#!/usr/bin/env python

"""

This package defines classes that simplify bit-wise creation, manipulation and

interpretation of data.

Classes:

Bits -- An immutable container for binary data.

BitArray -- A mutable container for binary data.

ConstBitStream -- An immutable container with streaming methods.

BitStream -- A mutable container with streaming methods.

Bits (base class)

/ \

+ mutating methods / \ + streaming methods

/ \

BitArray ConstBitStream

\ /

\ /

\ /

BitStream

Functions:

pack -- Create a BitStream from a format string.

Exceptions:

Error -- Module exception base class.

CreationError -- Error during creation.

InterpretError -- Inappropriate interpretation of binary data.

ByteAlignError -- Whole byte position or length needed.

ReadError -- Reading or peeking past the end of a bitstring.

http://python-bitstring.googlecode.com

"""

\_\_licence\_\_ = """

The MIT License

Copyright (c) 2006-2016 Scott Griffiths (dr.scottgriffiths@gmail.com)

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in

all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN

THE SOFTWARE.

"""

\_\_version\_\_ = "3.1.3"

\_\_author\_\_ = "Scott Griffiths"

import numbers

import copy

import sys

import re

import binascii

import mmap

import os

import struct

import operator

import collections

byteorder = sys.byteorder

bytealigned = False

"""Determines whether a number of methods default to working only on byte boundaries."""

# Maximum number of digits to use in \_\_str\_\_ and \_\_repr\_\_.

MAX\_CHARS = 250

# Maximum size of caches used for speed optimisations.

CACHE\_SIZE = 1000

class Error(Exception):

"""Base class for errors in the bitstring module."""

def \_\_init\_\_(self, \*params):

self.msg = params[0] if params else ''

self.params = params[1:]

def \_\_str\_\_(self):

if self.params:

return self.msg.format(\*self.params)

return self.msg

class ReadError(Error, IndexError):

"""Reading or peeking past the end of a bitstring."""

def \_\_init\_\_(self, \*params):

Error.\_\_init\_\_(self, \*params)

class InterpretError(Error, ValueError):

"""Inappropriate interpretation of binary data."""

def \_\_init\_\_(self, \*params):

Error.\_\_init\_\_(self, \*params)

class ByteAlignError(Error):

"""Whole-byte position or length needed."""

def \_\_init\_\_(self, \*params):

Error.\_\_init\_\_(self, \*params)

class CreationError(Error, ValueError):

"""Inappropriate argument during bitstring creation."""

def \_\_init\_\_(self, \*params):

Error.\_\_init\_\_(self, \*params)

class ConstByteStore(object):

"""Stores raw bytes together with a bit offset and length.

Used internally - not part of public interface.

"""

\_\_slots\_\_ = ('offset', '\_rawarray', 'bitlength')

def \_\_init\_\_(self, data, bitlength=None, offset=None):

"""data is either a bytearray or a MmapByteArray"""

self.\_rawarray = data

if offset is None:

offset = 0

if bitlength is None:

bitlength = 8 \* len(data) - offset

self.offset = offset

self.bitlength = bitlength

def getbit(self, pos):

assert 0 <= pos < self.bitlength

byte, bit = divmod(self.offset + pos, 8)

return bool(self.\_rawarray[byte] & (128 >> bit))

def getbyte(self, pos):

"""Direct access to byte data."""

return self.\_rawarray[pos]

def getbyteslice(self, start, end):

"""Direct access to byte data."""

c = self.\_rawarray[start:end]

return c

@property

def bytelength(self):

if not self.bitlength:

return 0

sb = self.offset // 8

eb = (self.offset + self.bitlength - 1) // 8

return eb - sb + 1

def \_\_copy\_\_(self):

return ByteStore(self.\_rawarray[:], self.bitlength, self.offset)

def \_appendstore(self, store):

"""Join another store on to the end of this one."""

if not store.bitlength:

return

# Set new array offset to the number of bits in the final byte of current array.

store = offsetcopy(store, (self.offset + self.bitlength) % 8)

if store.offset:

# first do the byte with the join.

joinval = (self.\_rawarray.pop() & (255 ^ (255 >> store.offset)) |

(store.getbyte(0) & (255 >> store.offset)))

self.\_rawarray.append(joinval)

self.\_rawarray.extend(store.\_rawarray[1:])

else:

self.\_rawarray.extend(store.\_rawarray)

self.bitlength += store.bitlength

def \_prependstore(self, store):

"""Join another store on to the start of this one."""

if not store.bitlength:

return

# Set the offset of copy of store so that it's final byte

# ends in a position that matches the offset of self,

# then join self on to the end of it.

store = offsetcopy(store, (self.offset - store.bitlength) % 8)

assert (store.offset + store.bitlength) % 8 == self.offset % 8

bit\_offset = self.offset % 8

if bit\_offset:

# first do the byte with the join.

store.setbyte(-1, (store.getbyte(-1) & (255 ^ (255 >> bit\_offset)) | \

(self.\_rawarray[self.byteoffset] & (255 >> bit\_offset))))

store.\_rawarray.extend(self.\_rawarray[self.byteoffset + 1: self.byteoffset + self.bytelength])

else:

store.\_rawarray.extend(self.\_rawarray[self.byteoffset: self.byteoffset + self.bytelength])

self.\_rawarray = store.\_rawarray

self.offset = store.offset

self.bitlength += store.bitlength

@property

def byteoffset(self):

return self.offset // 8

@property

def rawbytes(self):

return self.\_rawarray

class ByteStore(ConstByteStore):

"""Adding mutating methods to ConstByteStore

Used internally - not part of public interface.

"""

\_\_slots\_\_ = ()

def setbit(self, pos):

assert 0 <= pos < self.bitlength

byte, bit = divmod(self.offset + pos, 8)

self.\_rawarray[byte] |= (128 >> bit)

def unsetbit(self, pos):

assert 0 <= pos < self.bitlength

byte, bit = divmod(self.offset + pos, 8)

self.\_rawarray[byte] &= ~(128 >> bit)

def invertbit(self, pos):

assert 0 <= pos < self.bitlength

byte, bit = divmod(self.offset + pos, 8)

self.\_rawarray[byte] ^= (128 >> bit)

def setbyte(self, pos, value):

self.\_rawarray[pos] = value

def setbyteslice(self, start, end, value):

self.\_rawarray[start:end] = value

def offsetcopy(s, newoffset):

"""Return a copy of a ByteStore with the newoffset.

Not part of public interface.

"""

assert 0 <= newoffset < 8

if not s.bitlength:

return copy.copy(s)

else:

if newoffset == s.offset % 8:

return ByteStore(s.getbyteslice(s.byteoffset, s.byteoffset + s.bytelength), s.bitlength, newoffset)

newdata = []

d = s.\_rawarray

assert newoffset != s.offset % 8

if newoffset < s.offset % 8:

# We need to shift everything left

shiftleft = s.offset % 8 - newoffset

# First deal with everything except for the final byte

for x in range(s.byteoffset, s.byteoffset + s.bytelength - 1):

newdata.append(((d[x] << shiftleft) & 0xff) +\

(d[x + 1] >> (8 - shiftleft)))

bits\_in\_last\_byte = (s.offset + s.bitlength) % 8

if not bits\_in\_last\_byte:

bits\_in\_last\_byte = 8

if bits\_in\_last\_byte > shiftleft:

newdata.append((d[s.byteoffset + s.bytelength - 1] << shiftleft) & 0xff)

else: # newoffset > s.\_offset % 8

shiftright = newoffset - s.offset % 8

newdata.append(s.getbyte(0) >> shiftright)

for x in range(s.byteoffset + 1, s.byteoffset + s.bytelength):

newdata.append(((d[x - 1] << (8 - shiftright)) & 0xff) +\

(d[x] >> shiftright))

bits\_in\_last\_byte = (s.offset + s.bitlength) % 8

if not bits\_in\_last\_byte:

bits\_in\_last\_byte = 8

if bits\_in\_last\_byte + shiftright > 8:

newdata.append((d[s.byteoffset + s.bytelength - 1] << (8 - shiftright)) & 0xff)

new\_s = ByteStore(bytearray(newdata), s.bitlength, newoffset)

assert new\_s.offset == newoffset

return new\_s

def equal(a, b):

"""Return True if ByteStores a == b.

Not part of public interface.

"""

# We want to return False for inequality as soon as possible, which

# means we get lots of special cases.

# First the easy one - compare lengths:

a\_bitlength = a.bitlength

b\_bitlength = b.bitlength

if a\_bitlength != b\_bitlength:

return False

if not a\_bitlength:

assert b\_bitlength == 0

return True

# Make 'a' the one with the smaller offset

if (a.offset % 8) > (b.offset % 8):

a, b = b, a

# and create some aliases

a\_bitoff = a.offset % 8

b\_bitoff = b.offset % 8

a\_byteoffset = a.byteoffset

b\_byteoffset = b.byteoffset

a\_bytelength = a.bytelength

b\_bytelength = b.bytelength

da = a.\_rawarray

db = b.\_rawarray

# If they are pointing to the same data, they must be equal

if da is db and a.offset == b.offset:

return True

if a\_bitoff == b\_bitoff:

bits\_spare\_in\_last\_byte = 8 - (a\_bitoff + a\_bitlength) % 8

if bits\_spare\_in\_last\_byte == 8:

bits\_spare\_in\_last\_byte = 0

# Special case for a, b contained in a single byte

if a\_bytelength == 1:

a\_val = ((da[a\_byteoffset] << a\_bitoff) & 0xff) >> (8 - a\_bitlength)

b\_val = ((db[b\_byteoffset] << b\_bitoff) & 0xff) >> (8 - b\_bitlength)

return a\_val == b\_val

# Otherwise check first byte

if da[a\_byteoffset] & (0xff >> a\_bitoff) != db[b\_byteoffset] & (0xff >> b\_bitoff):

return False

# then everything up to the last

b\_a\_offset = b\_byteoffset - a\_byteoffset

for x in range(1 + a\_byteoffset, a\_byteoffset + a\_bytelength - 1):

if da[x] != db[b\_a\_offset + x]:

return False

# and finally the last byte

return (da[a\_byteoffset + a\_bytelength - 1] >> bits\_spare\_in\_last\_byte ==

db[b\_byteoffset + b\_bytelength - 1] >> bits\_spare\_in\_last\_byte)

assert a\_bitoff != b\_bitoff

# This is how much we need to shift a to the right to compare with b:

shift = b\_bitoff - a\_bitoff

# Special case for b only one byte long

if b\_bytelength == 1:

assert a\_bytelength == 1

a\_val = ((da[a\_byteoffset] << a\_bitoff) & 0xff) >> (8 - a\_bitlength)

b\_val = ((db[b\_byteoffset] << b\_bitoff) & 0xff) >> (8 - b\_bitlength)

return a\_val == b\_val

# Special case for a only one byte long

if a\_bytelength == 1:

assert b\_bytelength == 2

a\_val = ((da[a\_byteoffset] << a\_bitoff) & 0xff) >> (8 - a\_bitlength)

b\_val = ((db[b\_byteoffset] << 8) + db[b\_byteoffset + 1]) << b\_bitoff

b\_val &= 0xffff

b\_val >>= 16 - b\_bitlength

return a\_val == b\_val

# Compare first byte of b with bits from first byte of a

if (da[a\_byteoffset] & (0xff >> a\_bitoff)) >> shift != db[b\_byteoffset] & (0xff >> b\_bitoff):

return False

# Now compare every full byte of b with bits from 2 bytes of a

for x in range(1, b\_bytelength - 1):

# Construct byte from 2 bytes in a to compare to byte in b

b\_val = db[b\_byteoffset + x]

a\_val = ((da[a\_byteoffset + x - 1] << 8) + da[a\_byteoffset + x]) >> shift

a\_val &= 0xff

if a\_val != b\_val:

return False

# Now check bits in final byte of b

final\_b\_bits = (b.offset + b\_bitlength) % 8

if not final\_b\_bits:

final\_b\_bits = 8

b\_val = db[b\_byteoffset + b\_bytelength - 1] >> (8 - final\_b\_bits)

final\_a\_bits = (a.offset + a\_bitlength) % 8

if not final\_a\_bits:

final\_a\_bits = 8

if b.bytelength > a\_bytelength:

assert b\_bytelength == a\_bytelength + 1

a\_val = da[a\_byteoffset + a\_bytelength - 1] >> (8 - final\_a\_bits)

a\_val &= 0xff >> (8 - final\_b\_bits)

return a\_val == b\_val

assert a\_bytelength == b\_bytelength

a\_val = da[a\_byteoffset + a\_bytelength - 2] << 8

a\_val += da[a\_byteoffset + a\_bytelength - 1]

a\_val >>= (8 - final\_a\_bits)

a\_val &= 0xff >> (8 - final\_b\_bits)

return a\_val == b\_val

class MmapByteArray(object):

"""Looks like a bytearray, but from an mmap.

Not part of public interface.

"""

\_\_slots\_\_ = ('filemap', 'filelength', 'source', 'byteoffset', 'bytelength')

def \_\_init\_\_(self, source, bytelength=None, byteoffset=None):

self.source = source

source.seek(0, os.SEEK\_END)

self.filelength = source.tell()

if byteoffset is None:

byteoffset = 0

if bytelength is None:

bytelength = self.filelength - byteoffset

self.byteoffset = byteoffset

self.bytelength = bytelength

self.filemap = mmap.mmap(source.fileno(), 0, access=mmap.ACCESS\_READ)

def \_\_getitem\_\_(self, key):

try:

start = key.start

stop = key.stop

except AttributeError:

try:

assert 0 <= key < self.bytelength

return ord(self.filemap[key + self.byteoffset])

except TypeError:

# for Python 3

return self.filemap[key + self.byteoffset]

else:

if start is None:

start = 0

if stop is None:

stop = self.bytelength

assert key.step is None

assert 0 <= start < self.bytelength

assert 0 <= stop <= self.bytelength

s = slice(start + self.byteoffset, stop + self.byteoffset)

return bytearray(self.filemap.\_\_getitem\_\_(s))

def \_\_len\_\_(self):

return self.bytelength

# This creates a dictionary for every possible byte with the value being

# the key with its bits reversed.

BYTE\_REVERSAL\_DICT = dict()

# For Python 2.x/ 3.x coexistence

# Yes this is very very hacky.

try:

xrange

for i in range(256):

BYTE\_REVERSAL\_DICT[i] = chr(int("{0:08b}".format(i)[::-1], 2))

except NameError:

for i in range(256):

BYTE\_REVERSAL\_DICT[i] = bytes([int("{0:08b}".format(i)[::-1], 2)])

from io import IOBase as file

xrange = range

basestring = str

# Python 2.x octals start with '0', in Python 3 it's '0o'

LEADING\_OCT\_CHARS = len(oct(1)) - 1

def tidy\_input\_string(s):

"""Return string made lowercase and with all whitespace removed."""

s = ''.join(s.split()).lower()

return s

INIT\_NAMES = ('uint', 'int', 'ue', 'se', 'sie', 'uie', 'hex', 'oct', 'bin', 'bits',

'uintbe', 'intbe', 'uintle', 'intle', 'uintne', 'intne',

'float', 'floatbe', 'floatle', 'floatne', 'bytes', 'bool', 'pad')

TOKEN\_RE = re.compile(r'(?P<name>' + '|'.join(INIT\_NAMES) +

r')((:(?P<len>[^=]+)))?(=(?P<value>.\*))?$', re.IGNORECASE)

DEFAULT\_UINT = re.compile(r'(?P<len>[^=]+)?(=(?P<value>.\*))?$', re.IGNORECASE)

MULTIPLICATIVE\_RE = re.compile(r'(?P<factor>.\*)\\*(?P<token>.+)')

# Hex, oct or binary literals

LITERAL\_RE = re.compile(r'(?P<name>0(x|o|b))(?P<value>.+)', re.IGNORECASE)

# An endianness indicator followed by one or more struct.pack codes

STRUCT\_PACK\_RE = re.compile(r'(?P<endian><|>|@)?(?P<fmt>(?:\d\*[bBhHlLqQfd])+)$')

# A number followed by a single character struct.pack code

STRUCT\_SPLIT\_RE = re.compile(r'\d\*[bBhHlLqQfd]')

# These replicate the struct.pack codes

# Big-endian

REPLACEMENTS\_BE = {'b': 'intbe:8', 'B': 'uintbe:8',

'h': 'intbe:16', 'H': 'uintbe:16',

'l': 'intbe:32', 'L': 'uintbe:32',

'q': 'intbe:64', 'Q': 'uintbe:64',

'f': 'floatbe:32', 'd': 'floatbe:64'}

# Little-endian

REPLACEMENTS\_LE = {'b': 'intle:8', 'B': 'uintle:8',

'h': 'intle:16', 'H': 'uintle:16',

'l': 'intle:32', 'L': 'uintle:32',

'q': 'intle:64', 'Q': 'uintle:64',

'f': 'floatle:32', 'd': 'floatle:64'}

# Size in bytes of all the pack codes.

PACK\_CODE\_SIZE = {'b': 1, 'B': 1, 'h': 2, 'H': 2, 'l': 4, 'L': 4,

'q': 8, 'Q': 8, 'f': 4, 'd': 8}

\_tokenname\_to\_initialiser = {'hex': 'hex', '0x': 'hex', '0X': 'hex', 'oct': 'oct',

'0o': 'oct', '0O': 'oct', 'bin': 'bin', '0b': 'bin',

'0B': 'bin', 'bits': 'auto', 'bytes': 'bytes', 'pad': 'pad'}

def structparser(token):

"""Parse struct-like format string token into sub-token list."""

m = STRUCT\_PACK\_RE.match(token)

if not m:

return [token]

else:

endian = m.group('endian')

if endian is None:

return [token]

# Split the format string into a list of 'q', '4h' etc.

formatlist = re.findall(STRUCT\_SPLIT\_RE, m.group('fmt'))

# Now deal with mulitiplicative factors, 4h -> hhhh etc.

fmt = ''.join([f[-1] \* int(f[:-1]) if len(f) != 1 else

f for f in formatlist])

if endian == '@':

# Native endianness

if byteorder == 'little':

endian = '<'

else:

assert byteorder == 'big'

endian = '>'

if endian == '<':

tokens = [REPLACEMENTS\_LE[c] for c in fmt]

else:

assert endian == '>'

tokens = [REPLACEMENTS\_BE[c] for c in fmt]

return tokens

def tokenparser(fmt, keys=None, token\_cache={}):

"""Divide the format string into tokens and parse them.

