zerial Documentation

Release 0.2.0

Josh Reed

Nov 08, 2019

Contents

Zerial is the serialization tool that allows your model classes to be the Zingle Zource of Truth™ for your project. Let your model classes take whatever form or use whatever collection types they need, and just use metadata to define how that text gets serialized. With support for variant record types, you can even evolve your data models over time, and even create versioned models if need be.

Zerial is built on top of the excellent attrs library, which makes class creation and definition in Python very easy and very obvious. This library adds abritrarily recursive serialization and de-serialization of complex data classes.

Zerial was inspired because complex applications in spaces where requirements are hard to define up front call for a unique approach to modeling data. A solution has to be flexible enough to accomodate a growing and changing understanding of the underlying problem domain, while being rigorous enough to encapsulate these changes to data modules. External schemas like ORM or JSON Schema are both inflexible and instrinsically bound to specific data exchange formats (SQL and JSON) that you may or may not want to actually use. Implicit schemas, although sufficienty flexible, ultimately fail because they break separation of concerns, requiring every bit of code that touches data to understand how to create a validate an entire history of model versions for that data type.

This project aims to allow components to structure their data in a way that is convenient for people interacting with the code, provided that it can be destructured into simple types. The combination of attrs' rich support for defaults and value factories and zerial's support of variant records, you can evolve your data models over time, without breaking your client code or stored serialized data.

CHAPTER 1

Features

- Stucturing and destructuring of model classes
- Supports rich typing without any runtime dependency on stuff from the `typing` module, which has deeply inconsistent runtime behavior
- Model fields can be simple types or other model classes
- Collection types can be represented as any kind of Python object, as long as you can convert it to and from a list or dict with string keys.
- Variant records permit fields that accept multiple types of data, permitting extensibility. Variants with default types allow this to be added at any point in development (and even permits for *versioned* data models).

CHAPTER 2

Todo/Roadmap

1. Optional native support for numpy arrays
2. Export of schema definition formats from record classes
3. Debug tools for destructure/restructure failures
4. Wrappers around `attr.s` and `attr.ib` to make defining models cleaner
5. Better automated checking and testing tools for serializability
6. More extensive documentation

2.1 Contents:

2.1.1 Installation

At the command line either via easy_install or pip:

```
$ easy_install zerial
$ pip install zerial
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv zerial
$ pip install zerial
```

2.1.2 Usage

To use zerial in a project:

```
import zerial
```

2.1.3 zerial package

Module contents

class zerial.**Sequence**(*item_type*, *restructure_factory=<class 'list'>*, *destructure_factory=<class 'list'>*, *apparent_type=NOTHING*)

Bases: zerial._base.Ztype, typing.Generic

Metadata type for a sequence of items.

Handles any version of N number of items of type T. Works for sets, tuples, lists, and anything else that follows that interface.

Attribute item_type The type of the object contained in this sequence. Note that this can itself be another complex object or another Ztype, so these data types can be arbitrarily nested.

Attribute restructure_factory The callable that will convert a stored sequence into a live data type. Basically, the function that can rebuild your data into the collection type you want.

Attribute destructure_factory The callable that will convert a live sequence into storage data. Normally this will always be list, unless your serialization format supports other sequence types.

Attribute apparent_type To be deprecated

default_apparent_type()

destruct(*inst*, *ztr*)

Unstructure inst into a mapping.

The Ztructurer doing the destructuring is passed for its options and to permit further recursive descent if necessary.

Works with the Ztructurer to take apart more complex types.

restruct(*data*, *ztr*)

Structure the mapping into the appropriate type.

The Ztructurer doing the rebuilding is passed for its options and to permit recursive rebuilding if needed.

Works with Ztructurer to rebuild more complext types.

class zerial.**Mapping**(*key_type: Type[K]*, *val_type: Type[V]*, *restructure_factory=<class 'dict'>*, *destructure_factory=<class 'dict'>*, *extract_pairs=operator.methodcaller('items')*, *apparent_type=typing.MutableMapping*)

Bases: zerial._base.Ztype, typing.Generic

Ztype wrapper for a simple mapping/dict

Since many serialization formats require that keys for mappings be strings, we require that the key_type be convertible to-and-from string. This limits us to pretty primitive types, like int and str.

Attribute key_type The type of the keys of the mapping. Currently only supports types that can do a straight-forward one-to-one convertion to and from `str`.

Attribute val_type The type of the values of the mapping. Can be any type that will can be serialized, including other `Ztype` objects, allowing arbirarily complex value types.

Attribute restructure_factory The container type that will be used to rebuild your data type upon being restructured. It will be called with an iterable of key-value pairs. The default is dict, but if you want, say, a `collections.defaultdict` to come out with a default function of `int`, you could give this:

```
@zerial.record
@attr.s
class MyRecord:
    my_field = attr.ib(
        type=typing.Mapping[str, int],
        metadata=zerial.data(zerial.Mapping(
            str, int, partial(defaultdict, int)
        )),
    )
```

This will result in `my_field` being restructured ready for you to do all the fun `defaultdict` stuff you've always wanted to do like `my_record.my_field[arbitrary_key] += 1`, because `zerial` ensures that it gets restructured that way.

Attribute destructure_factory The container type that will be used to take apart your data to destructure it. Can be useful if your mapping is ordered, in which case you'd want to save it as a `list` to preserve the order even in a serialized dict. This feature is unstable and may be replaced by something more sophisticated in the future.

