nani Documentation

Release 0.2.0

Christopher Crouzet

January 19, 2017

Contents

Welcome! If you are just getting started, a recommended first read is the *Overview* as it shortly covers the *why*, *what*, and *how*'s of this library. From there, the *Installation* then the *Tutorial* sections should get you up to speed with the basics required to use it.

Looking how to use a specific function, class, or method? The whole public interface is described in the *API Reference* section.

Please report bugs and suggestions on GitHub.

CHAPTER 1

User's Guide

1.1 Overview

Upon getting started with NumPy, the rules to define `numpy.dtype` objects tend to quickly become *confusing*. Not only different syntaxes can create a same data type, but it also seems *arbitrary* and hence *difficult* to remember that sub-array data types can only be defined as tuples while structured data types exclusively require lists made of field tuples, and so on.

To address this point, Nani takes the stance of offering one—and only one—way of constructing `numpy.dtype` objects. Although this syntax makes the code more verbose, it also makes it easier to read and to reason about.

Nani's approach allows **type introspection** which brings additional benefits in the form of dynamically generated **default values** and **view types**. Default values facilitate the definition of new array elements while view types are useful to encapsulate interactions with NumPy and to expose a different public interface to your library users instead of the one provided with `numpy.ndarray`.

1.1.1 Features

- explicit syntax for defining `numpy.dtype` objects.
- generates default values and view types.
- allows for type introspection.

1.1.2 Usage

```
>>> import numpy
>>> import nani
>>> color_type = nani.Array(
...     element_type=nani.Number(type=numpy.uint8, default=255),
...     shape=3,
...     view=None)
>>> dtype, default, view = nani.resolve(color_type, name='Color')
>>> a = numpy.array([default] * 2, dtype=dtype)
>>> v = view(a)
>>> for color in v:
...     print(color)
[255, 255, 255]
[255, 255, 255]
```

The `color_type` above defines an array of 3 `numpy.uint8` elements having each a default value of `255`. The resulting `dtype` and `default` objects are used to initialize a new NumPy array of 10 color elements, while the `view` type is used to wrap that array into a standard collection interface.

See also:

The *Tutorial* section for more detailed examples and explanations on how to use Nani.

1.2 Installation

Nani is written in the Python language and hence requires a Python interpreter. Any of the following Python versions is supported: 2.7, 3.3, 3.4, 3.5, and 3.6. On top of that, Nani also depends the `numpy` package.

Note: Package dependencies are automatically being taken care off when using `pip`.

1.2.1 Installing pip

The recommended [1] approach for installing a Python package such as Nani is to use `pip`, a package manager for projects written in Python. If `pip` is not already installed on your system, you can do so by following these steps:

1. Download get-pip.py.
2. Run `python get-pip.py` in a shell.

Note: The installation commands described in this page might require `sudo` privileges to run successfully.

1.2.2 System-Wide Installation

Installing globally the most recent version of Nani and its dependencies can be done with `pip`:

```
$ pip install nani
```

Or using `easy_install` (provided with `setuptools`):

```
$ easy_install nani
```

1.2.3 Virtualenv

If you'd rather make Nani and its dependencies only available for your specific project, an alternative approach is to use `virtualenv`. First, make sure that it is installed:

```
$ pip install virtualenv
```

Then, an isolated environment needs to be created for your project before installing Nani in there:

[1] See the Python Packaging User Guide

```
$ mkdir myproject
$ cd myproject
$ virtualenv env
New python executable in /path/to/myproject/env/bin/python
Installing setuptools, pip, wheel...done.
$ source env/bin/activate
$ pip install nani
```

At this point, Nani is available for the project `myproject` as long as the virtual environment is activated.

To exit the virtual environment, run:

```
$ deactivate
```

Note: Instead of having to activate the virtual environment, it is also possible to directly use the `env/bin/python`, `env/bin/pip`, and the other executables found in the folder `env/bin`.

Note: For Windows, some code samples might not work out of the box. Mainly, activating `virtualenv` is done by running the command `env\Scripts\activate` instead.

1.2.4 Development Version

To stay cutting edge with the latest development progresses, it is possible to directly retrieve the source from the repository with the help of Git:

```
$ git clone https://github.com/christophercrouzet/nani.git
$ cd nani
$ pip install --editable .[dev]
```

Note: The `[dev]` part installs additional dependencies required to assist development on Nani.