Return stretchy token and list of [initialiser, length, value]

initialiser is one of: hex, oct, bin, uint, int, se, ue, 0x, 0o, 0b etc.

length is None if not known, as is value.

If the token is in the keyword dictionary (keys) then it counts as a

special case and isn't messed with.

tokens must be of the form: [factor\*][initialiser][:][length][=value]

"""

try:

return token\_cache[(fmt, keys)]

except KeyError:

token\_key = (fmt, keys)

# Very inefficient expanding of brackets.

fmt = expand\_brackets(fmt)

# Split tokens by ',' and remove whitespace

# The meta\_tokens can either be ordinary single tokens or multiple

# struct-format token strings.

meta\_tokens = (''.join(f.split()) for f in fmt.split(','))

return\_values = []

stretchy\_token = False

for meta\_token in meta\_tokens:

# See if it has a multiplicative factor

m = MULTIPLICATIVE\_RE.match(meta\_token)

if not m:

factor = 1

else:

factor = int(m.group('factor'))

meta\_token = m.group('token')

# See if it's a struct-like format

tokens = structparser(meta\_token)

ret\_vals = []

for token in tokens:

if keys and token in keys:

# Don't bother parsing it, it's a keyword argument

ret\_vals.append([token, None, None])

continue

value = length = None

if token == '':

continue

# Match literal tokens of the form 0x... 0o... and 0b...

m = LITERAL\_RE.match(token)

if m:

name = m.group('name')

value = m.group('value')

ret\_vals.append([name, length, value])

continue

# Match everything else:

m1 = TOKEN\_RE.match(token)

if not m1:

# and if you don't specify a 'name' then the default is 'uint':

m2 = DEFAULT\_UINT.match(token)

if not m2:

raise ValueError("Don't understand token '{0}'.".format(token))

if m1:

name = m1.group('name')

length = m1.group('len')

if m1.group('value'):

value = m1.group('value')

else:

assert m2

name = 'uint'

length = m2.group('len')

if m2.group('value'):

value = m2.group('value')

if name == 'bool':

if length is not None:

raise ValueError("You can't specify a length with bool tokens - they are always one bit.")

length = 1

if length is None and name not in ('se', 'ue', 'sie', 'uie'):

stretchy\_token = True

if length is not None:

# Try converting length to int, otherwise check it's a key.

try:

length = int(length)

if length < 0:

raise Error

# For the 'bytes' token convert length to bits.

if name == 'bytes':

length \*= 8

except Error:

raise ValueError("Can't read a token with a negative length.")

except ValueError:

if not keys or length not in keys:

raise ValueError("Don't understand length '{0}' of token.".format(length))

ret\_vals.append([name, length, value])

# This multiplies by the multiplicative factor, but this means that

# we can't allow keyword values as multipliers (e.g. n\*uint:8).

# The only way to do this would be to return the factor in some fashion

# (we can't use the key's value here as it would mean that we couldn't

# sensibly continue to cache the function's results. (TODO).

return\_values.extend(ret\_vals \* factor)

return\_values = [tuple(x) for x in return\_values]

if len(token\_cache) < CACHE\_SIZE:

token\_cache[token\_key] = stretchy\_token, return\_values

return stretchy\_token, return\_values

# Looks for first number\*(

BRACKET\_RE = re.compile(r'(?P<factor>\d+)\\*\(')

def expand\_brackets(s):

"""Remove whitespace and expand all brackets."""

s = ''.join(s.split())

while True:

start = s.find('(')

if start == -1:

break

count = 1 # Number of hanging open brackets

p = start + 1

while p < len(s):

if s[p] == '(':

count += 1

if s[p] == ')':

count -= 1

if not count:

break

p += 1

if count:

raise ValueError("Unbalanced parenthesis in '{0}'.".format(s))

if start == 0 or s[start - 1] != '\*':

s = s[0:start] + s[start + 1:p] + s[p + 1:]

else:

m = BRACKET\_RE.search(s)

if m:

factor = int(m.group('factor'))

matchstart = m.start('factor')

s = s[0:matchstart] + (factor - 1) \* (s[start + 1:p] + ',') + s[start + 1:p] + s[p + 1:]

else:

raise ValueError("Failed to parse '{0}'.".format(s))

return s

# This converts a single octal digit to 3 bits.

OCT\_TO\_BITS = ['{0:03b}'.format(i) for i in xrange(8)]

# A dictionary of number of 1 bits contained in binary representation of any byte

BIT\_COUNT = dict(zip(xrange(256), [bin(i).count('1') for i in xrange(256)]))

class Bits(object):

"""A container holding an immutable sequence of bits.

For a mutable container use the BitArray class instead.

Methods:

all() -- Check if all specified bits are set to 1 or 0.

any() -- Check if any of specified bits are set to 1 or 0.

count() -- Count the number of bits set to 1 or 0.

cut() -- Create generator of constant sized chunks.

endswith() -- Return whether the bitstring ends with a sub-string.

find() -- Find a sub-bitstring in the current bitstring.

findall() -- Find all occurrences of a sub-bitstring in the current bitstring.

join() -- Join bitstrings together using current bitstring.

rfind() -- Seek backwards to find a sub-bitstring.

split() -- Create generator of chunks split by a delimiter.

startswith() -- Return whether the bitstring starts with a sub-bitstring.

tobytes() -- Return bitstring as bytes, padding if needed.

tofile() -- Write bitstring to file, padding if needed.

unpack() -- Interpret bits using format string.

Special methods:

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

Properties:

bin -- The bitstring as a binary string.

bool -- For single bit bitstrings, interpret as True or False.

bytes -- The bitstring as a bytes object.

float -- Interpret as a floating point number.

floatbe -- Interpret as a big-endian floating point number.

floatle -- Interpret as a little-endian floating point number.

floatne -- Interpret as a native-endian floating point number.

hex -- The bitstring as a hexadecimal string.

int -- Interpret as a two's complement signed integer.

intbe -- Interpret as a big-endian signed integer.

intle -- Interpret as a little-endian signed integer.

intne -- Interpret as a native-endian signed integer.

len -- Length of the bitstring in bits.

oct -- The bitstring as an octal string.

se -- Interpret as a signed exponential-Golomb code.

ue -- Interpret as an unsigned exponential-Golomb code.

sie -- Interpret as a signed interleaved exponential-Golomb code.

uie -- Interpret as an unsigned interleaved exponential-Golomb code.

uint -- Interpret as a two's complement unsigned integer.

uintbe -- Interpret as a big-endian unsigned integer.

uintle -- Interpret as a little-endian unsigned integer.

uintne -- Interpret as a native-endian unsigned integer.

"""

\_\_slots\_\_ = ('\_datastore')

def \_\_init\_\_(self, auto=None, length=None, offset=None, \*\*kwargs):

"""Either specify an 'auto' initialiser:

auto -- a string of comma separated tokens, an integer, a file object,

a bytearray, a boolean iterable or another bitstring.

Or initialise via \*\*kwargs with one (and only one) of:

bytes -- raw data as a string, for example read from a binary file.

bin -- binary string representation, e.g. '0b001010'.

hex -- hexadecimal string representation, e.g. '0x2ef'

oct -- octal string representation, e.g. '0o777'.

uint -- an unsigned integer.

int -- a signed integer.

float -- a floating point number.

uintbe -- an unsigned big-endian whole byte integer.

intbe -- a signed big-endian whole byte integer.

floatbe - a big-endian floating point number.

uintle -- an unsigned little-endian whole byte integer.

intle -- a signed little-endian whole byte integer.

floatle -- a little-endian floating point number.

uintne -- an unsigned native-endian whole byte integer.

intne -- a signed native-endian whole byte integer.

floatne -- a native-endian floating point number.

se -- a signed exponential-Golomb code.

ue -- an unsigned exponential-Golomb code.

sie -- a signed interleaved exponential-Golomb code.

uie -- an unsigned interleaved exponential-Golomb code.

bool -- a boolean (True or False).

filename -- a file which will be opened in binary read-only mode.

Other keyword arguments:

length -- length of the bitstring in bits, if needed and appropriate.

It must be supplied for all integer and float initialisers.

offset -- bit offset to the data. These offset bits are

ignored and this is mainly intended for use when

initialising using 'bytes' or 'filename'.

"""

pass

def \_\_new\_\_(cls, auto=None, length=None, offset=None, \_cache={}, \*\*kwargs):

# For instances auto-initialised with a string we intern the

# instance for re-use.

try:

if isinstance(auto, basestring):

try:

return \_cache[auto]

except KeyError:

x = object.\_\_new\_\_(Bits)

try:

\_, tokens = tokenparser(auto)

except ValueError as e:

raise CreationError(\*e.args)

x.\_datastore = ConstByteStore(bytearray(0), 0, 0)

for token in tokens:

x.\_datastore.\_appendstore(Bits.\_init\_with\_token(\*token).\_datastore)

assert x.\_assertsanity()

if len(\_cache) < CACHE\_SIZE:

\_cache[auto] = x

return x

if isinstance(auto, Bits):

return auto

except TypeError:

pass

x = super(Bits, cls).\_\_new\_\_(cls)

x.\_initialise(auto, length, offset, \*\*kwargs)

return x

def \_initialise(self, auto, length, offset, \*\*kwargs):

if length is not None and length < 0:

raise CreationError("bitstring length cannot be negative.")

if offset is not None and offset < 0:

raise CreationError("offset must be >= 0.")

if auto is not None:

self.\_initialise\_from\_auto(auto, length, offset)

return

if not kwargs:

# No initialisers, so initialise with nothing or zero bits

if length is not None and length != 0:

data = bytearray((length + 7) // 8)

self.\_setbytes\_unsafe(data, length, 0)

return

self.\_setbytes\_unsafe(bytearray(0), 0, 0)

return

k, v = kwargs.popitem()

try:

init\_without\_length\_or\_offset[k](self, v)

if length is not None or offset is not None:

raise CreationError("Cannot use length or offset with this initialiser.")

except KeyError:

try:

init\_with\_length\_only[k](self, v, length)

if offset is not None:

raise CreationError("Cannot use offset with this initialiser.")

except KeyError:

if offset is None:

offset = 0

try:

init\_with\_length\_and\_offset[k](self, v, length, offset)

except KeyError:

raise CreationError("Unrecognised keyword '{0}' used to initialise.", k)

def \_initialise\_from\_auto(self, auto, length, offset):

if offset is None:

offset = 0

self.\_setauto(auto, length, offset)

return

def \_\_copy\_\_(self):

"""Return a new copy of the Bits for the copy module."""

# Note that if you want a new copy (different ID), use \_copy instead.

# The copy can return self as it's immutable.

return self

def \_\_lt\_\_(self, other):

raise TypeError("unorderable type: {0}".format(type(self).\_\_name\_\_))

def \_\_gt\_\_(self, other):

raise TypeError("unorderable type: {0}".format(type(self).\_\_name\_\_))

def \_\_le\_\_(self, other):

raise TypeError("unorderable type: {0}".format(type(self).\_\_name\_\_))

def \_\_ge\_\_(self, other):

raise TypeError("unorderable type: {0}".format(type(self).\_\_name\_\_))

def \_\_add\_\_(self, bs):

"""Concatenate bitstrings and return new bitstring.

bs -- the bitstring to append.

"""

bs = Bits(bs)

if bs.len <= self.len:

s = self.\_copy()

s.\_append(bs)

else:

s = bs.\_copy()

s = self.\_\_class\_\_(s)

s.\_prepend(self)

return s

def \_\_radd\_\_(self, bs):

"""Append current bitstring to bs and return new bitstring.

bs -- the string for the 'auto' initialiser that will be appended to.

"""

bs = self.\_converttobitstring(bs)

return bs.\_\_add\_\_(self)

def \_\_getitem\_\_(self, key):

"""Return a new bitstring representing a slice of the current bitstring.

Indices are in units of the step parameter (default 1 bit).

Stepping is used to specify the number of bits in each item.

>>> print BitArray('0b00110')[1:4]

'0b011'

>>> print BitArray('0x00112233')[1:3:8]

'0x1122'

"""

length = self.len

try:

step = key.step if key.step is not None else 1

except AttributeError:

# single element

if key < 0:

key += length

if not 0 <= key < length:

raise IndexError("Slice index out of range.")

# Single bit, return True or False

return self.\_datastore.getbit(key)

else:

if step != 1:

# convert to binary string and use string slicing

bs = self.\_\_class\_\_()

bs.\_setbin\_unsafe(self.\_getbin().\_\_getitem\_\_(key))

return bs

start, stop = 0, length

if key.start is not None:

start = key.start

if key.start < 0:

start += stop

if key.stop is not None:

stop = key.stop

if key.stop < 0:

stop += length

start = max(start, 0)

stop = min(stop, length)

if start < stop:

return self.\_slice(start, stop)

else:

return self.\_\_class\_\_()

def \_\_len\_\_(self):

"""Return the length of the bitstring in bits."""

return self.\_getlength()

def \_\_str\_\_(self):

"""Return approximate string representation of bitstring for printing.

Short strings will be given wholly in hexadecimal or binary. Longer

strings may be part hexadecimal and part binary. Very long strings will

be truncated with '...'.

"""

length = self.len

if not length:

return ''

if length > MAX\_CHARS \* 4:

# Too long for hex. Truncate...

return ''.join(('0x', self.\_readhex(MAX\_CHARS \* 4, 0), '...'))

# If it's quite short and we can't do hex then use bin

if length < 32 and length % 4 != 0:

return '0b' + self.bin

# If we can use hex then do so

if not length % 4:

return '0x' + self.hex

# Otherwise first we do as much as we can in hex

# then add on 1, 2 or 3 bits on at the end

bits\_at\_end = length % 4

return ''.join(('0x', self.\_readhex(length - bits\_at\_end, 0),

', ', '0b',

self.\_readbin(bits\_at\_end, length - bits\_at\_end)))

def \_\_repr\_\_(self):

"""Return representation that could be used to recreate the bitstring.

If the returned string is too long it will be truncated. See \_\_str\_\_().

"""

length = self.len

if isinstance(self.\_datastore.\_rawarray, MmapByteArray):

offsetstring = ''

if self.\_datastore.byteoffset or self.\_offset:

offsetstring = ", offset=%d" % (self.\_datastore.\_rawarray.byteoffset \* 8 + self.\_offset)

lengthstring = ", length=%d" % length

return "{0}(filename='{1}'{2}{3})".format(self.\_\_class\_\_.\_\_name\_\_,

self.\_datastore.\_rawarray.source.name, lengthstring, offsetstring)

else:

s = self.\_\_str\_\_()

lengthstring = ''

if s.endswith('...'):

lengthstring = " # length={0}".format(length)

return "{0}('{1}'){2}".format(self.\_\_class\_\_.\_\_name\_\_, s, lengthstring)

def \_\_eq\_\_(self, bs):

"""Return True if two bitstrings have the same binary representation.

>>> BitArray('0b1110') == '0xe'

True

"""

try:

bs = Bits(bs)

except TypeError:

return False

return equal(self.\_datastore, bs.\_datastore)

def \_\_ne\_\_(self, bs):

"""Return False if two bitstrings have the same binary representation.

>>> BitArray('0b111') == '0x7'

False

"""

return not self.\_\_eq\_\_(bs)

def \_\_invert\_\_(self):

"""Return bitstring with every bit inverted.

Raises Error if the bitstring is empty.

"""

if not self.len:

raise Error("Cannot invert empty bitstring.")

s = self.\_copy()

s.\_invert\_all()

return s

def \_\_lshift\_\_(self, n):

"""Return bitstring with bits shifted by n to the left.

n -- the number of bits to shift. Must be >= 0.

"""

if n < 0:

raise ValueError("Cannot shift by a negative amount.")

if not self.len:

raise ValueError("Cannot shift an empty bitstring.")

n = min(n, self.len)

s = self.\_slice(n, self.len)

s.\_append(Bits(n))

return s

def \_\_rshift\_\_(self, n):

"""Return bitstring with bits shifted by n to the right.

n -- the number of bits to shift. Must be >= 0.

