

Also its parameter has changed from ‘bitstringlist’ to ‘sequence’. This is technically a backward incompatibility in the unlikely event that you are referring to the parameter by name.

### `__init__` method accepts keywords

Rather than a long list of initialisers the `__init__` methods now use a `**kwargs` dictionary for all initialisers except ‘auto’. This should have no effect, except that this is a small backward incompatibility if you use positional arguments when initialising with anything other than auto (which would be rather unusual).

### More optimisations

A number of methods have been speeded up.

### Bug fixed in replace method

(it could fail if start != 0).

## 20.1.21 January 19th 2010: version 1.2.0 for Python 2.6 and 3.x released

## 20.1.22 New 'Bits' class

Introducing a brand new class, Bits, representing an immutable sequence of bits.

The Bits class is the base class for the mutable BitString. The differences between Bits and BitStrings are:

- Bits are immutable, so once they have been created their value cannot change. This of course means that mutating methods (append, replace, del etc.) are not available for Bits.
- Bits are hashable, so they can be used in sets and as keys in dictionaries.
- Bits are potentially more efficient than BitStrings, both in terms of computation and memory. The current implementation is only marginally more efficient though - this should improve in future versions.

You can switch from Bits to a BitString or vice versa by constructing a new object from the old.

```
>>> s = Bits('0xabcd')
>>> t = BitString(s)
>>> t.append('0xe')
>>> u = Bits(t)
```

The relationship between Bits and BitString is supposed to loosely mirror that between bytes and bytearray in Python 3.

### Deprecation messages turned on

A number of methods have been flagged for removal in version 2. Deprecation warnings will now be given, which include an alternative way to do the same thing. All of the deprecated methods have simpler equivalent alternatives.

```
>>> t = s.slice(0, 2)
__main__:1: DeprecationWarning: Call to deprecated function slice.
Instead of 's.slice(a, b, c)' use 's[a:b:c]'.
```

The deprecated methods are: `advancebit`, `advancebits`, `advancebyte`, `advancebytes`, `retreatbit`, `retreatbits`, `retreatbyte`, `retreatbytes`, `tell`, `seek`, `slice`, `delete`, `tellbyte`, `seekbyte`, `truncatestart` and `truncateend`.

### Initialise from bool

Booleans have been added to the list of types that can 'auto' initialise a bitstring.

```
>>> zerobit = BitString(False)
>>> onebit = BitString(True)
```

### Improved efficiency

More methods have been speeded up, in particular some deletions and insertions.

#### Bug fixes

A rare problem with truncating the start of bitstrings was fixed.

A possible problem outputting the final byte in `tofile()` was fixed.

### 20.1.23 December 22nd 2009: version 1.1.3 for Python 2.6 and 3.x released

This version hopefully fixes an installation problem for platforms with case-sensitive file systems. There are no new features or other bug fixes.

### 20.1.24 December 18th 2009: version 1.1.2 for Python 2.6 and 3.x released

This is a minor update with (almost) no new features.

#### Improved efficiency

The speed of many typical operations has been increased, some substantially.

Initialise from integer

A BitString of ‘0’ bits can be created using just an integer to give the length in bits. So instead of

```
>>> s = BitString(length=100)
```

you can write just

```
>>> s = BitString(100)
```

This matches the behaviour of bytearrays and (in Python 3) bytes.

- A defect related to using the set / unset functions on !BitStrings initialised from a file has been fixed.

### 20.1.25 November 24th 2009: version 1.1.0 for Python 2.6 and 3.x released

Note that this version will not work for Python 2.4 or 2.5. There may be an update for these Python versions some time next year, but it’s not a priority quite yet. Also note that only one version is now provided, which works for Python 2.6 and 3.x (done with the minimum of hackery!)

### 20.1.26 New features

#### Improved efficiency

A fair number of functions have improved efficiency, some quite dramatically.