1.3 Tutorial

Creating an array in NumPy requires to provide a `dtype` describing the data type for each element of the array. With Nani, a more explicit syntax is used to define the data type, as well as other properties such as the default values and the view types.

As a result, creating a NumPy array through Nani requires an additional step:

- describe a NumPy array's `dtype` with Nani, using the *data types* provided, such as *`Number`*, *`Array`*, *`Structure`*, and so on.
- resolve Nani's data type into a format compatible with NumPy, using the function *`resolve()`*.
- use the resolved properties to create the NumPy array through the usual `numpy`'s API, and to optionally offer an abstraction layer around it.

1.3.1 Flat Array of Integers

```
>>> import numpy
>>> import nani
>>> data_type = nani.Number(type=numpy.int32)
>>> dtype, default, view = nani.resolve(data_type)
>>> a = numpy.arange(15, dtype=dtype)
>>> a
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
>>> type(a)
<type 'numpy.ndarray'>
>>> v = view(a)
>>> v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> type(v)
<class 'nani.ArrayView'>
```

This example is the simplest usage possible, making it a good start to understand what's going on.

Firstly, an integral data type using *Number* is defined. This is describing the type of each element of the NumPy array that will be created. By default, *Number* has a `type` value set to `numpy.float_`, which needs to be overriden here to describe an integer instead.

Then the *resolve()* function returns, amongst other properties, a NumPy `dtype` which is directly used to initialize the NumPy array.

The view generated by *resolve()* can be used to wrap the whole NumPy array. Here it is nothing more than a simple emulation of a Python container [1]: it has a length, it is iterable, and it can be queried for membership using the `in` keyword. Of course, it is possible to provide a different interface.

1.3.2 Array of Vector2-like Elements

```
>>> import numpy
>>> import nani
>>> vector2_type = nani.Array(
...     element_type=nani.Number(),
...     shape=2,
...     name='Vector2')
>>> dtype, default, view = nani.resolve(vector2_type, name='Positions')
>>> a = numpy.zeros(3, dtype=dtype)
>>> v = view(a)
>>> for i, position in enumerate(v):
...     position[0] = i + 1
...     position[1] = i + 2
>>> v
[[1.0, 2.0], [2.0, 3.0], [3.0, 4.0]]
>>> type(v)
<class 'nani.Positions'>
>>> type(v[0])
<class 'nani.Vector2'>
```

Vector2 structures are best represented in NumPy using a sub-array of size 2. The same can be expressed in Nani and the view generated will correctly wrap the whole NumPy array into a container-like class [1], but accessing its elements will also return yet another object with a similar interface.

[1] See the `Collection` item in the table for Abstract Base Classes for Containers.

1.3.3 Vector2 Array With a Custom View

```
>>> import math
>>> import numpy
>>> import nani
>>> class Vector2(object):
...     __slots__ = ('_data',)
...     def __init__(self, data):
...         self._data = data
...     def __str__(self):
...         return "(%s, %s)" % (self.x, self.y)
...     @property
...     def x(self):
...         return self._data[0]
...     @x.setter
...     def x(self, value):
...         self._data[0] = value
...     @property
...     def y(self):
...         return self._data[1]
...     @y.setter
...     def y(self, value):
...         self._data[1] = value
...     def length(self):
...         return math.sqrt(self.x ** 2 + self.y ** 2)
>>> vector2_type = nani.Array(
...     element_type=nani.Number(),
...     shape=2,
...     view=Vector2)
>>> dtype, default, view = nani.resolve(vector2_type, name='Positions')
>>> a = numpy.array([(1.0, 3.0), (2.0, 4.0)], dtype=dtype)
>>> v = view(a)
>>> for position in v:
...     position.x *= 1.5
...     position.y *= 2.5
...     position.length()
7.64852927039
10.4403065089
>>> a
[[  1.5   7.5]
[  3.   10. ]]
>>>  v
[(1.5, 7.5), (3.0, 10.0)]
```

This time a custom view for the Vector2 elements is provided. As per the documentation for the Nani data type *Array*, the view is a class accepting a single parameter `data`.

This view defines a custom interface that allows accessing the Vector2 elements through the `x` and `y` properties, as well as retrieving the length of the vector.

Note: To expose a sequence-like interface, similar to what Nani generates dynamically, it is necessary to implement it manually.

