marshmallow

Release 2.21.0

unknown

Sep 07, 2020

CONTENTS

Release v2.21.0. (*Changelog*)

marshmallow is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes.

```
from datetime import date
from marshmallow import Schema, fields, pprint

class ArtistSchema(Schema):
    name = fields.Str()

class AlbumSchema(Schema):
    title = fields.Str()
    release_date = fields.Date()
    artist = fields.Nested(ArtistSchema())

bowie = dict(name='David Bowie')
album = dict(artist=bowie, title='Hunky Dory', release_date=date(1971, 12, 17))

schema = AlbumSchema()
result = schema.dump(album)
pprint(result.data, indent=2)
# { 'artist': {'name': 'David Bowie'},
#   'release_date': '1971-12-17',
#   'title': 'Hunky Dory'}
```

In short, marshmallow schemas can be used to:

- **Validate** input data.
- **Deserialize** input data to app-level objects.
- **Serialize** app-level objects to primitive Python types. The serialized objects can then be rendered to standard formats such as JSON for use in an HTTP API.

CHAPTER

ONE

GET IT NOW

```
$ pip install -U marshmallow
```

Ready to get started? Go on to the *Quickstart tutorial* or check out some *Examples*.

CHAPTER

TWO

UPGRADING FROM AN OLDER VERSION?

See the *Upgrading to Newer Releases* page for notes on getting your code up-to-date with the latest version.

CHAPTER

THREE

WHY ANOTHER LIBRARY?

See *this document* to learn about what makes marshmallow unique.

CHAPTER

FOUR

GUIDE

4.1 Installation

marshmallow requires Python >= 2.7 or >= 3.4. It has no external dependencies other than the Python standard library.

Note: The python-dateutil package is not a hard dependency, but it is recommended for robust datetime deserialization.

```
$ pip install 'python-dateutil>=2.7.0'
```

4.1.1 Installing/Upgrading from the PyPI

To install the latest stable version from the PyPI:

```
$ pip install -U marshmallow
```

To install the latest pre-release version from the PyPI:

```
$ pip install -U marshmallow --pre
```

To install marshmallow with the recommended soft dependencies:

```
$ pip install -U marshmallow[reco]
```

4.1.2 Get the Bleeding Edge Version

To get the latest development version of marshmallow, run

```
$ pip install -U git+https://github.com/marshmallow-code/marshmallow.git@dev
```

See also:

Need help upgrading to newer releases? See the *Upgrading to Newer Releases* page.

4.2 Quickstart

This guide will walk you through the basics of creating schemas for serializing and deserializing data.

4.2.1 Declaring Schemas

Let's start with a basic user "model".

```
import datetime as dt

class User(object):
    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.created_at = dt.datetime.now()

    def __repr__(self):
        return '<User(name={self.name!r})>'.format(self=self)
```

Create a schema by defining a class with variables mapping attribute names to *Field* objects.

```
from marshmallow import Schema, fields

class UserSchema(Schema):
    name = fields.Str()
    email = fields.Email()
    created_at = fields.DateTime()
```

See also:

For a full reference on the available field classes, see the *API Docs*.

4.2.2 Serializing Objects ("Dumping")

Serialize objects by passing them to your schema's *dump* method, which returns the formatted result (as well as a dictionary of validation errors, which we'll *revisit later*).

```
from marshmallow import pprint

user = User(name="Monty", email="monty@python.org")
schema = UserSchema()
result = schema.dump(user)
pprint(result.data)
# {"name": "Monty",
#  "email": "monty@python.org",
#  "created_at": "2014-08-17T14:54:16.049594+00:00"}
```

You can also serialize to a JSON-encoded string using *dumps*.

```
json_result = schema.dumps(user)
pprint(json_result.data)
# '{"name": "Monty", "email": "monty@python.org", "created_at": "2014-08-17T14:54:16.
↪049594+00:00"}'
```

Filtering output

You may not need to output all declared fields every time you use a schema. You can specify which fields to output with the `only` parameter.

```
summary_schema = UserSchema(only=('name', 'email'))
summary_schema.dump(user).data
# {"name": "Monty Python", "email": "monty@python.org"}
```

You can also exclude fields by passing in the `exclude` parameter.

