marshmallow-annotations Documentation

Release 2.3.0

Alec Nikolas Reiter

Dec 13, 2018

Contents

Version 2.3.0 (*Change Log*)

marshmallow-annotations allows you to create marshmallow schema from classes with annotations on them:

```
from marshmallow_annotations import AnnotationSchema
from .music import Album, Artist

class AlbumScheme(AnnotationSchema):
    class Meta:
        target = Album
        register_as_scheme = True


class ArtistScheme(AnnotationSchema):
    class Meta:
        target = Artist
        register_as_scheme = True


scheme = ArtistScheme()
scheme.dump(
    Artist(
        id=1, name="Abominable Putridity",
        albums=[
            Album(
                id=1,
                name="The Anomalies of Artificial Origin"
            )
        ]
    )
)

# {
#     "albums": [
#         {
#             "id": 1,
#             "name": "The Anomalies of Artificial Origin"
#         }
#     ],
#     "id": 1,
#     "name": "Abominable Putridity"
# }
```

CHAPTER 1

Installation

marshmallow-annotations is available on pypi and installable with:

```
pip install marshmallow-annotations
```

marshmallow-annotations supports Python 3.6+ and marshmallow 2.x.x

Note: If you are install `marshmallow-annotations` outside of a virtual environment, consider installing with `pip install --user marshmallow-annotations` rather than using sudo or adminstrator privileges to avoid installing it into your system Python.

CHAPTER 2

Content

2.1 Quickstart

This guide will walk you through the basics of using `marshmallow-annotations`.

2.1.1 Annotating your class

Before we can make use of `marshmallow-annotations` we first need to annotate some classes:

```
from typing import List

class Album:
    id: int
    name: str

    def __init__(self, id, name):
        self.id = id
        self.name = name

class Artist:
    id: int
    name: str
    albums: List[Album]

    def __init__(self, id, name, albums=None):
        self.id = id
        self.name = name
        self.albums = albums if albums is not None else []
```

Note: If writing this boiler plate is a turn off, consider using attrs in conjunction with this library as well:

```
import attr

@attr.s
class Album:
    id: int = attr.ib()
    name: str = attr.ib()

@attr.s
class Artist:
    id: int = attr.ib()
    name: str = attr.ib()
    albums: List[Album] = attr.ib(default=attr.Factory(list))
```

With these classes defined, we can declare our schema.

2.1.2 Building Schema

The most basic scheme with `marshmallow-annotations` is a Meta definition on an *AnnotationSchema* subclass:

```
from marshmallow_annotations import AnnotationSchema

class AlbumScheme(AnnotationSchema):
    class Meta:
        target = Album
        register_as_scheme = True

class ArtistScheme(AnnotationSchema):
    class Meta:
        target = Artist
        register_as_scheme = True
```

Behind the scenes, these short definitions expand into much larger defitions based on our annotations:

```
from marshmallow import fields, Schema

class AlbumScheme(Schema):
    id = fields.Integer(required=True, allow_none=False)
    name = fields.String(required=True, allow_none=False)


class ArtistScheme(Schema):
    id = fields.Integer(required=True, allow_none=False)
    name = fields.String(required=True, allow_none=False)
    albums = fields.Nested('AlbumScheme', many=True, required=True, allow_none=False)
```

If you're curious what the `register_as_scheme` option does, this causes the generated scheme to become associated with the target type in the internal type mapping and the type will be resolved to its scheme (e.g. in the above example, future references to `Album` in type hints will resolve to `AlbumScheme`).

With the schema defined we can serialize our `Artist` and `Album` classes down to primitives:

```
scheme = ArtistScheme()
scheme.dump(
    Artist(
```

(continues on next page)

(continued from previous page)

```
        id=1, name="Abominable Putridity",
        albums=[
            Album(
                id=1,
                name="The Anomalies of Artificial Origin"
            )
        ]
    )
)

# {
#     "albums": [
#         {
#             "id": 1,
#             "name": "The Anomalies of Artificial Origin"
#         }
#     ],
#     "id": 1,
#     "name": "Abominable Putridity"
# }
```

2.1.3 How Types Map

`marshmallow-annotations` comes preconfigured with a handful of Python types mapped to marshmallow fields, these fields and their mappings are:

- `bool` maps to `Boolean`
- `date` maps to `Date`
- `datetime` maps to `DateTime`
- `Decimal` maps to `Decimal`
- `float` maps to `Float`
- `int` maps to `Integer`
- `str` maps to `String`
- `time` maps to `Time`
- `timedelta` maps to `TimeDelta`
- `UUID` maps to `UUID`
- `dict` maps to `Dict`

Note: See below for details on the `dict` mapping.