1.3.4 Particle Structure

```
>>> import numpy
>>> import nani
>>> class Vector2(object):
...     __slots__ = ('_data',)
...     def __init__(self, data):
...         self._data = data
...     def __str__(self):
...         return "(%s, %s)" % (self.x, self.y)
...     @property
...     def x(self):
...         return self._data[0]
...     @x.setter
...     def x(self, value):
...         self._data[0] = value
...     @property
...     def y(self):
...         return self._data[1]
...     @y.setter
...     def y(self, value):
...         self._data[1] = value
>>> vector2_type = nani.Array(
...     element_type=nani.Number(),
...     shape=2,
...     view=Vector2)
>>> particle_type = nani.Structure(
...     fields=(
...         ('position', vector2_type),
...         ('velocity', vector2_type),
...         ('size', nani.Number(default=1.0)),
...     ),
...     name='Particle')
>>> dtype, default, view = nani.resolve(particle_type, name='Particles')
>>> a = numpy.array([default] * 2, dtype=dtype)
>>> v = view(a)
>>> for i, particle in enumerate(v):
...     particle.position.x = (i + 2) * 3
...     particle.velocity.y = (i + 2) * 4
...     particle.size *= 2
...     particle
Particle(position=(6.0, 0.0), velocity=(0.0, 8.0), size=2.0)
Particle(position=(9.0, 0.0), velocity=(0.0, 12.0), size=2.0)
>>> data = nani.get_data(v)
>>> data['position'] += data['velocity']
>>> data
[([6.0, 8.0], [0.0, 8.0], 1.0) ([9.0, 12.0], [0.0, 12.0], 2.0)]
```

Building upon the previous example, a particle data type is defined in the form of a NumPy structured array. The Vector2 data type is reused for the `position` and `velocity` fields, with its custom view still giving access to the `x` and `y` properties.

The default values returned by the *`resolve()`* function is also used here to initialize NumPy's array, ensuring that the `size` field is set to `1.0` for each particle.

At any time, the NumPy array data can be retrieved from an array view generated by Nani using the *`get_data()`* function, allowing the user to bypass the interface provided.

1.3.5 Atomic Views

When accessing or setting an atomic element—such as a number—in a NumPy array, its value is directly returned. The views dynamically generated by Nani follow this principle by default but also offer the possibility to add an extra layer between the user and the value. One use case could be to provide a more user-friendly interface to manipulate bit fields (or flags):

```
>>> import sys
>>> import numpy
>>> import nani
>>> if sys.version_info[0] == 2:
...     def iteritems(d):
...         return d.iteritems()
... else:
...     def iteritems(d):
...         return iter(d.items())
>>> _PLAYER_STATE_ALIVE = 1 << 0
>>> _PLAYER_STATE_MOVING = 1 << 1
>>> _PLAYER_STATE_SHOOTING = 1 << 2
>>> _PLAYER_STATE_LABELS = {
...     _PLAYER_STATE_ALIVE: 'alive',
...     _PLAYER_STATE_MOVING: 'moving',
...     _PLAYER_STATE_SHOOTING: 'shooting'
... }
>>> class PlayerState(object):
...     __slots__ = ('_data', '_index')
...     def __init__(self, data, index):
...         self._data = data
...         self._index = index
...     def __str__(self):
...         value = self._data[self._index]
...         return ('(%s)' % (', '.join([
...             "'%s'" % (name,)
...             for state, name in iteritems(_PLAYER_STATE_LABELS)
...             if value & state
...         ])))
...     @property
...     def alive(self):
...         return self._data[self._index] & _PLAYER_STATE_ALIVE != 0
...     @alive.setter
...     def alive(self, value):
...         self._data[self._index] |= _PLAYER_STATE_ALIVE
...     @property
...     def moving(self):
...         return self._data[self._index] & _PLAYER_STATE_MOVING != 0
...     @moving.setter
...     def moving(self, value):
...         self._data[self._index] |= _PLAYER_STATE_MOVING
...     @property
...     def shooting(self):
...         return self._data[self._index] & _PLAYER_STATE_SHOOTING != 0
...     @shooting.setter
...     def shooting(self, value):
...         self._data[self._index] |= _PLAYER_STATE_SHOOTING
>>> vector2_type = nani.Array(
...     element_type=nani.Number(),
...     shape=2)
>>> player_type = nani.Structure(
```

```
...     fields=(
...         ('name', nani.String(length=32, default='unnamed')),
...         ('position', vector2_type),
...         ('state', nani.Number(
...             type=numpy.uint8,
...             default=_PLAYER_STATE_ALIVE,
...             view=PlayerState)),
...     ),
...     name='Player')
>>> dtype, default, view = nani.resolve(player_type, name='Players')
>>> a = numpy.array([default] * 2, dtype=dtype)
>>> v = view(a)
>>> first_player = v[0]
>>> first_player
Player(name=unnamed, position=[0.0, 0.0], state=('alive'))
>>> first_player.state.moving = True
>>> first_player.state
('alive', 'moving')
>>> first_player.state.shooting
False
```

