bitstring

bitstring is a Python module to help make the creation and analysis of all types of bit-level binary data as simple and efficient as possible.

It has been maintained since 2006.

Overview

- Efficiently store and manipulate binary data in idiomatic Python.
- Create bitstrings from hex, octal, binary, files, formatted strings, bytes, integers and floats of different endiannesses.
- Powerful binary packing and unpacking functions.
- Bit level slicing, joining, searching, replacing and more.
- Read from and interpret bitstrings as streams of binary data.
- Create efficiently stored arrays of any fixed-length format.
- Rich API - chances are that whatever you want to do there's a simple and elegant way of doing it.
- Open source software, released under the MIT licence.

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

The bitstring module provides support many different bit formats, allowing easy and efficient storage, interpretation and construction.

Documentation Quick Start

You can take a look at the introductory walkthrough notebook here.

- `Bits` - an immutable container of bits.
- `BitArray` - adds mutating methods to `Bits`.
- `BitStream` - adds a bit position and read methods.
- `Array` - an array of bitstrings of the same type.

Mixed format bitstrings

If you have binary data (or want to construct it) from multiple types then you could use the `BitArray` class. The example below constructs a 28 bit bitstring from a hexadecimal string, then unpacks it into multiple bit interpretations. It also demonstrates how it can be flexibly modified and sliced using standard notation, and how properties such as `bin` and `float` can be used to interpret the data.

```
>>> s = bitstring.BitArray('0x4f8e220')
>>> s.unpack('uint12, hex8, bin')
[1272, 'e2', '00100000']
>>> '0b11000' in s
True
>>> s += 'f32=0.001'
>>> s.bin
'01001111100011100010001000000011101010000011000100100110111 1'
>>> s[-32:].float
0.0010000000474974513
```

The module also supplies the `BitStream` class, which adds a bit position so that 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.

There are also a companion classes called `Bits` and `ConstBitStream` which are immutable versions of `BitArray` and `BitStream` respectively. See the reference documentation for full details.

Arrays of bitstrings

fixed-length binary format.

< Page contents

Want an array of 5 bit unsigned integers, or of 8 or 16 bit floating point numbers? No problem. You can also easily change the data's interpretation, convert to another format, and freely modify the underlying data which is stored as a `BitArray` object.

```
>>> a = bitstring.Array('uint16', [0, 1, 4, 6, 11, 2, 8, 7])
>>> a.data
BitArray('0x0000000100040006000b000200080007')
>>> b = a.astype('uint5')
>>> b.data
BitArray('0x0048658907')
>>> a.tolist() == b.tolist()
True
```

You can also take and set slices as you'd expect, and apply operations to each element in the `Array`.

```
>>> a[::2] *= 5
>>> a
Array('uint16', [0, 1, 20, 6, 55, 2, 40, 7])
>>> a >> 2
Array('uint16', [0, 0, 5, 1, 13, 0, 10, 1])
```

Installation and download

To install just `pip install bitstring`.

To download the module, as well as for defect reports, enhancement requests and Git repository browsing go to the project's home on GitHub.

Release Notes

To see what been added, improved or fixed, and possibly also to see what's coming in the next version, see the release notes on GitHub.

Credits

Created by Scott Griffiths in 2006 to help with ad hoc parsing and creation of compressed video files. Maintained and expanded ever since as it became unexpectedly popular. Thanks to all those who have

Search

Searching for multiple words only shows matches that contain all words.

search

Quick Reference

This section gives a summary of the bitstring module's classes, functions and attributes.

There are four main classes that are bit containers, so that each element is a single bit. They differ based on whether they can be modified after creation and on whether they have the concept of a current bit position.

Class	Mutable?	Streaming methods?	
Bits	✘	✘	An efficient, immutable container of bits.
BitArray	✔	✘	Like `Bits` but it can be changed after creation.
ConstBitStream	✘	✔	Immutable like `Bits` but with a bit position and reading methods.
BitStream	✔	✔	Mutable like `BitArray` but with a bit position and reading methods.

The final class is a flexible container whose elements are fixed-length bitstrings.

Array	✔	✘	An efficient list-like container where each item has a fixed-length binary format.

Bits

`Bits` is the most basic class and is just a container of bits. It is immutable, so once created its value cannot change.

```
Bits(auto, /, length: Optional[int], offset: Optional[int], **kwargs)
```

The first parameter (usually referred to as *auto*) can be many different types, including parsable strings, a file handle, a bytes or bytearray object, an integer or an iterable.

A single initialiser from *kwargs* can be used instead of *auto*, including `bin`, `hex`, `oct`, `bool`, `uint`, `int`, `float`, `bytes` and `filename`.

Examples:

< Page contents

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.
- `copy` – Return a copy of the bitstring.
- `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.
- `fromstring` – Create a bitstring from a formatted string.
- `join` – Join bitstrings together using current bitstring.
- `pp` – Pretty print the 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.
- `tobitarray` – Return bitstring as a `bitarray` object from the bitarray package.
- `tobytes` – Return bitstring as bytes, padding if needed.
- `tofile` – Write bitstring to file, padding if needed.
- `unpack` – Interpret bits using format string.

Special methods

Also available are operators that will return a new bitstring (or check for equality):

- `==` / `!=` – Equality tests.
- `[]` – Get an element or slice.
- `+` – Concatenate with another bitstring.
- `*` – Concatenate multiple copies of the current bitstring.
- `~` – Invert every bit of the bitstring.
- `<<` – Shift bits to the left.
- `>>` – Shift bits to the right.
- `&` – Bit-wise AND between two bitstrings.
- `|` – Bit-wise OR between two bitstrings.
- `^` – Bit-wise XOR between two bitstrings.

required. Many require the bitstring to be specific lengths.

< Page contents

- `bin` / `b` – The bitstring as a binary string.
- `bool` – For single bit bitstrings, interpret as True or False.
- `bytes` – The bitstring as a bytes object.
- `float` / `floatbe` / `f` – 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` / `h` – The bitstring as a hexadecimal string.
- `int` / `i` – 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` / `o` – The bitstring as an octal string.
- `uint` / `u` – 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.

There are also various other flavours of 16-bit, 8-bit and smaller floating point types (see Exotic Floating Point Formats) and exponential-Golomb integer types (see Exponential-Golomb Codes) that are not listed here for brevity.

BitArray

`Bits` ⟶ `BitArray`

`BitArray` adds mutating methods to `Bits`. The constructor is the same as for `Bits`.

Additional methods

All of the methods listed above for the `Bits` class are available, plus:

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

Additional special methods

The special methods available for the `Bits` class are all available, plus some which will modify the bitstring:

- `[]` – Set an element or slice.
- `del` – Delete an element or slice.
- `+=` – Append bitstring to the current bitstring.
- `*=` – Concatenate multiple copies of the current bitstring.
- `<<=` – Shift bits in-place to the left.
- `>>=` – Shift bits in-place to the right.
- `&=` – In-place bit-wise AND between two bitstrings.
- `|=` – In-place bit-wise OR between two bitstrings.
- `^=` – In-place bit-wise XOR between two bitstrings.

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

ConstBitStream

`Bits` ⟶ `ConstBitStream`

`ConstBitStream` adds a bit position and methods to read and navigate in an immutable bitstream. If you wish to use streaming methods on a large file without changing it then this is often the best class to use.

The constructor is the same as for `Bits / BitArray` but with an optional current bit position.

```
ConstBitStream(auto, length: Optional[int], offset: Optional[int], pos: int = 0, **kwargs)
```

All of the methods, special methods and properties listed above for the `Bits` class are availabl

Additional methods

- `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.

Additional properties

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

BitStream

`Bits ⟶ BitArray / ConstBitStream ⟶ BitStream`

`BitStream` contains all of the 'stream' elements of `ConstBitStream` and adds all of the mutating methods of `BitArray`. The constructor is the same as for `ConstBitStream`. It has all the methods, special methods and properties of the `Bits`, `BitArray` and `ConstBitArray` classes.

It is the most general of the four classes, but it is usually best to choose the simplest class for your use case.

Array

A bitstring `Array` is a contiguously allocated sequence of bitstrings of the same type. It is similar to the `array` type in the array module, except that it is far more flexible.

```
Array(dtype: str | Dtype, initializer, trailing_bits)
```

The *dtype* can any single fixed-length token as described in Format tokens and Compact format strings.

The *inititalizer* will typically be an iterable such as a list, but can also be many other things including an open binary file, a bytes or bytearray object, another `bitstring.Array` or an `array.array`. It can also be an integer, in which case the `Array` will be zero-initialised with that many items.

The *trailing_bits* typically isn't used in construction, and specifies bits left over after interpreting the stored binary data according to the data type *dtype*.

Both the dtype and the underlying bit data (stored as a `BitArray`) can be freely modified after c… element-wise operations can be used on the `Array`. Modifying the data or format after creation may cause

< Page contents

```
Array('>H', [1, 10, 20])
Array('float16', a_file_object)
Array('int4', stored_bytes)
```

Methods

- `append` – Append a single item to the end of the Array.
- `astype` – Cast the Array to a new dtype.
- `byteswap` – Change byte endianness of all items.
- `count` – Count the number of occurrences of a value.
- `equals` – Compare with another Array for exact equality.
- `extend` – Append multiple items to the end of the Array from an iterable.
- `fromfile` – Append items read from a file object.
- `insert` – Insert an item at a given position.
- `pop` – Return and remove an item.
- `pp` – Pretty print the Array.
- `reverse` – Reverse the order of all items.
- `tobytes` – Return Array data as bytes object, padding with zero bits at the end if needed.
- `tofile` – Write Array data to a file, padding with zero bits at the end if needed.
- `tolist` – Return Array items as a list.

Special methods

These non-mutating special methods are available. Where appropriate they return a new `Array`.

- `[]` – Get an element or slice.
- `+` – Add value to each element.
- `-` – Subtract value from each element.
- `*` – Multiply each element by a value.
- `/` – Divide each element by a value.
- `//` – Floor divide each element by a value.
- `%` – Take modulus of each element with a value.
- `<<` – Shift value of each element to the left.
- `>>` – Shift value of each element to the right.
- `&` – Bit-wise AND of each element.
- `|` – Bit-wise OR of each element.
- `^` – Bit-wise XOR of each element.

```
>>> b = Array('i6', [30, -10, 1, 0])
>>> b >> 2
Array('i6', [7, -3, 0, 0])
>>> b + 1
Array('i6', [31, -9, 2, 1])
>>> b + b
Array('i6', [60, -20, 2, 0])
```

Comparison operators will output an `Array` with a `dtype` of `'bool'`.

- `==` / `!=` – Equality tests.
- `<` – Less than comparison.
- `<=` – Less than or equal comparison.
- `>` – Greater than comparison.
- `>=` – Greater than or equal comparison.