List[T]

`typing.List` maps to a special field factory that will attempt to locate it's type parameter, e.g. `List[int]` will map to `fields.List(fields.Integer())`. Alternatively, `List[T]` can generate a `fields.Nested(TScheme, many=True)` if its factory can determine that its subtype has a scheme factory registered rather than a field factory.

The success of mapping to its type parameter depends on *properly configuring your type mappings*. If List's interior typehint can't be resolved, then a `AnnotationConversionError` is raised.

Optional[T]

Another special type is `typing.Optional` (aka `typing.Union[T, None]`). When `marshmallow-annotations` encounters a type hint wrapped in `Optional` it generates the base fieldi but will default `required` to False and `allow_none` to True *unless overridden*.

Danger: Right now `marshmallow-annotations` will only inspect the first member of a Union if it thinks it's actually an Optional. The heuristics for this are simple and naive: if the type hint is a Union and the last parameter is NoneType then it's obviously an `Optional`.

The following hint will generate an int even though it's hinting at a type that may be either an int, a float or a None:

```
Union[int, float, None]
```

Dict[TKey, TValue]

`marshmallow-annotations` does support handling automatic conversions for `dict` mapped fields. However, `marshmallow` itself does not provide additional serialization and deserialization support for dictionary fields, so even if a field is mapped as `Dict[int, string]` the output of that field will be the raw input. If you require dictionary serialization, you may consider using a custom field.

Forward Declaration

`marshmallow-annotations` can handle forward declarations of a target type into itself if `register_as_scheme` is set to True:

```
class MyType:
    children: List[MyType]


class MyTypeSchema(AnnotationSchema):
    class Meta:
        target = MyType
        register_as_scheme = True
```

The `register_as_scheme` option is very eager and will set the generated schema class into the register as soon as it determines it can, which occurs before field generation happens.

Danger: Until Python 3.6.5, evaluation of forward declarations with `typing.get_type_hints()` – the method that `marshmallow-annotations` uses to gather hints – did not work properly. If you are using a class that has a forward reference to either itself or a class not yet defined, it will fail when used with `marshmallow-annotations`.

For these classes, it is recommended to not use forward declarations with this library unless you are using 3.6.5+ or backport 3.6.5's fixes to `typing.get_type_hints` into your application and monkey patch it in.

2.1.4 Configuring Fields

By default, fields will be generated with `required=True` and `allow_none=False` (however, as mentioned above, an `Optional` type hint flips these). However, sometimes a small adjustment is needed to the generated field. Rather than require writing out the entire definition, you can use `Meta.Fields` to declare how to build the generated fields.

For example, if `Artist` should receive a default name if one is not provided, it may be configured this way:

```
class ArtistScheme(AnnotationSchema):
    class Meta:
        target = Artist
        register_as_scheme = True

        class Fields:
            name = {"default": "One Man Awesome Band"}
```

Each individual field may be configured here with a dictionary and the values of the dictionary will be passed to the field's constructor when it is generated.

You may also predefine how fields should be configured on a parent scheme and the children will inherit those configurations:

```
class Track:
    id: Optional[UUID]
    name: str


class BaseScheme(AnnotationSchema):
    class Meta:
        class Fields:
            id = {"load_only": True}

class TrackScheme(BaseScheme):
    class Meta:
        target = Track

TrackScheme().dump({"name": "Wormhole Inversion", "id": str(uuid4())}).data
# {"name": "Wormhole Inversion"}
```

Children schema may choose to override the configuration and the scheme will piece together the correct configuration from the MRO resolution:

```
class TrackScheme(BaseScheme):  # as before
    class Meta:
        class Fields:
            id = {"missing": "bdff81f3-dadb-47a7-a0de-fbc892646f47"}

TrackScheme().dump({"name": "Wormhole Inversion", "id": str(uuid4())}).data
# {"name": "Wormhole Inversion"}

TrackScheme().load({"name": "Wormhole Inversion"}).data
# {
#   "name": "Wormhole Inversion",
#   "id": "bdff81f3-dadb-47a7-a0de-fbc892646f47"
# }
```