The NumPy array created here is made of elements each representing a `Player` from a game. The view class `PlayerState` allows to manipulate the state of the player (alive, moving, shooting) by abstracting the bitwise operations required to read/set the flags from/to the `numpy.uint8` data. As per the documentation of the data type *`Number`*, the view class' `__init__` method is required to accept 2 parameters: `data` and `index`.

1.4 API Reference

The whole public interface of Nani is described here.

All of the library's content is accessible from within the only module `nani`. This includes the types required to define a NumPy `dtype`, the field helpers when dealing with structured arrays, the functions, and the final output structure *`Nani`* returned from the main routine *`resolve()`*.

1.4.1 Data Types

These are the data types mimicking the NumPy's scalar hierarchy of types and allowing to describe NumPy's dtypes in the Nani format.

They can later on be converted into NumPy's `dtypes` by calling the *`resolve()`* function.

`Bool`	Type corresponding to `numpy.bool_`.
`Object`	Type corresponding to `numpy.object_`.
`Number`	Type corresponding to `numpy.number`.
`String`	Type corresponding to `numpy.string_`.
`Unicode`	Type corresponding to `numpy.unicode_`.
`Array`	Type corresponding to a NumPy (sub)array.
`Structure`	Type corresponding to a NumPy structured array.
`Bytes`	alias of *`String`*
`Str`	alias of *`Unicode`*

class nani.**Bool**(*default=False*, *view=None*)
Type corresponding to numpy.bool_.

default
bool

Default value.

view
type or None

If None, the owning array returns a direct reference to this boolean value, otherwise it is expected to be a class object wrapping it and accepting 2 parameters: data, the NumPy array owning the boolean value, and index, its position in the array.

---

class nani.**Object**(*default=None*, *view=None*)
Type corresponding to numpy.object_.

default
object

Default value.

view
type or None

If None, the owning array returns a direct reference to this Python object, otherwise it is expected to be a class object wrapping it and accepting 2 parameters: data, the NumPy array owning the Python object, and index, its position in the array.

---

class nani.**Number**(*type=numpy.float_*, *default=0*, *view=None*)
Type corresponding to numpy.number.

default
numpy.number type

Default value.

view
type or None

If None, the owning array returns a direct reference to this numeric value, otherwise it is expected to be a class object wrapping it and accepting 2 parameters: data, the NumPy array owning the numeric value, and index, its position in the array.

---

class nani.**String**(*length*, *default=''*, *view=None*)
Type corresponding to numpy.string_.

length
int

Number of characters.

default
str on PY2 or bytes on PY3

Default value.

view
type or None

If `None`, the owning array returns a direct reference to this string value, otherwise it is expected to be a class object wrapping it and accepting 2 parameters: `data`, the NumPy array owning the string value, and `index`, its position in the array.

---

class nani.**Unicode**(*length*, *default=''*, *view=None*)
Type corresponding to `numpy.unicode_`.

length
int

Number of characters.

default
unicode on PY2 or str on PY3

Default value.

view
type or None

If `None`, the owning array returns a direct reference to this unicode value, otherwise it is expected to be a class object wrapping it and accepting 2 parameters: `data`, the NumPy array owning the unicode value, and `index`, its position in the array.

---

class nani.**Array**(*element_type*, *shape*, *name=None*, *view=None*)
Type corresponding to a NumPy (sub)array.

element_type
nani type

Type of each element.

shape
int or tuple of int

Shape of the array. Passing an int defines a 1D array.

name
str or None

Name for the view type if *view* is `None`.

view
type or None

If `None`, a view for this array is dynamically generated by Nani, otherwise it is expected to be a class object wrapping it and accepting 1 parameter: `data`, the corresponding NumPy array.