4.2.3 Deserializing Objects ("Loading")

The opposite of the *dump* method is the *load* method, which deserializes an input dictionary to an application-level data structure.

By default, *load* will return a dictionary of field names mapped to the deserialized values.

```
from pprint import pprint

user_data = {
    'created_at': '2014-08-11T05:26:03.869245',
    'email': u'ken@yahoo.com',
    'name': u'Ken'
}
schema = UserSchema()
result = schema.load(user_data)
pprint(result.data)
# {'name': 'Ken',
#  'email': 'ken@yahoo.com',
#  'created_at': datetime.datetime(2014, 8, 11, 5, 26, 3, 869245)},
```

Notice that the datetime string was converted to a `datetime` object.

Deserializing to Objects

In order to deserialize to an object, define a method of your *Schema* and decorate it with *post_load*. The method receives a dictionary of deserialized data as its only parameter.

```
from marshmallow import Schema, fields, post_load

class UserSchema(Schema):
    name = fields.Str()
    email = fields.Email()
    created_at = fields.DateTime()

    @post_load
    def make_user(self, data):
        return User(**data)
```

Now, the *load* method will return a `User` object.

```
user_data = {
    'name': 'Ronnie',
    'email': 'ronnie@stones.com'
}
schema = UserSchema()
result = schema.load(user_data)
result.data  # => <User(name='Ronnie')>
```

4.2.4 Handling Collections of Objects

Iterable collections of objects are also serializable and deserializable. Just set `many=True`.

```
user1 = User(name="Mick", email="mick@stones.com")
user2 = User(name="Keith", email="keith@stones.com")
users = [user1, user2]
schema = UserSchema(many=True)
result = schema.dump(users)  # OR UserSchema().dump(users, many=True)
result.data
# [{'name': u'Mick',
#   'email': u'mick@stones.com',
#   'created_at': '2014-08-17T14:58:57.600623+00:00'}
#  {'name': u'Keith',
#   'email': u'keith@stones.com',
#   'created_at': '2014-08-17T14:58:57.600623+00:00'}]
```

4.2.5 Validation

Schema.load() (and its JSON-decoding counterpart, *Schema.loads()*) returns a dictionary of validation errors as the second element of its return value. Some fields, such as the *Email* and *URL* fields, have built-in validation.

```
data, errors = UserSchema().load({'email': 'foo'})
errors  # => {'email': ['"foo" is not a valid email address.']}
# OR, equivalently
result = UserSchema().load({'email': 'foo'})
result.errors  # => {'email': ['"foo" is not a valid email address.']}
```

When validating a collection, the errors dictionary will be keyed on the indices of invalid items.

```
class BandMemberSchema(Schema):
    name = fields.String(required=True)
    email = fields.Email()

user_data = [
    {'email': 'mick@stones.com', 'name': 'Mick'},
    {'email': 'invalid', 'name': 'Invalid'},  # invalid email
    {'email': 'keith@stones.com', 'name': 'Keith'},
    {'email': 'charlie@stones.com'},  # missing "name"
]

result = BandMemberSchema(many=True).load(user_data)
result.errors
# {1: {'email': ['"invalid" is not a valid email address.']},
#  3: {'name': ['Missing data for required field.']}}
```

You can perform additional validation for a field by passing it a `validate` callable (function, lambda, or object with `__call__` defined).

```
class ValidatedUserSchema(UserSchema):
    # NOTE: This is a contrived example.
    # You could use marshmallow.validate.Range instead of an anonymous function here
    age = fields.Number(validate=lambda n: 18 <= n <= 40)

in_data = {'name': 'Mick', 'email': 'mick@stones.com', 'age': 71}
```

