minform Documentation

Release 0.2.0

David Donna

November 22, 2015

Contents

CHAPTER 1

Examples

A simple BinaryForm might look like this:

```
import minform

class Person(minform.BinaryForm):
    '''
    This is a subclass of wtforms.Form: you can validate data with it,
    construct it from an HTML form, extract the data as a Python dict, etc.
    '''
    first_name = minform.BytesField('First Name', max_length=10)
    last_name = minform.BytesField('Last Name', max_length=10)
    age = minform.UInt8Field('Age')

#               first_name (10)          last_name (10)           age (1)
packed_data = b'David\x00\x00\x00\x00\x00Donna\x00\x00\x00\x00\x00\x18'
form = Person.unpack(packed_data)
assert form.data == {
    'first_name': b'David',
    'last_name': b'Donna',
    'age': 24,
}

next_form = Person(first_name=b'Foo', last_name=b'Barsson', age=100)
packed = next_form.pack()
assert packed == b'Foo\x00\x00\x00\x00\x00\x00\x00Barsson\x00\x00\x00\x64'
```

Compound BinaryFields allow you to create nested structures that still serialize into flat buffers.

```
class MyBigBadForm(minform.BinaryForm):
    """
    This is taking a turn for campy criminality.
    """
    riches = minform.Int16Field()
    goons = minform.BinaryFieldList(Person, max_entries=4, length=minform.EXPLICIT)

squad = MyBigBadForm(riches=55223, goons=[
    {'first_name': 'Joey', 'last_name': 'Schmoey', 'age': 32},
    {'first_name': 'Manny', 'last_name': 'The Man', 'age': 40},
    {'first_name': 'Gerta', 'last_name': 'Goethe', 'age': 52},
])
assert squad.pack() == (b'\xd7\xb7' +                                  # riches
                        b'\x03' +                                      # goons prefix
                        b'Joey\0\0\0\0\0\0Schmoey\0\0\0\x32' +         # goons[0]
```

```
                        b'Manny\0\0\0\0\0The Man\0\0\0\x40' +          # goons[1]
                        b'Gerta\0\0\0\0\0Goethe\0\0\0\0\x52' +         # goons[2]
                        b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0') # goons[3]
```

For more information, see the API doc.

CHAPTER 2

API

This documentation assumes that you're somewhat familiar with WTForms, since Minform is intentionally similar to (and substantively derived from) that project.

Minform provides a *BinaryForm* class, which subclasses from `wtforms.form.Form`. Instead of subclassing `Form` with `wtforms.fields.Field` instances as class variables, you need to subclass *BinaryForm*, and give it *BinaryItem* instances as class variables. The result will be a `Form` with additional *pack()* and *unpack()* methods.

Note: This documentation will often refer to `bytes` objects. This mostly applies to Python 3; if you're using Python 2, you can read `bytes` as `str`.

Python version	raw bytes type	unicode string type
Python 2	`str`	`unicode`
Python 3	`bytes`	`str`

2.1 Base Classes

class `minform.`**`BinaryForm`**(*formdata=None*, *obj=None*, *prefix=''*, *data=None*, *meta=None*, ***kwargs*)

Form with the power to serialize to and deserialize from packed bytes!

A *BinaryForm* is used much like a `wtforms.form.Form`. Instead of `wtforms.fields.Field` instances, however, the class members should be instances of *BinaryItem*.

When the class is created, the *BinaryItem* class members will be used, in order, to generate a binary protocol for serializing and deserializing instances of the form. Using the *BinaryForm* subclass's *unpack()* method will bind a form to the data represented by a buffer.

`size`
int

The number of bytes in a packed buffer of data for this class.

`order`
Byte ordering of numbers, etc. in corresponding buffers of packed data. See *Byte Order* for more.

`pack`(*order=None*)
Serialize this form's bound data into packed bytes.