Mutating versions of many of the methods are also available.

- `[]` – Set an element or slice.
- `del` – Delete an element or slice.
- `+=` – Add value to each element in-place.
- `-=` – Subtract value from each element in-place.
- `*=` – Multiply each element by a value in-place.
- `/=` – Divide each element by a value in-place.
- `//=` – Floor divide each element by a value in-place.
- `%=` – Take modulus of each element with a value in-place.
- `<<=` – Shift bits of each element to the left in-place.
- `>>=` – Shift bits of each element to the right in-place.
- `&=` – In-place bit-wise AND of each element.
- `|=` – In-place bit-wise OR of each element.
- `^=` – In-place bit-wise XOR of each element.

Example:

```
>>> a = Array('float16', [1.5, 2.5, 7, 1000])
>>> a[::2] *= 3.0  # Multiply every other float16 value in-place
>>> a
Array('float16', [4.5, 2.5, 21.0, 1000.0])
```

Properties

- `data` – The complete binary data in a `BitArray` object. Can be freely modified.
- `dtype` – The data type or typecode. Can be freely modified.
- `itemsize` – The length *in bits* of a single item. Read only.
- `trailing_bits` – If the data length is not a multiple of the *dtype* length, this `BitArray` gives the leftovers at the end of the data.

Dtype

A data type (or 'dtype') concept is used in the bitstring module to encapsulate how to create, parse and present different bit interpretations.

```
Dtype(token: str, /, length: int | None, scale: int | float | None = None)
```

Creates a `Dtype` object. Dtypes are immutable and cannot be changed after creation.

The first parameter is a format token string that can optionally include a length.

If appropriate, the *length* parameter can be used to specify the length of the bitstring.

The *scale* parameter can be used to specify a multiplicative scaling factor for the interpretation of the data.

Methods

- `build` – Create a bitstring from a value.
- `parse` – Parse a bitstring to find its value.

Properties

All properties are read-only.

- `bitlength` – The number of bits needed to represent a single instance of the data type.
- `bits_per_item` – The number of bits for each unit of length. Usually 1, but equals 8 for *bytes* type.
- `get_fn` – A function to get the value of the data type.
- `is_signed` – If True then the data type represents a signed quantity.
- `length` – The length of the data type in units of *bits_per_item*.
- `name` – A string giving the name of the data type.

- `set_fn` – A function to set the value of the data type.
- `variable_length` – If True then the length of the data type varies, and shouldn't be spe

< Page contents

General Information

Format tokens

Format strings are used when constructing bitstrings, as well as reading, packing and unpacking them, as well as giving the format for `Array` objects. They can also be auto promoted to bitstring when appropriate - see The auto initialiser.

`'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.
`'bool[:1]'`	next bit as a boolean (True or False).
`'pad:n'`	next `n` bits will be ignored (padding). Only applicable when reading, not cr

< Page contents

See also Exotic Floating Point Formats and Exponential-Golomb Codes for other types that c
format token strings.

Bitstring literals

To make a literal quantity (one that directly represents a sequence of bits) you can use any of the format tokens above followed by an `'='` and a value to initialise with. For example:

```
s = BitArray('float32=10.125, int7=-9')
s.append('hex:abc')
```

You can also create binary, octal and hexadecimal literals by starting a string with `'0b'`, `'0o'` and `'0x'` respectively:

```
t = BitArray('0b101')
t += '0x001f'
```

Compact format strings

Another option 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:

`'>'`	Big-endian
`'<'`	Little-endian
`'='`	Native-endian

Note

- For native-endian `'@'` and `'='` can both be used and are equivalent. The `'@'` character was required for native-endianness prior to version 4.1 of bitstring.
- 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

`'I'`	32 bit signed integer
`'L'`	32 bit unsigned integer
`'q'`	64 bit signed integer
`'Q'`	64 bit unsigned integer
`'e'`	16 bit floating point number
`'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	`intbe16`
`'<h'`	Little-endian 16 bit signed integer	`intle16`
`'=h'`	Native-endian 16 bit signed integer	`intne16`

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.

Module level

Functions

- `pack` – Create a new `BitStream` according to a format string and values.

Exceptions

- `Error` – Base class for module exceptions.
- `ReadError` – Reading or peeking past the end of a bitstring.
- `InterpretError` – Inappropriate interpretation of binary data.
- `ByteAlignError` – Whole-byte position or length needed.
- `CreationError` – Inappropriate argument during bitstring creation.

of the module.

< Page contents

- `bytealigned` – Determines whether a number of methods default to working only on b
- `lsb0` – If True, index bits with the least significant bit (the final bit) as bit zero.
- `mxfp_overflow` – Determines how values are converted to 8-bit MX floats. Can be either `'saturate'` (the default) or `'overflow'`. See Exotic Floating Point Formats.
- `no_color` – If True, don't use ANSI color codes in the pretty print methods. Defaults to False unless the NO_COLOR environment variable is set.

< Overview

Reference >

Reference

The Quick Reference gives one line summaries of all of the methods and properties of the bitstring classes.

This section gives more information and usually examples.

< Quick Reference

Introduction >

Appendices

This section contains supplementary information about the library, including details of some of the more specialist data types that are supported.


bitstring

bitstring

bitstring is a Python module to help make the creation and analysis of all types of bit-level binary data as simple and efficient as possible.

It has been maintained since 2006.

Overview

- Efficiently store and manipulate binary data in idiomatic Python.
- Create bitstrings from hex, octal, binary, files, formatted strings, bytes, integers and floats of different endiannesses.
- Powerful binary packing and unpacking functions.
- Bit level slicing, joining, searching, replacing and more.
- Read from and interpret bitstrings as streams of binary data.
- Create efficiently stored arrays of any fixed-length format.
- Rich API - chances are that whatever you want to do there's a simple and elegant way of doing it.
- Open source software, released under the MIT licence.

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

The bitstring module provides support many different bit formats, allowing easy and efficient storage, interpretation and construction.

Documentation Quick Start

You can take a look at the introductory walkthrough notebook here.

- `Bits` - an immutable container of bits.
- `BitArray` - adds mutating methods to `Bits`.
- `BitStream` - adds a bit position and read methods.
- `Array` - an array of bitstrings of the same type.

Mixed format bitstrings

If you have binary data (or want to construct it) from multiple types then you could use the `BitArray` class. The example below constructs a 28 bit bitstring from a hexadecimal string, then unpacks it into multiple bit interpretations. It also demonstrates how it can be flexibly modified and sliced using standard notation, and how properties such as `bin` and `float` can be used to interpret the data.

```
>>> s = bitstring.BitArray('0x4f8e220')
>>> s.unpack('uint12, hex8, bin')
[1272, 'e2', '00100000']
>>> '0b11000' in s
True
>>> s += 'f32=0.001'
>>> s.bin
'01001111100011100010001000000011101010000011000100100110111 1'
>>> s[-32:].float
0.0010000000474974513
```

The module also supplies the `BitStream` class, which adds a bit position so that 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.

There are also a companion classes called `Bits` and `ConstBitStream` which are immutable versions of `BitArray` and `BitStream` respectively. See the reference documentation for full details.

Arrays of bitstrings

fixed-length binary format.

< Page contents

Want an array of 5 bit unsigned integers, or of 8 or 16 bit floating point numbers? No problem. You can also easily change the data's interpretation, convert to another format, and freely modify the underlying data which is stored as a `BitArray` object.

```
>>> a = bitstring.Array('uint16', [0, 1, 4, 6, 11, 2, 8, 7])
>>> a.data
BitArray('0x0000000100040006000b000200080007')
>>> b = a.astype('uint5')
>>> b.data
BitArray('0x0048658907')
>>> a.tolist() == b.tolist()
True
```

You can also take and set slices as you'd expect, and apply operations to each element in the `Array`.

```
>>> a[::2] *= 5
>>> a
Array('uint16', [0, 1, 20, 6, 55, 2, 40, 7])
>>> a >> 2
Array('uint16', [0, 0, 5, 1, 13, 0, 10, 1])
```

Installation and download

To install just `pip install bitstring`.

To download the module, as well as for defect reports, enhancement requests and Git repository browsing go to the project's home on GitHub.

Release Notes

To see what been added, improved or fixed, and possibly also to see what's coming in the next version, see the release notes on GitHub.

Credits

Created by Scott Griffiths in 2006 to help with ad hoc parsing and creation of compressed video files. Maintained and expanded ever since as it became unexpectedly popular. Thanks to all those who have

Introduction

The bitstring classes

Five classes are provided by the bitstring module, four are simple containers of bits:

- `Bits` : This is the most basic class. It is immutable and so its contents can't be changed after creation.
- `BitArray` : This adds mutating methods to its base class.
- `ConstBitStream` : 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` : This is the most versatile class, having both the bitstream methods and the mutating methods.

`Bits` and `BitArray` are intended to loosely mirror the `bytes` and `bytearray` types in Python. The term 'bitstring' is used in this documentation to refer generically to any of these four classes.

The fifth class is `Array` which is a container of fixed-length bitstrings. The rest of this introduction mostly concerns the more basic types - for more details on `Array` you can go directly to the reference documentation, but understanding how bit format strings are specified will be helpful.

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 whole 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.

in the constructor ('auto' initialisation, described below), or using a keyword argument for a d

< Page contents

```
Bits(auto, /, length: Optional[int], offset: Optional[int], **kwargs)
```

Some of the keyword arguments that can be used are:

- `bytes` : A `bytes` object, 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.
- `bool` : A boolean (i.e. True or False).
- `filename` : Directly from a file, without reading into memory if using `Bits` or `ConstBitStream`.

There are also various other flavours of 16-bit, 8-bit and smaller floating point types (see Exotic Floating Point Formats) and exponential-Golomb integer types (see Exponential-Golomb Codes).

The `hex`, `oct`, `bin`, `float`, `int` and `uint` can all be shortened to just their initial letters. The data type name can be combined with its length if appropriate, or the length can be specified separately.

For example:

```
a = Bits(hex='deadbeef')
b = BitArray(f32=100.25)  # or = BitArray(float=100.25, length=32)
c = ConstBitStream(filename='a_big_file')
d = Bits(u12=105)
e = BitArray(bool=True)
```

Note that some types need a length to be specified, some don't need one, and others can infer the length from the value.

Another way to create a bitstring is via the `pack` function, which packs multiple values according to a given format. See the entry on `pack` for more information.

The auto initialiser

The first parameter when creating a bitstring is a positional only parameter, referred to as 'auto', a variety of types:

< Page contents

- A file object, opened in binary mode, from which the bitstring will be formed.
- A `bytearray` or `bytes` object.
- An `array` object from the built-in `array` module. This is used after being converted to it's constituent byte data via its `tobytes` method.
- A `bitarray` or `frozenbitarray` object from the 3rd party `bitarray` package.