#### New bit setting and checking functions

Although these functions don’t do anything that couldn’t be done before, they do make some common use cases much more efficient. If you need to set or check single bits then these are the functions you need.

- `set` / `unset` : Set bit(s) to 1 or 0 respectively.
- `allset` / `allunset` : Check if all bits are 1 or all 0.
- `anyset` / `anyunset` : Check if any bits are 1 or any 0.

```
>>> s = BitString(length=1000)
>>> s.set((10, 100, 44, 12, 1))
>>> s.allunset((2, 22, 222))
True
>>> s.anyset(range(7, 77))
True
```

### New rotate functions

`ror` / `rol` : Rotate bits to the right or left respectively.

```
>>> s = BitString('0b100000000')
>>> s.ror(2)
>>> s.bin
'0b001000000'
>>> s.rol(5)
>>> s.bin
'0b000000100'
```

### Floating point interpretations

New float initialisations and interpretations are available. These only work for BitStrings of length 32 or 64 bits.

```
>>> s = BitString(float=0.2, length=64)
>>> s.float
0.200000000000000001
>>> t = bitstring.pack('<3f', -0.4, 1e34, 17.0)
>>> t.hex
'0xcdccccbedf84f67700008841'
```

### 'bytes' token reintroduced

This token returns a bytes object (equivalent to a str in Python 2.7).

```
>>> s = BitString('0x010203')
>>> s.unpack('bytes:2, bytes:1')
['\x01\x02', '\x03']
```

### 'uint' is now the default token type

So for example these are equivalent:

```
a, b = s.readlist('uint:12, uint:12')
a, b = s.readlist('12, 12')
```

## 20.1.27 October 10th 2009: version 1.0.1 for Python 3.x released

This is a straight port of version 1.0.0 to Python 3.

For changes since the last Python 3 release read all the way down in this document to version 0.4.3.

This version will also work for Python 2.6, but there's no advantage to using it over the 1.0.0 release. It won't work for anything before 2.6.

## 20.1.28 October 9th 2009: version 1.0.0 for Python 2.x released

Version 1 is here!

This is the first release not to carry the 'beta' tag. It contains a couple of minor new features but is principally a release to fix the API. If you've been using an older version then you almost certainly will have to recode a bit. If you're not ready to do that then you may wish to delay updating.

So the bad news is that there are lots of small changes to the API. The good news is that all the changes are pretty trivial, the new API is cleaner and more 'Pythonic', and that by making it version 1.0 I'm promising not to tweak it again for some time.

## 20.1.29 API Changes

### New read / peek functions for returning multiple items

The functions read, readbits, readbytes, peek, peekbits and peekbytes now only ever return a single item, never a list.

The new functions readlist, readbitlist, readbytelist, peeklist, peekbitlist and peekbytelist can be used to read multiple items and will always return a list.

So a line like:

```
>>> a, b = s.read('uint:12, hex:32')
```

becomes

```
>>> a, b = s.readlist('uint:12, hex:32')
```

### Renaming / removing functions

Functions have been renamed as follows:

```
``seekbit`` -> ``seek``
```

```
``tellbit`` -> ``tell``
```

```
``reversebits`` -> ``reverse``
```

```
``deletebits`` -> ``delete``
```

```
``tostring`` -> ``tobytes``
```

and a couple have been removed altogether:

- `deletebytes` - use `delete` instead.
- `empty` - use `not s` rather than `s.empty()`.

### Renaming parameters

The parameters 'startbit' and 'endbit' have been renamed 'start' and 'end'. This affects the methods `slice`, `find`, `findall`, `rfind`, `reverse`, `cut` and `split`.

The parameter 'bitpos' has been renamed to 'pos'. The affects the methods `seek`, `tell`, `insert`, `overwrite` and `delete`.

