Bread Documentation

Release 3.0.0

Alex Rasmussen

Dec 03, 2018

Contents

Reading binary formats is a pain. `bread` (short for "binary read", but pronounced like the baked good) makes that simpler. `bread` understands a simple declarative specification of a binary format, which it uses to parse. It's more verbose, but the format is a lot easier to understand and the resulting object is a lot easier to use.

CHAPTER 1

User's Guide

1.1 Introduction

In this section, we'll discuss why I wrote `bread`, and give a rough sense of what it can do.

1.1.1 Motivation

Here's an example from the documentation for Python's `struct` library::

```
record = 'raymond   \x32\x12\x08\x01\x08'
name, serialnum, school, gradelevel = unpack('<10sHHb', record)
```

The format specification is dense, but it's also really hard to understand. What was `H` again? is `b` signed? Am I sure I'm unpacking those fields in the right order?

Now what happens if I have arrays of complex structures? Deeply nested structures? This can get messy really fast.

Here's `bread`'s format specification for the above example::

```
import bread as b
record_spec = [
    {"endianness" : b.LITTLE_ENDIAN},
    ("name", b.string(10)),
    ("serialnum", b.uint16),
    ("school", b.uint16),
    ("gradelevel", b.byte)
]
```

Here's how to parse using that specification::

```
>>> parsed_record = b.parse(record, record_spec)
>>> parsed_record.name
"raymond   "
```

(continues on next page)

(continued from previous page)

```
>>> parsed_record.school
264
```

Here's a more complicated format specification::

```
nested_array_struct = [
    {"endianness" : b.BIG_ENDIAN},
    ("first", b.uint8),
    ("matrix", b.array(3, b.array(3, b.uint8))),
    ("last", b.uint8)
]
```

And how to parse using it::

```
>>> data = bytearray([42, 0, 1, 2, 3, 4, 5, 6, 7, 8, 0xdb])
>>> nested_parsed = b.parse(data, nested_array_struct)
>>> print nested_parsed
first: 42
matrix: [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
last: 219
```

1.1.2 Goals (and Non-Goals)

`bread` was designed to read binary files into a read-only object format that could be used by other tools. It's not really meant for writing binary data at this point (although I can imagine future versions being able to do something like that if I find the need).

I wrote `bread` with ease of use, rather than speed of execution, as a first-order concern. That's not to say that `bread` is really slow, but if you're writing something that analyzes gigabytes of binary data and speed is your main concern, you may want to just roll your own optimized format reader in something like C and call it a day.

1.2 The Format Specification Language

`bread` reads binary data according to a *format specification*. Format specifications are just lists. Each element in the list is called a *field descriptor*. Field descriptors describe how to consume a piece of binary data from the input, usually to create a *field* in the resulting object. Field descriptors are consumed from the format specification one at a time until the entire list has been consumed.

1.2.1 Field Descriptors

Most field descriptors will consume a certain amount of binary data and produce a value of a certain basic type.

Integers

`intX(num_bits, signed)` - the next `num_bits` bits represent an integer. If `signed` is `True`, the integer is interpreted as a signed, twos-complement number.

For convenience and improved readability, the following shorthands are defined:

Field Descriptor	Width (Bits)	Signed
`bit`	1	no
`semi_nibble`	2	no
`nibble`	4	no
`byte`	8	no
`uint8`	8	no
`uint16`	16	no
`uint32`	32	no
`uint64`	64	no
`int8`	8	yes
`int16`	16	yes
`int32`	32	yes
`int64`	64	yes

Strings

`string(length, encoding)` - the next `length` bytes represent a string of the given length. You can pick an encoding for the strings to encode and decode in; the default is `utf-8`.

Booleans

`boolean` - the next bit represents a boolean value. 0 is `False`, 1 is `True`

Enumerations

`enum(length, values, default=None)` - the next `length` bits represent one of a set of values, whose values are given by the dictionary `values`. If `default` is specified, it will be returned if the bits do not correspond to any value in `values`. Otherwise, it raises a `ValueError`.

Here is an example of a 2-bit field representing a card suit:

```
import bread as b

("suit", b.enum(2, {
    0: "diamonds",
    1: "hearts",
    2: "spades",
    3: "clubs"
}))
```

Arrays

`array(count, field_or_struct)` - the next piece of data is `count` occurrences of `field_or_struct` which, as the name might imply, can be either a field (including another array) or a format specification.

