tinyobj Documentation

Release 0.1.0

Brian Hicks

February 24, 2014

Contents

a tiny dict -> object mapper

CHAPTER 1

Features

- TODO

1.1 Contents:

1.1.1 Installation

At the command line either via easy_install or pip:

```
$ easy_install tinyobj
$ pip install tinyobj
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv tinyobj
$ pip install tinyobj
```

1.1.2 Usage

Say you have a dictionary that looks sort of like this:

```
{
    'username': 'rabbit',
    'password': 'some-hash',
    'active': True
}
```

you'd define a schema like so:

```
    from tinyobj import TinyObj, fields

class User(TinyObj):
    username = fields.TextField()
    password = fields.TextField()
    active = fields.BoolField()
```

and then initialize it:

```
user = User(username='rabbit', password='some-hash', active=True)

# or
```

```
user = User(doc_from_db)

assert user.username == 'rabbit'
assert user.password == 'some-hash'
assert user.active == True
```

You can get a dictionary of fields back (for saving) with `to_dict`:

```
assert user.to_dict() == doc_from_db
```

1.1.3 Fields

Fields are the validation/cleaning mechanic of **tinyobj**. Each is responsible for receiving a value (from the database, for example), cleaning it, and returning the cleaned value. A reference to the original value is not kept at this time, so reserializing the data for your specific use case is left as an exercise to the reader.

The base object is `Field`, of which `TinyObj` will detect subclasses to use as fields:

class `tinyobj.fields.`**`Field`**

base for other fields

`clean`(*value*)

clean a value, returning the cleaned value

`initialize`(*value=()*)

initialize returns a cleaned value or the default, raising ValueErrors as necessary.

Subclasses

tinyobj implements a number of fields to do validation, etc.

class `tinyobj.fields.`**`NumberField`**(*t=<type 'float'>*, *allow_negative=True*, *allow_positive=True*)

accept and validate numbers

takes a type to convert values to, can be (EG) `float`, `int`, `long`, or `complex`.

`clean`(*value*)

clean a value, converting and performing bounds checking

class `tinyobj.fields.`**`BoolField`**(*default=False*)

accept and validate boolean values

note that this field will just call `bool` on values, this may not be your desired behavior so you might want to implement a subclass that parses truthy/falsey values in a way specific to your application

class `tinyobj.fields.`**`TextField`**

accept and validate text.

Uses the Python implementation's appropriate unicode value (IE `unicode` on 2.x and `str` on 3.x)

class `tinyobj.fields.`**`NoValidationField`**

doesn't validate at all, but returns the value passed (defaulting to None)

1.1.4 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:

Types of Contributions

Report Bugs

Report bugs at https://github.com/BrianHicks/tinyobj/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.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/BrianHicks/tinyobj/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 :)

Get Started!

Ready to contribute? Here’s how to set up *tinyobj* for local development.

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

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

3. Create a branch for local development:

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

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

   ```
   $ tox
   ```

To get tox, just pip install it.

5. 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
   ```

6. Submit a pull request through the GitHub website.

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, and 3.3, and for PyPy. Check https://travis-ci.org/BrianHicks/tinyobj under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test test/test_tinyobj.py
```

1.1.5 Credits

Development Lead

- Brian Hicks <brian@brianthicks.com>

Contributors

None yet. Why not be the first?

1.1.6 History

0.1.0 (2014-02-24)

- First release on PyPI.

1.2 Feedback

If you have any suggestions or questions about **tinyobj** feel free to email me at brian@brianthicks.com.

If you encounter any errors or problems with **tinyobj**, please let me know! Open an Issue at the GitHub https://github.com/BrianHicks/tinyobj main repository.