"""

if n < 0:

raise ValueError("Cannot shift by a negative amount.")

if not self.len:

raise ValueError("Cannot shift an empty bitstring.")

if not n:

return self.\_copy()

s = self.\_\_class\_\_(length=min(n, self.len))

s.\_append(self[:-n])

return s

def \_\_mul\_\_(self, n):

"""Return bitstring consisting of n concatenations of self.

Called for expression of the form 'a = b\*3'.

n -- The number of concatenations. Must be >= 0.

"""

if n < 0:

raise ValueError("Cannot multiply by a negative integer.")

if not n:

return self.\_\_class\_\_()

s = self.\_copy()

s.\_imul(n)

return s

def \_\_rmul\_\_(self, n):

"""Return bitstring consisting of n concatenations of self.

Called for expressions of the form 'a = 3\*b'.

n -- The number of concatenations. Must be >= 0.

"""

return self.\_\_mul\_\_(n)

def \_\_and\_\_(self, bs):

"""Bit-wise 'and' between two bitstrings. Returns new bitstring.

bs -- The bitstring to '&' with.

Raises ValueError if the two bitstrings have differing lengths.

"""

bs = Bits(bs)

if self.len != bs.len:

raise ValueError("Bitstrings must have the same length "

"for & operator.")

s = self.\_copy()

s.\_iand(bs)

return s

def \_\_rand\_\_(self, bs):

"""Bit-wise 'and' between two bitstrings. Returns new bitstring.

bs -- the bitstring to '&' with.

Raises ValueError if the two bitstrings have differing lengths.

"""

return self.\_\_and\_\_(bs)

def \_\_or\_\_(self, bs):

"""Bit-wise 'or' between two bitstrings. Returns new bitstring.

bs -- The bitstring to '|' with.

Raises ValueError if the two bitstrings have differing lengths.

"""

bs = Bits(bs)

if self.len != bs.len:

raise ValueError("Bitstrings must have the same length "

"for | operator.")

s = self.\_copy()

s.\_ior(bs)

return s

def \_\_ror\_\_(self, bs):

"""Bit-wise 'or' between two bitstrings. Returns new bitstring.

bs -- The bitstring to '|' with.

Raises ValueError if the two bitstrings have differing lengths.

"""

return self.\_\_or\_\_(bs)

def \_\_xor\_\_(self, bs):

"""Bit-wise 'xor' between two bitstrings. Returns new bitstring.

bs -- The bitstring to '^' with.

Raises ValueError if the two bitstrings have differing lengths.

"""

bs = Bits(bs)

if self.len != bs.len:

raise ValueError("Bitstrings must have the same length "

"for ^ operator.")

s = self.\_copy()

s.\_ixor(bs)

return s

def \_\_rxor\_\_(self, bs):

"""Bit-wise 'xor' between two bitstrings. Returns new bitstring.

bs -- The bitstring to '^' with.

Raises ValueError if the two bitstrings have differing lengths.

"""

return self.\_\_xor\_\_(bs)

def \_\_contains\_\_(self, bs):

"""Return whether bs is contained in the current bitstring.

bs -- The bitstring to search for.

"""

# Don't want to change pos

try:

pos = self.\_pos

except AttributeError:

pass

found = Bits.find(self, bs, bytealigned=False)

try:

self.\_pos = pos

except AttributeError:

pass

return bool(found)

def \_\_hash\_\_(self):

"""Return an integer hash of the object."""

# We can't in general hash the whole bitstring (it could take hours!)

# So instead take some bits from the start and end.

if self.len <= 160:

# Use the whole bitstring.

shorter = self

else:

# Take 10 bytes from start and end

shorter = self[:80] + self[-80:]

h = 0

for byte in shorter.tobytes():

try:

h = (h << 4) + ord(byte)

except TypeError:

# Python 3

h = (h << 4) + byte

g = h & 0xf0000000

if g & (1 << 31):

h ^= (g >> 24)

h ^= g

return h % 1442968193

# This is only used in Python 2.x...

def \_\_nonzero\_\_(self):

"""Return True if any bits are set to 1, otherwise return False."""

return self.any(True)

# ...whereas this is used in Python 3.x

\_\_bool\_\_ = \_\_nonzero\_\_

def \_assertsanity(self):

"""Check internal self consistency as a debugging aid."""

assert self.len >= 0

assert 0 <= self.\_offset, "offset={0}".format(self.\_offset)

assert (self.len + self.\_offset + 7) // 8 == self.\_datastore.bytelength + self.\_datastore.byteoffset

return True

@classmethod

def \_init\_with\_token(cls, name, token\_length, value):

if token\_length is not None:

token\_length = int(token\_length)

if token\_length == 0:

return cls()

# For pad token just return the length in zero bits

if name == 'pad':

return cls(token\_length)

if value is None:

if token\_length is None:

error = "Token has no value ({0}=???).".format(name)

else:

error = "Token has no value ({0}:{1}=???).".format(name, token\_length)

raise ValueError(error)

try:

b = cls(\*\*{\_tokenname\_to\_initialiser[name]: value})

except KeyError:

if name in ('se', 'ue', 'sie', 'uie'):

b = cls(\*\*{name: int(value)})

elif name in ('uint', 'int', 'uintbe', 'intbe', 'uintle', 'intle', 'uintne', 'intne'):

b = cls(\*\*{name: int(value), 'length': token\_length})

elif name in ('float', 'floatbe', 'floatle', 'floatne'):

b = cls(\*\*{name: float(value), 'length': token\_length})

elif name == 'bool':

if value in (1, 'True', '1'):

b = cls(bool=True)

elif value in (0, 'False', '0'):

b = cls(bool=False)

else:

raise CreationError("bool token can only be 'True' or 'False'.")

else:

raise CreationError("Can't parse token name {0}.", name)

if token\_length is not None and b.len != token\_length:

msg = "Token with length {0} packed with value of length {1} ({2}:{3}={4})."

raise CreationError(msg, token\_length, b.len, name, token\_length, value)

return b

def \_clear(self):

"""Reset the bitstring to an empty state."""

self.\_datastore = ByteStore(bytearray(0))

def \_setauto(self, s, length, offset):

"""Set bitstring from a bitstring, file, bool, integer, iterable or string."""

# As s can be so many different things it's important to do the checks

# in the correct order, as some types are also other allowed types.

# So basestring must be checked before Iterable

# and bytes/bytearray before Iterable but after basestring!

if isinstance(s, Bits):

if length is None:

length = s.len - offset

self.\_setbytes\_unsafe(s.\_datastore.rawbytes, length, s.\_offset + offset)

return

if isinstance(s, file):

if offset is None:

offset = 0

if length is None:

length = os.path.getsize(s.name) \* 8 - offset

byteoffset, offset = divmod(offset, 8)

bytelength = (length + byteoffset \* 8 + offset + 7) // 8 - byteoffset

m = MmapByteArray(s, bytelength, byteoffset)

if length + byteoffset \* 8 + offset > m.filelength \* 8:

raise CreationError("File is not long enough for specified "

"length and offset.")

self.\_datastore = ConstByteStore(m, length, offset)

return

if length is not None:

raise CreationError("The length keyword isn't applicable to this initialiser.")

if offset:

raise CreationError("The offset keyword isn't applicable to this initialiser.")

if isinstance(s, basestring):

bs = self.\_converttobitstring(s)

assert bs.\_offset == 0

self.\_setbytes\_unsafe(bs.\_datastore.rawbytes, bs.length, 0)

return

if isinstance(s, (bytes, bytearray)):

self.\_setbytes\_unsafe(bytearray(s), len(s) \* 8, 0)

return

if isinstance(s, numbers.Integral):

# Initialise with s zero bits.

if s < 0:

msg = "Can't create bitstring of negative length {0}."

raise CreationError(msg, s)

data = bytearray((s + 7) // 8)

self.\_datastore = ByteStore(data, s, 0)

return

if isinstance(s, collections.Iterable):

# Evaluate each item as True or False and set bits to 1 or 0.

self.\_setbin\_unsafe(''.join(str(int(bool(x))) for x in s))

return

raise TypeError("Cannot initialise bitstring from {0}.".format(type(s)))

def \_setfile(self, filename, length, offset):

"""Use file as source of bits."""

source = open(filename, 'rb')

if offset is None:

offset = 0

if length is None:

length = os.path.getsize(source.name) \* 8 - offset

byteoffset, offset = divmod(offset, 8)

bytelength = (length + byteoffset \* 8 + offset + 7) // 8 - byteoffset

m = MmapByteArray(source, bytelength, byteoffset)

if length + byteoffset \* 8 + offset > m.filelength \* 8:

raise CreationError("File is not long enough for specified "

"length and offset.")

self.\_datastore = ConstByteStore(m, length, offset)

def \_setbytes\_safe(self, data, length=None, offset=0):

"""Set the data from a string."""

data = bytearray(data)

if length is None:

# Use to the end of the data

length = len(data)\*8 - offset

self.\_datastore = ByteStore(data, length, offset)

else:

if length + offset > len(data) \* 8:

msg = "Not enough data present. Need {0} bits, have {1}."

raise CreationError(msg, length + offset, len(data) \* 8)

if length == 0:

self.\_datastore = ByteStore(bytearray(0))

else:

self.\_datastore = ByteStore(data, length, offset)

def \_setbytes\_unsafe(self, data, length, offset):

"""Unchecked version of \_setbytes\_safe."""

self.\_datastore = ByteStore(data[:], length, offset)

assert self.\_assertsanity()

def \_readbytes(self, length, start):

"""Read bytes and return them. Note that length is in bits."""

assert length % 8 == 0

assert start + length <= self.len

if not (start + self.\_offset) % 8:

return bytes(self.\_datastore.getbyteslice((start + self.\_offset) // 8,

(start + self.\_offset + length) // 8))

return self.\_slice(start, start + length).tobytes()

def \_getbytes(self):

"""Return the data as an ordinary string."""

if self.len % 8:

raise InterpretError("Cannot interpret as bytes unambiguously - "

"not multiple of 8 bits.")

return self.\_readbytes(self.len, 0)

def \_setuint(self, uint, length=None):

"""Reset the bitstring to have given unsigned int interpretation."""

try:

if length is None:

# Use the whole length. Deliberately not using .len here.

length = self.\_datastore.bitlength

except AttributeError:

# bitstring doesn't have a \_datastore as it hasn't been created!

pass

# TODO: All this checking code should be hoisted out of here!

if length is None or length == 0:

raise CreationError("A non-zero length must be specified with a "

"uint initialiser.")

if uint >= (1 << length):

msg = "{0} is too large an unsigned integer for a bitstring of length {1}. "\

"The allowed range is [0, {2}]."

raise CreationError(msg, uint, length, (1 << length) - 1)

if uint < 0:

raise CreationError("uint cannot be initialsed by a negative number.")

s = hex(uint)[2:]

s = s.rstrip('L')

if len(s) & 1:

s = '0' + s

try:

data = bytes.fromhex(s)

except AttributeError:

# the Python 2.x way

data = binascii.unhexlify(s)

# Now add bytes as needed to get the right length.

extrabytes = ((length + 7) // 8) - len(data)

if extrabytes > 0:

data = b'\x00' \* extrabytes + data

offset = 8 - (length % 8)

if offset == 8:

offset = 0

self.\_setbytes\_unsafe(bytearray(data), length, offset)

def \_readuint(self, length, start):

"""Read bits and interpret as an unsigned int."""

if not length:

raise InterpretError("Cannot interpret a zero length bitstring "

"as an integer.")

offset = self.\_offset

startbyte = (start + offset) // 8

endbyte = (start + offset + length - 1) // 8

b = binascii.hexlify(bytes(self.\_datastore.getbyteslice(startbyte, endbyte + 1)))

assert b

i = int(b, 16)

final\_bits = 8 - ((start + offset + length) % 8)

if final\_bits != 8:

i >>= final\_bits

i &= (1 << length) - 1

return i

def \_getuint(self):

"""Return data as an unsigned int."""

return self.\_readuint(self.len, 0)

def \_setint(self, int\_, length=None):

"""Reset the bitstring to have given signed int interpretation."""

# If no length given, and we've previously been given a length, use it.

if length is None and hasattr(self, 'len') and self.len != 0:

length = self.len

if length is None or length == 0:

raise CreationError("A non-zero length must be specified with an int initialiser.")

if int\_ >= (1 << (length - 1)) or int\_ < -(1 << (length - 1)):

raise CreationError("{0} is too large a signed integer for a bitstring of length {1}. "

"The allowed range is [{2}, {3}].", int\_, length, -(1 << (length - 1)),

(1 << (length - 1)) - 1)

if int\_ >= 0:

self.\_setuint(int\_, length)

return

# TODO: We should decide whether to just use the \_setuint, or to do the bit flipping,

# based upon which will be quicker. If the -ive number is less than half the maximum

# possible then it's probably quicker to do the bit flipping...

# Do the 2's complement thing. Add one, set to minus number, then flip bits.

int\_ += 1

self.\_setuint(-int\_, length)

self.\_invert\_all()

def \_readint(self, length, start):

"""Read bits and interpret as a signed int"""

ui = self.\_readuint(length, start)

if not ui >> (length - 1):

# Top bit not set, number is positive

return ui

# Top bit is set, so number is negative

tmp = (~(ui - 1)) & ((1 << length) - 1)

return -tmp

def \_getint(self):

"""Return data as a two's complement signed int."""

return self.\_readint(self.len, 0)

def \_setuintbe(self, uintbe, length=None):

"""Set the bitstring to a big-endian unsigned int interpretation."""

if length is not None and length % 8 != 0:

raise CreationError("Big-endian integers must be whole-byte. "

"Length = {0} bits.", length)

self.\_setuint(uintbe, length)

def \_readuintbe(self, length, start):

"""Read bits and interpret as a big-endian unsigned int."""

if length % 8:

raise InterpretError("Big-endian integers must be whole-byte. "

"Length = {0} bits.", length)

return self.\_readuint(length, start)

def \_getuintbe(self):

"""Return data as a big-endian two's complement unsigned int."""

return self.\_readuintbe(self.len, 0)

def \_setintbe(self, intbe, length=None):

"""Set bitstring to a big-endian signed int interpretation."""

if length is not None and length % 8 != 0:

raise CreationError("Big-endian integers must be whole-byte. "

"Length = {0} bits.", length)

self.\_setint(intbe, length)

def \_readintbe(self, length, start):

"""Read bits and interpret as a big-endian signed int."""

if length % 8:

raise InterpretError("Big-endian integers must be whole-byte. "

"Length = {0} bits.", length)

return self.\_readint(length, start)

def \_getintbe(self):

"""Return data as a big-endian two's complement signed int."""

return self.\_readintbe(self.len, 0)

def \_setuintle(self, uintle, length=None):

if length is not None and length % 8 != 0:

raise CreationError("Little-endian integers must be whole-byte. "

"Length = {0} bits.", length)

self.\_setuint(uintle, length)

self.\_reversebytes(0, self.len)

def \_readuintle(self, length, start):

"""Read bits and interpret as a little-endian unsigned int."""

if length % 8:

raise InterpretError("Little-endian integers must be whole-byte. "

"Length = {0} bits.", length)

assert start + length <= self.len

absolute\_pos = start + self.\_offset

startbyte, offset = divmod(absolute\_pos, 8)

val = 0

if not offset:

endbyte = (absolute\_pos + length - 1) // 8

chunksize = 4 # for 'L' format

while endbyte - chunksize + 1 >= startbyte:

val <<= 8 \* chunksize

val += struct.unpack('<L', bytes(self.\_datastore.getbyteslice(endbyte + 1 - chunksize, endbyte + 1)))[0]

endbyte -= chunksize

for b in xrange(endbyte, startbyte - 1, -1):

val <<= 8

val += self.\_datastore.getbyte(b)

else:

data = self.\_slice(start, start + length)

assert data.len % 8 == 0

data.\_reversebytes(0, self.len)

for b in bytearray(data.bytes):

val <<= 8

val += b

return val

def \_getuintle(self):

return self.\_readuintle(self.len, 0)

def \_setintle(self, intle, length=None):

if length is not None and length % 8 != 0:

raise CreationError("Little-endian integers must be whole-byte. "

"Length = {0} bits.", length)

self.\_setint(intle, length)

self.\_reversebytes(0, self.len)

def \_readintle(self, length, start):

"""Read bits and interpret as a little-endian signed int."""

ui = self.\_readuintle(length, start)

if not ui >> (length - 1):

# Top bit not set, number is positive

return ui

# Top bit is set, so number is negative

tmp = (~(ui - 1)) & ((1 << length) - 1)

return -tmp

def \_getintle(self):

return self.\_readintle(self.len, 0)

def \_setfloat(self, f, length=None):

# If no length given, and we've previously been given a length, use it.

if length is None and hasattr(self, 'len') and self.len != 0:

length = self.len

if length is None or length == 0:

raise CreationError("A non-zero length must be specified with a "

"float initialiser.")

if length == 32:

b = struct.pack('>f', f)

elif length == 64:

b = struct.pack('>d', f)

else:

raise CreationError("floats can only be 32 or 64 bits long, "

"not {0} bits", length)

self.\_setbytes\_unsafe(bytearray(b), length, 0)

def \_readfloat(self, length, start):

"""Read bits and interpret as a float."""

if not (start + self.\_offset) % 8:

startbyte = (start + self.\_offset) // 8

if length == 32:

f, = struct.unpack('>f', bytes(self.\_datastore.getbyteslice(startbyte, startbyte + 4)))

elif length == 64:

f, = struct.unpack('>d', bytes(self.\_datastore.getbyteslice(startbyte, startbyte + 8)))

else:

if length == 32:

f, = struct.unpack('>f', self.\_readbytes(32, start))

elif length == 64:

f, = struct.unpack('>d', self.\_readbytes(64, start))

try:

return f

except NameError:

raise InterpretError("floats can only be 32 or 64 bits long, not {0} bits", length)

def \_getfloat(self):

"""Interpret the whole bitstring as a float."""

return self.\_readfloat(self.len, 0)

def \_setfloatle(self, f, length=None):

# If no length given, and we've previously been given a length, use it.

if length is None and hasattr(self, 'len') and self.len != 0:

length = self.len

if length is None or length == 0:

raise CreationError("A non-zero length must be specified with a "

"float initialiser.")

if length == 32:

b = struct.pack('<f', f)

elif length == 64:

b = struct.pack('<d', f)

else:

raise CreationError("floats can only be 32 or 64 bits long, "

"not {0} bits", length)

self.\_setbytes\_unsafe(bytearray(b), length, 0)

def \_readfloatle(self, length, start):

"""Read bits and interpret as a little-endian float."""

startbyte, offset = divmod(start + self.\_offset, 8)

if not offset:

if length == 32:

f, = struct.unpack('<f', bytes(self.\_datastore.getbyteslice(startbyte, startbyte + 4)))

elif length == 64:

f, = struct.unpack('<d', bytes(self.\_datastore.getbyteslice(startbyte, startbyte + 8)))

else:

if length == 32:

f, = struct.unpack('<f', self.\_readbytes(32, start))

elif length == 64:

f, = struct.unpack('<d', self.\_readbytes(64, start))

try:

return f

except NameError:

raise InterpretError("floats can only be 32 or 64 bits long, "

"not {0} bits", length)

def \_getfloatle(self):

"""Interpret the whole bitstring as a little-endian float."""

return self.\_readfloatle(self.len, 0)

def \_setue(self, i):

"""Initialise bitstring with unsigned exponential-Golomb code for integer i.