If it is a string then that string will be parsed into tokens to construct the binary data:

- Starting with `'0x'` or `'hex='` implies hexadecimal. e.g. `'0x013ff'`, `'hex=013ff'`
- Starting with `'0o'` or `'oct='` implies octal. e.g. `'0o755'`, `'oct=755'`
- Starting with `'0b'` or `'bin='` 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. `'uint8=255'`, `'int4=-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. `'uintle32=1'`, `'intne16=-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. `'float64=0.2'`, `'floatle32=-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 `'uint4=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.

Promotion to bitstrings

Almost anywhere that a bitstring is expected you can substitute something that will get 'auto' promoted to a bitstring. For example:

```
>>> BitArray('0xf') == '0b1111'
True
```

Here the equals operator is expecting another bitstring so creates one from the string. The right hand side gets promoted to `Bits('0b1111')`.

Methods that need another bitstring as a parameter will also 'auto' promote, for example:


```
for bs in s.split('0x40'):
    if bs.endswith('0b111'):
```

which illustrates a variety of methods promoting strings, iterables and a bytes object to bitstrings.

Anything that can be used as the first parameter of the `Bits` constructor can be auto promoted to a bitstring where one is expected, with the exception of integers. Integers won't be auto promoted, but instead will raise a `TypeError`:

```
>>> a = BitArray(100)  # Create bitstring with 100 zeroed bits.
>>> a += 0xff          # TypeError - 0xff is the same as the integer 255.
>>> a += '0xff'        # Probably what was meant - append eight '1' bits.
>>> a += Bits(255)     # If you really want to do it then code it explicitly.
```

BitsType

```
class BitsType(
    Bits | str | Iterable[Any] | bool | BinaryIO | bytearray | bytes | memoryview |
bitarray.bitarray
)
```

The `BitsType` type is used in the documentation in a number of places where an object of any type that can be promoted to a bitstring is acceptable.

It's just a union of types rather than an actual class (though it's documented here as a class as I could find no alternative). It's not user accessible, but is just a shorthand way of saying any of the above types.

Keyword initialisers

If the 'auto' initialiser isn't used then at most one keyword initialiser can be used.

From a hexadecimal string

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

`__init__` this will work equally well:

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

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')
```

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')
```

From an integer

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

The 'auto' initialiser can be used by giving the length in bits immediately after the `int` or `ui` followed by an equals sign then the value:

```
>>> e = BitArray('uint12=45')
>>> f = BitArray('int7=-1')
```

The `uint` and `int` names can be shortened to just `u` and `i` respectively. For mutable bitstrings you can change value by assigning to a property with a length:

```
>>> e = BitArray()
>>> e.u12 = 45
>>> f = BitArray()
>>> f.i7 = -1
```

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

Big and little-endian integers

```
>>> big_endian = 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'
```

< Page contents

From a floating point number

```
>>> f1 = BitArray(float=10.3, length=32)
>>> f2 = BitArray('float64=5.4e31')
```

Floating point numbers can be used for initialisation provided that the bitstring is 16, 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. The 16 bit version has very limited range and is used mainly in specialised areas such as machine learning.

The exact bits used to represent the floating point number will conform to the IEEE 754 standard, even if the machine being used does not use that standard internally.

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')
```

See also Exotic Floating Point Formats for information on other floating point representations that are supported (bfloat and different 8-bit and smaller float formats).

From 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
'000000001100100101'
```

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 th
Exponential-Golomb Codes).

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

< Page contents

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

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` or a `bytes` object, either explicitly with a `bytes=some_bytearray` keyword or via the 'auto' initialiser:

```
c = BitArray(a_bytearray_object)
d = BitArray(b'\x23g$5')
```

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="my200GBfile")
```

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.

```
p = Bits(f)
```

< Page contents

Note

For the immutable types `Bits` and `ConstBitstream` the file is memory mapped (mmap) in a read-only mode for efficiency.

This behaves slightly differently depending on the platform; in particular Windows will lock the file against any further writing whereas Unix-like systems will not. This means that you won't be able to write to the file from Windows OS while the `Bits` or `ConstBitStream` object exists.

The work-arounds for this are to either (i) Delete the object before opening the file for writing, (ii) Use either `BitArray` or `BitStream` which will read the whole file into memory or (iii) Stop using Windows (or run in WSL).

< Reference

Interpreting Bitstrings >

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 GiB file object and ask for its binary string representation then that string will be around 80 GiB in size!

If you're in an interactive session then the pretty-print method `pp` can be useful as it will only convert the bitstring one chunk at a time for display.

Properties

Many of the more commonly used interpretations have single letter equivalents. The `hex`, `bin`, `oct`, `int`, `uint` and `float` properties can be shortened to `h`, `b`, `o`, `i`, `u` and `f` respectively. Properties can have bit lengths appended to them to make properties such as `f64`, `u32` or `floatle32`.

When used as a getter these just add an extra check on the bitstring's length - if the bitstring is not the stated length then an `InterpretError` is raised. When used as a setter they define the new length of the bitstring.

```
s = BitArray()  # Empty bitstring
s.f32 = 101.5   # New length is 32 bits, representing a float
```

For the properties described below we will use these:

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


bin / hex / oct

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

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

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.h
ValueError: Cannot convert to hex unambiguously - not multiple of 4 bits.
```

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

Integer types

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.u
7
```

```
>>> 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
```

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.i
-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.

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.

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 example we explicitly padded at the start to byte align before using the `bytes` property.

representation and will only work if the bitstring is 16, 32 or 64 bits long.

< Page contents

Different endiannesses are provided via `floatle` and `floatne`. Note that as floating point i
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.

Other floating point types

A range of floating point types that are mostly used in machine learning are also availabe. They include `bfloat16` which is a truncated `float32`, together with IEEE 8-bit formats and a range of OCP Microscaling 8-bit, 6-bit and 4-bit formats.

See Exotic Floating Point Formats for more information.

Exponential-Golomb types

Some variable length integer types are supported. The lengths of these types depends upon the data being read and they are mainly used in video codecs.

See Exponential-Golomb Codes for more information.

< Introduction

Dtypes >

Dtypes

A data type (or 'dtype') concept is used in the bitstring module to encapsulate how to create, parse and present different bit interpretations. The properties described above are all examples of dtypes.

```
class Dtype(
    token: str | Dtype,
    /,
    length: int | None = None,
    scale: int | float | None = None
)
```

Dtypes are immutable and cannot be changed after creation.

The first parameter is a format token string that can optionally include a length. For example `'ue'`, `'int'` or `'float16'`.

If the first parameter doesn't include a length and one is appropriate, the *length* parameter can be used to specify the length of the dtype.

The *scale* parameter can be used to specify a multiplicative scaling factor for the interpretation of the data. This is primarily intended for use with floating point formats of 8 bits or less, but can be used on other types.

In most situations the token string can be used instead of *Dtype* object when it is needed, and the *Dtype* will be constructed automatically, which is why the *Dtype* object is rarely used directly in this documentation. It can however be advantageous to to create *Dtype* objects directly for efficiency reasons, or for using dtypes programmatically.

If you need to use the *scale* parameter then there is no way to specify this in the format token string, so you must directly use a *Dtype* object.

Methods

```
Dtype.build(
    value: Any,
    /
) → Bits
```

```
>>> d.build(85)  # Equivalent to: Bits(u10=85)
Bits('0b0001010101')
```

< Page contents

```
Dtype.parse(
    b: BitsType,
    /
) → Any
```

Parse a bitstring to find its value. The *b* parameter should be a bitstring of the appropriate length, or an object that can be converted to a bitstring.

```
>>> d = Dtype('u10')
>>> d.parse('0b0001010101')  # Equivalent to: Bits('0b0001010101').u10
85
```

Properties

All properties are read-only.

```
Dtype.bitlength: int | None
```

The number of bits needed to represent a single instance of the data type. Will be set to `None` for variable length dtypes.

```
Dtype.bits_per_item: int
```

The number of bits for each unit of length. Usually 1, but equals 8 for *bytes* type.

```
Dtype.get_fn: Callable
```

A function to get the value of the data type.

If True then the data type represents a signed quantity.

< Page contents

```
Dtype.length: int | None
```

The length of the data type in units of *bits_per_item*. Will be set to `None` for variable length dtypes.

```
Dtype.name: str
```

A string giving the name of the data type.

```
Dtype.read_fn: Callable
```

A function to read the value of the data type.

```
Dtype.return_type: type
```

The type of the value returned by the *parse* method, such as `int`, `float` or `str`.

```
Dtype.scale: int | float | None
```

The multiplicative scale applied when interpreting the data.

```
Dtype.set_fn: Callable
```

A function to set the value of the data type.

```
Dtype.variable_length: bool
```

If True then the length of the data type depends on the data being interpreted, and must not be specified.

< Interpreting Bitstrings

Bits

The `Bits` class is the simplest type in the bitstring module, and represents an immutable sequence of bits. This is the best class to use if you will not need to modify the data after creation and don't need streaming methods.

```
class Bits(
    auto: BitsType | int | None,
    /,
    length: int | None = None,
    offset: int | None = None,
    **kwargs
)
```

Creates a new bitstring. You must specify either no initialiser, just an 'auto' value as the first parameter, or a keyword argument such as `bytes`, `bin`, `hex`, `oct`, `uint`, `int`, `float`, `bool` or `filename` (for example) to indicate the data type. 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 16, 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
```

See also The auto initialiser, which allows many different types to be used to initialise a bits

In the methods below we use `BitsType` to indicate that any of the types that can auto in used.

< Page contents

Methods

```
Bits.all(
    value: bool,
    pos: Iterable[int] | None = None
) → bool
```

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 < -len(s)` or `pos > len(s)`. It defaults to the whole bitstring.

```
>>> s = Bits('int15=-1')
>>> s.all(True, [3, 4, 12, 13])
True
>>> s.all(1)
True
```

```
Bits.any(
    value: bool,
    pos: Iterable[int] | None = None
) → bool
```

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 < -len(s)` or `pos > len(s)`. It defaults to the whole bitstring.