(continues on next page)

(continued from previous page)

```
result = ValidatedUserSchema().load(in_data)
result.errors  # => {'age': ['Validator <lambda>(71.0) is False']}
```

Validation functions either return a boolean or raise a `ValidationError`. If a *`ValidationError`* is raised, its message is stored when validation fails.

```
from marshmallow import Schema, fields, ValidationError

def validate_quantity(n):
    if n < 0:
        raise ValidationError('Quantity must be greater than 0.')
    if n > 30:
        raise ValidationError('Quantity must not be greater than 30.')

class ItemSchema(Schema):
    quantity = fields.Integer(validate=validate_quantity)

in_data = {'quantity': 31}
result, errors = ItemSchema().load(in_data)
errors  # => {'quantity': ['Quantity must not be greater than 30.']}
```

Note: If you have multiple validations to perform, you may also pass a collection (list, tuple, generator) of callables.

Note: *`Schema.dump()`* also returns a dictionary of errors, which will include any `ValidationErrors` raised during serialization. However, `required`, `allow_none`, `validate`, *`@validates`*, and *`@validates_schema`* only apply during deserialization.

Field Validators as Methods

It is often convenient to write validators as methods. Use the *`validates`* decorator to register field validator methods.

```
from marshmallow import fields, Schema, validates, ValidationError

class ItemSchema(Schema):
    quantity = fields.Integer()

    @validates('quantity')
    def validate_quantity(self, value):
        if value < 0:
            raise ValidationError('Quantity must be greater than 0.')
        if value > 30:
            raise ValidationError('Quantity must not be greater than 30.')
```

`strict` Mode

If you set `strict=True` in either the Schema constructor or as a `class Meta` option, an error will be raised when invalid data are passed in. You can access the dictionary of validation errors from the *`ValidationError.messages`* attribute.

```
from marshmallow import ValidationError

try:
    UserSchema(strict=True).load({'email': 'foo'})
except ValidationError as err:
    print(err.messages)# => {'email': ['"foo" is not a valid email address.
↪']}
```

See also:

You can register a custom error handler function for a schema by overriding the *handle_error* method. See the *Extending Schemas* page for more info.

See also:

Need schema-level validation? See the *Extending Schemas* page.

Required Fields

You can make a field required by passing `required=True`. An error will be stored if the the value is missing from the input to *Schema.load()*.

To customize the error message for required fields, pass a `dict` with a `required` key as the `error_messages` argument for the field.

```
class UserSchema(Schema):
    name = fields.String(required=True)
    age = fields.Integer(
        required=True,
        error_messages={'required': 'Age is required.'}
    )
    city = fields.String(
        required=True,
        error_messages={'required': {'message': 'City required', 'code': 400}}
    )
    email = fields.Email()

data, errors = UserSchema().load({'email': 'foo@bar.com'})
errors
# {'name': ['Missing data for required field.'],
#  'age': ['Age is required.'],
#  'city': {'message': 'City required', 'code': 400}}
```

Partial Loading

When using the same schema in multiple places, you may only want to check required fields some of the time when deserializing by specifying them in `partial`.

```
class UserSchema(Schema):
    name = fields.String(required=True)
    age = fields.Integer(required=True)

data, errors = UserSchema().load({'age': 42}, partial=('name',))
# OR UserSchema(partial=('name',)).load({'age': 42})
data, errors  # => ({'age': 42}, {})
```

Or you can ignore missing fields entirely by setting `partial=True`.

```
class UserSchema(Schema):
    name = fields.String(required=True)
    age = fields.Integer(required=True)

data, errors = UserSchema().load({'age': 42}, partial=True)
# OR UserSchema(partial=True).load({'age': 42})
data, errors  # => ({'age': 42}, {})
```