Raises CreationError if i < 0.

"""

if i < 0:

raise CreationError("Cannot use negative initialiser for unsigned "

"exponential-Golomb.")

if not i:

self.\_setbin\_unsafe('1')

return

tmp = i + 1

leadingzeros = -1

while tmp > 0:

tmp >>= 1

leadingzeros += 1

remainingpart = i + 1 - (1 << leadingzeros)

binstring = '0' \* leadingzeros + '1' + Bits(uint=remainingpart,

length=leadingzeros).bin

self.\_setbin\_unsafe(binstring)

def \_readue(self, pos):

"""Return interpretation of next bits as unsigned exponential-Golomb code.

Raises ReadError if the end of the bitstring is encountered while

reading the code.

"""

oldpos = pos

try:

while not self[pos]:

pos += 1

except IndexError:

raise ReadError("Read off end of bitstring trying to read code.")

leadingzeros = pos - oldpos

codenum = (1 << leadingzeros) - 1

if leadingzeros > 0:

if pos + leadingzeros + 1 > self.len:

raise ReadError("Read off end of bitstring trying to read code.")

codenum += self.\_readuint(leadingzeros, pos + 1)

pos += leadingzeros + 1

else:

assert codenum == 0

pos += 1

return codenum, pos

def \_getue(self):

"""Return data as unsigned exponential-Golomb code.

Raises InterpretError if bitstring is not a single exponential-Golomb code.

"""

try:

value, newpos = self.\_readue(0)

if value is None or newpos != self.len:

raise ReadError

except ReadError:

raise InterpretError("Bitstring is not a single exponential-Golomb code.")

return value

def \_setse(self, i):

"""Initialise bitstring with signed exponential-Golomb code for integer i."""

if i > 0:

u = (i \* 2) - 1

else:

u = -2 \* i

self.\_setue(u)

def \_getse(self):

"""Return data as signed exponential-Golomb code.

Raises InterpretError if bitstring is not a single exponential-Golomb code.

"""

try:

value, newpos = self.\_readse(0)

if value is None or newpos != self.len:

raise ReadError

except ReadError:

raise InterpretError("Bitstring is not a single exponential-Golomb code.")

return value

def \_readse(self, pos):

"""Return interpretation of next bits as a signed exponential-Golomb code.

Advances position to after the read code.

Raises ReadError if the end of the bitstring is encountered while

reading the code.

"""

codenum, pos = self.\_readue(pos)

m = (codenum + 1) // 2

if not codenum % 2:

return -m, pos

else:

return m, pos

def \_setuie(self, i):

"""Initialise bitstring with unsigned interleaved exponential-Golomb code for integer i.

Raises CreationError if i < 0.

"""

if i < 0:

raise CreationError("Cannot use negative initialiser for unsigned "

"interleaved exponential-Golomb.")

self.\_setbin\_unsafe('1' if i == 0 else '0' + '0'.join(bin(i + 1)[3:]) + '1')

def \_readuie(self, pos):

"""Return interpretation of next bits as unsigned interleaved exponential-Golomb code.

Raises ReadError if the end of the bitstring is encountered while

reading the code.

"""

try:

codenum = 1

while not self[pos]:

pos += 1

codenum <<= 1

codenum += self[pos]

pos += 1

pos += 1

except IndexError:

raise ReadError("Read off end of bitstring trying to read code.")

codenum -= 1

return codenum, pos

def \_getuie(self):

"""Return data as unsigned interleaved exponential-Golomb code.

Raises InterpretError if bitstring is not a single exponential-Golomb code.

"""

try:

value, newpos = self.\_readuie(0)

if value is None or newpos != self.len:

raise ReadError

except ReadError:

raise InterpretError("Bitstring is not a single interleaved exponential-Golomb code.")

return value

def \_setsie(self, i):

"""Initialise bitstring with signed interleaved exponential-Golomb code for integer i."""

if not i:

self.\_setbin\_unsafe('1')

else:

self.\_setuie(abs(i))

self.\_append(Bits([i < 0]))

def \_getsie(self):

"""Return data as signed interleaved exponential-Golomb code.

Raises InterpretError if bitstring is not a single exponential-Golomb code.

"""

try:

value, newpos = self.\_readsie(0)

if value is None or newpos != self.len:

raise ReadError

except ReadError:

raise InterpretError("Bitstring is not a single interleaved exponential-Golomb code.")

return value

def \_readsie(self, pos):

"""Return interpretation of next bits as a signed interleaved exponential-Golomb code.

Advances position to after the read code.

Raises ReadError if the end of the bitstring is encountered while

reading the code.

"""

codenum, pos = self.\_readuie(pos)

if not codenum:

return 0, pos

try:

if self[pos]:

return -codenum, pos + 1

else:

return codenum, pos + 1

except IndexError:

raise ReadError("Read off end of bitstring trying to read code.")

def \_setbool(self, value):

# We deliberately don't want to have implicit conversions to bool here.

# If we did then it would be difficult to deal with the 'False' string.

if value in (1, 'True'):

self.\_setbytes\_unsafe(bytearray(b'\x80'), 1, 0)

elif value in (0, 'False'):

self.\_setbytes\_unsafe(bytearray(b'\x00'), 1, 0)

else:

raise CreationError('Cannot initialise boolean with {0}.', value)

def \_getbool(self):

if self.length != 1:

msg = "For a bool interpretation a bitstring must be 1 bit long, not {0} bits."

raise InterpretError(msg, self.length)

return self[0]

def \_readbool(self, pos):

return self[pos], pos + 1

def \_setbin\_safe(self, binstring):

"""Reset the bitstring to the value given in binstring."""

binstring = tidy\_input\_string(binstring)

# remove any 0b if present

binstring = binstring.replace('0b', '')

self.\_setbin\_unsafe(binstring)

def \_setbin\_unsafe(self, binstring):

"""Same as \_setbin\_safe, but input isn't sanity checked. binstring mustn't start with '0b'."""

length = len(binstring)

# pad with zeros up to byte boundary if needed

boundary = ((length + 7) // 8) \* 8

padded\_binstring = binstring + '0' \* (boundary - length)\

if len(binstring) < boundary else binstring

try:

bytelist = [int(padded\_binstring[x:x + 8], 2)

for x in xrange(0, len(padded\_binstring), 8)]

except ValueError:

raise CreationError("Invalid character in bin initialiser {0}.", binstring)

self.\_setbytes\_unsafe(bytearray(bytelist), length, 0)

def \_readbin(self, length, start):

"""Read bits and interpret as a binary string."""

if not length:

return ''

# Get the byte slice containing our bit slice

startbyte, startoffset = divmod(start + self.\_offset, 8)

endbyte = (start + self.\_offset + length - 1) // 8

b = self.\_datastore.getbyteslice(startbyte, endbyte + 1)

# Convert to a string of '0' and '1's (via a hex string an and int!)

try:

c = "{:0{}b}".format(int(binascii.hexlify(b), 16), 8\*len(b))

except TypeError:

# Hack to get Python 2.6 working

c = "{0:0{1}b}".format(int(binascii.hexlify(str(b)), 16), 8\*len(b))

# Finally chop off any extra bits.

return c[startoffset:startoffset + length]

def \_getbin(self):

"""Return interpretation as a binary string."""

return self.\_readbin(self.len, 0)

def \_setoct(self, octstring):

"""Reset the bitstring to have the value given in octstring."""

octstring = tidy\_input\_string(octstring)

# remove any 0o if present

octstring = octstring.replace('0o', '')

binlist = []

for i in octstring:

try:

if not 0 <= int(i) < 8:

raise ValueError

binlist.append(OCT\_TO\_BITS[int(i)])

except ValueError:

raise CreationError("Invalid symbol '{0}' in oct initialiser.", i)

self.\_setbin\_unsafe(''.join(binlist))

def \_readoct(self, length, start):

"""Read bits and interpret as an octal string."""

if length % 3:

raise InterpretError("Cannot convert to octal unambiguously - "

"not multiple of 3 bits.")

if not length:

return ''

# Get main octal bit by converting from int.

# Strip starting 0 or 0o depending on Python version.

end = oct(self.\_readuint(length, start))[LEADING\_OCT\_CHARS:]

if end.endswith('L'):

end = end[:-1]

middle = '0' \* (length // 3 - len(end))

return middle + end

def \_getoct(self):

"""Return interpretation as an octal string."""

return self.\_readoct(self.len, 0)

def \_sethex(self, hexstring):

"""Reset the bitstring to have the value given in hexstring."""

hexstring = tidy\_input\_string(hexstring)

# remove any 0x if present

hexstring = hexstring.replace('0x', '')

length = len(hexstring)

if length % 2:

hexstring += '0'

try:

try:

data = bytearray.fromhex(hexstring)

except TypeError:

# Python 2.6 needs a unicode string (a bug). 2.7 and 3.x work fine.

data = bytearray.fromhex(unicode(hexstring))

except ValueError:

raise CreationError("Invalid symbol in hex initialiser.")

self.\_setbytes\_unsafe(data, length \* 4, 0)

def \_readhex(self, length, start):

"""Read bits and interpret as a hex string."""

if length % 4:

raise InterpretError("Cannot convert to hex unambiguously - "

"not multiple of 4 bits.")

if not length:

return ''

# This monstrosity is the only thing I could get to work for both 2.6 and 3.1.

# TODO: Is utf-8 really what we mean here?

s = str(binascii.hexlify(self.\_slice(start, start + length).tobytes()).decode('utf-8'))

# If there's one nibble too many then cut it off

return s[:-1] if (length // 4) % 2 else s

def \_gethex(self):

"""Return the hexadecimal representation as a string prefixed with '0x'.

Raises an InterpretError if the bitstring's length is not a multiple of 4.

"""

return self.\_readhex(self.len, 0)

def \_getoffset(self):

return self.\_datastore.offset

def \_getlength(self):

"""Return the length of the bitstring in bits."""

return self.\_datastore.bitlength

def \_ensureinmemory(self):

"""Ensure the data is held in memory, not in a file."""

self.\_setbytes\_unsafe(self.\_datastore.getbyteslice(0, self.\_datastore.bytelength),

self.len, self.\_offset)

@classmethod

def \_converttobitstring(cls, bs, offset=0, cache={}):

"""Convert bs to a bitstring and return it.

offset gives the suggested bit offset of first significant

bit, to optimise append etc.

"""

if isinstance(bs, Bits):

return bs

try:

return cache[(bs, offset)]

except KeyError:

if isinstance(bs, basestring):

b = cls()

try:

\_, tokens = tokenparser(bs)

except ValueError as e:

raise CreationError(\*e.args)

if tokens:

b.\_append(Bits.\_init\_with\_token(\*tokens[0]))

b.\_datastore = offsetcopy(b.\_datastore, offset)

for token in tokens[1:]:

b.\_append(Bits.\_init\_with\_token(\*token))

assert b.\_assertsanity()

assert b.len == 0 or b.\_offset == offset

if len(cache) < CACHE\_SIZE:

cache[(bs, offset)] = b

return b

except TypeError:

# Unhashable type

pass

return cls(bs)

def \_copy(self):

"""Create and return a new copy of the Bits (always in memory)."""

s\_copy = self.\_\_class\_\_()

s\_copy.\_setbytes\_unsafe(self.\_datastore.getbyteslice(0, self.\_datastore.bytelength),

self.len, self.\_offset)

return s\_copy

def \_slice(self, start, end):

"""Used internally to get a slice, without error checking."""

if end == start:

return self.\_\_class\_\_()

offset = self.\_offset

startbyte, newoffset = divmod(start + offset, 8)

endbyte = (end + offset - 1) // 8

bs = self.\_\_class\_\_()

bs.\_setbytes\_unsafe(self.\_datastore.getbyteslice(startbyte, endbyte + 1), end - start, newoffset)

return bs

def \_readtoken(self, name, pos, length):

"""Reads a token from the bitstring and returns the result."""

if length is not None and int(length) > self.length - pos:

raise ReadError("Reading off the end of the data. "

"Tried to read {0} bits when only {1} available.".format(int(length), self.length - pos))

try:

val = name\_to\_read[name](self, length, pos)

return val, pos + length

except KeyError:

if name == 'pad':

return None, pos + length

raise ValueError("Can't parse token {0}:{1}".format(name, length))

except TypeError:

# This is for the 'ue', 'se' and 'bool' tokens. They will also return the new pos.

return name\_to\_read[name](self, pos)

def \_append(self, bs):

"""Append a bitstring to the current bitstring."""

self.\_datastore.\_appendstore(bs.\_datastore)

def \_prepend(self, bs):

"""Prepend a bitstring to the current bitstring."""

self.\_datastore.\_prependstore(bs.\_datastore)

def \_reverse(self):

"""Reverse all bits in-place."""

# Reverse the contents of each byte

n = [BYTE\_REVERSAL\_DICT[b] for b in self.\_datastore.rawbytes]

# Then reverse the order of the bytes

n.reverse()

# The new offset is the number of bits that were unused at the end.

newoffset = 8 - (self.\_offset + self.len) % 8

if newoffset == 8:

newoffset = 0

self.\_setbytes\_unsafe(bytearray().join(n), self.length, newoffset)

def \_truncatestart(self, bits):

"""Truncate bits from the start of the bitstring."""

assert 0 <= bits <= self.len

if not bits:

return

if bits == self.len:

self.\_clear()

return

bytepos, offset = divmod(self.\_offset + bits, 8)

self.\_setbytes\_unsafe(self.\_datastore.getbyteslice(bytepos, self.\_datastore.bytelength), self.len - bits,

offset)

assert self.\_assertsanity()

def \_truncateend(self, bits):

"""Truncate bits from the end of the bitstring."""

assert 0 <= bits <= self.len

if not bits:

return

if bits == self.len:

self.\_clear()

return

newlength\_in\_bytes = (self.\_offset + self.len - bits + 7) // 8

self.\_setbytes\_unsafe(self.\_datastore.getbyteslice(0, newlength\_in\_bytes), self.len - bits,

self.\_offset)

assert self.\_assertsanity()

def \_insert(self, bs, pos):

"""Insert bs at pos."""

assert 0 <= pos <= self.len

if pos > self.len // 2:

# Inserting nearer end, so cut off end.

end = self.\_slice(pos, self.len)

self.\_truncateend(self.len - pos)

self.\_append(bs)

self.\_append(end)

else:

# Inserting nearer start, so cut off start.

start = self.\_slice(0, pos)

self.\_truncatestart(pos)

self.\_prepend(bs)

self.\_prepend(start)

try:

self.\_pos = pos + bs.len

except AttributeError:

pass

assert self.\_assertsanity()

def \_overwrite(self, bs, pos):

"""Overwrite with bs at pos."""

assert 0 <= pos < self.len

if bs is self:

# Just overwriting with self, so do nothing.

assert pos == 0

return

firstbytepos = (self.\_offset + pos) // 8

lastbytepos = (self.\_offset + pos + bs.len - 1) // 8

bytepos, bitoffset = divmod(self.\_offset + pos, 8)

if firstbytepos == lastbytepos:

mask = ((1 << bs.len) - 1) << (8 - bs.len - bitoffset)

self.\_datastore.setbyte(bytepos, self.\_datastore.getbyte(bytepos) & (~mask))

d = offsetcopy(bs.\_datastore, bitoffset)

self.\_datastore.setbyte(bytepos, self.\_datastore.getbyte(bytepos) | (d.getbyte(0) & mask))

else:

# Do first byte

mask = (1 << (8 - bitoffset)) - 1

self.\_datastore.setbyte(bytepos, self.\_datastore.getbyte(bytepos) & (~mask))

d = offsetcopy(bs.\_datastore, bitoffset)

self.\_datastore.setbyte(bytepos, self.\_datastore.getbyte(bytepos) | (d.getbyte(0) & mask))

# Now do all the full bytes

self.\_datastore.setbyteslice(firstbytepos + 1, lastbytepos, d.getbyteslice(1, lastbytepos - firstbytepos))

# and finally the last byte

bitsleft = (self.\_offset + pos + bs.len) % 8

if not bitsleft:

bitsleft = 8

mask = (1 << (8 - bitsleft)) - 1

self.\_datastore.setbyte(lastbytepos, self.\_datastore.getbyte(lastbytepos) & mask)

self.\_datastore.setbyte(lastbytepos,

self.\_datastore.getbyte(lastbytepos) | (d.getbyte(d.bytelength - 1) & ~mask))

assert self.\_assertsanity()

def \_delete(self, bits, pos):

"""Delete bits at pos."""

assert 0 <= pos <= self.len

assert pos + bits <= self.len

if not pos:

# Cutting bits off at the start.

self.\_truncatestart(bits)

return

if pos + bits == self.len:

# Cutting bits off at the end.

self.\_truncateend(bits)

return

if pos > self.len - pos - bits:

# More bits before cut point than after it, so do bit shifting

# on the final bits.

end = self.\_slice(pos + bits, self.len)

assert self.len - pos > 0

self.\_truncateend(self.len - pos)

self.\_append(end)

return

# More bits after the cut point than before it.

start = self.\_slice(0, pos)

self.\_truncatestart(pos + bits)

self.\_prepend(start)

return

def \_reversebytes(self, start, end):

"""Reverse bytes in-place."""