Attribute extract_pairs Function to extract item pairs from the object created by `destructure_factory`. By default, this calls `.items()` on the object it receives, but if you were to destructure into a `list`, you'll need to pass `None` here (which is converted to the identity function `lambda x: x`) since a list of pairs is already, well, a list of pairs. This feature is unstable and may be replaced by something better in the future.

destruct (*inst*, *ztr*)

Unstructure inst into a mapping.

The Ztructurer doing the destructuring is passed for its options and to permit further recursive descent if necessary.

Works with the Ztructurer to take apart more complex types.

restruct (*mapping*, *ztr*)

Structure the mapping into the appropriate type.

The Ztructurer doing the rebuilding is passed for its options and to permit recursive rebuilding if needed.

Works with Ztructurer to rebuild more complext types.

class zerial.**Variant**(*type_records*, *default=<object object>*, *name: str = NOTHING*, *enum: enum.Enum = NOTHING*)

Bases: `zerial._base.Ztype`

Variant that allows multiple types to occupy the same attribute slot.

This is an implementation of a sum type (called enums or variants, depending on the source) for serialization. It allows multiple types to be saved in the same slot, and deserialized intelligently into the correct type. It accomplishes this by storing a name corresponding to the type along with the type information.

Attribute _type_records Iterable of types, or a mapping of names to types. The types should be concrete types, in the sense that they can instantiate a real object of their class from provided data. They are treated as invariant. If you have multiple subclasses of a type, they must all be specified here. The `Variant` has to be able to store the type in the destructured data and consistently map it to the desired runtime type in the restuctured model, so it needs a full accounting of which types are available to it at definition time. If passed a mapping, the keys are taken as the type names. If passed a plain iterable, the type names are taken from the `__name__` attribute of the class. Collisions are invalid, so if you have two types with the same `__name__`, use a mapping to give them unique names in this context.

Attribute default Default type to use if no type information is found in the destructured data. This permits previously non-variant field to become variants without invalidating previously saved data.

NO_DEFAULT = <object object>

apparent_type
Special type indicating an unconstrained type.

- Any is compatible with every type.
- Any assumed to have all methods.
- All values assumed to be instances of Any.

Note that all the above statements are true from the point of view of static type checkers. At runtime, Any should not be used with instance or class checks.

destruct (*inst*, *ztr*)
Unstructure inst into a mapping.

The Ztructurer doing the destructuring is passed for its options and to permit further recursive descent if necessary.

Works with the Ztructurer to take apart more complex types.

name

restruct (*data*, *ztr*)
Structure the mapping into the appropriate type.

The Ztructurer doing the rebuilding is passed for its options and to permit recursive rebuilding if needed.

Works with Ztructurer to rebuild more complext types.

types

class zerial.**Serializer**(*to_outer: Callable[T, U], to_inner: Callable[U, T]*)
Bases: zerial._base.Ztype, typing.Generic

When you just need a serializer that goes forward and back.

Attribute to_outer Function that destructures the data.

Attribute to_inner Function that restructures the data.

If the other metadata types fail, or there are obvious standard string representations of your data type (like in dates), you can just use a *Serializer* to take care of it for you:

```
import datetime

@zerial.record
@attr.s
class ImportantDate:
    name = attr.ib(type=str)
    date = attr.ib(
        type=datetime.date,
        metadata=zerial.data(zerial.Serializer(
            lambda d: d.isoformat(),
            lambda s: datetime.datetime.strptime(s, '%Y-%m-%d').date(),
        )),
    )
```

It doesn't necessarily have to be a string either. If your mapping variant is too complex for *Mapping*, then you could use this class as a last resort for defining a method of destructuring it.

destruct(*inst*, *_*)
Unstructure inst into a mapping.

The Ztructurer doing the destructuring is passed for its options and to permit further recursive descent if necessary.

Works with the Ztructurer to take apart more complex types.

restruct(*data*, *_*)
Structure the mapping into the appropriate type.

The Ztructurer doing the rebuilding is passed for its options and to permit recursive rebuilding if needed.

Works with Ztructurer to rebuild more complext types.

class zerial.**Ztructurer**(*dict_factory=<class 'dict'>*, *metachar: str = '%'*, *permitted_passthrus: tuple = (<class 'bool'>, <class 'int'>, <class 'float'>, <class 'str'>)*, *is_serializable_field=operator.attrgetter('init')*, *encoders=NOTHING*, *can_structure=<function has>*)

Bases: object

can_encode(*type*)

can_pass_thru(*val*)

can_structure(*tobj*)

destructure(*inst*, *type=None*)

get_encoder(*type*)

get_metakey(*key*)

record(*cls*)
Decorator to indicate that a class is serializable

Currently does nothing special, but in the future it may be used to add automated features, and could potentially become required to serialize objects in certain contexts.

restructure(*type*, *mapping*)

zerial.**zdata**(*ztype*)

zerial.**Zapping**
alias of zerial._data.Mapping

zerial.**Zariant**
alias of zerial._data.Variant

zerial.**Zequence**
alias of zerial._data.Sequence

zerial.**Zerializer**
alias of zerial._data.Serializer

2.1.4 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/jriddy/zerial/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

zerial could always use more documentation, whether as part of the official zerial 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/jriddy/zerial/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 *zerial* for local development.

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

```
$ git clone git@github.com:your_name_here/zerial.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 2.7, 3.5, and 3.6. Other versions are planned by not yet fully supported. Check https://travis-ci.org/jriddy/zerial 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_zerial.py
```

2.1.5 Credits

Development Lead

- Josh Reed <jriddy@gmail.com>

Contributors

None yet. Why not be the first?

2.1.6 History

0.1.0 (2018-10-21)

- First release on PyPI.

2.2 Feedback

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

Python Module Index

z

Index