The BitArray class

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

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

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

append(*bs*)

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

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

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

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

The *fmt* can be an integer, an iterable of integers or a compact format string similar to those used in **pack** (described in Compact format strings). It defaults to 0, which means reverse as many bytes as possible. The *fmt* gives a pattern of byte sizes to use to swap the endianness of the **BitArray**. Note that if you use a compact format string then the endianness identifier (`<`, `>` or `@`) is not needed, and if present it will be ignored.

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

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

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

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

clear()

Removes all bits from the bitstring.

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

copy()

Returns a copy of the bitstring.

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

insert(*bs*, *pos*)

Inserts *bs* at *pos*.

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

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

invert([*pos*])

Inverts one or many bits from `1` to `0` or vice versa.

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

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

overwrite(*bs*, *pos*)

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

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

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

prepend(*bs*)

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

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

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

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

If *bytealigned* is `True` then replacements will only be made on byte boundaries. *start* and *end* give the search range and default to `0` and len respectively. If *count* is specified then no more than this many replacements will be made.

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

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

Reverses bits in the **BitArray** in-place.

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

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

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

Rotates the contents of the **BitArray** in-place by *bits* bits to the left.

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

Raises **ValueError** if `bits < 0`.

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

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

Rotates the contents of the **BitArray** in-place by *bits* bits to the right.

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

Raises **ValueError** if `bits < 0`.

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

Sets one or many bits to either `1` (if *value* is `True`) or `0` (if *value* isn't `True`). *pos* can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise **IndexError** if `pos < -s.len` or `pos > s.len`. The default is to set every bit in the **BitArray**.

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

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

bin

Writable version of **Bits.bin**.

bool

Writable version of **Bits.bool**.

bytes

Writable version of **Bits.bytes**.

hex

Writable version of **Bits.hex**.

int

Writable version of **Bits.int**.

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

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

intbe

Writable version of **Bits.intbe**.

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

intle

Writable version of **Bits.intle**.

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

intne

Writable version of **Bits.intne**.

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

float

floatbe

Writable version of **Bits.float**.

floatle

Writable version of **Bits.floatle**.

floatne

Writable version of **Bits.floatne**.

oct

Writable version of **Bits.oct**.

se

Writable version of **Bits.se**.

ue

Writable version of **Bits.uie**.

sie

Writable version of Bits.sie.

uie

Writable version of Bits.ue.

uint

Writable version of Bits.uint.

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

uintbe

Writable version of Bits.uintbe.

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

uintle

Writable version of Bits.uintle.

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

uintne

Writable version of Bits.uintne.

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

__delitem__(*key*)

`del s[start:end:step]`

Deletes the slice specified.

__iadd__(*bs*)

`s1 += s2`

Appends *bs* to the current bitstring.

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

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

__iand__(*bs*)

`s &= bs`

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

__ilshift__(*n*)

`s <<= n`

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

__imul__(*n*)

`s *= n`

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

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

__ior__(*bs*)

`s |= bs`

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

__irshift__(*n*)

`s >>= n`

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

__ixor__(*bs*)

`s ^= bs`

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

__setitem__(*key*, *value*)

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

Replaces the slice specified with a new value.

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