# Make the start occur on a byte boundary

# TODO: We could be cleverer here to avoid changing the offset.

newoffset = 8 - (start % 8)

if newoffset == 8:

newoffset = 0

self.\_datastore = offsetcopy(self.\_datastore, newoffset)

# Now just reverse the byte data

toreverse = bytearray(self.\_datastore.getbyteslice((newoffset + start) // 8, (newoffset + end) // 8))

toreverse.reverse()

self.\_datastore.setbyteslice((newoffset + start) // 8, (newoffset + end) // 8, toreverse)

def \_set(self, pos):

"""Set bit at pos to 1."""

assert 0 <= pos < self.len

self.\_datastore.setbit(pos)

def \_unset(self, pos):

"""Set bit at pos to 0."""

assert 0 <= pos < self.len

self.\_datastore.unsetbit(pos)

def \_invert(self, pos):

"""Flip bit at pos 1<->0."""

assert 0 <= pos < self.len

self.\_datastore.invertbit(pos)

def \_invert\_all(self):

"""Invert every bit."""

set = self.\_datastore.setbyte

get = self.\_datastore.getbyte

for p in xrange(self.\_datastore.byteoffset, self.\_datastore.byteoffset + self.\_datastore.bytelength):

set(p, 256 + ~get(p))

def \_ilshift(self, n):

"""Shift bits by n to the left in place. Return self."""

assert 0 < n <= self.len

self.\_append(Bits(n))

self.\_truncatestart(n)

return self

def \_irshift(self, n):

"""Shift bits by n to the right in place. Return self."""

assert 0 < n <= self.len

self.\_prepend(Bits(n))

self.\_truncateend(n)

return self

def \_imul(self, n):

"""Concatenate n copies of self in place. Return self."""

assert n >= 0

if not n:

self.\_clear()

return self

m = 1

old\_len = self.len

while m \* 2 < n:

self.\_append(self)

m \*= 2

self.\_append(self[0:(n - m) \* old\_len])

return self

def \_inplace\_logical\_helper(self, bs, f):

"""Helper function containing most of the \_\_ior\_\_, \_\_iand\_\_, \_\_ixor\_\_ code."""

# Give the two bitstrings the same offset (modulo 8)

self\_byteoffset, self\_bitoffset = divmod(self.\_offset, 8)

bs\_byteoffset, bs\_bitoffset = divmod(bs.\_offset, 8)

if bs\_bitoffset != self\_bitoffset:

if not self\_bitoffset:

bs.\_datastore = offsetcopy(bs.\_datastore, 0)

else:

self.\_datastore = offsetcopy(self.\_datastore, bs\_bitoffset)

a = self.\_datastore.rawbytes

b = bs.\_datastore.rawbytes

for i in xrange(len(a)):

a[i] = f(a[i + self\_byteoffset], b[i + bs\_byteoffset])

return self

def \_ior(self, bs):

return self.\_inplace\_logical\_helper(bs, operator.ior)

def \_iand(self, bs):

return self.\_inplace\_logical\_helper(bs, operator.iand)

def \_ixor(self, bs):

return self.\_inplace\_logical\_helper(bs, operator.xor)

def \_readbits(self, length, start):

"""Read some bits from the bitstring and return newly constructed bitstring."""

return self.\_slice(start, start + length)

def \_validate\_slice(self, start, end):

"""Validate start and end and return them as positive bit positions."""

if start is None:

start = 0

elif start < 0:

start += self.len

if end is None:

end = self.len

elif end < 0:

end += self.len

if not 0 <= end <= self.len:

raise ValueError("end is not a valid position in the bitstring.")

if not 0 <= start <= self.len:

raise ValueError("start is not a valid position in the bitstring.")

if end < start:

raise ValueError("end must not be less than start.")

return start, end

def unpack(self, fmt, \*\*kwargs):

"""Interpret the whole bitstring using fmt and return list.

fmt -- A single string or a list of strings with comma separated tokens

describing how to interpret the bits in the bitstring. Items

can also be integers, for reading new bitstring of the given length.

kwargs -- A dictionary or keyword-value pairs - the keywords used in the

format string will be replaced with their given value.

Raises ValueError if the format is not understood. If not enough bits

are available then all bits to the end of the bitstring will be used.

See the docstring for 'read' for token examples.

"""

return self.\_readlist(fmt, 0, \*\*kwargs)[0]

def \_readlist(self, fmt, pos, \*\*kwargs):

tokens = []

stretchy\_token = None

if isinstance(fmt, basestring):

fmt = [fmt]

# Not very optimal this, but replace integers with 'bits' tokens

# TODO: optimise

for i, f in enumerate(fmt):

if isinstance(f, numbers.Integral):

fmt[i] = "bits:{0}".format(f)

for f\_item in fmt:

stretchy, tkns = tokenparser(f\_item, tuple(sorted(kwargs.keys())))

if stretchy:

if stretchy\_token:

raise Error("It's not possible to have more than one 'filler' token.")

stretchy\_token = stretchy

tokens.extend(tkns)

if not stretchy\_token:

lst = []

for name, length, \_ in tokens:

if length in kwargs:

length = kwargs[length]

if name == 'bytes':

length \*= 8

if name in kwargs and length is None:

# Using default 'uint' - the name is really the length.

value, pos = self.\_readtoken('uint', pos, kwargs[name])

lst.append(value)

continue

value, pos = self.\_readtoken(name, pos, length)

if value is not None: # Don't append pad tokens

lst.append(value)

return lst, pos

stretchy\_token = False

bits\_after\_stretchy\_token = 0

for token in tokens:

name, length, \_ = token

if length in kwargs:

length = kwargs[length]

if name == 'bytes':

length \*= 8

if name in kwargs and length is None:

# Default 'uint'.

length = kwargs[name]

if stretchy\_token:

if name in ('se', 'ue', 'sie', 'uie'):

raise Error("It's not possible to parse a variable"

"length token after a 'filler' token.")

else:

if length is None:

raise Error("It's not possible to have more than "

"one 'filler' token.")

bits\_after\_stretchy\_token += length

if length is None and name not in ('se', 'ue', 'sie', 'uie'):

assert not stretchy\_token

stretchy\_token = token

bits\_left = self.len - pos

return\_values = []

for token in tokens:

name, length, \_ = token

if token is stretchy\_token:

# Set length to the remaining bits

length = max(bits\_left - bits\_after\_stretchy\_token, 0)

if length in kwargs:

length = kwargs[length]

if name == 'bytes':

length \*= 8

if name in kwargs and length is None:

# Default 'uint'

length = kwargs[name]

if length is not None:

bits\_left -= length

value, pos = self.\_readtoken(name, pos, length)

if value is not None:

return\_values.append(value)

return return\_values, pos

def \_findbytes(self, bytes\_, start, end, bytealigned):

"""Quicker version of find when everything's whole byte

and byte aligned.

"""

assert self.\_datastore.offset == 0

assert bytealigned is True

# Extract data bytes from bitstring to be found.

bytepos = (start + 7) // 8

found = False

p = bytepos

finalpos = end // 8

increment = max(1024, len(bytes\_) \* 10)

buffersize = increment + len(bytes\_)

while p < finalpos:

# Read in file or from memory in overlapping chunks and search the chunks.

buf = bytearray(self.\_datastore.getbyteslice(p, min(p + buffersize, finalpos)))

pos = buf.find(bytes\_)

if pos != -1:

found = True

p += pos

break

p += increment

if not found:

return ()

return (p \* 8,)

def \_findregex(self, reg\_ex, start, end, bytealigned):

"""Find first occurrence of a compiled regular expression.

Note that this doesn't support arbitrary regexes, in particular they

must match a known length.

"""

p = start

length = len(reg\_ex.pattern)

# We grab overlapping chunks of the binary representation and

# do an ordinary string search within that.

increment = max(4096, length \* 10)

buffersize = increment + length

while p < end:

buf = self.\_readbin(min(buffersize, end - p), p)

# Test using regular expressions...

m = reg\_ex.search(buf)

if m:

pos = m.start()

# pos = buf.find(targetbin)

# if pos != -1:

# if bytealigned then we only accept byte aligned positions.

if not bytealigned or (p + pos) % 8 == 0:

return (p + pos,)

if bytealigned:

# Advance to just beyond the non-byte-aligned match and try again...

p += pos + 1

continue

p += increment

# Not found, return empty tuple

return ()

def find(self, bs, start=None, end=None, bytealigned=None):

"""Find first occurrence of substring bs.

Returns a single item tuple with the bit position if found, or an

empty tuple if not found. The bit position (pos property) will

also be set to the start of the substring if it is found.

bs -- The bitstring to find.

start -- The bit position to start the search. Defaults to 0.

end -- The bit position one past the last bit to search.

Defaults to self.len.

bytealigned -- If True the bitstring will only be

found on byte boundaries.

Raises ValueError if bs is empty, if start < 0, if end > self.len or

if end < start.

>>> BitArray('0xc3e').find('0b1111')

(6,)

"""

bs = Bits(bs)

if not bs.len:

raise ValueError("Cannot find an empty bitstring.")

start, end = self.\_validate\_slice(start, end)

if bytealigned is None:

bytealigned = globals()['bytealigned']

if bytealigned and not bs.len % 8 and not self.\_datastore.offset:

p = self.\_findbytes(bs.bytes, start, end, bytealigned)

else:

p = self.\_findregex(re.compile(bs.\_getbin()), start, end, bytealigned)

# If called from a class that has a pos, set it

try:

self.\_pos = p[0]

except (AttributeError, IndexError):

pass

return p

def findall(self, bs, start=None, end=None, count=None, bytealigned=None):

"""Find all occurrences of bs. Return generator of bit positions.

bs -- The bitstring to find.

start -- The bit position to start the search. Defaults to 0.

end -- The bit position one past the last bit to search.

Defaults to self.len.

count -- The maximum number of occurrences to find.

bytealigned -- If True the bitstring will only be found on

byte boundaries.

Raises ValueError if bs is empty, if start < 0, if end > self.len or

if end < start.

Note that all occurrences of bs are found, even if they overlap.

"""

if count is not None and count < 0:

raise ValueError("In findall, count must be >= 0.")

bs = Bits(bs)

start, end = self.\_validate\_slice(start, end)

if bytealigned is None:

bytealigned = globals()['bytealigned']

c = 0

if bytealigned and not bs.len % 8 and not self.\_datastore.offset:

# Use the quick find method

f = self.\_findbytes

x = bs.\_getbytes()

else:

f = self.\_findregex

x = re.compile(bs.\_getbin())

while True:

p = f(x, start, end, bytealigned)

if not p:

break

if count is not None and c >= count:

return

c += 1

try:

self.\_pos = p[0]

except AttributeError:

pass

yield p[0]

if bytealigned:

start = p[0] + 8

else:

start = p[0] + 1

if start >= end:

break

return

def rfind(self, bs, start=None, end=None, bytealigned=None):

"""Find final occurrence of substring bs.

Returns a single item tuple with the bit position if found, or an

empty tuple if not found. The bit position (pos property) will

also be set to the start of the substring if it is found.

bs -- The bitstring to find.

start -- The bit position to end the reverse search. Defaults to 0.

end -- The bit position one past the first bit to reverse search.

Defaults to self.len.

bytealigned -- If True the bitstring will only be found on byte

boundaries.

Raises ValueError if bs is empty, if start < 0, if end > self.len or

if end < start.

"""

bs = Bits(bs)

start, end = self.\_validate\_slice(start, end)

if bytealigned is None:

bytealigned = globals()['bytealigned']

if not bs.len:

raise ValueError("Cannot find an empty bitstring.")

# Search chunks starting near the end and then moving back

# until we find bs.

increment = max(8192, bs.len \* 80)

buffersize = min(increment + bs.len, end - start)

pos = max(start, end - buffersize)

while True:

found = list(self.findall(bs, start=pos, end=pos + buffersize,

bytealigned=bytealigned))

if not found:

if pos == start:

return ()

pos = max(start, pos - increment)

continue

return (found[-1],)

def cut(self, bits, start=None, end=None, count=None):

"""Return bitstring generator by cutting into bits sized chunks.

bits -- The size in bits of the bitstring chunks to generate.

start -- The bit position to start the first cut. Defaults to 0.

end -- The bit position one past the last bit to use in the cut.

Defaults to self.len.

count -- If specified then at most count items are generated.

Default is to cut as many times as possible.

"""

start, end = self.\_validate\_slice(start, end)

if count is not None and count < 0:

raise ValueError("Cannot cut - count must be >= 0.")

if bits <= 0:

raise ValueError("Cannot cut - bits must be >= 0.")

c = 0

while count is None or c < count:

c += 1

nextchunk = self.\_slice(start, min(start + bits, end))

if nextchunk.len != bits:

return

assert nextchunk.\_assertsanity()

yield nextchunk

start += bits

return

def split(self, delimiter, start=None, end=None, count=None,

bytealigned=None):

"""Return bitstring generator by splittling using a delimiter.

The first item returned is the initial bitstring before the delimiter,

which may be an empty bitstring.

delimiter -- The bitstring used as the divider.

start -- The bit position to start the split. Defaults to 0.

end -- The bit position one past the last bit to use in the split.

Defaults to self.len.

count -- If specified then at most count items are generated.

Default is to split as many times as possible.

bytealigned -- If True splits will only occur on byte boundaries.

Raises ValueError if the delimiter is empty.

"""

delimiter = Bits(delimiter)

if not delimiter.len:

raise ValueError("split delimiter cannot be empty.")

start, end = self.\_validate\_slice(start, end)

if bytealigned is None:

bytealigned = globals()['bytealigned']

if count is not None and count < 0:

raise ValueError("Cannot split - count must be >= 0.")

if count == 0:

return

if bytealigned and not delimiter.len % 8 and not self.\_datastore.offset:

# Use the quick find method

f = self.\_findbytes

x = delimiter.\_getbytes()

else:

f = self.\_findregex

x = re.compile(delimiter.\_getbin())

found = f(x, start, end, bytealigned)

if not found:

# Initial bits are the whole bitstring being searched

yield self.\_slice(start, end)

return

# yield the bytes before the first occurrence of the delimiter, even if empty

yield self.\_slice(start, found[0])

startpos = pos = found[0]

c = 1

while count is None or c < count:

pos += delimiter.len

found = f(x, pos, end, bytealigned)

if not found:

# No more occurrences, so return the rest of the bitstring

yield self.\_slice(startpos, end)

return

c += 1

yield self.\_slice(startpos, found[0])

startpos = pos = found[0]

# Have generated count bitstrings, so time to quit.

return

def join(self, sequence):

"""Return concatenation of bitstrings joined by self.

sequence -- A sequence of bitstrings.

"""

s = self.\_\_class\_\_()

i = iter(sequence)

print 'join...'

try:

nn=next(i)

print nn

dd=Bits(nn)

print dd

s.\_append(dd)

while True:

n = next(i)

s.\_append(self)

s.\_append(Bits(n))

except StopIteration:

pass

return s

def tobytes(self):

"""Return the bitstring as bytes, padding with zero bits if needed.

Up to seven zero bits will be added at the end to byte align.

"""

d = offsetcopy(self.\_datastore, 0).rawbytes

# Need to ensure that unused bits at end are set to zero

unusedbits = 8 - self.len % 8

if unusedbits != 8:

d[-1] &= (0xff << unusedbits)

return bytes(d)

def tofile(self, f):

"""Write the bitstring to a file object, padding with zero bits if needed.

Up to seven zero bits will be added at the end to byte align.