---

class nani.**Structure**(*fields*, *name=None*, *view=None*)
Type corresponding to a NumPy structured array.

fields
tuple of nani.Field or compatible tuple

Fields defining the structure.

name
str or None

Name for the view type if *view* is None.

view
type or None

If None, a view for this structured array is dynamically generated by Nani, otherwise it is expected to be a class object wrapping it and accepting 1 parameter: data, the corresponding NumPy structured array.

---

class nani.**Bytes**(*length*, *default=''*, *view=None*)
Alias for *String*.

---

class nani.**Str**(*length*, *default=''*, *view=None*)
Alias for *String* on PY2 or *Unicode* on PY3.

1.4.2 Field Helpers

Structured arrays are described as a sequence of fields. Each field can be defined as a *Field* or as a tuple compatible with the *Field* structure.

A constant *READ_ONLY* equivalent to the boolean value True is provided to make the code more readable when setting the *Field.read_only* attribute without explicitely writing the read_only keyword. Example:

```
>>> import nani
>>> data_type = nani.Structure(
...     fields=(
...         ('do_not_touch', nani.Number(), nani.READ_ONLY),
...     )
... )
```

Field	Describe a field of a structured array.
READ_ONLY	Constant to use for the *Field.read_only* attribute's value.

---

class nani.**Field**(*name*, *type*, *read_only=False*)
Describe a field of a structured array.

name
str

Name of the field.

type
nani data type

Type of the field.

read_only
bool

True to not define a setter property in the structured array view if it is set to be dynamically generated by Nani.

---

nani.**READ_ONLY = True**
Constant to use for the *Field.read_only* attribute's value. To use for readability reasons when the read_only keyword is not explicitely written.

1.4.3 Functions

validate	Check if a data type is well-formed.
resolve	Retrieve the properties for a given data type.
update	Update a data type with different values.
get_data	Retrieve the NumPy data from an array view generated by Nani.
get_element_view	Retrieve the element view from an array view generated by Nani.

---

nani.**validate**(*data_type*)
Check if a data type is well-formed.

Parameters **data_type** (*nani data type*) – Data type.

Returns True if the data type is well-formed.

Return type bool

Raises *TypeError or ValueError* – The data type isn't well-formed.

---

nani.**resolve**(*data_type*, *name=None*, *listify_default=False*)
Retrieve the properties for a given data type.

This is the main routine where most of the work is done. It converts Nani's data types into properties that can be used to define a new NumPy array and to wrap it into a view object.

Use *validate()* to check if the input data type is well-formed.

Parameters

- **data_type** (*nani data type*) – Type of the array elements.
- **name** (*str*) – Name for the view to be generated for the array.
- **listify_default** (*bool*) – True to output the default values with lists in place of tuples. This might cause the output to be incompatible with array creation routines such as numpy.array but it should still work for element assignment.

Returns The properties to use to initalize a NumPy array around the data type.

Return type *nani.Nani*

Examples

Create a NumPy array where each element represents a color:

```
>>> import numpy
>>> import nani
>>> color_type = nani.Array(
...     element_type=nani.Number(type=numpy.uint8, default=255),
...     shape=3,
...     view=None)
```

```
>>> dtype, default, view = nani.resolve(color_type, name='Color')
>>> a = numpy.array([default] * element_count, dtype=dtype)
>>> v = view(a)
>>> type(v)
<class 'nani.Color'>
>>> for color in v:
...     color
[255, 255, 255]
[255, 255, 255]
```

---

nani.**update**(*data_type*, ***kwargs*)

Update a data type with different values.

The operation is not made in place, instead a copy is returned.

Parameters

- **data_type** (*nani data type*) – Data type.
- **kwargs** – Keyword arguments to update.

Returns The updated version of the data type.

Return type nani data type

Examples

Update the shape of an array data type and the default value of its elements:

```
>>> import nani
>>> data_type = nani.Array(
...     element_type=nani.Number(),
...     shape=2)
>>> new_data_type = nani.update(
...     data_type,
...     element_type=nani.update(data_type.element_type, default=123),
...     shape=3)
```

---

nani.**get_data**(*view*)

Retrieve the NumPy data from an array view generated by Nani.

Parameters **view** (*nani.AtomicArrayView or nani.CompositeArrayView*) – Array view.

Returns The NumPy array, None otherwise.

Return type *type*

Examples

```
>>> import numpy
>>> import nani
>>> data_type = nani.Number(type=numpy.int32)
>>> dtype, _, view = nani.resolve(data_type)
>>> a = numpy.a_range(10, dtype=dtype)
```

```
>>> v = view(a)
>>> nani.get_data(v) is a
True
```

---

nani.**get_element_view**(*view*)

Retrieve the element view from an array view generated by Nani.