2.1.5 Meta Options

In addition to the `Fields` declaration, `marshmallow-annotations` also provides several other options that can be set in the "Meta" object on a scheme:

- `target`: The annotated class to generate fields from, if this is not provided no fields will be generated however all options related to it will be preserved for children schema.
- `converter_factory`: A callable that accepts a *`TypeRegistry`* by keyword argument `registry` and produces a *`AbstractConverter`* instance. By default this is *`BaseConverter`*.
- `registry`: A registry to use in place of the global type registry, must be an instance of *`TypeRegistry`*.
- `register_as_scheme`: If set to true, this will register the generated scheme into supplied registry as the type handler for the `target` type.

2.2 Custom Constructor, Types and Registries

There are several ways to customize how fields are generated from types in your application.

2.2.1 Custom Field Mapping

The most straight forward is registering custom field mappings into the global registry:

```
from marshmallow_annotations import registry
from ipaddress import IPv4Address
from some_extension IPAddressField


registry.register_field_for_type(IPv4Address, IPAddressField)
```

And now `marshmallow_annotations` will recognize `IPv4Address` hints as generating an `IPAddressField`.

Danger: Registrations don't check for preexisting entries before being added, and it's entirely possible to overwrite an existing registration. In some circumstances this may be useful however it may lead to hard to track down bugs.

For more complicated types, an entire scheme may be registered into the registry as well:

```
from myapp.entities import Game
from myapp.schema import GameScheme

registry.register_scheme_constructor(Game, GameScheme)
# alternatively:
registry.register_scheme_constructor(Game, 'GameScheme')
```

If more customized behavior is needed, `field_factory` and `register` are also exposed. `field_factory` is the decorator form of `register` so it will not be covered here. `register` accepts two arguments:

1. The type to associate with
2. A *Field Factory*, a callable that accepts:
   (a) An *`AbstractConverter`* instance

(b) A tuple of type hints

(c) A dictionary of configuration values for the underlying field

And returns a fully instantiated marshmallow Field instance. For example, `typing.List` has a custom factory that resembles:

```
def _list_converter(converter, subtypes, opts):
    return fields.List(converter.convert(subtypes[0]), **opts)
```

Under the hood, `register_scheme_constructor` and `register_field_for_type` use generalized versions of such a converter, these are exposed as *`scheme_factory()`* and *`field_factory()`* and are availabe for use if needed.

2.2.2 Using a non-Global registry

Since mutable, global state doesn't sit well with everyone it is also possible to use a non-global registry by creating your own instance of `marshmallow_annotations.registry.TypeRegistry`:

```
from marshmallow_annotations import DefaultTypeRegistry

my_registry = DefaultTypeRegistry()
```

It's also possible to pass a dictionary that maps a type to a field factory into the initializer and pre-configure your registry instance with those types:

```
conf = {IPv4Address: field_factory(IPAddressField)}

my_registry = DefaultTypeRegistry(conf)
```

A specific registry instance can be attached to a scheme by declaring the `registry` field on the Meta options:

```
class MyScheme(AnnotationSchema):
    class Meta:
        registry = my_registry
```

Now this specific Scheme will derive fields from the mappings found in `my_registry`.

2.2.3 Custom Registry

If *`DefaultTypeRegistry`* isn't meeting your needs, *`TypeRegistry`* is available for implementation as well.

2.2.4 Custom Converters

Another customization point is implementing your own *`AbstractConverter`* class as well to provide to schema definitions:

```
class MyConverter(AbstractConverter):
    # impl contract


class SomeSchema(AnnotationSchema):
    class Meta:
        target = MyType
        converter_factory = MyConverter
```

2.3 marshmallow_annotations API

Warning: Scary looking type signatures a head.

2.3.1 Type API

This section will hopeful ease some of the stress and tension that will inevitably arise from looking at the following type signatures. Seriously, they're not pretty.