### Mutating methods return None rather than self

This means that you can't chain functions together so

```
>>> s.append('0x00').prepend('0xff')
>>> t = s.reverse()
```

Needs to be rewritten

```
>>> s.append('0x00')
>>> s.prepend('0xff')
>>> s.reverse()
>>> t = s
```

Affects `truncatestart`, `truncateend`, `insert`, `overwrite`, `delete`, `append`, `prepend`, `reverse` and `reversebytes`.

### Properties renamed

The 'data' property has been renamed to 'bytes'. Also if the BitString is not a whole number of bytes then a ValueError exception will be raised when using 'bytes' as a 'getter'.

Properties 'len' and 'pos' have been added to replace 'length' and 'bitpos', although the longer names have not been removed so you can continue to use them if you prefer.

### Other changes

- The `unpack` method now always returns a list, never a single item.
- BitStrings are now 'unhashable', so calling hash on one or making a set will fail.
- The colon separating the token name from its length is now mandatory. So for example `BitString('uint12=100')` becomes `BitString('uint:12=100')`.
- Removed support for the 'bytes' token in format strings. Instead of `s.read('bytes:4')` use `s.read('bits:32')`.

## 20.1.30 New features

### Added endswith and startswith functions

These do much as you'd expect; they return True or False depending on whether the BitString starts or ends with the parameter.

```
>>> BitString('0xef342').startswith('0b11101')
True
```

## 20.1.31 September 11th 2009: version 0.5.2 for Python 2.x released

### Finally some tools for dealing with endianness!

New interpretations are now available for whole-byte BitStrings that treat them as big, little, or native-endian

```
>>> big = BitString(intbe=1, length=16) # or BitString('intbe:16=1') if you prefer.
>>> little = BitString(intle=1, length=16)
>>> print big.hex, little.hex
0x0001 0x0100
```

(continues on next page)

(continued from previous page)

```
>>> print big.intbe, little.intle
1 1
```

### 'Struct'-like compact format codes

To save some typing when using `pack`, `unpack`, `read` and `peek`, compact format codes based on those used in the struct and array modules have been added. These must start with a character indicating the endianness (>, < or `@` for big, little and native-endian), followed by characters giving the format:

```
b   1-byte signed int
B   1-byte unsigned int
h   2-byte signed int
H   2-byte unsigned int
l   4-byte signed int
L   4-byte unsigned int
q   8-byte signed int
Q   8-byte unsigned int
```

For example:

```
>>> s = bitstring.pack('<4h', 0, 1, 2, 3)
```

creates a BitString with four little-endian 2-byte integers. While

```
>>> x, y, z = s.read('>hhl')
```

reads them back as two big-endian two-byte integers and one four-byte big endian integer.

Of course you can combine this new format with the old ones however you like:

```
>>> s.unpack('<h, intle:24, uint:5, bin')
[0, 131073, 0, '0b0000000001100000000']
```

## 20.1.32 August 26th 2009: version 0.5.1 for Python 2.x released

This update introduces pack and unpack functions for creating and dissembling BitStrings.

### New pack() and unpack() functions

The `pack` function provides a flexible new method for creating BitStrings. Tokens for BitString 'literals' can be used in the same way as in the constructor.

```
>>> from bitstring import BitString, pack
>>> a = pack('0b11, 0xff, 0o77, int:5=-1, se=33')
```

You can also leave placeholders in the format, which will be filled in by the values provided.

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

Finally you can use a dictionary or keywords.

```
>>> c = pack('bin=a, hex=b, bin=a', a='010', b='ef')
```

The `unpack` method is similar to the `read` method except that it always unpacks from the start of the BitString.

```
>>> x, y = b.unpack('uint:10, hex')
```

If a token is given without a length (as above) then it will expand to fill the remaining bits in the BitString. This also now works with `read` and `peek`.

