EULcore Documentation

Release 0.15.0

Emory University Libraries

October 14, 2011

CONTENTS

EULcommon is one of a collection of reusable Python components from Emory University Libraries. The library contains both released and unreleased beta components. Except where noted otherwise, components documented here are released and ready for production use.

CHAPTER

ONE

CONTENTS

1.1 Change & Version Information

The following is a summary of changes and improvements to `eulcommon`. New features in each version should be listed, with any necessary information about installation or upgrade notes.

1.1.1 Initial Release

- Split out and re-organized common, useful components (`binfile`, `djangoextras`) from `eulcore` into `eulcommon` for easier re-use.

1.2 `eulcore.binfile` – Map binary data to Python objects

Map binary data on-disk to read-only Python objects.

This module facilitates exposing stored binary data using common Pythonic idioms. Fields in relocatable binary objects map to Python attributes using a priori knowledge about how the binary structure is organized. This is akin to the standard `struct` module, but with some slightly different use cases. `struct`, for instance, offers a more terse syntax, which is handy for certain simple structures. `struct` is also a bit faster since it's implemented in C. This module's more verbose `BinaryStructure` definitions give it a few advantages over `struct`, though:

- This module allows users to define their own field types, where `struct` field types are basically inextensible.
- The object-based nature of `BinaryStructure` makes it easy to add non-structural properties and methods to subclasses, which would require a bit of reimplementing and wrapping from a `struct` tuple.
- `BinaryStructure` instances access fields through named properties instead of indexed tuples. `struct` tuples are fine for structures a few fields long, but when a packed binary structure grows to dozens of fields, navigating its `struct` tuple grows perilous.
- `BinaryStructure` unpacks fields only when they're accessed, allowing us to define libraries of structures scores of fields long, understanding that any particular application might access only one or two of them.
- Fields in a `BinaryStructure` can overlap eachother, greatly simplifying both C unions and fields with multiple interpretations (integer/string, signed/unsigned).
- This module makes sparse structures easy. If you're reverse-engineering a large binary structure and discover a 4-byte integer in the middle of 68 bytes of unidentified mess, this module makes it easy to add an `IntegerField` at a known structure offset. `struct` requires you to split your `'68x'` into a `'32xI32x'` (or was that a `'30xi34x'`? Better recount.)

This package exports the following names:

- `BinaryStructure` – a base class for binary data structures
- `ByteField` – a field that maps fixed-length binary data to Python strings
- `LengthPrependedStringField` – a field that maps variable-length binary strings to Python strings
- `IntegerField` – a field that maps fixed-length binary data to Python numbers

1.2.1 `BinaryStructure` Subclasses

`eulcommon.binfile.eudora` – Eudora email index files

Map binary email table of contents files for the Eudora mail client to Python objects.

The Eudora email client has a long history through the early years of email. It supported versions for early Mac systems as well as early Windows OSes. Unfortuntely, most of them use binary file formats that are entirely incompatible with one another. This module is aimed at one day reading all of them, but for now practicality and immediate needs demand that it focus on the files saved by a particular version on mid-90s Mac System 7.

That Eudora version stores email in flat (non-hierarchical) folders. It stores each folder's email data in a single file akin to a Unix mbox file, but with some key differences, described below. In addition to this folder data file, each folder also stores a binary "table of contents" index. In this version, a folder called `In` stores its index in a file called `In.toc`. This file consists of a fixed-size binary header with folder metadata, followed by fixed-size binary email records containing cached email header metadata as well as the location of the full email in the mbox-like data file. As the contents of the folder are updated, these fixed-size binary email records are added, removed, and reordered, apparently compacting the file as necessary so that it matches the folder contents displayed to the application end user.

With the index serving to dictate the order of the emails and their contents, their locations and sizes inside the data storage file become less important. When emails are deleted from a folder, the index is updated, but they are not removed immediately from the data file. Instead that data space is marked as inactive and might be reused later when a new email is added to the folder. As a result, the folder data file may contain stale and out-of-order data and thus **cannot be read directly as a standard mbox file**.

This module, then, provides classes for parsing the binary structures of the index file and mapping them to Python objects. This binary file has gone through many formats. Only one is represented in this module, though it could certainly be expanded to support more. Parsers and information about other versions of the index file are available at http://eudora2unix.sourceforge.net/ and http://users.starpower.net/ksimler/eudora/toc.html; these were immensely helpful in reverse-engineering the version represented by this module.

This module exports the following names:

- `Toc` – a `BinaryStructure` for the index file header
- `Message` – a `BinaryStructure` for the fixed-length email metadata entries in the index files

class `eulcommon.binfile.eudora.`**`Message`**(*fobj=None*, *mm=None*, *offset=0*)

A `BinaryStructure` for a single email's metadata cached in the index file.

Only a few fields are currently represented; other fields contain interesting data but have not yet been reverse-engineered.