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

```
Bits.copy() → Bits
```

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.

```
Bits.count(
    value: bool
) → int
```

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

If you need to count more than just single bits you can use `findall`, for example `len(list(s.findall('0xabc')))`. Note that if the bitstring is very sparse, as in the example here, it could be quicker to find and count all the set bits with something like `len(list(s.findall('0b1')))`. For bitstrings with more entropy the `count` method will be much quicker than finding.

```
Bits.cut(
    bits: int,
    start: int | None = None,
    end: int | None = None,
    count: int | None = None
) → Iterator[Bits]
```

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

```
Bits.endswith(
    bs: BitsType,
    start: int | None = None,
    end: int | None = None
) → bool
```

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
```

```
Bits.find(
    bs: BitsType,
    start: int | None = None,
    end: int | None = None,
    bytealigned: bool | None = None
) → Tuple[int] | Tuple[()]
```

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,)
```

```
Bits.findall(
    bs: BitsType,
    start: int | None = None,
    end: int | None = None,
    count: int | None = None,
```

If *bytealigned* is `True` then *bs* will only be looked for at byte aligned positions. *start* and 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]
```

```
classmethod Bits.fromstring(
    s: str,
    /
) → Bits
```

Creates a new bitstring from the formatted string *s*. It is equivalent to creating a new bitstring using *s* as the first parameters, but can be clearer to write and will be slightly faster.

```
>>> b1 = Bits('int16=91')
>>> b2 = Bits.fromstring('int16=91')
>>> b1 == b2
True
```

```
Bits.join(
    sequence: Iterable
) → Bits
```

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
```



```
Bits.pp(
    fmt: str | None = None,
    width: int = 120,
```

Pretty print the bitstring's value according to the *fmt*. Either a single, or two comma separ can be specified, together with options for setting the maximum display *width*, the number of bits to display in each group, and the separator to print between groups.

```
>>> s = Bits('0b10111100101101001')*20
>>> s.pp(width=80)
<Bits, fmt='bin8, hex', length=340 bits> [
  0: 10111100 10110100 11011110 01011010 01101111 00101101 : bc b4 de 5a 6f 2d
 48: 00110111 10010110 10011011 11001011 01001101 11100101 : 37 96 9b cb 4d e5
 96: 10100110 11110010 11010011 01111001 01101001 10111100 : a6 f2 d3 79 69 bc
144: 10110100 11011110 01011010 01101111 00101101 00110111 : b4 de 5a 6f 2d 37
192: 10010110 10011011 11001011 01001101 11100101 10100110 : 96 9b cb 4d e5 a6
240: 11110010 11010011 01111001 01101001 10111100 10110100 : f2 d3 79 69 bc b4
288: 11011110 01011010 01101111 00101101 00110111 10010110 : de 5a 6f 2d 37 96
] + trailing_bits = 0x9
```

```
>>> s.pp('int20, hex', width=80, show_offset=False, sep=' / ')
<Bits, fmt='int20, hex', length=340 bits> [
-275635 / -107921 /  185209 /  433099 : bcb4d / e5a6f / 2d379 / 69bcb
 319066 /  455379 /  497307 / -215842 : 4de5a / 6f2d3 / 7969b / cb4de
 370418 / -182378 / -410444 / -137818 : 5a6f2 / d3796 / 9bcb4 / de5a6
 -53961 / -431684 / -307739 / -364755 : f2d37 / 969bc / b4de5 / a6f2d
 227689                               : 37969
]
```

The available formats are any fixed-length dtypes, for example `'bin'`, `'oct'`, `'hex'` and `'bytes'` together with types with explicit lengths such as `'uint5'` and `'float16'`. A bit length can be specified after the format (with an optional :) to give the number of bits represented by each group, otherwise the default is based on the format or formats selected.

For the `'bytes'` format, characters from the 'Latin Extended-A' unicode block are used for non-ASCII and unprintable characters.

If the bitstring cannot be represented in a format due to its length not being a multiple of the number of bits represented by each character then an `InterpretError` will be raised.

An output *stream* can be specified. This should be an object with a `write` method and the default is `sys.stdout`.

By default the output will have colours added in the terminal. This can be disabled - see `bitstring.options.no_color` for more information.


```
Bits.rfind(
    bs: BitsType,
    start: int | None = None,
```

< Page contents

Searches backwards for *bs* in the current bitstring and sets `pos` to the start of *bs* and ret
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,)
```

```
Bits.split(
    delimiter: BitsType,
    start: int | None = None,
    end: int | None = None,
    count: int | None = None,
    bytealigned: bool | None = None
) → Iterable[Bits]
```

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']
```

```
Bits.startswith(
    bs: BitsType,
    start: int | None = None,
    end: int | None = None
) → bool
```

```
>>> s = BitArray('0xef133')
>>> s.startswith('0b111011')
True
```

```
Bits.tobitarray() → bitarray.bitarray
```

Returns the bitstring as a `bitarray` object.

Converts the bitstring to an equivalent `bitarray` object from the `bitarray` package. This shouldn't be confused with the `BitArray` type provided in the `bitstring` package - the `bitarray` package is a separate third-party way of representing binary objects.

Note that `BitStream` and `ConstBitStream` types that have a bit position do support this method but the bit position information will be lost.

```
Bits.tobytes() → bytes
```

Returns the bitstring as a `bytes` object.

The returned value will be padded at the end with between zero and seven `0` bits to make it byte aligned. This differs from using the plain `bytes` property which will not pad with zero bits and instead raises an exception if the bitstring is not a whole number of bytes long.

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@'
```

This is equivalent to casting to a bytes object directly:

```
>>> bytes(s)
b'hello@'
```



```
Bits.tofile(
    f: BinaryIO
```

The data written will be padded at the end with between zero and seven `0` bits to make i

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

```
Bits.unpack(
    fmt: str | list[str | int],
    **kwargs
) → list[float | int | str | None | Bits]
```

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 Format tokens for details.

```
>>> s = Bits('int4=-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.

The `unpack` method is a natural complement of the `pack` function.

```
s = bitstring.pack('uint10, hex, int13, 0b11', 130, '3d', -23)
a, b, c, d = s.unpack('uint10, hex, int13, bin2')
```

Properties

The many ways to interpret bitstrings can be accessed via properties. These properties will be read-only for a `Bits` object, but are also writable for derived mutable types such as `BitArray` and `BitStream`

This list isn't exhaustive - see for example Exotic Floating Point Formats for information on l
many 8-bit and smaller floating point formats. Also see Exponential-Golomb Codes for some
variable length integer formats.

< Page contents

Note that the `bin`, `oct`, `hex`, `int`, `uint` and `float` properties can all be shortened to their initial letter.

```
Bits.bin: str
```

Property for the representation of the bitstring as a binary string. Can be shortened to just `b`.

```
Bits.bool: 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`.

```
Bits.bytes: 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'
```

```
Bits.hex: str
```

Property representing the hexadecimal value of the bitstring. Can be shortened to just `h`.

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'
```

`i`.

```
Bits.intbe: int
```

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.

```
Bits.intle: int
```

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.

```
Bits.intne: int
```

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.

```
Bits.float: float
```

```
Bits.floatbe: float
```

Property for the floating point representation of the bitstring. Can be shortened to just `f`.

The bitstring must be 16, 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`.

```
Bits.floatne: float
```

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

```
Bits.len: int
```

```
Bits.length: int
```

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

Using the `len()` built-in function is preferred in almost all cases, but these properties are available for backward compatibility. The only occasion where the properties are needed is if a 32-bit build of Python is being used and you have a bitstring whose length doesn't fit in a 32-bit unsigned integer. In that case `len(s)` may fail with an `OverflowError`, whereas `s.len` will still work. With 64-bit Python the problem shouldn't occur unless you have more than a couple of exabytes of data!

```
Bits.oct: str
```

Property for the octal representation of the bitstring. Can be shortened to just `o`.

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

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

```
Bits.uint: int
```

Property for the unsigned base-2 integer representation of the bitstring. Can be shortened to just `u`.


```
Bits.uintbe: int
```

```
Bits.uintle: int
```

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

```
Bits.uintne: int
```

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

Special Methods

```
Bits.__add__(
    bs
)
```

```
Bits.__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
```

```
Bits.__and__(
    bs
)
```

```
Bits.__rand__(
    bs
)
```

ValueError is raised.

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

```
Bits.__bool__()
```

if s:

Returns `False` if the bitstring is empty (has zero length), otherwise returns `True`.

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

```
Bits.__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
```

```
Bits.__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.

< Page contents

```
>>> s_copy3 = s[:]
>>> s == s_copy1 == s_copy2 == s_copy3
True
```

```
Bits.__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
```

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).

```
Bits.__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.

```
True
```

```
Bits.__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.

```
Bits.__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
>>> ~~s == s
True
```

```
Bits.__len__()
```

len(s)

Returns the length of the bitstring in bits.

If you are using a 32-bit Python build (which is quite unlikely these days) it's recommended that you use the `len` property rather than the `len` function because of the function will raise a `OverflowError` if the length is greater than `sys.maxsize`.


```
Bits.__lshift__(
    n
```

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

< Page contents

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

```
Bits.__mul__(
    n
)
```

```
Bits.__rmul__(
    n
)
```

`s * n / n * s`

Return bitstring consisting of *n* concatenations of another.

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

```
Bits.__ne__(
    bs
)
```

`s1 != s2`

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

```
Bits.__nonzero__()
```

See `__bool__`.

```
Bits.__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
```

```
Bits.__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')
```

```
Bits.__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')
```

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
```

See also the `pp` method for ways to pretty-print the bitstring.

```
Bits.__xor__(
    bs
)
```

```
Bits.__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
```

BitArray

```
class BitArray(
    auto: BitsType | int | None,
    /,
    length: int | None = None,
    offset: int | None = None,
    **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 the same as for `Bits`.

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.

Methods

```
BitArray.append(
    bs: BitsType
) → None
```

Join a `BitArray` to the end of the current `BitArray`.

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

```
BitArray.byteswap(
    fmt: str | int | Iterable[int] | None = None,
    start: int | None = None,
    end: int | None = None,
    repeat: bool = True
) → int
```


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

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

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

```
>>> s = BitArray('uintle32=1234')
>>> s.byteswap()
>>> print(s.uintbe)
1234
```

```
BitArray.clear() → None
```

Removes all bits from the bitstring.

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

```
BitArray.insert(
    bs: BitsType,
    pos: int
) → None
```

Inserts *bs* at *pos*.

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

< Page contents

```
>>> s.insert('0x00')
>>> s
BitStream('0xccd00ee')
```

```
BitArray.invert(
    pos: int | Iterable[int] | None = None
) → None
```

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 < -len(s)` or `pos > len(s)`. 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'
```

```
BitArray.overwrite(
    bs: BitsType,
    pos: int
) → None
```

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

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

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

```
BitArray.replace(
    old: BitsType,
    new: BitsType,
    start: int | None = None,
    end: int | None = None,
    count: int | None = None,
    bytealigned: bool | None = None
) → int
```

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 the start and end of the bitstring. 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
```

```
BitArray.reverse(
    start: int | None = None,
    end: int | None = None
) → None
```

Reverses bits in the `BitArray` in-place.

start and *end* give the range of bits to reverse and default to the start and end of the bitstrin

```
>>> a.reverse(0, 4)
>>> a.bin
'110100000'
```

```
BitArray.rol(
    bits: int,
    start: int | None = None,
    end: int | None = None
) → None
```

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 the start and end of the bitstring.

Raises `ValueError` if `bits < 0`.