### New tostring() and tofile() methods

The `tostring` method just returns the data as a string, with up to seven zero bits appended to byte align. The `tofile` method does the same except writes to a file object.

```
>>> f = open('myfile', 'wb')
>>> BitString('0x1234ff').tofile(f)
```

### Other changes

The use of = is now mandatory in 'auto' initialisers. Tokens like `uint12 100` will no longer work. Also the use of a `:` before the length is encouraged, but not yet mandated. So the previous example should be written as `uint:12=100`.

The 'auto' initialiser will now take a file object.

```
>>> f = open('myfile', 'rb')
>>> s = BitString(f)
```

## 20.1.33 July 19th 2009: version 0.5.0 for Python 2.x released

This update breaks backward compatibility in a couple of areas. The only one you probably need to be concerned about is the change to the default for bytealigned in `find`, `replace`, `split`, etc.

See the user manual for more details on each of these items.

### Expanded abilities of 'auto' initialiser

More types can be initialised through the 'auto' initialiser. For example instead of

```
>>> a = BitString(uint=44, length=16)
```

you can write

```
>>> a = BitString('uint16=44')
```

Also, different comma-separated tokens will be joined together, e.g.

```
>>> b = BitString('0xff') + 'int8=-5'
```

can be written

```
>>> b = BitString('0xff, int8=-5')
```

### New formatted read and peek methods

These takes a format string similar to that used in the auto initialiser. If only one token is provided then a single value is returned, otherwise a list of values is returned.

```
>>> start_code, width, height = s.read('hex32, uint12, uint12')
```

is equivalent to

```
>>> start_code = s.readbits(32).hex
>>> width = s.readbits(12).uint
>>> height = s.readbits(12).uint
```

The tokens are:

```
int n   : n bits as an unsigned integer.
uint n  : n bits as a signed integer.
hex n   : n bits as a hexadecimal string.
oct n   : n bits as an octal string.
bin n   : n bits as a binary string.
ue      : next bits as an unsigned exp-Golomb.
se      : next bits as a signed exp-Golomb.
bits n  : n bits as a new BitString.
bytes n : n bytes as a new BitString.
```

See the user manual for more details.

### `hex` and `oct` methods removed

The special methods for `hex` and `oct` have been removed. Please use the `hex` and `oct` properties instead.

```
>>> hex(s)
```

becomes

```
>>> s.hex
```

### `join` made a method

The `join` function must now be called on a BitString object, which will be used to join the list together. You may need to recode slightly:

```
>>> s = bitstring.join('0x34', '0b1001', '0b1')
```

becomes

```
>>> s = BitString().join('0x34', '0b1001', '0b1')
```

### More than one value allowed in readbits, readbytes, peekbits and peekbytes

If you specify more than one bit or byte length then a list of BitStrings will be returned.

```
>>> a, b, c = s.readbits(10, 5, 5)
```

is equivalent to

```
>>> a = readbits(10)
>>> b = readbits(5)
>>> c = readbits(5)
```

### bytealigned defaults to False, and is at the end of the parameter list

Functions that have a bytealigned parameter have changed so that it now defaults to False rather than True. Also its position in the parameter list has changed to be at the end. You may need to recode slightly (sorry!)

### readue and readse methods have been removed

Instead you should use the new read function with a 'ue' or 'se' token:

```
>>> i = s.readue()
```

becomes

```
>>> i = s.read('ue')
```

This is more flexible as you can read multiple items in one go, plus you can now also use the peek method with ue and se.

### Minor bugs fixed

See the issue tracker for more details.

## 20.1.34 June 15th 2009: version 0.4.3 for Python 2.x released

This is a minor update. This release is the first to bundle the bitstring manual. This is a PDF and you can find it in the docs directory.