`LENGTH = 220`

the size of a single message header

`body_offset`

the offset of the body within the raw email data

`date`

a date value copied from email headers

offset
the offset of the raw email data in the folder data file

priority
some kind of unspecified single-byte priority field

size
the size of the raw email data in the folder data file

status
some kind of unspecified single-byte status field

subject
the email subject copied from email headers

to
a recipient copied from email headers

class eulcommon.binfile.eudora.**Toc** (*fobj=None*, *mm=None*, *offset=0*)
A BinaryStructure for an email folder index header.

Only a few fields are currently represented; other fields contain interesting data but have not yet been reverse-engineered.

LENGTH = 278
the size of this binary header

messages
a generator yielding the Message structures in the index

name
the user-displayed folder name, e.g., "In" for the default inbox

version
the file format version

1.2.2 General Usage

Suppose we have an 8-byte file whose binary data consists of the bytes 0, 1, 2, 3, etc.:

```
>>> with open('numbers.bin') as f:
...     f.read()
...
'\x00\x01\x02\x03\x04\x05\x06\x07'
```

Suppose further that these contents represent sensible binary data, laid out such that the first two bytes are a literal string value. Except that sometimes, in the binary format we're parsing, it might sometimes be necessary to interpret those first two bytes not as a literal string, but instead as a number, encoded as a big-endian unsigned integer. Following that is a variable-length string, encoded with the total string length in the third byte.

This structure might be represented as:

```
from eulcommon.binfile import *
class MyObject(BinaryStructure):
    mybytes = ByteField(0, 2)
    myint = IntegerField(0, 2)
    mystring = LengthPrepededStringField(2)
```

Client code might then read data from that file:

```
>>> f = open('numbers.bin')
>>> obj = MyObject(f)
>>> obj.mybytes
'\x00\x01'
>>> obj.myint
1
>>> obj.mystring
'\x03\x04'
```

It's not uncommon for such binary structures to be repeated at different points within a file. Consider if we overlay the same structure on the same file, but starting at byte 1 instead of byte 0:

```
>>> f = open('numbers.bin')
>>> obj = MyObject(f, offset=1)
>>> obj.mybytes
'\x01\x02'
>>> obj.myint
258
>>> obj.mystring
'\x04\x05\x06'
```

1.2.3 BinaryStructure

class eulcommon.binfile.**BinaryStructure**(*fobj=None*, *mm=None*, *offset=0*)

A superclass for binary data structures superimposed over files.

Typical users will create a subclass containing field objects (e.g., ByteField, IntegerField). Each subclass instance is created with a file and with an optional offset into that file. When code accesses fields on the instance, they are calculated from the underlying binary file data.

Instead of a file, it is occasionally appropriate to overlay an mmap structure (from the mmap standard library). This happens most often when one BinaryStructure instance creates another, passing self.mmap to the secondary object's constructor. In this case, the caller may specify the *mm* argument instead of an *fobj*.

Parameters

- **fobj** – a file object or filename to overlay
- **mm** – a mmap object to overlay
- **offset** – the offset into the file where the structured data begins

1.2.4 Field classes

class eulcommon.binfile.**ByteField**(*start*, *end*)

A field mapping fixed-length binary data to Python strings.

Parameters

- **start** – The offset into the structure of the beginning of the byte data.
- **end** – The offset into the structure of the end of the byte data. This is actually one past the last byte of data, so a four-byte ByteField starting at index 4 would be defined as ByteField(4, 8) and would include bytes 4, 5, 6, and 7 of the binary structure.

Typical users will create a *ByteField* inside a BinaryStructure subclass definition:

```
class MyObject(BinaryStructure):
    myfield = ByteField(0, 4) # the first 4 bytes of the file
```

When you instantiate the subclass and access the field, its value will be the literal bytes at that location in the structure:

```
>>> o = MyObject('file.bin')
>>> o.myfield
'ABCD'
```

class eulcommon.binfile.**LengthPrependedStringField**(*offset*)
A field mapping variable-length binary strings to Python strings.

This field accesses strings encoded with their length in their first byte and string data following that byte.

Parameters **offset** – The offset of the single-byte string length.

Typical users will create a `LengthPrependedStringField` inside a `BinaryStructure` subclass definition:

```
class MyObject(BinaryStructure):
    myfield = LengthPrependedStringField(0)
```

When you instantiate the subclass and access the field, its length will be read from that location in the structure, and its data will be the bytes immediately following it. So with a file whose first bytes are `'\x04ABCD'`:

```
>>> o = MyObject('file.bin')
>>> o.myfield
'ABCD'
```

class eulcommon.binfile.**IntegerField**(*start*, *end*)
A field mapping fixed-length binary data to Python numbers.

This field accessses arbitrary-length integers encoded as binary data. Currently only big-endian, unsigned integers are supported.

Parameters

- **start** – The offset into the structure of the beginning of the byte data.
- **end** – The offset into the structure of the end of the byte data. This is actually one past the last byte of data, so a four-byte `IntegerField` starting at index 4 would be defined as `IntegerField(4, 8)` and would include bytes 4, 5, 6, and 7 of the binary structure.