Schema.validate

If you only need to validate input data (without deserializing to an object), you can use *Schema.validate()*.

```
errors = UserSchema().validate({'name': 'Ronnie', 'email': 'invalid-email'})
errors  # {'email': ['"invalid-email" is not a valid email address.']}
```

4.2.6 Specifying Attribute Names

By default, `Schemas` will marshal the object attributes that are identical to the schema's field names. However, you may want to have different field and attribute names. In this case, you can explicitly specify which attribute names to use.

```
class UserSchema(Schema):
    name = fields.String()
    email_addr = fields.String(attribute="email")
    date_created = fields.DateTime(attribute="created_at")

user = User('Keith', email='keith@stones.com')
ser = UserSchema()
result, errors = ser.dump(user)
pprint(result)
# {'name': 'Keith',
#  'email_addr': 'keith@stones.com',
#  'date_created': '2014-08-17T14:58:57.600623+00:00'}
```

4.2.7 Specifying Deserialization Keys

By default `Schemas` will unmarshal an input dictionary to an output dictionary whose keys are identical to the field names. However, if you are consuming data that does not exactly match your schema, you can specify additional keys to load values by passing the `load_from` argument.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.Email(load_from='emailAddress')

data = {
    'name': 'Mike',
    'emailAddress': 'foo@bar.com'
}
s = UserSchema()
result, errors = s.load(data)
#{'name': u'Mike',
# 'email': 'foo@bar.com'}
```

4.2.8 Specifying Serialization Keys

If you want to marshal a field to a different key than the field name you can use `dump_to`, which is analogous to `load_from`.

```
class UserSchema(Schema):
    name = fields.String(dump_to='TheName')
    email = fields.Email(load_from='CamelCasedEmail', dump_to='CamelCasedEmail')

data = {
    'name': 'Mike',
    'email': 'foo@bar.com'
}
s = UserSchema()
result, errors = s.dump(data)
#{'TheName': u'Mike',
# 'CamelCasedEmail': 'foo@bar.com'}
```

4.2.9 Refactoring: Implicit Field Creation

When your model has many attributes, specifying the field type for every attribute can get repetitive, especially when many of the attributes are already native Python datatypes.

The *class Meta* paradigm allows you to specify which attributes you want to serialize. Marshmallow will choose an appropriate field type based on the attribute's type.

Let's refactor our User schema to be more concise.

```
# Refactored schema
class UserSchema(Schema):
    uppername = fields.Function(lambda obj: obj.name.upper())
    class Meta:
        fields = ("name", "email", "created_at", "uppername")
```

Note that `name` will be automatically formatted as a *`String`* and `created_at` will be formatted as a *`DateTime`*.

Note: If instead you want to specify which field names to include *in addition* to the explicitly declared fields, you can use the `additional` option.

The schema below is equivalent to above:

```
class UserSchema(Schema):
    uppername = fields.Function(lambda obj: obj.name.upper())
    class Meta:
        # No need to include 'uppername'
        additional = ("name", "email", "created_at")
```

4.2.10 Ordering Output

For some use cases, it may be useful to maintain field ordering of serialized output. To enable ordering, set the `ordered` option to `True`. This will instruct marshmallow to serialize data to a `collections.OrderedDict`.

```
from collections import OrderedDict

class UserSchema(Schema):
    uppername = fields.Function(lambda obj: obj.name.upper())
    class Meta:
        fields = ("name", "email", "created_at", "uppername")
        ordered = True

u = User('Charlie', 'charlie@stones.com')
schema = UserSchema()
result = schema.dump(u)
assert isinstance(result.data, OrderedDict)
# marshmallow's pprint function maintains order
pprint(result.data, indent=2)
# {
#   "name": "Charlie",
#   "email": "charlie@stones.com",
#   "created_at": "2014-10-30T08:27:48.515735+00:00",
#   "uppername": "CHARLIE"
# }
```

4.2.11 "Read-only" and "Write-only" Fields

In the context of a web API, the `dump_only` and `load_only` parameters are conceptually equivalent to "read-only" and "write-only" fields, respectively.