### New 'cut' method

This method returns a generator for constant sized chunks of a BitString.

```
>>> for byte in s.cut(8):
...     do_something_with(byte)
```

You can also specify a startbit and endbit, as well as a count, which limits the number of items generated:

```
>>> first100TSPackets = list(s.cut(188*8, count=100))
```

### 'slice' method now equivalent to `__getitem__`

This means that a step can also be given to the slice method so that the following are now the same thing, and it's just a personal preference which to use:

```
>>> s1 = s[a:b:c]
>>> s2 = s.slice(a, b, c)
```

### findall gets a 'count' parameter

So now

```
>>> list(a.findall(s, count=n))
```

is equivalent to

```
>>> list(a.findall(s))[:n]
```

except that it won't need to generate the whole list and so is much more efficient.

### Changes to 'split'

The split method now has a 'count' parameter rather than 'maxsplit'. This makes the interface closer to that for cut, replace and findall. The final item generated is now no longer the whole of the rest of the BitString.

- A couple of minor bugs were fixed. See the issue tracker for details.

## 20.1.35 May 25th 2009: version 0.4.2 for Python 2.x released

This is a minor update, and almost doesn't break compatibility with version 0.4.0, but with the slight exception of findall() returning a generator, detailed below.

### Stepping in slices

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

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

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

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

### Removed restrictions on offset

You can now specify an offset of greater than 7 bits when creating a BitString, and the use of offset is also now permitted when using the filename initialiser. This is useful when you want to create a BitString from the middle of a file without having to read the file into memory.

```
>>> f = BitString(filename='reallybigfile', offset=8000000, length=32)
```

### Integers can be assigned to slices

You can now assign an integer to a slice of a BitString. If the integer doesn't fit in the size of slice given then a ValueError exception is raised. So this is now allowed and works as expected:

```
>>> s[8:16] = 106
```

and is equivalent to

```
>>> s[8:16] = BitString(uint=106, length=8)
```

### Less exceptions raised

Some changes have been made to slicing so that less exceptions are raised, bringing the interface closer to that for lists. So for example trying to delete past the end of the BitString will now just delete to the end, rather than raising a ValueError.

### Initialisation from lists and tuples

A new option for the auto initialiser is to pass it a list or tuple. The items in the list or tuple are evaluated as booleans and the bits in the BitString are set to 1 for True items and 0 for False items. This can be used anywhere the auto initialiser can currently be used. For example:

```
>>> a = BitString([True, 7, False, 0, ()])      # 0b11000
>>> b = a + ['Yes', '']                          # Adds '0b10'
>>> (True, True, False) in a
True
```

### Miscellany

- `reversebits` now has optional startbit and endbit parameters.
- As an optimisation `findall` will return a generator, rather than a list. If you still want the whole list then of course you can just call `list()` on the generator.
- Improved efficiency of `rfind`.
- A couple of minor bugs were fixed. See the issue tracker for details.

## 20.1.36 April 23rd 2009: Python 3 only version 0.4.1 released

This version is just a port of version 0.4.0 to Python 3. All the unit tests pass, but beyond that only limited ad hoc testing has been done and so it should be considered an experimental release. That said, the unit test coverage is very good - I'm just not sure if anyone even wants a Python 3 version!

## 20.1.37 April 11th 2009: version 0.4.0 released

### New methods

Added `rfind`, `findall` and `replace`. These do pretty much what you'd expect - see the docstrings or the wiki for more information.

### More special methods

Some missing methods were added: `__repr__`, `__contains__`, `__rand__`, `__ror__`, `__rxor__` and `__delitem__`.

### Miscellany

A couple of small bugs were fixed (see the issue tracker).

There are some small backward incompatibilities relative to version 0.3.2:

### Combined `find` and `findbytealigned`

`findbytealigned` has been removed, and becomes part of `find`. The default start position has changed on both `find` and `split` to be the start of the BitString. You may need to recode:

```
>>> s1.find(bs)
>>> s2.findbytealigned(bs)
>>> s2.split(bs)
```

becomes

```
>>> s1.find(bs, bytealigned=False, startbit=s1.bitpos)
>>> s2.find(bs, startbit=s1.bitpos)  # bytealigned defaults to True
>>> s2.split(bs, startbit=s2.bitpos)
```

### Reading off end of BitString no longer raises exception

Previously a read or peek function that encountered the end of the BitString would raise a ValueError. It will now instead return the remainder of the BitString, which could be an empty BitString. This is closer to the file object interface.