"""

# If the bitstring is file based then we don't want to read it all

# in to memory.

chunksize = 1024 \* 1024 # 1 MB chunks

if not self.\_offset:

a = 0

bytelen = self.\_datastore.bytelength

p = self.\_datastore.getbyteslice(a, min(a + chunksize, bytelen - 1))

while len(p) == chunksize:

f.write(p)

a += chunksize

p = self.\_datastore.getbyteslice(a, min(a + chunksize, bytelen - 1))

f.write(p)

# Now the final byte, ensuring that unused bits at end are set to 0.

bits\_in\_final\_byte = self.len % 8

if not bits\_in\_final\_byte:

bits\_in\_final\_byte = 8

f.write(self[-bits\_in\_final\_byte:].tobytes())

else:

# Really quite inefficient...

a = 0

b = a + chunksize \* 8

while b <= self.len:

f.write(self.\_slice(a, b).\_getbytes())

a += chunksize \* 8

b += chunksize \* 8

if a != self.len:

f.write(self.\_slice(a, self.len).tobytes())

def startswith(self, prefix, start=None, end=None):

"""Return whether the current bitstring starts with prefix.

prefix -- The bitstring to search for.

start -- The bit position to start from. Defaults to 0.

end -- The bit position to end at. Defaults to self.len.

"""

prefix = Bits(prefix)

start, end = self.\_validate\_slice(start, end)

if end < start + prefix.len:

return False

end = start + prefix.len

return self.\_slice(start, end) == prefix

def endswith(self, suffix, start=None, end=None):

"""Return whether the current bitstring ends with suffix.

suffix -- The bitstring to search for.

start -- The bit position to start from. Defaults to 0.

end -- The bit position to end at. Defaults to self.len.

"""

suffix = Bits(suffix)

start, end = self.\_validate\_slice(start, end)

if start + suffix.len > end:

return False

start = end - suffix.len

return self.\_slice(start, end) == suffix

def all(self, value, pos=None):

"""Return True if one or many bits are all set to value.

value -- If value is True then checks for bits set to 1, otherwise

checks for bits set to 0.

pos -- An iterable of bit positions. Negative numbers are treated in

the same way as slice indices. Defaults to the whole bitstring.

"""

value = bool(value)

length = self.len

if pos is None:

pos = xrange(self.len)

for p in pos:

if p < 0:

p += length

if not 0 <= p < length:

raise IndexError("Bit position {0} out of range.".format(p))

if not self.\_datastore.getbit(p) is value:

return False

return True

def any(self, value, pos=None):

"""Return True if any of one or many bits are set to value.

value -- If value is True then checks for bits set to 1, otherwise

checks for bits set to 0.

pos -- An iterable of bit positions. Negative numbers are treated in

the same way as slice indices. Defaults to the whole bitstring.

"""

value = bool(value)

length = self.len

if pos is None:

pos = xrange(self.len)

for p in pos:

if p < 0:

p += length

if not 0 <= p < length:

raise IndexError("Bit position {0} out of range.".format(p))

if self.\_datastore.getbit(p) is value:

return True

return False

def count(self, value):

"""Return count of total number of either zero or one bits.

value -- If True then bits set to 1 are counted, otherwise bits set

to 0 are counted.

>>> Bits('0xef').count(1)

7

"""

if not self.len:

return 0

# count the number of 1s (from which it's easy to work out the 0s).

# Don't count the final byte yet.

count = sum(BIT\_COUNT[self.\_datastore.getbyte(i)] for i in xrange(self.\_datastore.bytelength - 1))

# adjust for bits at start that aren't part of the bitstring

if self.\_offset:

count -= BIT\_COUNT[self.\_datastore.getbyte(0) >> (8 - self.\_offset)]

# and count the last 1 - 8 bits at the end.

endbits = self.\_datastore.bytelength \* 8 - (self.\_offset + self.len)

count += BIT\_COUNT[self.\_datastore.getbyte(self.\_datastore.bytelength - 1) >> endbits]

return count if value else self.len - count

# Create native-endian functions as aliases depending on the byteorder

if byteorder == 'little':

\_setfloatne = \_setfloatle

\_readfloatne = \_readfloatle

\_getfloatne = \_getfloatle

\_setuintne = \_setuintle

\_readuintne = \_readuintle

\_getuintne = \_getuintle

\_setintne = \_setintle

\_readintne = \_readintle

\_getintne = \_getintle

else:

\_setfloatne = \_setfloat

\_readfloatne = \_readfloat

\_getfloatne = \_getfloat

\_setuintne = \_setuintbe

\_readuintne = \_readuintbe

\_getuintne = \_getuintbe

\_setintne = \_setintbe

\_readintne = \_readintbe

\_getintne = \_getintbe

\_offset = property(\_getoffset)

len = property(\_getlength,

doc="""The length of the bitstring in bits. Read only.

""")

length = property(\_getlength,

doc="""The length of the bitstring in bits. Read only.

""")

bool = property(\_getbool,

doc="""The bitstring as a bool (True or False). Read only.

""")

hex = property(\_gethex,

doc="""The bitstring as a hexadecimal string. Read only.

""")

bin = property(\_getbin,

doc="""The bitstring as a binary string. Read only.

""")

oct = property(\_getoct,

doc="""The bitstring as an octal string. Read only.

""")

bytes = property(\_getbytes,

doc="""The bitstring as a bytes object. Read only.

""")

int = property(\_getint,

doc="""The bitstring as a two's complement signed int. Read only.

""")

uint = property(\_getuint,

doc="""The bitstring as a two's complement unsigned int. Read only.

""")

float = property(\_getfloat,

doc="""The bitstring as a floating point number. Read only.

""")

intbe = property(\_getintbe,

doc="""The bitstring as a two's complement big-endian signed int. Read only.

""")

uintbe = property(\_getuintbe,

doc="""The bitstring as a two's complement big-endian unsigned int. Read only.

""")

floatbe = property(\_getfloat,

doc="""The bitstring as a big-endian floating point number. Read only.

""")

intle = property(\_getintle,

doc="""The bitstring as a two's complement little-endian signed int. Read only.

""")

uintle = property(\_getuintle,

doc="""The bitstring as a two's complement little-endian unsigned int. Read only.

""")

floatle = property(\_getfloatle,

doc="""The bitstring as a little-endian floating point number. Read only.

""")

intne = property(\_getintne,

doc="""The bitstring as a two's complement native-endian signed int. Read only.

""")

uintne = property(\_getuintne,

doc="""The bitstring as a two's complement native-endian unsigned int. Read only.

""")

floatne = property(\_getfloatne,

doc="""The bitstring as a native-endian floating point number. Read only.

""")

ue = property(\_getue,

doc="""The bitstring as an unsigned exponential-Golomb code. Read only.

""")

se = property(\_getse,

doc="""The bitstring as a signed exponential-Golomb code. Read only.

""")

uie = property(\_getuie,

doc="""The bitstring as an unsigned interleaved exponential-Golomb code. Read only.

""")

sie = property(\_getsie,

doc="""The bitstring as a signed interleaved exponential-Golomb code. Read only.

""")

# Dictionary that maps token names to the function that reads them.

name\_to\_read = {'uint': Bits.\_readuint,

'uintle': Bits.\_readuintle,

'uintbe': Bits.\_readuintbe,

'uintne': Bits.\_readuintne,

'int': Bits.\_readint,

'intle': Bits.\_readintle,

'intbe': Bits.\_readintbe,

'intne': Bits.\_readintne,

'float': Bits.\_readfloat,

'floatbe': Bits.\_readfloat, # floatbe is a synonym for float

'floatle': Bits.\_readfloatle,

'floatne': Bits.\_readfloatne,

'hex': Bits.\_readhex,

'oct': Bits.\_readoct,

'bin': Bits.\_readbin,

'bits': Bits.\_readbits,

'bytes': Bits.\_readbytes,

'ue': Bits.\_readue,

'se': Bits.\_readse,

'uie': Bits.\_readuie,

'sie': Bits.\_readsie,

'bool': Bits.\_readbool,

}

# Dictionaries for mapping init keywords with init functions.

init\_with\_length\_and\_offset = {'bytes': Bits.\_setbytes\_safe,

'filename': Bits.\_setfile,

}

init\_with\_length\_only = {'uint': Bits.\_setuint,

'int': Bits.\_setint,

'float': Bits.\_setfloat,

'uintbe': Bits.\_setuintbe,

'intbe': Bits.\_setintbe,

'floatbe': Bits.\_setfloat,

'uintle': Bits.\_setuintle,

'intle': Bits.\_setintle,

'floatle': Bits.\_setfloatle,

'uintne': Bits.\_setuintne,

'intne': Bits.\_setintne,

'floatne': Bits.\_setfloatne,

}

init\_without\_length\_or\_offset = {'bin': Bits.\_setbin\_safe,

'hex': Bits.\_sethex,

'oct': Bits.\_setoct,

'ue': Bits.\_setue,

'se': Bits.\_setse,

'uie': Bits.\_setuie,

'sie': Bits.\_setsie,

'bool': Bits.\_setbool,

}

class BitArray(Bits):

"""A container holding a mutable sequence of bits.

Subclass of the immutable Bits class. Inherits all of its

methods (except \_\_hash\_\_) and adds mutating methods.

Mutating methods:

append() -- Append a bitstring.

byteswap() -- Change byte endianness in-place.

insert() -- Insert a bitstring.

invert() -- Flip bit(s) between one and zero.

overwrite() -- Overwrite a section with a new bitstring.

prepend() -- Prepend a bitstring.

replace() -- Replace occurrences of one bitstring with another.

reverse() -- Reverse bits in-place.

rol() -- Rotate bits to the left.

ror() -- Rotate bits to the right.

set() -- Set bit(s) to 1 or 0.

Methods inherited from Bits:

all() -- Check if all specified bits are set to 1 or 0.

any() -- Check if any of specified bits are set to 1 or 0.

count() -- Count the number of bits set to 1 or 0.

cut() -- Create generator of constant sized chunks.

endswith() -- Return whether the bitstring ends with a sub-string.

find() -- Find a sub-bitstring in the current bitstring.

findall() -- Find all occurrences of a sub-bitstring in the current bitstring.

join() -- Join bitstrings together using current bitstring.

rfind() -- Seek backwards to find a sub-bitstring.

split() -- Create generator of chunks split by a delimiter.

startswith() -- Return whether the bitstring starts with a sub-bitstring.

tobytes() -- Return bitstring as bytes, padding if needed.

tofile() -- Write bitstring to file, padding if needed.

unpack() -- Interpret bits using format string.

Special methods:

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

in addition to the inherited [], ==, !=, +, \*, ~, <<, >>, &, | and ^.

Properties:

bin -- The bitstring as a binary string.

bool -- For single bit bitstrings, interpret as True or False.

bytepos -- The current byte position in the bitstring.

bytes -- The bitstring as a bytes object.

float -- Interpret as a floating point number.

floatbe -- Interpret as a big-endian floating point number.

floatle -- Interpret as a little-endian floating point number.

floatne -- Interpret as a native-endian floating point number.

hex -- The bitstring as a hexadecimal string.

int -- Interpret as a two's complement signed integer.

intbe -- Interpret as a big-endian signed integer.

intle -- Interpret as a little-endian signed integer.

intne -- Interpret as a native-endian signed integer.

len -- Length of the bitstring in bits.

oct -- The bitstring as an octal string.

pos -- The current bit position in the bitstring.

se -- Interpret as a signed exponential-Golomb code.

ue -- Interpret as an unsigned exponential-Golomb code.

sie -- Interpret as a signed interleaved exponential-Golomb code.

uie -- Interpret as an unsigned interleaved exponential-Golomb code.

uint -- Interpret as a two's complement unsigned integer.

uintbe -- Interpret as a big-endian unsigned integer.

uintle -- Interpret as a little-endian unsigned integer.

uintne -- Interpret as a native-endian unsigned integer.

"""

\_\_slots\_\_ = ()

# As BitArray objects are mutable, we shouldn't allow them to be hashed.

\_\_hash\_\_ = None

def \_\_init\_\_(self, auto=None, length=None, offset=None, \*\*kwargs):

"""Either specify an 'auto' initialiser:

auto -- a string of comma separated tokens, an integer, a file object,

a bytearray, a boolean iterable or another bitstring.

Or initialise via \*\*kwargs with one (and only one) of:

bytes -- raw data as a string, for example read from a binary file.

bin -- binary string representation, e.g. '0b001010'.

hex -- hexadecimal string representation, e.g. '0x2ef'

oct -- octal string representation, e.g. '0o777'.

uint -- an unsigned integer.

int -- a signed integer.

float -- a floating point number.

uintbe -- an unsigned big-endian whole byte integer.

intbe -- a signed big-endian whole byte integer.

floatbe - a big-endian floating point number.

uintle -- an unsigned little-endian whole byte integer.

intle -- a signed little-endian whole byte integer.

floatle -- a little-endian floating point number.

uintne -- an unsigned native-endian whole byte integer.

intne -- a signed native-endian whole byte integer.

floatne -- a native-endian floating point number.

se -- a signed exponential-Golomb code.

ue -- an unsigned exponential-Golomb code.

sie -- a signed interleaved exponential-Golomb code.

uie -- an unsigned interleaved exponential-Golomb code.

bool -- a boolean (True or False).

filename -- a file which will be opened in binary read-only mode.

Other keyword arguments:

length -- length of the bitstring in bits, if needed and appropriate.

It must be supplied for all integer and float initialisers.

offset -- bit offset to the data. These offset bits are

ignored and this is intended for use when

initialising using 'bytes' or 'filename'.

"""

# For mutable BitArrays we always read in files to memory:

if not isinstance(self.\_datastore, ByteStore):

self.\_ensureinmemory()

def \_\_new\_\_(cls, auto=None, length=None, offset=None, \*\*kwargs):

x = super(BitArray, cls).\_\_new\_\_(cls)

y = Bits.\_\_new\_\_(BitArray, auto, length, offset, \*\*kwargs)

x.\_datastore = y.\_datastore

return x

def \_\_iadd\_\_(self, bs):

"""Append bs to current bitstring. Return self.

bs -- the bitstring to append.

"""

self.append(bs)

return self

def \_\_copy\_\_(self):

"""Return a new copy of the BitArray."""

s\_copy = BitArray()

if not isinstance(self.\_datastore, ByteStore):

# Let them both point to the same (invariant) array.

# If either gets modified then at that point they'll be read into memory.

s\_copy.\_datastore = self.\_datastore

else:

s\_copy.\_datastore = copy.copy(self.\_datastore)

return s\_copy

def \_\_setitem\_\_(self, key, value):

"""Set item or range to new value.

Indices are in units of the step parameter (default 1 bit).

Stepping is used to specify the number of bits in each item.

If the length of the bitstring is changed then pos will be moved

to after the inserted section, otherwise it will remain unchanged.

>>> s = BitArray('0xff')

>>> s[0:1:4] = '0xe'

>>> print s

'0xef'

>>> s[4:4] = '0x00'

>>> print s

'0xe00f'

"""

try:

# A slice

start, step = 0, 1

if key.step is not None:

step = key.step

except AttributeError:

# single element

if key < 0:

key += self.len

if not 0 <= key < self.len:

raise IndexError("Slice index out of range.")

if isinstance(value, numbers.Integral):

if not value:

self.\_unset(key)

return

if value in (1, -1):

self.\_set(key)

return

raise ValueError("Cannot set a single bit with integer {0}.".format(value))

value = Bits(value)

if value.len == 1:

# TODO: this can't be optimal

if value[0]:

self.\_set(key)

else:

self.\_unset(key)

else:

self.\_delete(1, key)

self.\_insert(value, key)

return

else:

if step != 1:

# convert to binary string and use string slicing

# TODO: Horribly inefficent

temp = list(self.\_getbin())

v = list(Bits(value).\_getbin())

temp.\_\_setitem\_\_(key, v)

self.\_setbin\_unsafe(''.join(temp))

return

# If value is an integer then we want to set the slice to that

# value rather than initialise a new bitstring of that length.

if not isinstance(value, numbers.Integral):

try:

# TODO: Better way than calling constructor here?

value = Bits(value)

except TypeError:

raise TypeError("Bitstring, integer or string expected. "

"Got {0}.".format(type(value)))

if key.start is not None:

start = key.start

if key.start < 0:

start += self.len

if start < 0:

start = 0

stop = self.len

if key.stop is not None:

stop = key.stop

if key.stop < 0:

stop += self.len

if start > stop:

# The standard behaviour for lists is to just insert at the

# start position if stop < start and step == 1.

stop = start

if isinstance(value, numbers.Integral):

if value >= 0:

value = self.\_\_class\_\_(uint=value, length=stop - start)

else:

value = self.\_\_class\_\_(int=value, length=stop - start)

stop = min(stop, self.len)

start = max(start, 0)

start = min(start, stop)

if (stop - start) == value.len:

if not value.len:

return

if step >= 0:

self.\_overwrite(value, start)

else:

self.\_overwrite(value.\_\_getitem\_\_(slice(None, None, 1)), start)

else:

# TODO: A delete then insert is wasteful - it could do unneeded shifts.

# Could be either overwrite + insert or overwrite + delete.

self.\_delete(stop - start, start)

if step >= 0:

self.\_insert(value, start)

else:

self.\_insert(value.\_\_getitem\_\_(slice(None, None, 1)), start)

# pos is now after the inserted piece.

return

def \_\_delitem\_\_(self, key):

"""Delete item or range.