Here's an example way of representing a deck of playing cards:

```
import bread as b

# A card is made up of a 2-bit suit and a 4-bit card number
```

(continues on next page)

(continued from previous page)

```
card = [
    ("suit", b.enum(2, {
        0: "diamonds",
        1: "hearts",
        2: "spades",
        3: "clubs"
    })),
    ("number", b.intX(4))]

# A deck consists of 52 cards, for a total of 312 bits or 39 bytes of data

deck = [("cards", b.array(52, card))]
```

1.2.2 Field Options

A dictionary of field options can be specified as the last argument to any field. A dictionary of global field options can also be defined at the beginning of the format spec (before any fields). Options defined on fields override these global options.

The following field options are defined:

- `str_format` - the function that should be used to format a field in the structure's human-readable representation. For example:

```
>>> import bread as b

# Format spec without str_format ...
>>> simple_spec = [('addr', b.uint8)]
>>> parsed_data = b.parse(bytearray([42]), simple_spec)
>>> print parsed_data
addr: 42

# ... and with str_format
>>> simple_spec_hex = [('addr', b.uint8, {"str_format": hex})]
>>> parsed_data = b.parse(bytearray([42]), simple_spec_hex)
>>> print parsed_data
addr: 0x2a
```

- `endianness` - for integer types, the endianness of the bytes that make up that integer. Can either be `LITTLE_ENDIAN` or `BIG_ENDIAN`. Default is little-endian.

  A simple example:

```
endianness_test = [
  ("big_endian", b.uint32, {"endianness" : b.BIG_ENDIAN}),
  ("little_endian", b.uint32, {"endianness" : b.LITTLE_ENDIAN}),
  ("default_endian", b.uint32)]

data = bytearray([0x01, 0x02, 0x03, 0x04] * 3)
test = b.parse(data, endianness_test)

>>> test.big_endian == 0x01020304
True
>>> test.little_endian == 0x04030201
True
```

(continues on next page)

(continued from previous page)

```
>>> test.default_endian == test.little_endian
True
```

- `offset` - for integer types, the amount to add to the number after it has been parsed. Specifying a negative number will subtract that amount from the number.

1.2.3 Conditionals

Conditionals allow the format specification to branch based on the value of a previous field. Conditional field descriptors are specified as follows:

```
(CONDITIONAL "field_name", options)
```

where `field_name` is the name of the field whose value determines the course of the conditional, and `options` is a dictionary giving format specifications to evaluate based on the field's value.

This is perhaps best illustrated by example:

```
import bread as b

# There are three kinds of widgets: type A, type B and type C. Each has
# its own format spec.

widget_A = [...]
widget_B = [...]
widget_C = [...]

# A widget may be of any of the three types, determined by its type field

widget = [
    ("type", b.string(1)),
    (b.CONDITIONAL, "type", {
        "A": widget_A,
        "B": widget_B,
        "C": widget_C
    })]
```

1.2.4 Padding

`padding(num_bits)` - indicates that the next `num_bits` bits should be ignored. Useful in situations where only the first few bits of a byte are meaningful, or where the format skips multiple bits or bytes.

1.3 Parsing

Currently, `bread` can parse data contained in strings, byte arrays, or files. In all three cases, data parsing is done with the function `parse(data, spec)`.

An example of parsing files:

```
import bread as b

format_spec = [...]
```

(continues on next page)

(continued from previous page)

```
with open('raw_file.bin', 'rb') as fp:
    parsed_obj = b.parse(fp, format_spec)
```

An example with byte arrays and strings:

```
import bread as b

format_spec = [("greeting", b.string(5))]

bytes = bytearray([0x68, 0x65, 0x6c, 0x6c, 0x6f])
string = "hello"

parsed_bytes = b.parse(bytes, format_spec)
parsed_string = b.parse(string, format_spec)
```

1.4 Parsed Object Methods

Objects produced by bread can produce JSON representations of themselves. Calling the object's `as_json()` method will produce its data as a JSON string.