### Removed visibility of offset

The offset property was previously read-only, and has now been removed from public view altogether. As it is used internally for efficiency reasons you shouldn't really have needed to use it. If you do then use the `_offset` parameter instead (with caution).

## 20.1.38 March 11th 2009: version 0.3.2 released

### Better performance

A number of methods (especially `find` and `findbytealigned`) have been sped up considerably.

### Bit-wise operations

Added support for bit-wise AND (`&`), OR (`|`) and XOR (`^`). For example:

```
>>> a = BitString('0b00111')
>>> print a & '0b10101'
0b00101
```

### Miscellany

Added `seekbit` and `seekbyte` methods. These complement the 'advance' and 'retreat' functions, although you can still just use bitpos and bytepos properties directly.

```
>>> a.seekbit(100)                   # Equivalent to a.bitpos = 100
```

Allowed comparisons between BitString objects and strings. For example this will now work:

```
>>> a = BitString('0b00001111')
>>> a == '0x0f'
True
```

## 20.1.39 February 26th 2009: version 0.3.1 released

This version only adds features and fixes bugs relative to 0.3.0, and doesn't break backwards compatibility.

### Octal interpretation and initialisation

The oct property now joins bin and hex. Just prefix octal numbers with '0o':

```
>>> a = BitString('0o755')
>>> print a.bin
0b111101101
```

### Simpler copying

Rather than using `b = copy.copy(a)` to create a copy of a BitString, now you can just use `b = BitString(a)`.

### More special methods

Lots of new special methods added, for example bit-shifting via << and >>, equality testing via == and `!=`, bit inversion (~) and concatenation using `*`.

Also `__setitem__` is now supported so BitString objects can be modified using standard index notation.

### Proper installer

Finally got round to writing the distutils script. To install just `python setup.py install`.

## 20.1.40 February 15th 2009: version 0.3.0 released

### Simpler initialisation from binary and hexadecimal

The first argument in the BitString constructor is now called 'auto' and will attempt to interpret the type of a string. Prefix binary numbers with '0b' and hexadecimals with '0x':

```
>>> a = BitString('0b0')          # single zero bit
>>> b = BitString('0xffff')       # two bytes
```

Previously the first argument was 'data', so if you relied on this then you will need to recode:

```
>>> a = BitString('\x00\x00\x01\xb3')    # Don't do this any more!
```

becomes

```
>>> a = BitString(data='\x00\x00\x01\xb3')
```

or just

```
>>> a = BitString('0x000001b3')
```

This new notation can also be used in functions that take a BitString as an argument. For example:

```
>>> a = BitString('0x0011') + '0xff'
>>> a.insert('0b001', 6)
>>> a.find('0b1111')
```

### BitString made more mutable

The methods `append`, `deletebits`, `insert`, `overwrite`, `truncatestart` and `truncateend` now modify the BitString that they act upon. This allows for cleaner and more efficient code, but you may need to rewrite slightly if you depended upon the old behaviour:

```
>>> a = BitString(hex='0xffff')
>>> a = a.append(BitString(hex='0x00'))
>>> b = a.deletebits(10, 10)
```

becomes

```
>>> a = BitString('0xffff')
>>> a.append('0x00')
>>> b = copy.copy(a)
>>> b.deletebits(10, 10)
```

Thanks to Frank Aune for suggestions in this and other areas.

#### Changes to printing

The binary interpretation of a BitString is now prepended with '0b'. This is in keeping with the Python 2.6 (and 3.0) bin function. The prefix is optional when initialising using *bin=*.