Parameters **order** – *byte order* constant dictating the endianness of packed integers. *If* `self.order` *is set, this parameter will be ignored.*
Returns bytes object with length `self.size`
Return type bytes

pack_into(*buffer*, *offset*, *order=None*)
Pack data from this item into an existing buffer.
Parameters
- **buffer** – a mutable byte buffer (e.g. `bytearray`), into which the data will be written
- **offset** (*int*) – the starting index of *buffer* to write data to
- **data** – see `pack()`
- **order** – see `pack()`

classmethod unpack(*buffer*, *order=None*)
Parameters
- **buffer** (*bytes*) – bytes object of length `size`
- **order** – *byte order* constant for integer endianness. *If* `order` *is set, this parameter will be ignored.*

Returns form bound to the data stored in the buffer
Return type *BinaryForm*
Raises `ValueError` – if `buffer` has the wrong size.

classmethod unpack_from(*buffer*, *offset=0*, *order=None*)
Unpack data from a specific portion of a buffer.
Parameters
- **buffer** – a byte buffer (e.g. a `bytes` object) that contains the serialized data at some offset
- **offset** (*int*) – the index in *buffer* where the serialized data starts
- **order** – see `unpack()`

class `minform.`**BinaryItem**
Item that occupies a block of bytes in a `BinaryForm`

A number of `BinaryItem` subclasses have already been provided; see *items* for more.

size
The number of bytes that will be used to store the item when the parent form is packed in a buffer.

Note: If you subclass `BinaryItem`, you need to ensure that the object will have an appropriate `size` property, since it is used by the form to split up buffer data for unpacking, and to assembled packed data.

---

form_field
This property is optional; for example, `BlankBytes` instances do not have a `form_field`. If present, it should be an instance of `wtforms.Field`. This field will then become a member of the form, just like a field in a `wtforms.form.Form`.

order
byte order constant that will override the order of the containing form or field. This will only be necessary if you need to serialize/deserialize with mixed byte ordering.

pack(*data*, *order=None*)
Serialize a chunk of data into packed bytes.
Parameters
- **data** – data to serialize, e.g. stored by a corresponding form field

- **order** – *byte order* constant dictating the endianness of packed integers. *If* `self.order` *is set, this parameter will be ignored.*

Returns bytes object with length `size`

Return type bytes

pack_into(*buffer*, *offset*, *data*, *order=None*)

Pack data from this item into an existing buffer.

Parameters

- **buffer** – a mutable byte buffer (e.g. `bytearray`), into which the data will be written
- **offset** (*int*) – the starting index of *buffer* to write data to
- **data** – see `pack()`
- **order** – see `pack()`

unpack(*buffer*, *order=None*)

Deserialize packed bytes into data.

Parameters

- **buffer** – bytes object of length `size`
- **order** – *byte order* constant for integer endianness. *If* `self.order` *is set, this parameter will be ignored.*

Returns data stored in the buffer

Raises `ValueError` – if *buffer* has the wrong size.

unpack_from(*buffer*, *offset=0*, *order=None*)

Unpack data from a specific portion of a buffer.

Parameters

- **buffer** – a byte buffer (e.g. a `bytes` object) that contains the serialized data at some offset
- **offset** (*int*) – the index in *buffer* where the serialized data starts
- **order** – see `unpack()`

2.2 Binary Items

2.2.1 Blank Bytes

class `minform.`**BlankBytes**(*size*)

Add padding to a form when serialized.

The *size* argument will set the `size`.

A `BlankBytes` instance can be placed anywhere in the list of fields in a `BinaryForm` definition. It doesn't matter what name you give it; when the form's fields are processed, the `BlankBytes` object itself will be removed from the class's namespace.

The corresponding bytes will be null when the form is packed, and ignored when a data buffer is unpacked. Likewise, the bytes in a packed buffer will be ignored, and unpacking blank bytes will always return `None`.

Because `BlankBytes` objects lack a `form_field` attribute, there will be no corresponding attribute in a parent `BinaryForm`'s data.

2.2.2 Basic Binary Fields

class `minform.`**BinaryField**

`BinaryItem` that corresponds to a form field.

Note: This class should not be instantiated directly. Instead, use one of its subclasses, described below.

The following classes all have *form_field* attributes, and their constructors accept a superset of the construction parameters for a wtforms.fields.Field. In general, constructor arguments whose names correspond to *BinaryItem* construction parameters will be passed in to the constructor for the corresponding wtforms.fields.Field. So, for example, you can set a label for HTML rendering, or add extra validators.