Objects produced by bread can also produce representations of themselves as Pythonic `list` s, `dict` s, etc. Calling the object's `as_native()` method will produce its data in this form.

1.5 Creating Empty Objects

Sometimes, you want to write a binary format without having to read anything first. To do this in Bread, you can use the function `new(spec)`.

Here's an example of `new()` in action:

```
format_spec = [("greeting", b.string(5)),
               ("age", b.nibble)]

empty_struct = b.new(format_spec)

empty_struct.greeting = 'hello'
empty_struct.age = 0xb

output_bytes = b.write(empty_struct)
```

1.6 Writing

New in version 1.2.

`write(parsed_obj, spec, filename=None)`

`bread` allows you to parse data, modify it, and then write the modified version back out again.

An example of reading, modifying and writing a file:

```
import bread as b

format_spec = [
    ('x', b.boolean),
    ('y', b.uint16)
]

with open('raw_file.bin', 'rb') as fp:
    parsed_obj = b.parse(fp, format_spec)

parsed_obj.y = 37

# When called without a 'filename' argument, write() returns the raw
# written data as a bytearray

modified_data = write(parsed_obj, format_spec)

# When called with a filename, write() writes the data to the named file
write(parsed_obj, format_spec, filename='raw_file.bin.modified')
```

1.7 Examples

1.7.1 NSF Headers

The following parses the header for an NES Sound Format (NSF) file and prints it in a human-readable format:

```
import bread as b
import sys

def hex_array(x):
    return str(map(hex, x))

nsf_header = [
    ('magic_number', b.array(5, b.byte),
     {"str_format": hex_array}),
    ('version', b.byte),
    ('total_songs', b.byte),
    ('starting_song', b.byte),
    ('load_addr', b.uint16, {"str_format": hex}),
    ('init_addr', b.uint16, {"str_format": hex}),
    ('play_addr', b.uint16, {"str_format": hex}),
    ('title', b.string(32)),
    ('artist', b.string(32)),
    ('copyright', b.string(32)),
    ('ntsc_speed', b.uint16),
    ('bankswitch_init', b.array(8, b.byte), {"str_format": hex_array}),
    ('pal_speed', b.uint16),
    ('ntsc', b.boolean),
    ('pal', b.boolean),
    ('ntsc_and_pal', b.boolean),
    (b.padding(6)),
    ('vrc6', b.boolean),
    ('vrc7', b.boolean),
    ('fds', b.boolean),
```

(continues on next page)

(continued from previous page)

```
    ('mmc5', b.boolean),
    ('namco_106', b.boolean),
    ('fme07', b.boolean),
    (b.padding(2)),
    (b.padding(32))
]

with open(sys.argv[1], 'r') as fp:
    header = b.parse(fp, nsf_header)
    print header
```

Here are a couple of examples of its output:

```
$ python nsf_header.py Mega_Man_2.nsf

magic_number: ['0x4e', '0x45', '0x53', '0x4d', '0x1a']
version: 1
total_songs: 24
starting_song: 1
load_addr: 0x8000
init_addr: 0x8003
play_addr: 0x8000
title: Mega Man 2
artist: Ogeretsu,Manami,Ietel,YuukiChan
copyright: 1988,1989 Capcom Co. Ltd.
ntsc_speed: 16666
bankswitch_init: ['0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0']
pal_speed: 0
ntsc: False
pal: False
ntsc_and_pal: False
vrc6: False
vrc7: False
fds: False
mmc5: False
namco_106: False
fme07: False

$ python nsf_header.py Super_Mario_Bros.nsf

magic_number: ['0x4e', '0x45', '0x53', '0x4d', '0x1a']
version: 1
total_songs: 18
starting_song: 1
load_addr: 0x8dc4
init_addr: 0xbe34
play_addr: 0xf2d0
title: Super Mario Bros.
artist: Koji Kondo
copyright: 1985 Nintendo
ntsc_speed: 16666
bankswitch_init: ['0x0', '0x0', '0x0', '0x0', '0x1', '0x1', '0x1', '0x1']
pal_speed: 0
ntsc: False
pal: False
ntsc_and_pal: False
vrc6: False
```

(continues on next page)

(continued from previous page)

```
vrc7: False
fds: False
mmc5: False
namco_106: False
fme07: False
```