Also, if you just print a BitString with no interpretation it will pick something appropriate - hex if it is an integer number of bytes, otherwise binary. If the BitString representation is very long it will be truncated by '...' so it is only an approximate interpretation.

```
>>> a = BitString('0b0011111')
>>> print a
0b0011111
>>> a += '0b0'
>>> print a
0x3e
```

#### More convenience functions

Some missing methods such as `advancebit` and `deletebytes` have been added. Also a number of 'peek' methods make an appearance as have `prepend` and `reversebits`. See the Tutorial for more details.

### 20.1.41 January 13th 2009: version 0.2.0 released

Some fairly minor updates, not really deserving of a whole version point update.

### 20.1.42 December 29th 2008: version 0.1.0 released

First release!

# Python Module Index

b

`bitstring`, 43

# Index

## Symbols

`__add__()` (*bitstring.Bits method*), 52
`__and__()` (*bitstring.Bits method*), 52
`__bool__()` (*bitstring.Bits method*), 52
`__contains__()` (*bitstring.Bits method*), 53
`__copy__()` (*bitstring.Bits method*), 53
`__delitem__()` (*bitstring.BitArray method*), 61
`__eq__()` (*bitstring.Bits method*), 53
`__getitem__()` (*bitstring.Bits method*), 53
`__hash__()` (*bitstring.Bits method*), 54
`__iadd__()` (*bitstring.BitArray method*), 61
`__iand__()` (*bitstring.BitArray method*), 61
`__ilshift__()` (*bitstring.BitArray method*), 61
`__imul__()` (*bitstring.BitArray method*), 62
`__invert__()` (*bitstring.Bits method*), 54
`__ior__()` (*bitstring.BitArray method*), 62
`__irshift__()` (*bitstring.BitArray method*), 62
`__ixor__()` (*bitstring.BitArray method*), 62
`__len__()` (*bitstring.Bits method*), 54
`__lshift__()` (*bitstring.Bits method*), 54
`__mul__()` (*bitstring.Bits method*), 54
`__ne__()` (*bitstring.Bits method*), 54
`__nonzero__()` (*bitstring.Bits method*), 55
`__or__()` (*bitstring.Bits method*), 55
`__radd__()` (*bitstring.Bits method*), 52
`__rand__()` (*bitstring.Bits method*), 52
`__repr__()` (*bitstring.Bits method*), 55
`__rmul__()` (*bitstring.Bits method*), 54
`__ror__()` (*bitstring.Bits method*), 55
`__rshift__()` (*bitstring.Bits method*), 55
`__rxor__()` (*bitstring.Bits method*), 55
`__setitem__()` (*bitstring.BitArray method*), 62
`__str__()` (*bitstring.Bits method*), 55
`__xor__()` (*bitstring.Bits method*), 55

## A

`all()` (*bitstring.Bits method*), 47
`any()` (*bitstring.Bits method*), 48
`append()` (*bitstring.BitArray method*), 57

## B

`bin` (*bitstring.BitArray attribute*), 60
`bin` (*bitstring.Bits attribute*), 50
`BitArray` (*class in bitstring*), 57
`bitpos` (*bitstring.ConstBitStream attribute*), 65
`Bits` (*class in bitstring*), 47
`BitStream` (*class in bitstring*), 67
`bitstring` (*module*), 43
`bool` (*bitstring.BitArray attribute*), 60
`bool` (*bitstring.Bits attribute*), 50
`bytealign()` (*bitstring.ConstBitStream method*), 63
`ByteAlignError`, 71
`bytepos` (*bitstring.ConstBitStream attribute*), 65
`bytes` (*bitstring.BitArray attribute*), 60
`bytes` (*bitstring.Bits attribute*), 50
`byteswap()` (*bitstring.BitArray method*), 57

## C

`clear()` (*bitstring.BitArray method*), 58
`ConstBitStream` (*class in bitstring*), 63
`copy()` (*bitstring.BitArray method*), 58
`count()` (*bitstring.Bits method*), 48
`CreationError`, 71
`cut()` (*bitstring.Bits method*), 48