```
>>> s = BitArray('0b01000001')
>>> s.rol(2)
>>> s.bin
'00000101'
```

```
BitArray.ror(
    bits: int,
    start: int | None = None,
    end: int | None = None
) → None
```

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 the start and end of the bitstring.

Raises `ValueError` if `bits < 0`.

```
BitArray.set(
    value: bool,
    pos: int | Iterable[int] | None = None
) → None
```

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

"0b1" or s.overwrite('0b1', x), especially if many bits are being set. In particular usi
object as an iterable is treated as a special case and is done efficiently.

< Page contents

```
>>> 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'
>>> s.set(1, range(0, len(s), 2))
>>> s.bin
'1010101010101010'
```

Properties

Note that the bin, oct, hex, int, uint and float properties can all be shortened to their initial letter. Properties can also have a length in bits appended to them to make properties such as u8 or floatle64 (with the exception of the bytes property which uses a unit of bytes instead of bits, so bytes4 is 32 bits long). These properties with lengths can be used to quickly create a new bitstring.

```
>>> a = BitArray()
>>> a.f32 = 17.6
>>> a.h
'418ccccd'
>>> a.i7 = -1
>>> a.b
'1111111'
```

The binary interpretation properties of the Bits class all become writable in the BitArray class.

For integer types, the properties can have a bit length appended to it such as u32 or int5 to specify the new length of the bitstring. Using a length too small to contain the value given will raise a CreationError.

When used as a setter without a new length the value must fit into the current length of the BitArray, else a ValueError will be raised.

```
ValueError: int 1232 is too large for a BitArray of length 8.
```

< Page contents

Other types also have restrictions on their lengths, and using an invalid length will raise a `CreationError`. For example trying to create a 20 bit floating point number or a two bit bool will raise this exception.

Special Methods

```
BitArray.__delitem__(
    key
)
```

`del s[start:end:step]`

Deletes the slice specified.

```
BitArray.__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
```

```
BitArray.__iand__(
    bs
)
```

ValueError is raised.

```
BitArray.__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.

```
BitArray.__imul__(
    n
)
```

s *= n

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

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

```
BitArray.__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.

```
BitArray.__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.

`s ^= bs`

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

```
BitArray.__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
```

< Bits

ConstBitStream >

ConstBitStream

```
class ConstBitStream(
    auto: BitsType | int | None,
    /,
    length: int | None = None,
    offset: int | None = None,
    pos: int = 0,
    **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 the same as for `Bits` except that an initial bit position `pos` can be given (defaults to 0).

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

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 default 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.

Reading using format strings

The `read` / `readlist` methods can also take a format string similar to that used in the auto in one token should be provided to `read` and a single value is returned. To read multiple tokens use `readlist`,

For example we can read and interpret three quantities from a bitstream with:

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

and we also could have combined the three reads as:

```
start_code, width, height = s.readlist('hex32, 2*uint12')
```

where here we are also using a multiplier to combine the format of 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('intle16, intle24, 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('pad12, uint4, pad4, uint8')
```

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':
```

Methods

```
ConstBitStream.bytealign() → int
```

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
```

```
ConstBitStream.peek(
    fmt: str | int
) → int | float | str | Bits | bool | bytes | None
```

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('hex8')
'12'
```



```
ConstBitStream.peeklist(
    fmt: str | list[str | int],
```

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.

```
ConstBitStream.read(
    fmt: str | int
) → int | float | str | Bits | bool | bytes | None
```

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. A full list of the tokens is given in Format tokens.

For example:

```
>>> s = ConstBitStream('0x23ef55302')
>>> s.read('hex12')
'23e'
>>> s.read('bin4')
'1111'
>>> s.read('u5')
10
>>> s.read('bits4')
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.

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 Format tokens for information on the format strings.

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

```
>>> s = ConstBitStream('0x43fe01ff21')
>>> s.readlist('hex8, uint6')
['43', 63]
>>> s.readlist(['bin3', 'intle16'])
['100', -509]
>>> s.pos = 0
>>> s.readlist('hex:b, uint:d', b=8, d=6)
['43', 63]
```

```
ConstBitStream.readto(
    bs: BitsType,
    bytealigned: bool | None = None
) → ConstBitStream
```

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('0x470001020347040506 47')
>>> s.readto('0x47', bytealigned=True)
ConstBitStream('0x47')
>>> s.readto('0x47', bytealigned=True)
ConstBitStream('0x0001020347')
>>> s.readto('0x47', bytealigned=True)
ConstBitStream('0x04050647')
```

be set to zero by default on construction, and will be modified by many of the methods desc
the stream is being read.

< Page contents

Using `find` or `rfind` will move `pos` to the start of the substring if it is found.

Note that the `pos` property isn't considered a part of the bitstring's identity; this allows it to vary for immutable `ConstBitStream` objects and means that it doesn't affect equality or hash values. It also will be reset to zero if a bitstring is copied.

```
ConstBitStream.bytepos: int
```

Property for setting and getting the current byte position in the bitstring. The value of `pos` will always be `bytepos * 8` as the two values are not independent.

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

```
ConstBitStream.pos: int
```

```
ConstBitStream.bitpos: int
```

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(s)`.

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

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

< BitArray

BitStream >

BitStream

```
class BitStream(
    auto: BitsType | int | None,
    /,
    length: int | None = None,
    offset: int | None = None,
    pos: int = 0,
    **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 the same as for `ConstBitStream`.

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.

The `pos` will also used as a default for the `BitArray.overwrite` and `BitArray.insert` methods.

The bit position is modified by methods that read bits, as described in `pos`, but for the mutable `BitStream` it is also modified by other methods:

- If a methods extends the bitstring (`+=`, `append`) the `pos` will move to the end of the bitstring.
- If a method otherwise changes the length of the bitstring (`prepend`, `insert`, sometimes `replace`) the `pos` becomes invalid and will be reset to `0`.

Array

```
class Array(
    dtype: str | Dtype,
    initializer: Iterable | int | Array | array.array | Bits | bytes | bytearray |
memoryview | BinaryIO | None = None,
    trailing_bits: BitsType | None = None
)
```

Create a new `Array` whose elements are set by the *dtype* (data-type) string or `Dtype`. This can be any format which has a fixed length. See Format tokens and Compact format strings for details on allowed dtype strings, noting that only formats with well defined bit lengths are allowed.

The *inititalizer* will typically be an iterable such as a list, but can also be many other things including an open binary file, a bytes or bytearray object, another `bitstring.Array` or an `array.array`. It can also be an integer, in which case the `Array` will be zero-initialised with that many items.

```
>>> bitstring.Array('i4', 8)
Array('int4', [0, 0, 0, 0, 0, 0, 0, 0])
```

The *trailing_bits* typically isn't used in construction, and specifies bits left over after interpreting the stored binary data according to the data type *dtype*.

The `Array` class is a way to efficiently store data that has a single type with a set length. The `bitstring.Array` type is meant as a more flexible version of the standard `array.array`, and can be used the same way.

```
import array
import bitstring

x = array.array('f', [1.0, 2.0, 3.14])
y = bitstring.Array('=f', [1.0, 2.0, 3.14])

assert x.tobytes() == y.tobytes()
```

This example packs three 32-bit floats into objects using both libraries. The only difference is the explicit native endianness for the format string of the bitstring version. The bitstring Array's advantage lies in the way that any fixed-length bitstring format can be used instead of just the dozen or so typecodes by the `array` module.

help illustrate:

```
from bitstring import Array

# Each unsigned int is stored in 4 bits
a = Array('uint4', [0, 5, 5, 3, 2])

# Convert and store floats in 8 bits each
b = Array('p3binary', [-56.0, 0.123, 99.6])

# Each element is a  7 bit signed integer
c = Array('int7', [-3, 0, 120])
```

You can then access and modify the `Array` with the usual notation:

```
a[1:4]  # Array('uint4', [5, 5, 3])
b[0]    # -56.0
c[-1]   # 120

a[0] = 2
b.extend([0.0, -1.5])
```

Conversion between `Array` types can be done using the `astype` method. If elements of the old array don't fit or don't make sense in the new array then the relevant exceptions will be raised.

```
>>> x = Array('float64', [89.3, 1e34, -0.00000001, 34])
>>> y = x.astype('float16')
>>> y
Array('float16', [89.3125, inf, -0.0, 34.0])
>>> y = y.astype('p4binary')
>>> y
Array('p4binary', [88.0, 240.0, 0.0, 32.0])
>>> y.astype('uint8')
Array('uint8', [88, 240, 0, 32])
>>> y.astype('uint7')
bitstring.CreationError: 240 is too large an unsigned integer for a bitstring of length 7. Th
```

You can also reinterpret the data by changing the `dtype` property directly. This will not copy any data but will cause the current data to be shown differently.

```
>>> x = Array('int16', [-5, 100, -4])
>>> x
Array('int16', [-5, 100, -4])
>>> x.dtype = 'int8'
```

< Page contents

property. You can freely manipulate the internal data using all of the methods available for t class.

The `Array` object also has a `trailing_bits` read-only data member, which consists of the end bits of the `data` that are left over when the `Array` is interpreted using the `dtype`. Typically `trailing_bits` will be an empty `BitArray` but if you change the length of the `data` or change the `dtype` specification there may be some bits left over.

Some methods, such as `append` and `extend` will raise an exception if used when `trailing_bits` is not empty, as it not clear how these should behave in this case. You can however still use `insert` which will always leave the `trailing_bits` unchanged.

The `dtype` string can be a type code such as `'>H'` or `'=d'` but it can also be a string defining any format which has a fixed-length in bits, for example `'int12'`, `'bfloat'`, `'bytes5'` or `'bool'`.

Note that the typecodes must include an endianness character to give the byte ordering. This is more like the `struct` module typecodes, and is different to the `array.array` typecodes which are always native-endian.

The correspondence between the big-endian type codes and bitstring dtype strings is given in the table below.