The only notable exceptions are the *order* and *length* parameters, which are used to set the *byte order* and *length policy*, and will not be passed through to the Field.

class minform.**BinaryBooleanField**(*label=''*, *validators=None*, *order=None*, ***kwargs*)
Store either True or False as b'\x01' or b'\x00' (respectively).

size
always 1

form_field
A wtforms.fields.BooleanField instance.

class minform.**CharField**(*label=''*, *validators=None*, *order=None*, ***kwargs*)
Store a single byte as a one-character str (in Python 2) or bytes object (in Python 3).

size
always 1

form_field
A wtforms.fields.StringField instance.

class minform.**BinaryIntegerField**(*label=''*, *validators=None*, *order=None*, ***kwargs*)
This class should not be instantiated directly; instead, you should use one of its subclasses, which determine what kind of int is stored, and how. Those subclasses are:

Name	size	Min	Max
Int8Field	1	-128	127
UInt8Field	1	0	255
Int16Field	2	-32768	32767
UInt16Field	2	0	65535
Int32Field	4	-2^{31}	$2^{31} - 1$
UInt32Field	4	0	$2^{32} - 1$
Int64Field	8	-2^{63}	$2^{63} - 1$
UInt64Field	8	0	$2^{64} - 1$

form_field
A wtforms.fields.Integerfield instance.

class minform.**Float32Field**(*label=''*, *validators=None*, *order=None*, ***kwargs*)
Store a float in four bytes.

size
Always 4.

form_field
A wtforms.fields.FloatField instance.

class minform.**Float64Field**(*label=''*, *validators=None*, *order=None*, ***kwargs*)
Store a float in eight bytes.

size
Always 8.

form_field
A wtforms.fields.FloatField instance.

class minform.**BytesField**(*label=''*, *validators=None*, *max_length=None*, *length='automatic'*, *order=None*, ***kwargs*)
Store *N* bytes.

max_length
Maximum number of bytes in the stored string. Note that this may not be equal to *size*.

size
The *size* of a *BytesField* with max_length *N* varies based on the *length* argument used to construct it.

If *length* is *FIXED* or *AUTOMATIC*, size will be *N*.

If *length* is *EXPLICIT*, there will be one or more extra bytes at the beginning of the packed data, which store the number of bytes used by the string. This will be the smallest number of bytes needed to store a number up to max_length. So, size can be *N+1*, *N+2*, *N+4*, or *N+8*. (For more information, see the documentation for *EXPLICIT*.)

form_field
A wtforms.fields.StringField instance.

2.2.3 Compound Binary Fields

These fields allow data to be nested. If your data may include several items with the same type, you can use a *BinaryFieldList* to manage them. If you want to re-use a set of items (or nest a more complicated data type in a *BinaryFieldList*), you can use a *BinaryFormField* to do so.

class minform.**BinaryFieldList**(*inner_field*, *label=''*, *validators=None*, *max_entries=None*, *length='explicit'*, *order=None*, ***kwargs*)
Store a homogeneous list of information.

inner_field
A BinaryField instance.

max_entries
The maximum number of items that can be stored in the list.

length
A *Length* constant.

size
If *length* is *minform.FIXED*, *size* will be equal to max_size * inner_field.length.

If *length* is *minform.EXPLICIT*, *size* will be prefix_length + (max_size * inner_field.length). The value of prefix_length follows the documentation for *Length*.

form_field
A wtforms.fields.FieldList instance.

class minform.**BinaryFormField**(*form_class*, *label=''*, *validators=None*, *order=None*, ***kwargs*)
Nest one *BinaryForm* inside another.

form_class
The *BinaryForm* subclass that describes the contents of this field. A *BinaryFormField* instance will have the same *size* as its *form_class*, and will pack and unpack data in the same ways.

form_field
A `wtforms.fields.FormField` instance.

2.2.4 Custom BinaryItems

When creating a custom *BinaryItem*, you need to be sure to include:

- A *size* attribute. This is used to determine how many bytes will be required by the *unpack()* method, and how many will be expected to be returned by the *pack()* method. This attribute is required even if you write custom *pack()* and *BinaryItem.unpack()* methods that don't refer to it!
- A *pack()* method. The type of *data* should be compatible with the type returned by the *unpack()* method (below).
- An *unpack()* method. You can expect *buf* to have *self.size* bytes when the method is invoked in the course of using a *BinaryForm*.

2.3 Length

The following constants are used as the *length* argument when constructing a *BytesField* or a *BinaryFieldList*; they control whether and how the packed buffer signals the length of the data.