Typical users will create an *IntegerField* inside a `BinaryStructure` subclass definition:

```
class MyObject(BinaryStructure):
    myfield = IntegerField(3, 6) # integer encoded in bytes 3, 4, 5
```

When you instantiate the subclass and access the field, its value will be big-endian unsigned integer encoded at that location in the structure. So with a file whose bytes 3, 4, and 5 are `'\x00\x01\x04'`:

```
>>> o = MyObject('file.bin')
>>> o.myfield
260
```

1.3 eulcommon.djangoextras – Extensions and additions to django

1.3.1 auth - Customized permission decorators

Customized decorators that enhance the default behavior of `django.contrib.auth.decorators.permission_required(`

The default behavior of `django.contrib.auth.decorators.permission_required()` for any user does not meet the required permission level is to redirect them to the login page– even if that user is already logged in. For more discussion of this behavior and current status in Django, see: http://code.djangoproject.com/ticket/4617

These decorators work the same way as the Django equivalents, with the added feature that if the user is already logged in and does not have the required permission, they will see 403 page instead of the login page.

The decorators should be used exactly the same as their django equivalents.

The code is based on the django snippet code at http://djangosnippets.org/snippets/254/

static `auth.`**`user_passes_test_with_403`**(*test_func*, *login_url=None*)
View decorator that checks to see if the user passes the specified test. See `django.contrib.auth.decorators.user_passes_test()`.

Anonymous users will be redirected to login_url, while logged in users that fail the test will be given a 403 error. In the case of a 403, the function will render the **403.html** template.

static `auth.`**`permission_required_with_403`**(*perm*, *login_url=None*)
Decorator for views that checks whether a user has a particular permission enabled, redirecting to the login page or rendering a 403 as necessary.

See `django.contrib.auth.decorators.permission_required()`.

static `auth.`**`user_passes_test_with_ajax`**(*test_func*, *login_url=None*, *redirect_field_name='next'*)
Decorator for views that checks that the user passes the given test, redirecting to the log-in page if necessary. The test should be a callable that takes the user object and returns True if the user passes.

Returns special response to ajax calls instead of blindly redirecting.

To use with class methods instead of functions, use `django.utils.decorators.method_decorator()`. See http://docs.djangoproject.com/en/dev/releases/1.2/#user-passes-test-login-required-and-permission-required

static `auth.`**`login_required_with_ajax`**(*function=None*, *redirect_field_name='next'*)
Decorator for views that checks that the user is logged in, redirecting to the log-in page if necessary, but returns a special response for ajax requests. See `eulcore.django.auth.decorators.user_passes_test_with_ajax()`.

static `auth.`**`permission_required_with_ajax`**(*perm*, *login_url=None*)
Decorator for views that checks whether a user has a particular permission enabled, redirecting to the log-in page if necessary, but returns a special response for ajax requests. See `eulcore.django.auth.decorators.user_passes_test_with_ajax()`.

`formfields` - Custom form fields & widgets

Custom generic form fields for use with Django forms.

---

class `eulcommon.djangoextras.formfields.`**`W3CDateField`**(*max_length=None*, *min_length=None*, **args*, ***kwargs*)
W3C date field that uses a `W3CDateWidget` for presentation and uses a simple regular expression to do basic validation on the input (but does not actually test that it is a valid date).

`widget`
alias of `W3CDateWidget`

class eulcommon.djangoextras.formfields.**W3CDateWidget**(*attrs=None*)
Multi-part date widget that generates three text input boxes for year, month, and day. Expects and generates dates in any of these W3C formats, depending on which fields are filled in: YYYY-MM-DD, YYYY-MM, or YYYY.

create_textinput(*name*, *field*, *value*, ***extra_attrs*)
Generate and render a `django.forms.widgets.TextInput` for a single year, month, or day input.

If size is specified in the extra attributes, it will also be used to set the maximum length of the field.

Parameters

- **name** – base name of the input field
- **field** – pattern for this field (used with name to generate input name)
- **value** – initial value for the field
- **extra_attrs** – any extra widget attributes

Returns rendered HTML output for the text input

render(*name*, *value*, *attrs=None*)
Render the widget as HTML inputs for display on a form.

Parameters

- **name** – form field base name
- **value** – date value
- **attrs** –
  - unused

Returns HTML text with three inputs for year/month/day

value_from_datadict(*data*, *files*, *name*)
Generate a single value from multi-part form data. Constructs a W3C date based on values that are set, leaving out day and month if they are not present.

Parameters

- **data** – dictionary of data submitted by the form
- **files** –
  - unused
- **name** – base name of the form field

Returns string value

class eulcommon.djangoextras.formfields.**DynamicChoiceField**(*choices=None*, *widget=None*, **args*, ***kwargs*)
A `django.forms.ChoiceField` whose choices are not static, but instead generated dynamically when referenced.

Parameters **choices** – callable; this will be called to generate choices each time they are referenced

widget
alias of `DynamicSelect`

class eulcommon.djangoextras.formfields.**DynamicSelect**(*attrs=None*, *choices=None*)
A `Select` widget whose choices are not static, but instead generated dynamically when referenced.

Parameters **choices** – callable; this will be called to generate choices each time they are referenced.

1.3.2 `http` - Content Negotiation for Django views

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

b

d