## E

`endswith()` (*bitstring.Bits method*), 48
`Error`, 71

## F

`find()` (*bitstring.Bits method*), 48
`findall()` (*bitstring.Bits method*), 49
`float` (*bitstring.BitArray attribute*), 60
`float` (*bitstring.Bits attribute*), 51
`floatbe` (*bitstring.BitArray attribute*), 60
`floatbe` (*bitstring.Bits attribute*), 51
`floatle` (*bitstring.BitArray attribute*), 60
`floatle` (*bitstring.Bits attribute*), 51
`floatne` (*bitstring.BitArray attribute*), 60
`floatne` (*bitstring.Bits attribute*), 51

## H

`hex` (*bitstring.BitArray attribute*), 60
`hex` (*bitstring.Bits attribute*), 50

## I

`insert()` (*bitstring.BitArray method*), 58

`int` (*bitstring.BitArray attribute*), 60
`int` (*bitstring.Bits attribute*), 51
`intbe` (*bitstring.BitArray attribute*), 60
`intbe` (*bitstring.Bits attribute*), 51
`InterpretError`, 71
`intle` (*bitstring.BitArray attribute*), 60
`intle` (*bitstring.Bits attribute*), 51
`intne` (*bitstring.BitArray attribute*), 60
`intne` (*bitstring.Bits attribute*), 51
`invert()` (*bitstring.BitArray method*), 58

J

`join()` (*bitstring.Bits method*), 49

L

`len` (*bitstring.Bits attribute*), 51
`length` (*bitstring.Bits attribute*), 51

O

`oct` (*bitstring.BitArray attribute*), 60
`oct` (*bitstring.Bits attribute*), 51
`overwrite()` (*bitstring.BitArray method*), 58

P

`pack()` (*in module bitstring*), 69
`peek()` (*bitstring.ConstBitStream method*), 63
`peeklist()` (*bitstring.ConstBitStream method*), 63
`pos` (*bitstring.ConstBitStream attribute*), 65
`prepend()` (*bitstring.BitArray method*), 59

R

`read()` (*bitstring.ConstBitStream method*), 64
`ReadError`, 71
`readlist()` (*bitstring.ConstBitStream method*), 64
`readto()` (*bitstring.ConstBitStream method*), 65
`replace()` (*bitstring.BitArray method*), 59
`reverse()` (*bitstring.BitArray method*), 59
`rfind()` (*bitstring.Bits method*), 49
`rol()` (*bitstring.BitArray method*), 59
`ror()` (*bitstring.BitArray method*), 59

S

`se` (*bitstring.BitArray attribute*), 61
`se` (*bitstring.Bits attribute*), 51
`set()` (*bitstring.BitArray method*), 59
`sie` (*bitstring.BitArray attribute*), 61
`sie` (*bitstring.Bits attribute*), 52
`split()` (*bitstring.Bits method*), 49
`startswith()` (*bitstring.Bits method*), 49

T

`tobytes()` (*bitstring.Bits method*), 50
`tofile()` (*bitstring.Bits method*), 50

U

`ue` (*bitstring.BitArray attribute*), 61
`ue` (*bitstring.Bits attribute*), 52
`uie` (*bitstring.BitArray attribute*), 61
`uie` (*bitstring.Bits attribute*), 52
`uint` (*bitstring.BitArray attribute*), 61
`uint` (*bitstring.Bits attribute*), 52
`uintbe` (*bitstring.BitArray attribute*), 61
`uintbe` (*bitstring.Bits attribute*), 52
`uintle` (*bitstring.BitArray attribute*), 61
`uintle` (*bitstring.Bits attribute*), 52
`uintne` (*bitstring.BitArray attribute*), 61
`uintne` (*bitstring.Bits attribute*), 52
`unpack()` (*bitstring.Bits method*), 50