pybdg Documentation

Release 1.0.dev2

Outernet Inc

April 17, 2016

Contents

Bitloads, or bit payloads, are compact payloads containing binary representations of data. It's a generic binary serialization format for Python objects.

Today, serializing data in text-based formats JSON and XML is quite popular. These formats are easy to handle, flexible, and don't rquire too much effort to use. The tradeoff is the size in bytes serialized payload consumes. When transmitting data over connections with limited bandwidth and/or high bandwidth cost, a more efficient way of storing data may be needed. This library was created to serve this need.

In bitloads, the structure and the *size* of each piece of data that goes into the payload must be known in advance. It does not support complex sturctures either. The lack the flexibility and convenience of text-based formats, is a tradeoff for optimizing for size. Bitloads are somewhat similar to structs, but they allow for even tighter packing of data.

The pybtl uses a declarative syntax for describing conversion of python objects into bitloads and vice versa, and provides the tools for performing the conversions. The name of the library has also been shotened to reflect its purpose. :)

CHAPTER 1

Source code

The source code is available on GitHub

CHAPTER 2

License

pybtl is licensed under BSD license. Please see the `LICENSE` file in the source tree for more information.

CHAPTER 3

Documentation

3.1 Introduction to bitloads

To illustrate the practical advantages of bitloads, let's consider a simple message that has two fields:

- 'id', which is an MD5 hexdigest
- 'count', which is a number between 0 and 15

An MD5 hexdigest has 32 hex digits, which translates to 128 bits (4 bits per digit). Since 'count' can never be larger than 15, we can use 4 bits to represent it. Our final bitload therefore has 128 + 4 bits, which is 132 bits in total. To round it to whole bytes, we may add 4 more bits of pading. That is 17 bytes in total (or 17 ascii characters) to represent this information.

In comparison, assuming a JSON representation of a dict that has the 'id' and 'count' keys, and using a string hexdigest and an integer number, we get a string that typically uses around 440 bits (416 with whitespace stripped away).

```
>>> len('{"count":12,"id":"acbd18db4cc2f85cedef654fccc4a4d8"}') * 8
416
```

The MD5 hash alone will use twice as many bits as the entire bitload!

When we are dealing with networks with low availability and high bandwidth cost, transmitting small amounts of data quickly can mean the difference between successful and unsuccessful transmission. Smaller payload size means quicker transmission and more efficient use of the available bandwidth, and this is what bitloads do well.

On the other hands, there are things bitloads are not very good at. In order to tightly pack the data, the length of each field in the bitload must be known in advance. This means that it is not possible to include data with arbitrary length that is not known in advance (it is *technically* possible, but it would make deserialization more involved).

3.2 Describing binary bitload fields

Each bitload is a sequence of simple python values (numbers, bytestrings, booleans) in binary format. The slots in which each of the values fit are caled 'fields'. Each field is associated with a name, data type, and, in most cases, length of the field in bits.

Let's take a look at a simple example:

```
>>> message_format = (
...     ('id', 'hex', 128),
...     ('count', 'int', 4)
... )
```

This example defines two fields, 'id', and 'count'. The 'id' field is a hex field and it's 128 bits long. The other field is 4 bits long and its data type is an integer.

The following data types are supported:

- 'str': string
- 'bytes': raw bytestring
- 'int': unsigned integer number
- 'hex': hexadecimal representation of a number in string format
- 'bool': boolean value
- 'pad': padding bits

Here are some more examples:

```
>>> another_message = (
...     ('done', 'bool'),
...     (None, 'pad', 2)
...     ('note', 'str', 256),
... )
```

Note: The 'bool' data type does not need a length, since it is always 1 bit.

Note: The name of a field with 'pad' data type is ignored. By convention, we use `None` so it stands out, but you can use names like `'*** PADDING ***'` for a more dramatic effect.

The order in which the fields are added to bitloads is the order in which they appear in the tuple/iterable.

It is important to understand the limits of your data. There are no checks to make sure the source data will fit the bitload field, so you may get unexpected results if you are no careful (e.g., inserting a 10-bit integer into a 4-bit field will yield the wrong value after deserialization).

3.2.1 About the built-in types

The built-in types have conversion functions in the *utils* module. The functions use names that follow the `'{type}_to_bita'` and `'bita_to_{type}'` format. The following table gives an overview of possible input (serializable) and output (deserialized) values:

type	inputs	outputs
str	`bytes`, `str`/`unicode`	`str`/`unicode`
bytes	`bytes`	`bytes`
int	`int` (unsigned long long)	`int` (unsigned long long)
hex	`bytes`, `str`/`unicode` (hex number as a string)	`bytes` (hex number as a string)
bool	any value (coerced using `bool()`)	`bool`
pad	n/a	n/a

Note: All unicode strings are stored as UTF-8.