```
class UserSchema(Schema):
    name = fields.Str()
    # password is "write-only"
    password = fields.Str(load_only=True)
    # created_at is "read-only"
    created_at = fields.DateTime(dump_only=True)
```

4.2.12 Next Steps

- Need to represent relationships between objects? See the *Nesting Schemas* page.
- Want to create your own field type? See the *Custom Fields* page.
- Need to add schema-level validation, post-processing, or error handling behavior? See the *Extending Schemas* page.
- For example applications using marshmallow, check out the *Examples* page.

4.3 Nesting Schemas

Schemas can be nested to represent relationships between objects (e.g. foreign key relationships). For example, a `Blog` may have an author represented by a User object.

```
import datetime as dt

class User(object):
    def __init__(self, name, email):
```

(continues on next page)

(continued from previous page)

```
        self.name = name
        self.email = email
        self.created_at = dt.datetime.now()
        self.friends = []
        self.employer = None

class Blog(object):
    def __init__(self, title, author):
        self.title = title
        self.author = author  # A User object
```

Use a *Nested* field to represent the relationship, passing in a nested schema class.

```
from marshmallow import Schema, fields, pprint

class UserSchema(Schema):
    name = fields.String()
    email = fields.Email()
    created_at = fields.DateTime()

class BlogSchema(Schema):
    title = fields.String()
    author = fields.Nested(UserSchema)
```

The serialized blog will have the nested user representation.

```
user = User(name="Monty", email="monty@python.org")
blog = Blog(title="Something Completely Different", author=user)
result, errors = BlogSchema().dump(blog)
pprint(result)
# {'title': u'Something Completely Different',
# {'author': {'name': u'Monty',
#             'email': u'monty@python.org',
#             'created_at': '2014-08-17T14:58:57.600623+00:00'}}
```

Note: If the field is a collection of nested objects, you must set `many=True`.

```
collaborators = fields.Nested(UserSchema, many=True)
```

4.3.1 Specifying Which Fields to Nest

You can explicitly specify which attributes of the nested objects you want to serialize with the `only` argument.

```
class BlogSchema2(Schema):
    title = fields.String()
    author = fields.Nested(UserSchema, only=["email"])

schema = BlogSchema2()
result, errors = schema.dump(blog)
pprint(result)
# {
#     'title': u'Something Completely Different',
```

(continues on next page)

(continued from previous page)

```
#     'author': {'email': u'monty@python.org'}
# }
```

You can represent the attributes of deeply nested objects using dot delimiters.

```
class SiteSchema(Schema):
    blog = fields.Nested(BlogSchema2)

schema = SiteSchema(only=['blog.author.email'])
result, errors = schema.dump(site)
pprint(result)
# {
#     'blog': {
#         'author': {'email': u'monty@python.org'}
#     }
# }
```

Note: If you pass in a string field name to `only`, only a single value (or flat list of values if `many=True`) will be returned.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.Email()
    friends = fields.Nested('self', only='name', many=True)
# ... create ``user`` ...
result, errors = UserSchema().dump(user)
pprint(result)
# {
#     "name": "Steve",
#     "email": "steve@example.com",
#     "friends": ["Mike", "Joe"]
# }
```

You can also exclude fields by passing in an `exclude` list. This argument also allows representing the attributes of deeply nested objects using dot delimiters.

4.3.2 Two-way Nesting

If you have two objects that nest each other, you can refer to a nested schema by its class name. This allows you to nest Schemas that have not yet been defined.