Type code	bitstring dtype
`'>b'`	`'int8'`
`'>B'`	`'uint8'`
`'>h'`	`'int16'`
`'>H'`	`'uint16'`
`'>l'`	`'int32'`
`'>L'`	`'uint32'`
`'>q'`	`'int64'`
`'>Q'`	`'uint64'`
`'>e'`	`'float16'`
`'>f'`	`'float32'`
`'>d'`	`'float64'`

The endianness character can be `'>'` for big-endian, `'<'` for little-endian or `'='` for native-endian (`'@'` can also be used for native-endian). In the bitstring dtypes the default is big-endian, but you ca or native endian using `'le'` or `'ne'` modifiers, for example:

```
'<H'          'uintle16'
```

Note that:

- The `array` module's native endianness means that different packed binary data will be created on different types of machines. Users may find that behaviour unexpected which is why endianness must be explicitly given as in the rest of the bitstring module.
- The `'u'` type code from the `array` module isn't supported as its length is platform dependent.
- The `'e'` type code isn't one of the `array` supported types, but it is used in the `struct` module and we support it here.
- The `'b'` and `'B'` type codes need to be preceded by an endianness character even though it makes no difference which one you use as they are only 1 byte long.

Methods

```
Array.append(
    x: float | int | str | bytes
) → None
```

Add a new element with value *x* to the end of the Array. The type of *x* should be appropriate for the type of the Array.

Raises a `ValueError` if the Array's bit length is not a multiple of its dtype length (see `trailing_bits`).

```
Array.astype(
    dtype: Dtype | str
) → Array
```

Cast the `Array` to the new *dtype* and return the result.

```
>>> a = Array('float64', [-990, 34, 1, 0.25])
>>> a.data
BitArray('0xc08ef0000000000040410000000000003ff0000000000003fd0000000000000')
>>> b = a.astype('float16')
>>> b.data
BitArray('0xe3bc50403c003400')
>>> a == b
Array('bool', [True, True, True, True])
```

Raises a `ValueError` if the format is not an integer number of bytes long.

```
>>> a = Array('uint32', [100, 1, 999])
>>> a.byteswap()
>>> a
Array('uint32', [1677721600, 16777216, 3875733504])
>>> a.dtype = 'uintle32'
>>> a
Array('uintle32', [100, 1, 999])
```

```
Array.count(
    value: float | int | str | bytes
) → int
```

Returns the number of elements set to *value*.

```
>>> a = Array('hex4')
>>> a.data += '0xdeadbeef'
>>> a
Array('hex4', ['d', 'e', 'a', 'd', 'b', 'e', 'e', 'f'])
>>> a.count('e')
3
```

For floating point types, using a *value* of `float('nan')` will count the number of elements for which `math.isnan()` returns `True`.

```
Array.equals(
    other: Any
) → bool
```

Equality test - *other* can be either another bitstring Array or an `array`. Returns `True` if the dtypes are equivalent and the underlying bit data is the same, otherwise returns `False`.

```
>>> a = Array('u8', [1, 2, 3, 2, 1])
>>> a[0:3].equals(a[-1:-4:-1])
True
>>> b = Array('i8', [1, 2, 3, 2, 1])
>>> a.equals(b)
False
```

```
True
```

Note that the `==` operator will perform an element-wise equality check and return a new `Array` of dtype `'bool'` (or raise an exception).

```
>>> a == b
Array('bool', [True, True, True, True, True])
```

```
Array.extend(
    iterable: Iterable | Array
) → None
```

Extend the Array by constructing new elements from the values in a list or other iterable.

The *iterable* can be another `Array` or an `array.array`, but only if the dtype is the same.

```
>>> a = Array('int5', [-5, 0, 10])
>>> a.extend([3, 2, 1])
>>> a.extend(a[0:3] // 5)
>>> a
Array('int5', [-5, 0, 10, 3, 2, 1, -1, 0, 2])
```

```
Array.fromfile(
    f: BinaryIO,
    n: int | None = None
) → None
```

Append items read from a file object.

```
Array.insert(
    i: int,
    x: float | int | str | bytes
) → None
```

Insert an item at a given position.

```
>>> a = Array('p3binary', [-10, -5, -0.5, 5, 10])
>>> a.insert(3, 0.5)
```

```
Array.pop(
    i: int | None = None
) → float | int | str | bytes
```

Remove and return the item at position i.

If a position isn't specified the final item is returned and removed.

```
>>> Array('bytes3', [b'ABC', b'DEF', b'ZZZ'])
>>> a.pop(0)
b'ABC'
>>> a.pop()
b'ZZZ'
>>> a.pop()
b'DEF'
```

```
Array.pp(
    fmt: str | None = None,
    width: int = 120,
    show_offset: bool = True,
    stream: TextIO = sys.stdout
) → None
```

Pretty print the Array.

The format string *fmt* defaults to the Array's current `dtype`, but any other valid Array format string can be used.

If a *fmt* doesn't have an explicit length, the Array's `itemsize` will be used.

A pair of comma-separated format strings can also be used - if both formats specify a length they must be the same. For example `'float, hex16'` or `'u4, b4'`.

The output will try to stay within *width* characters per line, but will always output at least one element value.

Setting *show_offset* to `False` will hide the element index on each line of the output.

An output *stream* can be specified. This should be an object with a `write` method and the default is `sys.stdout`.


```
>>> a = Array('u20', bytearray(range(100)))
>>> a.pp(width=70, show_offset=False)
<Array fmt='u20', length=40, itemsize=20 bits, total data size=100 bytes> [
     16  131844   20576  460809   41136  789774   61697   70163
```

```
>>> a.pp('hex32', width=70)
<Array fmt='hex32', length=25, itemsize=32 bits, total data size=100 bytes> [
  0: 00010203 04050607 08090a0b 0c0d0e0f 10111213 14151617 18191a1b
  7: 1c1d1e1f 20212223 24252627 28292a2b 2c2d2e2f 30313233 34353637
 14: 38393a3b 3c3d3e3f 40414243 44454647 48494a4b 4c4d4e4f 50515253
 21: 54555657 58595a5b 5c5d5e5f 60616263
]
```

```
>>> a.pp('i12, hex', show_offset=False, width=70)
<Array fmt='i12, hex', length=66, itemsize=12 bits, total data size=100 bytes> [
    0   258    48  1029    96  1800 : 000 102 030 405 060 708
  144 -1525   192  -754   241    17 : 090 a0b 0c0 d0e 0f1 011
  289   788   337  1559   385 -1766 : 121 314 151 617 181 91a
  433  -995   481  -224   530   547 : 1b1 c1d 1e1 f20 212 223
  578  1318   626 -2007   674 -1236 : 242 526 272 829 2a2 b2c
  722  -465   771   306   819  1077 : 2d2 e2f 303 132 333 435
  867  1848   915 -1477   963  -706 : 363 738 393 a3b 3c3 d3e
 1012    65  1060   836  1108  1607 : 3f4 041 424 344 454 647
 1156 -1718  1204  -947  1252  -176 : 484 94a 4b4 c4d 4e4 f50
 1301   595  1349  1366  1397 -1959 : 515 253 545 556 575 859
 1445 -1188  1493  -417  1542   354 : 5a5 b5c 5d5 e5f 606 162
] + trailing_bits = 0x63
```

By default the output will have colours added in the terminal. This can be disabled - see `bitstring.options.no_color` for more information.

```
Array.reverse() → None
```

Reverse the order of all items in the Array.

```
>>> a = Array('>L', [100, 200, 300])
>>> a.reverse()
>>> a
Array('>L', [300, 200, 100])
```

```
Array.tobytes() → bytes
```

Return Array data as bytes object, padding with zero bits at the end if needed.


```
>>> a = Array('i4', [3, -6, 2, -3, 2, -7])
>>> a.tobytes()
```

```
Array.tofile(
    f: BinaryIO
) → None
```

Write Array data to a file, padding with zero bits at the end if needed.

```
Array.tolist() → List[float | int | str | bytes]
```

Return Array items as a list.

Each packed element of the Array is converted to an ordinary Python object such as a `float` or an `int` depending on the Array's format, and returned in a Python list.

Special Methods

Type promotion

Many operations can be performed between two `Array` objects. For these to be valid the dtypes of the `Array` objects must be numerical, that is they must represent an integer or floating point value. Some operations have tighter restrictions, such as the shift operators `<<` and `>>` requiring integers only.

The dtype of the resulting `Array` is calculated by applying these rules:

Rule 0: For comparison operators (`<`, `>=`, `==`, `!=` etc.) the result is always an `Array` of dtype `'bool'`.

For other operators, one of the two input `Array` dtypes is used as the output dtype by applying the remaining rules in order until a winner is found:

- **Rule 1**: Floating point types always win against integer types.
- **Rule 2**: Signed integer types always win against unsigned integer types.
- **Rule 3**: Longer types win against shorter types.
- **Rule 4**: In a tie the first type wins.

Some examples should help illustrate:

Rule 0	`'uint8'`	`<=`	`'float64'`	→	`'bool'`
Rule 1	`'int32'`	`+`	`'float16'`	→	`'float16'`
Rule 2	`'uint20'`	`//`	`'int10'`	→	`'int10'`

Comparison operators

Comparison operators can operate between two `Array` objects, or between an `Array` and a scalar quantity (usually a number).

Note that they always produce an `Array` of `dtype` `'bool'`, including the equality and inequality operators.

To test the boolean equality of two Arrays use the `equals` method instead.

```
Array.__eq__(
    self,
    other: int | float | str | BitsType | Array
) → Array
```

```
a1 == a2
```

```
Array.__ne__(
    self,
    other: int | float | str | BitsType | Array
) → Array
```

```
a1 != a2
```

```
Array.__lt__(
    self,
    other: int | float | Array
) → Array
```

```
a1 < a2
```

```
Array.__le__(
    self,
    other: int | float | Array
) → Array
```

```
a1 <= a2
```

```
Array.__gt__(
    self,
    other: int | float | Array
) → Array
```

< Page contents

```
Array.__ge__(
    self,
    other: int | float | Array
) → Array
```

```
a1 >= a2
```

Numerical operators

```
Array.__add__(
    other: int | float | Array
) → Array
```

```
a + x
```

```
Array.__sub__(
    self,
    other: int | float | Array
) → Array
```

```
a - x
```

```
Array.__mul__(
    self,
    other: int | float | Array
) → Array
```

```
a * x
```

```
Array.__truediv__(
    self,
    other: int | float | Array
) → Array
```

```
a / x
```

```
Array.__floordiv__(
    self,
    other: int | float | Array
) → Array
```

< Page contents

```
Array.__rshift__(
    self,
    other: int | Array
) → Array
```

a >> i

```
Array.__lshift__(
    self,
    other: int | Array
) → Array
```

a << i

```
Array.__mod__(
    self,
    other: int | Array
) → Array
```

a % i

```
Array.__neg__(
    self
) → Array
```

-a

```
Array.__abs__(
    self
) → Array
```

abs(a)

Bitwise operators

```
Array.__and__(
    self,
    other: Bits
) → Array
```


a & bs

```
Array.__or__(
    self,
    other: Bits
) → Array
```

a | bs

```
>>> a |= '0x7fff'
```

```
Array.__xor__(
    self,
    other: Bits
) → Array
```

a ^ bs

```
>>> a ^= bytearray([56, 23])
```

Python language operators

```
Array.__len__(
    self
) → int
```

len(a)

Return the number of elements in the Array.

```
>>> a = Array('uint20', [1, 2, 3])
>>> len(a)
3
>>> a.dtype = 'uint1'
>>> len(a)
60
```



```
Array.__getitem__(
    self,
```

```
a[start:end:step]
```

```
Array.__setitem__(
    self,
    key: int | slice,
    value
) → None
```

```
a[i] = x

a[start:end:step] = x
```

```
Array.__delitem__(
    self,
    key: int | slice
) → None
```

```
del a[i]

del[start:end:step]
```

Properties

```
Array.data: BitArray
```

The bit data of the `Array`, as a `BitArray`. Read and write, and can be freely manipulated with all `BitArray` methods.