3.2.2 Dealing with other types of data

In order to deal with values that aren't directly supported by one of the standard types, there are two possible strategies we can employ.

One strategy is to adapt the python values. For example, `datetime` objects can be represented as unsigned integers. Floats can also be represented as a product of an integer and negative power of 10, and we can therefore store only the integer and restore the float by multiplying with the same negative power of 10 after deserializing it. Signed integers can be represented by scaling them such as 0 represents the smallest negative value.

Another strategy is to use a custom type. This data type allows one to add completely new types with relative ease. The tuple for this data type looks like this:

```
>>> ('myfield', 'user', 24, serializer, deserializer)
```

Note: The use of `'user'` type name is just an example. Any type that is not one of the types listed in this section can be used (i.e., any type other than 'str', 'bytes', 'int', 'hex', 'bool', and 'pad').

Two additional elements are the `serializer` and `deserializer` functions.

The `serializer` function takes a python value, and is expected to return a `bitarray` instance. The length of the output is not important as it will be adjusted to the correct length during serialization by padding with 0 or trimming off surplus bits. Keep in mind, though, that surplus bits *are* going to be trimmed off, which may not be what you want.

The `deserializer` function takes a `bitarray` instance, and is expected to return a python value. There are no restrictions on the return value.

Note: The bitarray documentation can be found on GitHub.

3.3 Serializing and deserializing

The conversion of python values into a bitload is called 'serializing', and the opposite operation is called 'deserializing'. In both cases, we need to supply the message format description (see Describing binary bitload fields). During serialization, data is supplied as a Python dict. After deserialization, the data comes out as a Python dict (technically a `OrderedDict` instance). The keys in the dict will correspond to field names in the bitload description.

In the examples in this section we will use a description tuple that looks like this:

```
>>> message_format = (
...     ('id', 'hex', 128),
...     ('count', 'int', 4),
...     (None, 'pad', 4),
... )
```

For both serializing and deserializing, a *`btl.bitload.Bitload`* class is used. The *`Bitload`* objects are instantiated with the description tuple:

```
>>> from btl import Bitload
>>> b = Bitload(message_format)
```

For serializing and deserializing, we can now call methods on this object.

3.3.1 Serializing

Let's create the source values.

```
>>> data = {
...     'id': 'acbd18db4cc2f85cedef654fccc4a4d8',
...     'count': 12,
... }
```

To serialize this dict, we will use the serialize function:

```
>>> b.serialize(data)
'\xab\xcd\x18\xdbL\xc2\xf8\\\xed\xefeO\xcc\xc4\xa4\xd8\xc0'
```

A shortcut for one-off serialization is the `serailize()` function:

```
>>> from btl import serialize
>>> serialize(message_format, data)
'\xab\xcd\x18\xdbL\xc2\xf8\\\xed\xefeO\xcc\xc4\xa4\xd8\xc0'
```

Note: In the input dict, keys that do not appear in the bitload description will simply be ignored. Keys that do appear in the description, but do not appear in the input dict with result in a `KeyError` exception.

3.3.2 Deserializing

Deserializing is equally simple as serializing:

```
>>> bitload = '\xab\xcd\x18\xdbL\xc2\xf8\\\xed\xefeO\xcc\xc4\xa4\xd8\xc0'
>>> b.deserialize(bitload)
OrderedDict([('id', 'abcd18db4cc2f85cedef654fccc4a4d8'), ('count', 12)])
```

We can see that the return value is an `OrderedDict` object. This is so that the order of the key is predictable, and in line with the bitload description. Unlike the plain dict, you can always depend on the order of the keys.

A shortcut for one-off deserialization is the *`deserialize()`* function:

```
>>> from btl import deserialize
>>> deserialize(message_format, bitload)
OrderedDict([('id', 'abcd18db4cc2f85cedef654fccc4a4d8'), ('count', 12)])
```

3.4 API documentation

This section provides the dry technical documentation of the pybtl library contents.

3.4.1 btl.bitload

This module contains the bitload class which is used to serialize and deserialize python values.

class `btl.bitload.`**`Bitload`**(*description*, *autopad=True*)

This class returns an object that is used to serialize and deserialize python values. The object is instantiated with the bitload description and maintains a map of the bitload layout. This map is used by its methods to insert or extract regions of the binary data that belong to individual fields.