Indices are in units of the step parameter (default 1 bit).

Stepping is used to specify the number of bits in each item.

>>> a = BitArray('0x001122')

>>> del a[1:2:8]

>>> print a

0x0022

"""

try:

# A slice

start = 0

step = key.step if key.step is not None else 1

except AttributeError:

# single element

if key < 0:

key += self.len

if not 0 <= key < self.len:

raise IndexError("Slice index out of range.")

self.\_delete(1, key)

return

else:

if step != 1:

# convert to binary string and use string slicing

# TODO: Horribly inefficent

temp = list(self.\_getbin())

temp.\_\_delitem\_\_(key)

self.\_setbin\_unsafe(''.join(temp))

return

stop = key.stop

if key.start is not None:

start = key.start

if key.start < 0 and stop is None:

start += self.len

if start < 0:

start = 0

if stop is None:

stop = self.len

if start > stop:

return

stop = min(stop, self.len)

start = max(start, 0)

start = min(start, stop)

self.\_delete(stop - start, start)

return

def \_\_ilshift\_\_(self, n):

"""Shift bits by n to the left in place. Return self.

n -- the number of bits to shift. Must be >= 0.

"""

if n < 0:

raise ValueError("Cannot shift by a negative amount.")

if not self.len:

raise ValueError("Cannot shift an empty bitstring.")

if not n:

return self

n = min(n, self.len)

return self.\_ilshift(n)

def \_\_irshift\_\_(self, n):

"""Shift bits by n to the right in place. Return self.

n -- the number of bits to shift. Must be >= 0.

"""

if n < 0:

raise ValueError("Cannot shift by a negative amount.")

if not self.len:

raise ValueError("Cannot shift an empty bitstring.")

if not n:

return self

n = min(n, self.len)

return self.\_irshift(n)

def \_\_imul\_\_(self, n):

"""Concatenate n copies of self in place. Return self.

Called for expressions of the form 'a \*= 3'.

n -- The number of concatenations. Must be >= 0.

"""

if n < 0:

raise ValueError("Cannot multiply by a negative integer.")

return self.\_imul(n)

def \_\_ior\_\_(self, bs):

bs = Bits(bs)

if self.len != bs.len:

raise ValueError("Bitstrings must have the same length "

"for |= operator.")

return self.\_ior(bs)

def \_\_iand\_\_(self, bs):

bs = Bits(bs)

if self.len != bs.len:

raise ValueError("Bitstrings must have the same length "

"for &= operator.")

return self.\_iand(bs)

def \_\_ixor\_\_(self, bs):

bs = Bits(bs)

if self.len != bs.len:

raise ValueError("Bitstrings must have the same length "

"for ^= operator.")

return self.\_ixor(bs)

def replace(self, old, new, start=None, end=None, count=None,

bytealigned=None):

"""Replace all occurrences of old with new in place.

Returns number of replacements made.

old -- The bitstring to replace.

new -- The replacement bitstring.

start -- Any occurrences that start before this will not be replaced.

Defaults to 0.

end -- Any occurrences that finish after this will not be replaced.

Defaults to self.len.

count -- The maximum number of replacements to make. Defaults to

replace all occurrences.

bytealigned -- If True replacements will only be made on byte

boundaries.

Raises ValueError if old is empty or if start or end are

out of range.

"""

old = Bits(old)

new = Bits(new)

if not old.len:

raise ValueError("Empty bitstring cannot be replaced.")

start, end = self.\_validate\_slice(start, end)

if bytealigned is None:

bytealigned = globals()['bytealigned']

# Adjust count for use in split()

if count is not None:

count += 1

sections = self.split(old, start, end, count, bytealigned)

lengths = [s.len for s in sections]

if len(lengths) == 1:

# Didn't find anything to replace.

return 0 # no replacements done

if new is self:

# Prevent self assignment woes

new = copy.copy(self)

positions = [lengths[0] + start]

for l in lengths[1:-1]:

# Next position is the previous one plus the length of the next section.

positions.append(positions[-1] + l)

# We have all the positions that need replacements. We do them

# in reverse order so that they won't move around as we replace.

positions.reverse()

try:

# Need to calculate new pos, if this is a bitstream

newpos = self.\_pos

for p in positions:

self[p:p + old.len] = new

if old.len != new.len:

diff = new.len - old.len

for p in positions:

if p >= newpos:

continue

if p + old.len <= newpos:

newpos += diff

else:

newpos = p

self.\_pos = newpos

except AttributeError:

for p in positions:

self[p:p + old.len] = new

assert self.\_assertsanity()

return len(lengths) - 1

def insert(self, bs, pos=None):

"""Insert bs at bit position pos.

bs -- The bitstring to insert.

pos -- The bit position to insert at.

Raises ValueError if pos < 0 or pos > self.len.

"""

bs = Bits(bs)

if not bs.len:

return self

if bs is self:

bs = self.\_\_copy\_\_()

if pos is None:

try:

pos = self.\_pos

except AttributeError:

raise TypeError("insert require a bit position for this type.")

if pos < 0:

pos += self.len

if not 0 <= pos <= self.len:

raise ValueError("Invalid insert position.")

self.\_insert(bs, pos)

def overwrite(self, bs, pos=None):

"""Overwrite with bs at bit position pos.

bs -- The bitstring to overwrite with.

pos -- The bit position to begin overwriting from.

Raises ValueError if pos < 0 or pos + bs.len > self.len

"""

bs = Bits(bs)

if not bs.len:

return

if pos is None:

try:

pos = self.\_pos

except AttributeError:

raise TypeError("overwrite require a bit position for this type.")

if pos < 0:

pos += self.len

if pos < 0 or pos + bs.len > self.len:

raise ValueError("Overwrite exceeds boundary of bitstring.")

self.\_overwrite(bs, pos)

try:

self.\_pos = pos + bs.len

except AttributeError:

pass

def append(self, bs):

"""Append a bitstring to the current bitstring.

bs -- The bitstring to append.

"""

# The offset is a hint to make bs easily appendable.

bs = self.\_converttobitstring(bs, offset=(self.len + self.\_offset) % 8)

self.\_append(bs)

def prepend(self, bs):

"""Prepend a bitstring to the current bitstring.

bs -- The bitstring to prepend.

"""

bs = Bits(bs)

self.\_prepend(bs)

def reverse(self, start=None, end=None):

"""Reverse bits in-place.

start -- Position of first bit to reverse. Defaults to 0.

end -- One past the position of the last bit to reverse.

Defaults to self.len.

Using on an empty bitstring will have no effect.

Raises ValueError if start < 0, end > self.len or end < start.

"""

start, end = self.\_validate\_slice(start, end)

if start == 0 and end == self.len:

self.\_reverse()

return

s = self.\_slice(start, end)

s.\_reverse()

self[start:end] = s

def set(self, value, pos=None):

"""Set one or many bits to 1 or 0.

value -- If True bits are set to 1, otherwise they are set to 0.

pos -- Either a single bit position or an iterable of bit positions.

Negative numbers are treated in the same way as slice indices.

Defaults to the entire bitstring.

Raises IndexError if pos < -self.len or pos >= self.len.

"""

f = self.\_set if value else self.\_unset

if pos is None:

pos = xrange(self.len)

try:

length = self.len

for p in pos:

if p < 0:

p += length

if not 0 <= p < length:

raise IndexError("Bit position {0} out of range.".format(p))

f(p)

except TypeError:

# Single pos

if pos < 0:

pos += self.len

if not 0 <= pos < length:

raise IndexError("Bit position {0} out of range.".format(pos))

f(pos)

def invert(self, pos=None):

"""Invert one or many bits from 0 to 1 or vice versa.

pos -- Either a single bit position or an iterable of bit positions.

Negative numbers are treated in the same way as slice indices.

Raises IndexError if pos < -self.len or pos >= self.len.

"""

if pos is None:

self.\_invert\_all()

return

if not isinstance(pos, collections.Iterable):

pos = (pos,)

length = self.len

for p in pos:

if p < 0:

p += length

if not 0 <= p < length:

raise IndexError("Bit position {0} out of range.".format(p))

self.\_invert(p)

def ror(self, bits, start=None, end=None):

"""Rotate bits to the right in-place.

bits -- The number of bits to rotate by.

start -- Start of slice to rotate. Defaults to 0.

end -- End of slice to rotate. Defaults to self.len.

Raises ValueError if bits < 0.

"""

if not self.len:

raise Error("Cannot rotate an empty bitstring.")

if bits < 0:

raise ValueError("Cannot rotate right by negative amount.")

start, end = self.\_validate\_slice(start, end)

bits %= (end - start)

if not bits:

return

rhs = self.\_slice(end - bits, end)

self.\_delete(bits, end - bits)

self.\_insert(rhs, start)

def rol(self, bits, start=None, end=None):

"""Rotate bits to the left in-place.

bits -- The number of bits to rotate by.

start -- Start of slice to rotate. Defaults to 0.

end -- End of slice to rotate. Defaults to self.len.

Raises ValueError if bits < 0.

"""

if not self.len:

raise Error("Cannot rotate an empty bitstring.")

if bits < 0:

raise ValueError("Cannot rotate left by negative amount.")

start, end = self.\_validate\_slice(start, end)

bits %= (end - start)

if not bits:

return

lhs = self.\_slice(start, start + bits)

self.\_delete(bits, start)

self.\_insert(lhs, end - bits)

def byteswap(self, fmt=None, start=None, end=None, repeat=True):

"""Change the endianness in-place. Return number of repeats of fmt done.

fmt -- A compact structure string, an integer number of bytes or

an iterable of integers. Defaults to 0, which byte reverses the

whole bitstring.

start -- Start bit position, defaults to 0.

end -- End bit position, defaults to self.len.

repeat -- If True (the default) the byte swapping pattern is repeated

as much as possible.

"""

start, end = self.\_validate\_slice(start, end)

if fmt is None or fmt == 0:

# reverse all of the whole bytes.

bytesizes = [(end - start) // 8]

elif isinstance(fmt, numbers.Integral):

if fmt < 0:

raise ValueError("Improper byte length {0}.".format(fmt))

bytesizes = [fmt]

elif isinstance(fmt, basestring):

m = STRUCT\_PACK\_RE.match(fmt)

if not m:

raise ValueError("Cannot parse format string {0}.".format(fmt))

# Split the format string into a list of 'q', '4h' etc.

formatlist = re.findall(STRUCT\_SPLIT\_RE, m.group('fmt'))

# Now deal with multiplicative factors, 4h -> hhhh etc.

bytesizes = []

for f in formatlist:

if len(f) == 1:

bytesizes.append(PACK\_CODE\_SIZE[f])

else:

bytesizes.extend([PACK\_CODE\_SIZE[f[-1]]] \* int(f[:-1]))

elif isinstance(fmt, collections.Iterable):

bytesizes = fmt

for bytesize in bytesizes:

if not isinstance(bytesize, numbers.Integral) or bytesize < 0:

raise ValueError("Improper byte length {0}.".format(bytesize))

else:

raise TypeError("Format must be an integer, string or iterable.")

repeats = 0

totalbitsize = 8 \* sum(bytesizes)

if not totalbitsize:

return 0

if repeat:

# Try to repeat up to the end of the bitstring.

finalbit = end

else:

# Just try one (set of) byteswap(s).

finalbit = start + totalbitsize

for patternend in xrange(start + totalbitsize, finalbit + 1, totalbitsize):

bytestart = patternend - totalbitsize

for bytesize in bytesizes:

byteend = bytestart + bytesize \* 8

self.\_reversebytes(bytestart, byteend)

bytestart += bytesize \* 8

repeats += 1

return repeats

def clear(self):

"""Remove all bits, reset to zero length."""

self.\_clear()

def copy(self):

"""Return a copy of the bitstring."""

return self.\_copy()

int = property(Bits.\_getint, Bits.\_setint,

doc="""The bitstring as a two's complement signed int. Read and write.

""")

uint = property(Bits.\_getuint, Bits.\_setuint,

doc="""The bitstring as a two's complement unsigned int. Read and write.

""")

float = property(Bits.\_getfloat, Bits.\_setfloat,

doc="""The bitstring as a floating point number. Read and write.

""")

intbe = property(Bits.\_getintbe, Bits.\_setintbe,

doc="""The bitstring as a two's complement big-endian signed int. Read and write.

""")

uintbe = property(Bits.\_getuintbe, Bits.\_setuintbe,

doc="""The bitstring as a two's complement big-endian unsigned int. Read and write.

""")

floatbe = property(Bits.\_getfloat, Bits.\_setfloat,

doc="""The bitstring as a big-endian floating point number. Read and write.

""")

intle = property(Bits.\_getintle, Bits.\_setintle,

doc="""The bitstring as a two's complement little-endian signed int. Read and write.

""")

uintle = property(Bits.\_getuintle, Bits.\_setuintle,

doc="""The bitstring as a two's complement little-endian unsigned int. Read and write.

""")

floatle = property(Bits.\_getfloatle, Bits.\_setfloatle,

doc="""The bitstring as a little-endian floating point number. Read and write.

""")

intne = property(Bits.\_getintne, Bits.\_setintne,

doc="""The bitstring as a two's complement native-endian signed int. Read and write.

""")

uintne = property(Bits.\_getuintne, Bits.\_setuintne,

doc="""The bitstring as a two's complement native-endian unsigned int. Read and write.

""")

floatne = property(Bits.\_getfloatne, Bits.\_setfloatne,

doc="""The bitstring as a native-endian floating point number. Read and write.

""")

ue = property(Bits.\_getue, Bits.\_setue,

doc="""The bitstring as an unsigned exponential-Golomb code. Read and write.

""")

se = property(Bits.\_getse, Bits.\_setse,

doc="""The bitstring as a signed exponential-Golomb code. Read and write.

""")

uie = property(Bits.\_getuie, Bits.\_setuie,

doc="""The bitstring as an unsigned interleaved exponential-Golomb code. Read and write.

""")

sie = property(Bits.\_getsie, Bits.\_setsie,

doc="""The bitstring as a signed interleaved exponential-Golomb code. Read and write.

""")

hex = property(Bits.\_gethex, Bits.\_sethex,

doc="""The bitstring as a hexadecimal string. Read and write.

""")

bin = property(Bits.\_getbin, Bits.\_setbin\_safe,

doc="""The bitstring as a binary string. Read and write.

""")

oct = property(Bits.\_getoct, Bits.\_setoct,

doc="""The bitstring as an octal string. Read and write.

""")

bool = property(Bits.\_getbool, Bits.\_setbool,

doc="""The bitstring as a bool (True or False). Read and write.

""")

bytes = property(Bits.\_getbytes, Bits.\_setbytes\_safe,

doc="""The bitstring as a ordinary string. Read and write.

""")

class ConstBitStream(Bits):

"""A container or stream holding an immutable sequence of bits.

For a mutable container use the BitStream class instead.

Methods inherited from Bits:

all() -- Check if all specified bits are set to 1 or 0.

any() -- Check if any of specified bits are set to 1 or 0.

count() -- Count the number of bits set to 1 or 0.

cut() -- Create generator of constant sized chunks.

endswith() -- Return whether the bitstring ends with a sub-string.

find() -- Find a sub-bitstring in the current bitstring.

findall() -- Find all occurrences of a sub-bitstring in the current bitstring.

join() -- Join bitstrings together using current bitstring.

rfind() -- Seek backwards to find a sub-bitstring.

split() -- Create generator of chunks split by a delimiter.

startswith() -- Return whether the bitstring starts with a sub-bitstring.

tobytes() -- Return bitstring as bytes, padding if needed.

tofile() -- Write bitstring to file, padding if needed.

unpack() -- Interpret bits using format string.

Other methods:

bytealign() -- Align to next byte boundary.

peek() -- Peek at and interpret next bits as a single item.

peeklist() -- Peek at and interpret next bits as a list of items.

read() -- Read and interpret next bits as a single item.

readlist() -- Read and interpret next bits as a list of items.

Special methods:

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

Properties:

bin -- The bitstring as a binary string.

bool -- For single bit bitstrings, interpret as True or False.

bytepos -- The current byte position in the bitstring.

bytes -- The bitstring as a bytes object.

float -- Interpret as a floating point number.

floatbe -- Interpret as a big-endian floating point number.

floatle -- Interpret as a little-endian floating point number.

floatne -- Interpret as a native-endian floating point number.

hex -- The bitstring as a hexadecimal string.

int -- Interpret as a two's complement signed integer.

intbe -- Interpret as a big-endian signed integer.

intle -- Interpret as a little-endian signed integer.

intne -- Interpret as a native-endian signed integer.

len -- Length of the bitstring in bits.

oct -- The bitstring as an octal string.

pos -- The current bit position in the bitstring.

se -- Interpret as a signed exponential-Golomb code.

ue -- Interpret as an unsigned exponential-Golomb code.

sie -- Interpret as a signed interleaved exponential-Golomb code.

uie -- Interpret as an unsigned interleaved exponential-Golomb code.

uint -- Interpret as a two's complement unsigned integer.

uintbe -- Interpret as a big-endian unsigned integer.

uintle -- Interpret as a little-endian unsigned integer.

uintne -- Interpret as a native-endian unsigned integer.