Note that some `Array` methods such as `append` and `extend` require the `data` to have a length that is a multiple of the `Array`'s `itemsize`.

```
Array.dtype: Dtype
```

The data type used to initialise the `Array` type. Read and write.

Note that some `Array` methods such as `append` and `extend` require the bit data to hav a multiple of the `Array`'s `itemsize`.

< Page contents

```
Array.itemsize: int
```

The size *in bits* of each item in the `Array`. Read-only.

Note that this gives a value in bits, unlike the equivalent in the `array` module which gives a value in bytes.

```
>>> a = Array('>h')
>>> b = Array('bool')
>>> a.itemsize
16
>>> b.itemsize
1
```

```
Array.trailing_bits: BitArray
```

A `BitArray` object equal to the end of the `data` that is not a multiple of the `itemsize`. Read only.

This will typically be an empty `BitArray`, but if the `dtype` or the `data` of an `Array` object has been altered after its creation then there may be left-over bits at the end of the data.

Note that any methods that append items to the `Array` will fail with a `ValueError` if there are any trailing bits.

< BitStream

Functions >

Functions

pack

```
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, see Format tokens and Compact format strings for details.

The tokens can be 'literals', like `0xef`, `0b110`, `uint8=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('bin3, hex4', '001', 'f')
>>> b = pack('uint10', 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('int8=a, bin=b, int4=a', a=7, b='0b110')
```


Plain names can also be used as follows:

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

< Page contents

```
>>> 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.

As an example of using just the `*values` arguments we can say:

```
s = bitstring.pack('hex32, uint12, uint12', '0x000001b3', 352, 288)
```

which is equivalent to initialising as:

```
s = BitStream('0x0000001b3, uint12=352, uint12=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('uint8, 0b110, int6, 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 = 'hex32=start_code, uint12=width, uint12=height'
```


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

Another method is to pass the same information as keywords at the end of pack's paramete

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

You can include constant bitstring tokens such as '0x101', '0xff', 'uint7=81' etc. and also use a keyword for the length specifier in the token, for example:

```
s = bitstring.pack('0xabc, int:n=-1', n=100)
```

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

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

Options

The bitstring module has an `options` object that allows certain module-wide behaviours to be set.

lsb0

bitstring.options.lsb0 : bool

By default bit numbering in the bitstring module is done from 'left' to 'right'. That is, from bit `0` at the start of the data to bit `n - 1` at the end. This allows bitstrings to be treated like an ordinary Python container that is only allowed to contain single bits.

The `lsb0` option allows bitstrings to use Least Significant Bit Zero (LSB0) bit numbering; that is the right-most bit in the bitstring will be bit 0, and the left-most bit will be bit (n-1), rather than the other way around. LSB0 is a more natural numbering system in many fields, but is the opposite to Most Significant Bit Zero (MSB0) numbering which is the natural option when thinking of bitstrings as standard Python c

bit index	0	1	2	3	4	5	6	7	8
value	0	1	0	0	0	1	1	1	1

In MSB0 everything behaves like an ordinary Python container. Bit zero is the left-most bit and reads/slices happen from left to right.

← LSB0

bit index	8	7	6	5	4	3	2	1	0
value	0	1	0	0	0	1	1	1	1

In LSB0 the final, right-most bit is labelled as bit zero. Reads and slices happen from right to left.

When bitstrings (or slices of bitstrings) are interpreted as integers and other types the left-most bit is considered as the most significant bit. It's important to note that this is the case irrespective of whether the first or last bit is considered the bit zero, so for example if you were to interpret a whole bitstring as an integer, its value would be the same with and without *lsb0* being set to *True*.

To illustrate this, for the example above this means that the bin and int representations would be `0b010001111` and `143` respectively for both MSB0 and LSB0 bit numbering.

To switch from the default MSB0, use `bitstring.options.lsb0`. This defaults to `False` and unless explicitly stated all examples and documentation related to the bitstring module use the default MSB0 indexing.

```
>>> bitstring.options.lsb0 = True
```

Slicing is still done with the start bit smaller than the end bit. For example:

```
>>> s = Bits('0b010001111')
>>> s[0:5]  # LSB0 so this is the right-most five bits
Bits('0b01111')
>>> s[0]
True
```

Note

In some standards and documents using LSB0 notation the slice of the final five bits would be shown as `s[5:0]`, which is reasonable as bit 5 comes before bit 0 when reading left to right, but this notation isn't used in this module as it clashes too much with the usual Python notation.

Negative indices work as you'd expect, with the first stored bit being `s[-1]` and the final stored [illegible] `s[-n]`.

For `BitStream` and `ConstBitStream` objects changing the value of `bitstring.options.lsb` the current position in the bitstring, unless that value is `0`, and future results are undefined. Basically don't perform reads or change the current bit position before switching the bit numbering system!

bytealigned

`bitstring.options.bytealigned : bool`

A number of methods take a bytealigned parameter to indicate that they should only work on byte boundaries (e.g. `find`, `findall`, `split` and `replace`). This parameter defaults to `bitstring.options.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.options.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.

mxfp_overflow

`bitstring.options.mxfp_overflow : str`

This option can be used to change the out-of-range behaviour of some 8-bit floating point types. The default value is `'saturate'` but it can also be set to `'overflow'`. See Exotic Floating Point Formats for details.

no_color

< Page contents

The bitstring module can use ANSI escape codes to colourise the output of the `Bits.pp` and methods. If a `NO_COLOR` environment variable is found and is not an empty string then this c to `True`, otherwise it defaults to `False`. See https://no-color.org for more information.

The terminal colours can also be turned off by setting `bitstring.options.no_color` to `True`.

Command Line Usage

The bitstring module can be called from the command line to perform simple operations. For example:

```
$ python -m bitstring int16=-400
0xfe70

$ python -m bitstring float32=0.2 bin
00111110010011001100110011001101

$ python -m bitstring 0xff "3*0b01,0b11" uint
65367

$ python -m bitstring hex=01, uint12=352.hex
01160
```

Command-line parameters are concatenated and a bitstring created from them. If the final parameter is either an interpretation string or ends with a `.` followed by an interpretation string then that interpretation of the bitstring will be used when printing it. If no interpretation is given then the bitstring is just printed.

Exceptions

```
exception Error(
    Exception
)
```

Base class for all module exceptions.

Inappropriate interpretation of binary data. For example using the 'bytes' property on a bitstring that isn't a whole number of bytes long.

```
exception ByteAlignError(
    Error
)
```

Whole-byte position or length needed.

```
exception CreationError(
    Error,
    ValueError
)
```

Inappropriate argument during bitstring creation.

```
exception ReadError(
    Error,
    IndexError
)
```

Reading or peeking past the end of a bitstring.

< Array

Appendices >

Exotic Floating Point Formats

Python floats are typically 64 bits long, but 32 and 16 bit sizes are also supported through the `struct` module. These are the well-known IEEE formats. Recently, lower precision floating points have become more widely used, largely driven by the requirements of machine learning algorithms and hardware.

As well as the 'half' precision 16 bit standard, a truncated version of the 32 bit standard called 'bfloat16' is used which has the range of a 32-bit float but less precision.

The #bits value in the tables below show how the available bits are split into *sign* + *exponent* + *mantissa*. There's always 1 bit to determine the sign of the floating point value. The more bits in the exponent, the larger the range that can be represented. The more bits in the mantissa, the greater the precision (~significant figures) of the values.

Type	# bits	Standard	+ive Range	bitstring / struct format
Double precision	1 + 11 + 52	IEEE 754	$10^{-308} \rightarrow 10^{308}$	`'float64' / 'd'`
Single precision	1 + 8 + 23	IEEE 754	$10^{-38} \rightarrow 10^{38}$	`'float32' / 'f'`
Half precision	1 + 5 + 10	IEEE 754	$6\times10^{-8} \rightarrow 65504$	`'float16' / 'e'`
bfloat	1 + 8 + 7	-	$10^{-38} \rightarrow 10^{38}$	`'bfloat' / -`

An example of creation and interpretation of a bfloat:

```
>>> a = Bits(bfloat=4.5e23)  # No need to specify length as always 16 bits
>>> a
Bits('0x66be')
>>> a.bfloat
4.486248158726163e+23  # Converted to Python float
```

IEEE 8-bit Floating Point Types

Note

This is an experimental feature and may be modified in future point releases.

In bitstring prior to version 4.2 the *p4binary8* and *p3binary8* formats were called *e4m3float* and *e5m2float* r
The two formats are almost identical, the difference being the addition of *inf* values that replace the largest
negative values that were previously available.

< Page contents

based on a publicly available draft of the standard. There are seven formats defined in the d
but only two are supported here. If you'd like the other precisions supported then raise a feat

The `p4binary8` has a single sign bit, 4 bits for the exponent and 3 bits for the mantissa. For a bit more range and less precision you can use `p3binary8` which has 5 bits for the exponent and only 2 for the mantissa. Note that in the standard they are called *binary8p4* and *binary8p3*, but for bitstring types any final digits should be the length of the type, so these slightly modified names were chosen.

Type	# bits	+ive Range	bitstring format
binary8p4	1 + 4 + 3	$10^{-3} \rightarrow 224$	`'p4binary8'`
binary8p3	1 + 5 + 2	$8{\times}10^{-6} \rightarrow 49152$	`'p3binary8'`

As there are just 256 possible values, both the range and precision of these formats are extremely limited. It's remarkable that any useful calculations can be performed, but both inference and training of large machine learning models can be done with these formats.

You can easily examine every possible value that these formats can represent using a line like this:

```
>>> [Bits(uint=x, length=8).p3binary8 for x in range(256)]
```

or using the `Array` type it's even more concise - we can create an Array and pretty print all the values with this line:

```
>>> Array('p4binary8', bytearray(range(256))).pp(width=90)
<Array fmt='p4binary', length=256, itemsize=8 bits, total data size=256 bytes>
[
   0:           0.0  0.0009765625   0.001953125  0.0029296875    0.00390625  0.0048828125
   6:   0.005859375  0.0068359375     0.0078125  0.0087890625   0.009765625  0.0107421875
  12:    0.01171875  0.0126953125   0.013671875  0.0146484375      0.015625   0.017578125
  18:    0.01953125   0.021484375     0.0234375   0.025390625    0.02734375   0.029296875
  24:       0.03125    0.03515625     0.0390625    0.04296875       0.046875    0.05078125
  30:     0.0546875    0.05859375        0.0625     0.0703125       0.078125     0.0859375
  36:       0.09375     0.1015625      0.109375     0.1171875          0.125      0.140625
  42:       0.15625      0.171875        0.1875      0.203125        0.21875      0.234375
  48:          0.25       0.28125        0.3125       0.34375          0.375       0.40625
  54:        0.4375       0.46875           0.5        0.5625          0.625        0.6875
  60:          0.75        0.8125         0.875        0.9375            1.0         1.125
  66:          1.25         1.375           1.5         1.625           1.75         1.875
  72:           2.0          2.25           2.5          2.75            3.0          3.25
  78:           3.5          3.75           4.0           4.5            5.0           5.5
  84:           6.0           6.5           7.0           7.5            8.0           9.0
  90:          10.0          11.0          12.0          13.0           14.0          15.0
  96:          16.0          18.0          20.0          22.0           24.0          26.0
 102:          28.0          30.0          32.0          36.0           40.0          44.0
 108:          48.0          52.0          56.0          60.0           64.0          72.0
 114:          80.0          88.0          96.0         104.0          112.0         120.0
```

```
 150:  -0.02734375  -0.029296875  -0.03125  -0.03515625  -0.0390625
 156:   -0.046875   -0.05078125  -0.0546875  -0.05859375    -0.0625
 162:   -0.078125    -0.0859375   -0.09375   -0.1015625   -0.109375
 168:      -0.125     -0.140625    -0.15625    -0.171875     -0.1875    -0.203125
 174:    -0.21875     -0.234375       -0.25     -0.28125     -0.3125     -0.34375
 180:      -0.375      -0.40625      -0.4375     -0.46875        -0.5      -0.5625
 186:      -0.625       -0.6875       -0.75       -0.8125       -0.875      -0.9375
 192:        -1.0        -1.125       -1.25        -1.375         -1.5        -1.625
 198:        -1.75       -1.875        -2.0         -2.25         -2.5         -2.75
 204:         -3.0        -3.25        -3.5         -3.75         -4.0          -4.5
 210:         -5.0         -5.5         -6.0          -6.5         -7.0          -7.5
 216:         -8.0         -9.0        -10.0         -11.0        -12.0         -13.0
 222:        -14.0        -15.0        -16.0         -18.0        -20.0         -22.0
 228:        -24.0        -26.0        -28.0         -30.0        -32.0         -36.0
 234:        -40.0        -44.0        -48.0         -52.0        -56.0         -60.0
 240:        -64.0        -72.0        -80.0         -88.0        -96.0        -104.0
 246:       -112.0       -120.0       -128.0        -144.0       -160.0        -176.0
 252:       -192.0       -208.0       -224.0          -inf
]
```

< Page contents

You'll see that there is only 1 zero value and only one 'nan' value, together with positive and negative 'inf' values.

When converting from a Python float (which will typically be stored in 64-bits) unrepresented values are rounded to nearest, with ties-to-even. This is the standard method used in IEEE 754.

Microscaling Formats

Note

This is an experimental feature and may be modified in future point releases.

A range of formats from the Microscaling Formats (MX) Alliance are supported. These are part of the Open Compute Project, and will usually have an external scale factor associated with them.

Eight-bit floats similar to the IEEE *p3binary8* and *p4binary8* are available, though these seem rather arbitrary and ugly in places in comparison to the IEEE definitions. There is also a format to use for the scaling factor, an int-like format which is really a float, and some sensible six and four bit float formats.

Type	# bits	+ive Range	bitstring format
E5M2	1 + 5 + 2	10^{-6} → 57344	`'e5m2mxfp'`
E4M3	1 + 4 + 3	$2{\times}10^{-3}$ → 448	`'e4m3mxfp'`
E3M2	1 + 3 + 2	0.0625 → 28	`'e3m2mxfp'`
E2M3	1 + 2 + 3	0.125 → 7.5	`'e2m3mxfp'`

E8M0	0 + 8 + 0	10^{-38} → 10^{38}	`'e8m0mxfp'`
INT8	8	0.015625 → 1.984375	`'mxint'`

- The E8M0 format is unsigned and designed to use as a scaling factor for blocks of the other formats.
- The INT8 format is like a signed two's complement 8-bit integer but with a scaling factor of 2^{-6}. So despite its name it is actually a float. The standard doesn't specify whether the largest negative value (-2.0) is a supported number or not. This implementation allows it.
- The E4M3 format is similar to the *p4binary8* format but with a different exponent bias and it wastes some values. It has no 'inf' values, instead opting to have two 'nan' values and two zero values.
- The E5M2 format is similar to the *p3binary8* format but wastes even more values. It does have positive and negative 'inf' values, but also six 'nan' values and two zero values.

The MX formats are designed to work with an external scaling factor. This should be in the E8M0 format, which uses a byte to encode the powers of two from 2^{-127} to 2^{127}, plus a 'nan' value. This can be specified in bitstring as part of the *Dtype*, and is very useful inside an `Array`.

```
>>> d = b'some_byte_data'
>>> a = Array(Dtype('e2m1mxfp', scale=2**10), d)
>>> a.pp()
<Array dtype='Dtype('e2m1mxfp', scale=2 ** 10)', length=28, itemsize=4 bits, total data size=
  0:  6144.0  1536.0  4096.0 -6144.0  4096.0 -3072.0  4096.0  3072.0  3072.0 -6144.0  4096.0
 14:  6144.0  2048.0  4096.0  3072.0  3072.0 -6144.0  4096.0  2048.0  4096.0   512.0  6144.0
]
```

To change the scale, replace the dtype in the *Array*:

```
>>> a.dtype = Dtype('e2m1mxfp', scale=2**6)
```

When initialising an *Array* from a list of values, you can also use the string `'auto'` as the scale, and an appropriate scale will be calculated based on the data.

```
>>> a = Array(Dtype('e2m1mxfp', scale='auto'), [0.0, 0.5, 40.5, 106.25, -52.0, -8.0])
>>> a.pp()
<Array dtype='Dtype('e2m1mxfp', scale=2 ** 4)', length=6, itemsize=4 bits, total data size=3
 0:    0.0    0.0   32.0   96.0  -48.0   -8.0
]
```

The scale is calculated based on the maximum absolute value of the data and the maximum representable value of the format. The auto-scale feature is only available for 8-bit and smaller floating point the IEEE 16-bit format. If all of the data is zero, then the scale is set to 1. For more details on this and these formats in general see the OCP Microscaling formats specification.

even rule is used. This is the same as is used in the IEEE 754 standard.

< Page contents

Note that for efficiency reasons Python floats are converted to 16-bit IEEE floats before being their final destination. This can mean that in edge cases the rounding to the 16-bit float will cause the next rounding to go in the other direction. The 16-bit float has 11 bits of precision, whereas the final format has at most 4 bits of precision, so this shouldn't be a real-world problem, but it could cause discrepancies when comparing with other methods. I could add a slower, more accurate mode if this is a problem (add a bug report).

Values that are out of range after rounding are dealt with as follows:

- `p3binary8` - Out of range values are set to `+inf` or `-inf`.
- `p4binary8` - Out of range values are set to `+inf` or `-inf`.
- `e5m2mxfp` - Out of range values are dealt with according to the `bitstring.options.mxfp_overflow` setting:
  - `'saturate'` (the default): values are set to the largest positive or negative finite value, as appropriate. Infinities will also be set to the largest finite value, despite the fact that the format has infinities.
  - `'overflow'`: Out of range values are set to `+inf` or `-inf`.
- `e4m3mxfp` - Out of range values are dealt with according to the `bitstring.options.mxfp_overflow` setting:
  - `'saturate'` (the default): values are set to the largest positive or negative value, as appropriate.
  - `'overflow'`: Out of range values are set to `nan`.
- `e3m2mxfp` - Out of range values are saturated to the largest positive or negative value.
- `e2m3mxfp` - Out of range values are saturated to the largest positive or negative value.
- `e2m1mxfp` - Out of range values are saturated to the largest positive or negative value.
- `mxint` - Out of range values are saturated to the largest positive or negative value. Note that the most negative value in this case is `-2.0`, which it is optional for an implementation to support.
- `e8m0mxfp` - No rounding is done. A `ValueError` will be raised if you try to convert a non-representable value. This is because the format is designed as a scaling factor, so it should generally be specified exactly.

< Appendices

Exponential-Golomb Codes >

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.

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.

ue

```
>>> s = BitStream(ue=12)
>>> s.bin
'0001101'
>>> s.append('ue=3')
>>> print(s.unpack('2*ue'))
[12, 3]
```

se

The `se` property does much the same as `ue` and the provisos there all apply. The obvious difference is that it interprets the bitstring as a signed exponential-Golomb rather than unsigned.

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

Exercise

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

`00100111011011010110001001001O1`

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 = Bits().join(f'ue={i}' for i in [3,0,0,2,2,1,0,0,8,4])
```

and to unpack it again:


```
>>> a.unpack('10*ue')
```

which uses these types of code a lot. There are other ways to map the bitstrings to integers:

< Page contents

Interleaved 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

010010	3
010011	-5
...	...

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

< Exotic Floating Point Formats

Optimisation Techniques >

Optimisation Techniques

The `bitstring` module aims to be as fast as reasonably possible, and since version 4.1 has used the `bitarray` C extension to power its core.

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.

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('uint12')
height = s.read('uint12')
flags = s.read('bin4')
```

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

```
width, height, flags = s.readlist('uint12, uint12, bin4')
```

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.

< Page contents

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.

Another problem is that the string `0xab` needs to be converted to a bitstring on every iteration. There are cacheing mechanisms that will make this faster after the first time, but if there is a constant conversion happening in a loop like this it is better to hoist it out of the loop by declaring `ab = Bits('0xab')` first and then adding this object instead of the string.

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()
```

```
    do_something()
```

< Page contents

If the pattern of setting or getting can be expressed as a `range` then it is much faster to pass in the range object so that it can be used to optimize the pattern. For example, instead of

```
for i in range(0, len(s), 2):
    s.set(True, i)
```

you should just write

```
s.set(True, range(0, len(s), 2))
```

< Exponential-Golomb Codes