Field Factory

A field factory **is not** the field's instance constructor, rather it is any callable that accepts:

1. An *AbstractConverter* instance
2. A tuple of type hints
3. A dictionary of configuration values for the underlying field

And returns a fully instantiated marshmallow Field instance, for example:

```
from marshmallow import List as ListField

def sequence_converter(converter, subtypes, opts):
    return ListField(converter.convet(subtypes[0]), **opts)
```

This might be registered against `Tuple`:

```
registry.register(typing.Tuple, sequence_converter)
```

A method that accepts a field factory contains the following signature:

```
Callable[[AbstractConverter, Tuple[type], Dict[str, Any]], FieldABC]
```

2.3.2 Registry

class marshmallow_annotations.base.**TypeRegistry**

Abstraction representation of a registry mapping Python types to marshmallow field types.

This abstract class is provided primarily for type hinting purposes but also allows implementations outside of the default implementation in this library.

field_factory(*target: type*) → Callable[Callable[[marshmallow_annotations.base.AbstractConverter, Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC], Callable[[marshmallow_annotations.base.AbstractConverter, Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC]]

Decorator form of register:

```
@register.field_factor(bytes)
def custom(converter, subtypes, opts):
    return fields.Raw(**opts)
```

Returns the original function so it can be used again if needed.

abstractmethod get(*target: type*) → Callable[[marshmallow_annotations.base.AbstractConverter, Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC]

Retrieves a field factory from the registry. If it doesn't exist, this may raise a `AnnotationConversionError`:

```
registry.get(str)  # string field factory
registry.get(object)  # raises AnnotationConversionError
```

abstractmethod has(*target: type*) → bool

Allows safely checking if a type has a companion field mapped already:

```
registry.has(int)     # True
registry.has(object)  # False
```

May also be used with in:

```
int in registry     # True
object in registry  # False
```

abstractmethod register(*target: type, constructor: Callable[[marshmallow_annotations.base.AbstractConverter, Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC]*) → None

Registers a raw field factory for the specified type:

```
from marshmallow import fields

def custom(converter, subtypes, opts):
    return fields.Raw(**opts)

registry.register(bytes, custom)
```

abstractmethod register_field_for_type(*target: type, field: marshmallow.base.FieldABC*) → None

Registers a raw marshmallow field to be associated with a type:

```
from typing import NewType
Email = NewType("Email", str)

registry.register_field_for_type(Email, EmailField)
```

abstractmethod register_scheme_factory(*target: type, scheme_or_name: Union[str, marshmallow.base.SchemaABC]*) → None

Registers an existing scheme or scheme name to be associated with a type:

```
from myapp.schema import ArtistScheme
from myapp.entities import Artist

registry.register_scheme_factory(Artist, ArtistScheme)
```

class marshmallow_annotations.registry.**DefaultTypeRegistry**(*registry: Dict[type, Callable[[marshmallow_annotations.base.Abs Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC]] = None*) → None

Default implementation of *TypeRegistry*.

Provides default mappings of:

- bool -> fields.Boolean

- date -> fields.Date
- datetime -> fields.DateTime
- Decimal -> fields.Decimal
- float -> fields.Float
- int -> fields.Integer
- str -> fields.String
- time -> fields.Time
- timedelta -> fields.TimeDelta
- UUID -> fields.UUID
- dict -> fields.Dict
- typing.Dict -> fields.Dict

As well as a special factory for typing.List[T] that will generate either fields.List or fields.Nested

marshmallow_annotations.registry.**field_factory**(*field: marshmallow.base.FieldABC*) → Callable[[marshmallow_annotations.base.AbstractConverter, Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC]

Maps a marshmallow field into a field factory

marshmallow_annotations.registry.**scheme_factory**(*scheme_name: str*) → Callable[[marshmallow_annotations.base.AbstractConverter, Tuple[type], Dict[str, Any]], marshmallow.base.FieldABC]

Maps a scheme or scheme name into a field factory

2.3.3 Converter

class marshmallow_annotations.base.**AbstractConverter**

Converters handle gathering type hints and consulting with a *TypeRegistry* in order to produce marshmallow field instances:

```
from marshmallow_annotations import BaseConverter

converter = BaseConverter(registry)
converter.convert(int, {"required": False})
# marshmallow.fields.Integer(...)
```

This abstract class is provided primarily for type hinting purposes but also allows implementations outside of the default implementation in this library.

abstractmethod convert(*typehint: type, opts: Union[typing.Dict[str, typing.Any], NoneType] = None, *, field_name: str = None, target: type = None*) → marshmallow.base.FieldABC

Used to convert a type hint into a `FieldABC` instance.