The bitload layout is accessible through the `layout` property, which is an instance of `collections.OrderedDict`. The keys correspond to field names, and the values are `namedtuple` instances with the following attributes:

- `start`: starting index of the field
- `end`: index one after the final index of the field
- `length`: length of the field (in bits)
- `serializer`: function used for serializing the python value of the field
- `deserializer`: function used for deserializing the binary data into the python value of the field

These attributes are read-only and cannot be assigned to.

After the layout is processed, the object's `length` property is set to the total length of the bitload.

> **Warning:** The object's `length` property is *not* read-only, but you should not try to change its value unless you know exactly what you are doing.

By default, the length of the bitload is automatically padded to whole bytes. To disable the padding, we can pass `autopad=False` to the constructur, in which case the length of the bitload is a simple sum of lengths of the fields (incuding 'pad' fields).

deserialize(*bitload*)
Return a `OrderedDict` object that contains the data from the specified bitload. The `bitload` argument should be a bytestring.

If the bitload does not have the correct length, a `ValueError` exception is raised.

serialize(*data*)
Return a bytestring containing serialized and packed data. The `data` object must be a dictionary or a dict-like object that implements key access.

`btl.bitload.`**deserialize**(*description*, *bitload*, *padded=True*)
Return an `OrderedDict` object using the specified bitload description and bitload. This is a shortcut for instantiating a *`Bitload`* object and calling its *`deserialize()`* method.

The `padded` argument is used to specify whether the incoming bitload has been padded to whole bytes.

`btl.bitload.`**serialize**(*description*, *data*, *autopad=True*)
Return a serialized bytestring using the specified bitload description and data. This is a shortcut for instantiating a *`Bitload`* object and calling its *`serialize()`* method.

The `autopad` argument can be used to automatically pad the bitload to whole bytes.

3.4.2 btl.utils

This module contains lower-level functions for converting values to binary format and vice versa.

Note: The `bitarray.bitarray` instances passed to and returned from the functions in this module, are expected to be big-endian.

`btl.utils.`**bita_to_bool**(*b*)
Return a boolean value represented by the specified bitarray instance.

Example:

```
>>> bita_to_bool(bitarray('0'))
False
```

btl.utils.**bita_to_bytes**(*b*)
Return a bytestring that is represented by the specified bitarray instance.

Note: This function does exactly the same thing as calling the `tobytes()` method.

Example:

```
>>> ba = bitarray('011001100110111101101111')
>>> bita_to_bytes(ba)
'foo'
```

btl.utils.**bita_to_hex**(*b*)
Return a hex number represented by the specified bitarray instance.

Example:

```
>>> bita_to_hex(bitarray('1111000100101010'))
'f12a'
```

btl.utils.**bita_to_int**(*b*)
Return an integer that corresponds to the specified bitarray. The bitarray must represent a number that falls in the C long long range. Any significant bits beyond the 64 bits required to build a long long will be stripped away, and additional bits will be padded as needed to match the 64 bit length.

Example:

```
>>> bita_to_int(bitarray('1100'))
12
```

btl.utils.**bita_to_str**(*b*)
Return a string represented by the specified bitarray instance. The output is a unicode string. It is assumed that the input represents a UTF-8-encoded bytestring.

Example:

```
>>> bita = bitarray('011001100110111101101111')
>>> bita_to_str(bita)
u'foo'
```

btl.utils.**bool_to_bita**(*b*)
Return a bitarray instance representing the boolean value.

Example:

```
>>> bool_to_bita(True)
bitarray('1')
```

btl.utils.**bytes_to_bita**(*s*)
Return a bitarray instance representing the bytestring. There is no limit to the length of the input.

Note: This function does exactly the same thing as packing bits into a `bitarray` object using the `frombytes()` method.

Example:

```
>>> bytes_to_bita('foo')
bitarray('011001100110111101101111')
```

btl.utils.**hex_digit_to_bita**(*h*)

Return a 4-bit bitearray instance that represents a single hex digit.

Example:

```
>>> hex_digit_to_bita('a')
bitarray('1010')
```

btl.utils.**hex_to_bita**(*h*)

Return a bitarray instance representing the hex number in the input string. There is no limit to the length of the input.

Example:

```
>>> hex_to_bita('f12a')
bitarray('1111000100101010')
```

btl.utils.**int_to_bita**(*n*)

Return a bitarray instance representing the specified number. The number must be an usigned integer. The number must be in C long long range.

The return value will contain 64 bits (size of long long).

Example:

```
>>> int_to_bita(12)[-8:]
bitarray('00001100')
```

btl.utils.**str_to_bita**(*s*)

Return a bitarray instance representing the specified string. The input can be a unicode string or a bytestring. Unicode strings will be encoded as UTF-8.

Example:

```
>>> str_to_bita(u'foo')
bitarray('011001100110111101101111')
```

b

B

D

H

I

S