Parameters **view** (*nani.AtomicArrayView or nani.CompositeArrayView*) – Array view.

Returns The element view, None otherwise.

Return type *type*

Examples

```
>>> import numpy
>>> import nani
>>> vector2_type = nani.Array(
...     element_type=nani.Number(),
...     shape=2,
...     name='Vector2')
>>> dtype, default, view = nani.resolve(vector2_type, name='Positions')
>>> a = numpy.zeros(3, dtype=dtype)
>>> v = view(a)
>>> type(v)
<class 'nani.Positions'>
>>> nani.get_element_view(v)
<class 'nani.Vector2'>
```

1.4.4 Output Structure

This regroups the output structure from the main routine *resolve()*. The *Nani* object regroups the properties helping to define NumPy's arrays and to optionally wrap them into an abstraction layer.

Nani	Output structure of the function *resolve*.

---

class nani.**Nani**(*dtype*, *default*, *view*)

Output structure of the function *resolve*.

dtype

numpy.dtype

NumPy's dtype, that is the data type of the array elements.

default

object

Default value(s) for a single array element.

view

type

A class to use as a wrapper around the NumPy array.

CHAPTER 2

Developer's Guide

2.1 Running the Tests

After making any code change in Nani, tests need to be evaluated to ensure that the library still behaves as expected.

Note: Some of the commands below are wrapped into `make` targets for convenience, see the file `Makefile`.

2.1.1 unittest

The tests are written using Python's built-in `unittest` module. They are available in the `tests` directory and can be fired through the `tests/run.py` file:

```
$ python tests/run.py
```

It is possible to run specific tests by passing a space-separated list of partial names to match:

```
$ python tests/run.py ThisTestClass and_that_function
```

The `unittest`'s command line interface is also supported:

```
$ python -m unittest discover -s tests -v
```

Finally, each test file is a **standalone** and can be directly executed.

2.1.2 tox

Test environments have been set-up with `tox` to allow testing Nani against each supported version of Python:

```
$ tox
```

2.1.3 coverage

The package `coverage` is used to help localize code snippets that could benefit from having some more testing:

```
$ coverage run --source nani -m unittest discover -s tests
$ coverage report
$ coverage html
```

In no way should `coverage` be a race to the 100% mark since it is *not* always meaningful to cover each single line of code. Furthermore, **having some code fully covered isn't synonym to having quality tests**. This is our responsability, as developers, to write each test properly regardless of the coverage status.

CHAPTER 3

Additional Information

3.1 Changelog

Version numbers comply with the Sementic Versioning Specification (SemVer).

3.1.1 v0.2.0 (2017-01-18)

Added

- Add a `validate()` function to the public interface.
- Add equality operators for the views.
- Add support for coverage and tox.
- Add continuous integration with Travis and coveralls.
- Add a few bling-bling badges to the readme.
- Add a Makefile to regroup common actions for developers.

Changed

- Improve the documentation.
- Improve the unit testing workflow.
- Allow array shapes and structure fields to be defined as lists.
- Improve the error messages.
- Replace `__str__()` with `__repr__()` for the views.
- Update some namedtuple typenames.
- Refocus the content of the readme.
- Define the 'long_description' and 'extras_require' metadata to setuptools' setup.
- Update the documentation's Makefile with a simpler template.
- Rework the '.gitignore' files.
- Rename the changelog to 'CHANGELOG'!
- Make minor tweaks to the code.

Fixed

- Fix the duplicate field finder.
- Fix a call to the wrong 'contains' implementation in a view.

3.1.2 v0.1.1 (2016-10-24)

Changed

- Remove the module index from the documentation.

Fixed

- Fix Read the Docs not building the documentation.

3.1.3 v0.1.0 (2016-10-24)

- Initial release.

3.2 Versioning

Version numbers comply with the Sementic Versioning Specification (SemVer).

In summary, version numbers are written in the form `MAJOR.MINOR.PATCH` where:

- incompatible API changes increment the MAJOR version.
- functionalities added in a backwards-compatible manner increment the MINOR version.
- backwards-compatible bug fixes increment the PATCH version.

Major version zero (0.y.z) is considered a special case denoting an initial development phase. Anything may change at any time without the MAJOR version being incremented.

3.3 License

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.