"""

\_\_slots\_\_ = ('\_pos')

def \_\_init\_\_(self, auto=None, length=None, offset=None, \*\*kwargs):

"""Either specify an 'auto' initialiser:

auto -- a string of comma separated tokens, an integer, a file object,

a bytearray, a boolean iterable or another bitstring.

Or initialise via \*\*kwargs with one (and only one) of:

bytes -- raw data as a string, for example read from a binary file.

bin -- binary string representation, e.g. '0b001010'.

hex -- hexadecimal string representation, e.g. '0x2ef'

oct -- octal string representation, e.g. '0o777'.

uint -- an unsigned integer.

int -- a signed integer.

float -- a floating point number.

uintbe -- an unsigned big-endian whole byte integer.

intbe -- a signed big-endian whole byte integer.

floatbe - a big-endian floating point number.

uintle -- an unsigned little-endian whole byte integer.

intle -- a signed little-endian whole byte integer.

floatle -- a little-endian floating point number.

uintne -- an unsigned native-endian whole byte integer.

intne -- a signed native-endian whole byte integer.

floatne -- a native-endian floating point number.

se -- a signed exponential-Golomb code.

ue -- an unsigned exponential-Golomb code.

sie -- a signed interleaved exponential-Golomb code.

uie -- an unsigned interleaved exponential-Golomb code.

bool -- a boolean (True or False).

filename -- a file which will be opened in binary read-only mode.

Other keyword arguments:

length -- length of the bitstring in bits, if needed and appropriate.

It must be supplied for all integer and float initialisers.

offset -- bit offset to the data. These offset bits are

ignored and this is intended for use when

initialising using 'bytes' or 'filename'.

"""

self.\_pos = 0

def \_\_new\_\_(cls, auto=None, length=None, offset=None, \*\*kwargs):

x = super(ConstBitStream, cls).\_\_new\_\_(cls)

x.\_initialise(auto, length, offset, \*\*kwargs)

return x

def \_setbytepos(self, bytepos):

"""Move to absolute byte-aligned position in stream."""

self.\_setbitpos(bytepos \* 8)

def \_getbytepos(self):

"""Return the current position in the stream in bytes. Must be byte aligned."""

if self.\_pos % 8:

raise ByteAlignError("Not byte aligned in \_getbytepos().")

return self.\_pos // 8

def \_setbitpos(self, pos):

"""Move to absolute postion bit in bitstream."""

if pos < 0:

raise ValueError("Bit position cannot be negative.")

if pos > self.len:

raise ValueError("Cannot seek past the end of the data.")

self.\_pos = pos

def \_getbitpos(self):

"""Return the current position in the stream in bits."""

return self.\_pos

def \_clear(self):

Bits.\_clear(self)

self.\_pos = 0

def \_\_copy\_\_(self):

"""Return a new copy of the ConstBitStream for the copy module."""

# Note that if you want a new copy (different ID), use \_copy instead.

# The copy can use the same datastore as it's immutable.

s = ConstBitStream()

s.\_datastore = self.\_datastore

# Reset the bit position, don't copy it.

s.\_pos = 0

return s

def \_\_add\_\_(self, bs):

"""Concatenate bitstrings and return new bitstring.

bs -- the bitstring to append.

"""

s = Bits.\_\_add\_\_(self, bs)

s.\_pos = 0

return s

def read(self, fmt):

"""Interpret next bits according to the format string and return result.

fmt -- Token string describing how to interpret the next bits.

Token examples: 'int:12' : 12 bits as a signed integer

'uint:8' : 8 bits as an unsigned integer

'float:64' : 8 bytes as a big-endian float

'intbe:16' : 2 bytes as a big-endian signed integer

'uintbe:16' : 2 bytes as a big-endian unsigned integer

'intle:32' : 4 bytes as a little-endian signed integer

'uintle:32' : 4 bytes as a little-endian unsigned integer

'floatle:64': 8 bytes as a little-endian float

'intne:24' : 3 bytes as a native-endian signed integer

'uintne:24' : 3 bytes as a native-endian unsigned integer

'floatne:32': 4 bytes as a native-endian float

'hex:80' : 80 bits as a hex string

'oct:9' : 9 bits as an octal string

'bin:1' : single bit binary string

'ue' : next bits as unsigned exp-Golomb code

'se' : next bits as signed exp-Golomb code

'uie' : next bits as unsigned interleaved exp-Golomb code

'sie' : next bits as signed interleaved exp-Golomb code

'bits:5' : 5 bits as a bitstring

'bytes:10' : 10 bytes as a bytes object

'bool' : 1 bit as a bool

'pad:3' : 3 bits of padding to ignore - returns None

fmt may also be an integer, which will be treated like the 'bits' token.

The position in the bitstring is advanced to after the read items.

Raises ReadError if not enough bits are available.

Raises ValueError if the format is not understood.

"""

if isinstance(fmt, numbers.Integral):

if fmt < 0:

raise ValueError("Cannot read negative amount.")

if fmt > self.len - self.\_pos:

raise ReadError("Cannot read {0} bits, only {1} available.",

fmt, self.len - self.\_pos)

bs = self.\_slice(self.\_pos, self.\_pos + fmt)

self.\_pos += fmt

return bs

p = self.\_pos

\_, token = tokenparser(fmt)

if len(token) != 1:

self.\_pos = p

raise ValueError("Format string should be a single token, not {0} "

"tokens - use readlist() instead.".format(len(token)))

name, length, \_ = token[0]

if length is None:

length = self.len - self.\_pos

value, self.\_pos = self.\_readtoken(name, self.\_pos, length)

return value

def readlist(self, fmt, \*\*kwargs):

"""Interpret next bits according to format string(s) and return list.

fmt -- A single string or list of strings with comma separated tokens

describing how to interpret the next bits in the bitstring. Items

can also be integers, for reading new bitstring of the given length.

kwargs -- A dictionary or keyword-value pairs - the keywords used in the

format string will be replaced with their given value.

The position in the bitstring is advanced to after the read items.

Raises ReadError is not enough bits are available.

Raises ValueError if the format is not understood.

See the docstring for 'read' for token examples. 'pad' tokens are skipped

and not added to the returned list.

>>> h, b1, b2 = s.readlist('hex:20, bin:5, bin:3')

>>> i, bs1, bs2 = s.readlist(['uint:12', 10, 10])

"""

value, self.\_pos = self.\_readlist(fmt, self.\_pos, \*\*kwargs)

return value

def readto(self, bs, bytealigned=None):

"""Read up to and including next occurrence of bs and return result.

bs -- The bitstring to find. An integer is not permitted.

bytealigned -- If True the bitstring will only be

found on byte boundaries.

Raises ValueError if bs is empty.

Raises ReadError if bs is not found.

"""

if isinstance(bs, numbers.Integral):

raise ValueError("Integers cannot be searched for")

bs = Bits(bs)

oldpos = self.\_pos

p = self.find(bs, self.\_pos, bytealigned=bytealigned)

if not p:

raise ReadError("Substring not found")

self.\_pos += bs.len

return self.\_slice(oldpos, self.\_pos)

def peek(self, fmt):

"""Interpret next bits according to format string and return result.

fmt -- Token string describing how to interpret the next bits.

The position in the bitstring is not changed. If not enough bits are

available then all bits to the end of the bitstring will be used.

Raises ReadError if not enough bits are available.

Raises ValueError if the format is not understood.

See the docstring for 'read' for token examples.

"""

pos\_before = self.\_pos

value = self.read(fmt)

self.\_pos = pos\_before

return value

def peeklist(self, fmt, \*\*kwargs):

"""Interpret next bits according to format string(s) and return list.

fmt -- One or more strings with comma separated tokens describing

how to interpret the next bits in the bitstring.

kwargs -- A dictionary or keyword-value pairs - the keywords used in the

format string will be replaced with their given value.

The position in the bitstring is not changed. If not enough bits are

available then all bits to the end of the bitstring will be used.

Raises ReadError if not enough bits are available.

Raises ValueError if the format is not understood.

See the docstring for 'read' for token examples.

"""

pos = self.\_pos

return\_values = self.readlist(fmt, \*\*kwargs)

self.\_pos = pos

return return\_values

def bytealign(self):

"""Align to next byte and return number of skipped bits.

Raises ValueError if the end of the bitstring is reached before

aligning to the next byte.

"""

skipped = (8 - (self.\_pos % 8)) % 8

self.pos += self.\_offset + skipped

assert self.\_assertsanity()

return skipped

pos = property(\_getbitpos, \_setbitpos,

doc="""The position in the bitstring in bits. Read and write.

""")

bitpos = property(\_getbitpos, \_setbitpos,

doc="""The position in the bitstring in bits. Read and write.

""")

bytepos = property(\_getbytepos, \_setbytepos,

doc="""The position in the bitstring in bytes. Read and write.

""")

class BitStream(ConstBitStream, BitArray):

"""A container or stream holding a mutable sequence of bits

Subclass of the ConstBitStream and BitArray classes. Inherits all of

their methods.

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.

append() -- Append a bitstring.

bytealign() -- Align to next byte boundary.

byteswap() -- Change byte endianness in-place.

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-string.

find() -- Find a sub-bitstring in the current bitstring.

findall() -- Find all occurrences of a sub-bitstring in the current bitstring.

insert() -- Insert a bitstring.

invert() -- Flip bit(s) between one and zero.

join() -- Join bitstrings together using current bitstring.

overwrite() -- Overwrite a section with a new bitstring.

peek() -- Peek at and interpret next bits as a single item.

peeklist() -- Peek at and interpret next bits as a list of items.

prepend() -- Prepend a bitstring.

read() -- Read and interpret next bits as a single item.

readlist() -- Read and interpret next bits as a list of items.

replace() -- Replace occurrences of one bitstring with another.

reverse() -- Reverse bits in-place.

rfind() -- Seek backwards to find a sub-bitstring.

rol() -- Rotate bits to the left.

ror() -- Rotate bits to the right.

set() -- Set bit(s) to 1 or 0.

split() -- Create generator of chunks split by a delimiter.

startswith() -- Return whether the bitstring starts with a sub-bitstring.

tobytes() -- Return bitstring as bytes, padding if needed.

tofile() -- Write bitstring to file, padding if needed.

unpack() -- Interpret bits using format string.

Special methods:

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

in addition to [], ==, !=, +, \*, ~, <<, >>, &, | and ^.

Properties:

bin -- The bitstring as a binary string.

bool -- For single bit bitstrings, interpret as True or False.

bytepos -- The current byte position in the bitstring.

bytes -- The bitstring as a bytes object.

float -- Interpret as a floating point number.

floatbe -- Interpret as a big-endian floating point number.

floatle -- Interpret as a little-endian floating point number.

floatne -- Interpret as a native-endian floating point number.

hex -- The bitstring as a hexadecimal string.

int -- Interpret as a two's complement signed integer.

intbe -- Interpret as a big-endian signed integer.

intle -- Interpret as a little-endian signed integer.

intne -- Interpret as a native-endian signed integer.

len -- Length of the bitstring in bits.

oct -- The bitstring as an octal string.

pos -- The current bit position in the bitstring.

se -- Interpret as a signed exponential-Golomb code.

ue -- Interpret as an unsigned exponential-Golomb code.

sie -- Interpret as a signed interleaved exponential-Golomb code.

uie -- Interpret as an unsigned interleaved exponential-Golomb code.

uint -- Interpret as a two's complement unsigned integer.

uintbe -- Interpret as a big-endian unsigned integer.

uintle -- Interpret as a little-endian unsigned integer.

uintne -- Interpret as a native-endian unsigned integer.

"""

\_\_slots\_\_ = ()

# As BitStream objects are mutable, we shouldn't allow them to be hashed.

\_\_hash\_\_ = None

def \_\_init\_\_(self, auto=None, length=None, offset=None, \*\*kwargs):

"""Either specify an 'auto' initialiser:

auto -- a string of comma separated tokens, an integer, a file object,

a bytearray, a boolean iterable or another bitstring.

Or initialise via \*\*kwargs with one (and only one) of:

bytes -- raw data as a string, for example read from a binary file.

bin -- binary string representation, e.g. '0b001010'.

hex -- hexadecimal string representation, e.g. '0x2ef'

oct -- octal string representation, e.g. '0o777'.

uint -- an unsigned integer.

int -- a signed integer.

float -- a floating point number.

uintbe -- an unsigned big-endian whole byte integer.

intbe -- a signed big-endian whole byte integer.

floatbe - a big-endian floating point number.

uintle -- an unsigned little-endian whole byte integer.

intle -- a signed little-endian whole byte integer.

floatle -- a little-endian floating point number.

uintne -- an unsigned native-endian whole byte integer.

intne -- a signed native-endian whole byte integer.

floatne -- a native-endian floating point number.

se -- a signed exponential-Golomb code.

ue -- an unsigned exponential-Golomb code.

sie -- a signed interleaved exponential-Golomb code.

uie -- an unsigned interleaved exponential-Golomb code.

bool -- a boolean (True or False).

filename -- a file which will be opened in binary read-only mode.

Other keyword arguments:

length -- length of the bitstring in bits, if needed and appropriate.

It must be supplied for all integer and float initialisers.

offset -- bit offset to the data. These offset bits are

ignored and this is intended for use when

initialising using 'bytes' or 'filename'.

"""

self.\_pos = 0

# For mutable BitStreams we always read in files to memory:

if not isinstance(self.\_datastore, ByteStore):

self.\_ensureinmemory()

def \_\_new\_\_(cls, auto=None, length=None, offset=None, \*\*kwargs):

x = super(BitStream, cls).\_\_new\_\_(cls)

x.\_initialise(auto, length, offset, \*\*kwargs)

return x

def \_\_copy\_\_(self):

"""Return a new copy of the BitStream."""

s\_copy = BitStream()

s\_copy.\_pos = 0

if not isinstance(self.\_datastore, ByteStore):

# Let them both point to the same (invariant) array.

# If either gets modified then at that point they'll be read into memory.

s\_copy.\_datastore = self.\_datastore

else:

s\_copy.\_datastore = ByteStore(self.\_datastore.\_rawarray[:],

self.\_datastore.bitlength,

self.\_datastore.offset)

return s\_copy

def prepend(self, bs):

"""Prepend a bitstring to the current bitstring.

bs -- The bitstring to prepend.

"""

bs = self.\_converttobitstring(bs)

self.\_prepend(bs)

self.\_pos += bs.len

def pack(fmt, \*values, \*\*kwargs):

"""Pack the values according to the format string and return a new BitStream.

fmt -- A single string or a list of strings with comma separated tokens

describing how to create the BitStream.

values -- Zero or more values to pack according to the format.

kwargs -- A dictionary or keyword-value pairs - the keywords used in the

format string will be replaced with their given value.

Token examples: 'int:12' : 12 bits as a signed integer

'uint:8' : 8 bits as an unsigned integer

'float:64' : 8 bytes as a big-endian float

'intbe:16' : 2 bytes as a big-endian signed integer

'uintbe:16' : 2 bytes as a big-endian unsigned integer

'intle:32' : 4 bytes as a little-endian signed integer

'uintle:32' : 4 bytes as a little-endian unsigned integer

'floatle:64': 8 bytes as a little-endian float

'intne:24' : 3 bytes as a native-endian signed integer

'uintne:24' : 3 bytes as a native-endian unsigned integer

'floatne:32': 4 bytes as a native-endian float

'hex:80' : 80 bits as a hex string

'oct:9' : 9 bits as an octal string

'bin:1' : single bit binary string

'ue' / 'uie': next bits as unsigned exp-Golomb code

'se' / 'sie': next bits as signed exp-Golomb code

'bits:5' : 5 bits as a bitstring object

'bytes:10' : 10 bytes as a bytes object

'bool' : 1 bit as a bool

'pad:3' : 3 zero bits as padding

>>> s = pack('uint:12, bits', 100, '0xffe')

>>> t = pack(['bits', 'bin:3'], s, '111')

>>> u = pack('uint:8=a, uint:8=b, uint:55=a', a=6, b=44)

"""

tokens = []

if isinstance(fmt, basestring):

fmt = [fmt]

try:

for f\_item in fmt:

\_, tkns = tokenparser(f\_item, tuple(sorted(kwargs.keys())))

tokens.extend(tkns)

except ValueError as e:

raise CreationError(\*e.args)

value\_iter = iter(values)

s = BitStream()

try:

for name, length, value in tokens:

# If the value is in the kwd dictionary then it takes precedence.

if value in kwargs:

value = kwargs[value]

# If the length is in the kwd dictionary then use that too.

if length in kwargs:

length = kwargs[length]

# Also if we just have a dictionary name then we want to use it

if name in kwargs and length is None and value is None:

s.append(kwargs[name])

continue

if length is not None:

length = int(length)

if value is None and name != 'pad':

# Take the next value from the ones provided

value = next(value\_iter)

s.\_append(BitStream.\_init\_with\_token(name, length, value))

except StopIteration:

raise CreationError("Not enough parameters present to pack according to the "

"format. {0} values are needed.", len(tokens))

try:

next(value\_iter)

except StopIteration:

# Good, we've used up all the \*values.

return s

raise CreationError("Too many parameters present to pack according to the format.")

# Aliases for backward compatibility

ConstBitArray = Bits

BitString = BitStream

\_\_all\_\_ = ['ConstBitArray', 'ConstBitStream', 'BitStream', 'BitArray',

'Bits', 'BitString', 'pack', 'Error', 'ReadError',

'InterpretError', 'ByteAlignError', 'CreationError', 'bytealigned']