Versionchanged 2.2.0 Added field_name and target optional keyword only arguments

abstractmethod convert_all(*target: type, ignore: AbstractSet[str] = frozenset(), configs: Union[typing.Dict[str, typing.Union[typing.Dict[str, typing.Any], NoneType]], NoneType] = None*) → Dict[str, marshmallow.base.FieldABC]

Used to transform a type with annotations into a dictionary mapping the type's attribute names to

`FieldABC` instances.

abstractmethod is_scheme(*typehint: type*) → bool
Used to check if the typehint passed if associated to a scheme or a regular field constructor.

class marshmallow_annotations.converter.**BaseConverter**(*, *registry: marshmallow_annotations.base.TypeRegistry = <marshmallow_annotations.registry.DefaultTypeRegistry object>*) → None
Default implementation of *AbstractConverter*.

Handles parsing types for type hints and mapping those type hints into marshmallow field instances by way of a *TypeRegistry* instance.

Versionchanged 2.1.0 Added non-public hook `_get_field_defaults`

Versionchanged 2.2.0 Added non-public hooks `_preprocess_typehint` and `_postprocess_typehint`

_get_field_defaults(*item*)
Non-public hookpoint to read default values for all fields from the target

_postprocess_typehint(*typehint*, *kwargs*, *field_name*, *target*)
Non-public hookpoint for any postprocessing of typehint parsing for a given field.

_preprocess_typehint(*typehint*, *kwargs*, *field_name*, *target*)
Non-public hookpoint for any preprocessing of typehint parsing for a given field

2.3.4 Schema

class marshmallow_annotations.scheme.**AnnotationSchemaMeta**(*name*, *bases*, *attrs*)
Metaclass that handles produces the *AnnotationSchema* class. Provided for integration into other libraries and toolkits

class marshmallow_annotations.scheme.**AnnotationSchema**(*extra=None*, *only=None*, *exclude=()*, *prefix=''*, *strict=None*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)
Base class for creating annotation schema with:

```
from marshmallow_annotations import AnnotationSchema
from my.app.entities import Artist

class ArtistScheme(AnnotationSchema):
    class Meta:
        target = Artist
        register_as_scheme = True
```

class marshmallow_annotations.scheme.**AnnotationSchemaOpts**(*meta*, *schema=None*)
marshmallow-annotations specific SchemaOpts implementation, provides:

- converter_factory
- registry
- register_as_scheme
- target

- field_configs
- converter

2.3.5 Exceptions

class marshmallow_annotations.exceptions.**MarshmallowAnnotationError**

class marshmallow_annotations.exceptions.**AnnotationConversionError**

2.4 Extensions

Extensions are provided for specific use cases.

2.4.1 attrs

If you are using attrs, you can use the extension *AttrsSchema* to generate your schema.

attrs Extension Installation

This extension ships with `marshmallow-annotations` beginning in v2.2.0 and is available for import normally. You may specify `marshmallow-annotations[attrs]` as the install target to automatically install attrs along side marshmallow-annotations.

attrs Integration API

This extension modifies loading behavior to deserialize directly into instances of the target class. To handle automatic generation, you must specify your attr class to have the `auto_attribs` option to generate matching `attr.ib` instances from purely typehinted fields:

```
from datetime import timedelta
import attr
from marshmallow_annotations.ext.attrs import AttrsSchema


@attr.s(auto_attribs=True)
class Track:
    name: str  # no attr.ib -- attrs will generate one with auto_attribs
    length: timedelta = attr.ib(factory=timedelta)

class TrackSchema(AttrsSchema):
    class Meta:
        target = Track

serializer = TrackSchema()
loaded = serializer.load({"name": "Letting Them Fall"}).data
# Track(name="Letting Them Fall", length=timedelta(0))
serializer.dump(loaded).data
# {"name": "Letting Them Fall", "length": 0}
```

The attrs integration makes a few changes to the normal assumptions the normal schema generation makes:

- If an `attr.ib` has a default value (not a factory), it is put on the generated field as the missing value and the field is marked optional (though not `allow_none`).
- If an `attr.ib` has a default factory, the field is marked optional but the factory is not moved to the generated field (this is despite that marshmallow fields can accept callables as `missing`, some factories accept the new instance as an argument and marshmallow cannot handle this).
- If an `attr.ib` is passed `init=False` then the generated field is marked `dump_only=True` even if the `Meta.Fields` setting set it to `dump_only=False` since the instance constructor cannot accept this value.