`minform.`**FIXED**
If the length is `FIXED`, all of the packed information, including terminal null bytes, will be considered part of the data.

```
    fixed_bytes = BytesField(max_length=6, length=FIXED)
    fixed_bytes.unpack(b'foobar\0\0\0\0') == b'foobar\0\0\0\0'

    fixed_list = BinaryFieldList(UInt16Field(), max_entries=4, length=FIXED)
    fixed_list.unpack(b'\x12\x34\x56\x78\x9a\x00\x00\x00') == \
        [0x1234, 0x5678, 0x9a00, 0x0000]
```

`minform.`**EXPLICIT**
If length is *EXPLICIT*, the packed buffer will start with an unsigned int that gives the length of the data (the number of bytes in a *BytesField*, or the number of entries in a *BinaryFieldList*). This prefix will be sized according to necessity; it will always be big enough to store the *max_length* or *max_entries* of the field:

maximum	prefix type	prefix size
up to 255	UInt8	1 byte
256 - 65535	UInt16	2 bytes
65535 - 4294967296	UInt32	4 bytes
larger	UInt64	8 bytes

If the max is larger than 2^{64}, a `ValueError` will be thrown. Here are some examples of the use of *EXPLICIT* length fields:

```
    explicit_bytes = BytesField(max_length=9, length=EXPLICIT)

    # The first byte is the length of the string.
    explicit_bytes.pack(b'foobar') == b'\x06foobar\0\0\0'

    # If you manually include the null bytes, they'll be preserved.
    explicit_bytes.pack(b'foo\0\0\0') == b'\x06foo\0\0\0\0\0\0'

    # The unpacking process respects the explicit size given.
    explicit_bytes.unpack(b'\x05hey\0\0\0\0\0\0') == b'hey\0\0'
    explicit_bytes.unpack(b'\x02hey\0\0\0\0\0\0') == b'he'

    explicit_list = BinaryFieldList(UInt16Field, max_entries=4,
                                    length=EXPLICIT)

    explicit_list.pack([0x1234, 0x5678, 0x9abc, 0xdef0]) == \
        b'\x04\x12\x34\x56\x78\x9a\xbc\xde\xf0'
    explicit_list.pack([0x1234, 0x5678]) == b'\x02\x12\x34\x56\x78\0\0\0\0'
```

minform.**AUTOMATIC**

The *AUTOMATIC* option is only available for *BytesField*, and has very simple semantics: strings shorter than *max_length* will be padded with null bytes when packed, and null bytes will be trimmed from the end when unpacking a buffer.

```
    auto_bytes = BytesField(max_length=10, length=AUTOMATIC)

    auto_bytes.pack(b'1234554321') == b'1234554321'
    auto_bytes.pack(b'foobar') == b'foobar\0\0\0\0'

    auto_bytes.unpack(b'abc\0def\0\0\0') == b'abc\0def'
```

2.4 Byte order

minform.**NATIVE**

minform.**LITTLE_ENDIAN**

minform.**BIG_ENDIAN**

minform.**NETWORK**

These constants operate according to the byte order constants from the struct module. The *minform.NATIVE* constant corresponds to the `'='` prefix, rather than `'@'`.

Note: Setting the *order* property on a *BinaryForm* or *BinaryItem* will override the *order* argument of *pack()* and *unpack()* methods. For clarity, we recommend that you use **either** the attribute **or** the *pack()*/*unpack()* argument.

Likewise, the *order* of a *BinaryItem* will override the *order* of the form or field that contains it.

You can think of it as the order cascading down from the *BinaryForm.unpack* *order* argument, through the class, to each of that form's items, and easy nested item, until it is overridden by an *order* attribute.

CHAPTER 3

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.1 Types of Contributions

3.1.1 Report Bugs

Report bugs at https://github.com/daviddonna/minform/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

3.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

3.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

3.1.4 Write Documentation

Minform could always use more documentation, whether as part of the official Minform docs, in docstrings, or even on the web in blog posts, articles, and such.

3.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/daviddonna/minform/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.2 Get Started!

Ready to contribute? Here's how to set up *minform* for local development.

1. Fork the *minform* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/minform.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv minform
$ cd minform/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 minform tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4. Check https://travis-ci.org/daviddonna/minform/pull_requests and make sure that the tests pass for all supported Python versions.

3.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_core
```

CHAPTER 4

Credits

4.1 Development Lead

- David Donna <davidadonna@gmail.com>

4.2 Contributors

None yet. Why not be the first?

4.3 Miscellaneous

This package was built on @audreyr's marvellous cookiecutter-pypackage template.

This package is available on github, at https://github.com/daviddonna/minform.

CHAPTER 5

Installation

At the command line:

```
$ easy_install minform
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv minform
$ pip install minform
```

m