It is possible to override the `missing` value and `required=False` by changing these in the `Meta.Fields`.

Beginning with v2.4.0, if metadata is provided to the `attr.ib` by its `metadata` argument, it will be propagated into the generated field. This metadata can be used by other projects such as apispec but is otherwise ignored by this library at this time.

Warning: If you use attrs to generate a class and then create a subclass not handled by attrs, this extension with throw an *AnnotationConversionError* if additional non-class level fields are added:

```
@attr.s(auto_attribs=True)
class SomeClass:
    x: int
    y: int

class SomeSubClass(SomeClass):
    z: int
```

Attempting to generate an *AttrsSchema* from `SomeSubClass` will fail as there is no matching `attr.ib` for `z`.

Provided Classes

class marshmallow_annotations.ext.attrs.**AttrsConverter**(*, *registry: marshmallow_annotations.base.TypeRegistry = <marshmallow_annotations.registry.DefaultTypeRegistry object>*) → None

class marshmallow_annotations.ext.attrs.**AttrsSchema**(*extra=None*, *only=None*, *exclude=()*, *prefix=''*, *strict=None*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)

Schema for handling `attrs` based targets, adds automatic load conversion into the target class and specifies the *AttrsConverter* as the converter factory.

2.4.2 typing.NamedTuple

If you are working with `typing.NamedTuple` definitions, you may use the extension *NamedTupleSchema* to generate your schema.

NamedTuple Type API

This extension modifies loading behavior to deserialize directly to instances of your defined `NamedTuple`:

```
from marshmallow_annotations import NamedTupleSchema
from typing import NamedTuple, Optional

class Vector(NamedTuple):
    x: int
    y: Optional[int]
    z: Optional[int] = 5

class VectorSchema(NamedTupleSchema):
    class Meta:
        target = Vector

schema = VectorSchema()
schema.load({'x': 1}).data

# Vector(x=1, y=None, z=5)

schema.dump(Vector(x=1, y=None, z=5)).data

# {'x': 1, 'y': None, 'z': 5}
```

Additionally, the `Meta` class provides you with the option flag `dump_default_args` to control whether attribute values matching defaults should be dumped or ignored; by default, matches are dumped:

```
class VectorSchemaDropDefaults(NamedTupleSchema):
    class Meta:
        target = Vector
        dump_default_args = False

schema = VectorSchemaDropDefaults()
schema.dump(Vector(x=1, y=None, z=5)).data

# {'x': 1}
```

NamedTuple Converter

class marshmallow_annotations.ext.namedtuple.**NamedTupleConverter**(*, *registry: marshmallow_annotations.base.TypeRegistry = <marshmallow_annotations.registry.DefaultTypeRegistry object>*) → None

NamedTuple Schema

class marshmallow_annotations.ext.namedtuple.**NamedTupleSchema**(*extra=None*, *only=None*, *exclude=()*, *prefix=''*, *strict=None*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)

Derived class for creating typing.NamedTuple schema with automatic post-load conversion to namedtuple instances.

class marshmallow_annotations.ext.namedtuple.**NamedTupleSchemaOpts**(*meta*, **args*, ***kwargs*)

NamedTuple specific AnnotationSchemaOpts, additionally provides:

- dump_default_fields

2.5 Change Log

2.5.1 Version 2.4.0 (2018-12-12)

- Dict field converter (see docs for details and cavaets) (#21)
- Fixed issue with AttrsConverter that would treat non-attrs based classes in containers as if they were attrs based (#25)
- Retain metadata attached to attr.ib fields when generating schema (#26)

2.5.2 Version 2.3.0 (2018-08-05)

- Python 3.7 support

2.5.3 Version 2.2.0 (2018-07-14)

- Specialized extension handling for attrs based classes

2.5.4 Version 2.1.0 (2018-07-01)

- Specialized extension handling for typing.NamedTuple

2.5.5 Version 2.0.0 (2018-05-02)

- Build schema from annotated class definitions
- Register custom types and handlers to generate fields
- Documentation

2.5.6 Version 1.0.0 (2017-06-17)

- It lives
- Create schemas using type annotations

Index

Symbols

A

B

C

D

F

G

H

I

M

N

R

S

T