For example, a representation of an `Author` model might include the books that have a foreign-key (many-to-one) relationship to it. Correspondingly, a representation of a `Book` will include its author representation.

```
class AuthorSchema(Schema):
    # Make sure to use the 'only' or 'exclude' params
    # to avoid infinite recursion
    books = fields.Nested('BookSchema', many=True, exclude=('author', ))
    class Meta:
        fields = ('id', 'name', 'books')

class BookSchema(Schema):
    author = fields.Nested(AuthorSchema, only=('id', 'name'))
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        fields = ('id', 'title', 'author')
```

```
from marshmallow import pprint
from mymodels import Author, Book

author = Author(name='William Faulkner')
book = Book(title='As I Lay Dying', author=author)
book_result, errors = BookSchema().dump(book)
pprint(book_result, indent=2)
# {
#   "id": 124,
#   "title": "As I Lay Dying",
#   "author": {
#     "id": 8,
#     "name": "William Faulkner"
#   }
# }

author_result, errors = AuthorSchema().dump(author)
pprint(author_result, indent=2)
# {
#   "id": 8,
#   "name": "William Faulkner",
#   "books": [
#     {
#       "id": 124,
#       "title": "As I Lay Dying"
#     }
#   ]
# }
```

Note: If you need to, you can also pass the full, module-qualified path to `fields.Nested`.

```
books = fields.Nested('path.to.BookSchema',
                      many=True, exclude=('author', ))
```

4.3.3 Nesting A Schema Within Itself

If the object to be marshalled has a relationship to an object of the same type, you can nest the `Schema` within itself by passing `"self"` (with quotes) to the *`Nested`* constructor.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.Email()
    friends = fields.Nested('self', many=True)
    # Use the 'exclude' argument to avoid infinite recursion
    employer = fields.Nested('self', exclude=('employer', ), default=None)

user = User("Steve", 'steve@example.com')
user.friends.append(User("Mike", 'mike@example.com'))
user.friends.append(User('Joe', 'joe@example.com'))
```

(continues on next page)

(continued from previous page)

```
user.employer = User('Dirk', 'dirk@example.com')
result = UserSchema().dump(user)
pprint(result.data, indent=2)
# {
#     "name": "Steve",
#     "email": "steve@example.com",
#     "friends": [
#         {
#             "name": "Mike",
#             "email": "mike@example.com",
#             "friends": [],
#             "employer": null
#         },
#         {
#             "name": "Joe",
#             "email": "joe@example.com",
#             "friends": [],
#             "employer": null
#         }
#     ],
#     "employer": {
#         "name": "Dirk",
#         "email": "dirk@example.com",
#         "friends": []
#     }
# }
```

4.3.4 Next Steps

- Want to create your own field type? See the *Custom Fields* page.
- Need to add schema-level validation, post-processing, or error handling behavior? See the *Extending Schemas* page.
- For example applications using marshmallow, check out the *Examples* page.

4.4 Custom Fields

There are three ways to create a custom-formatted field for a `Schema`:

- Create a custom *`Field`* class
- Use a *`Method`* field
- Use a *`Function`* field

The method you choose will depend on the manner in which you intend to reuse the field.

4.4.1 Creating A Field Class

To create a custom field class, create a subclass of *`marshmallow.fields.Field`* and implement its *`_serialize`*, and/or *`_deserialize`* methods.

```
from marshmallow import fields

class Titlecased(fields.Field):
    def _serialize(self, value, attr, obj):
        if value is None:
            return ''
        return value.title()

class UserSchema(Schema):
    name = fields.String()
    email = fields.String()
    created_at = fields.DateTime()
    titlename = TitleCased(attribute="name")
```

4.4.2 Method Fields

A *Method* field will serialize to the value returned by a method of the Schema. The method must take an `obj` parameter which is the object to be serialized.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.String()
    created_at = fields.DateTime()
    since_created = fields.Method("get_days_since_created")

    def get_days_since_created(self, obj):
        return dt.datetime.now().day - obj.created_at.day
```

4.4.3 Function Fields

A *Function* field will serialize the value of a function that is passed directly to it. Like a *Method* field, the function must take a single argument `obj`.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.String()
    created_at = fields.DateTime()
    uppername = fields.Function(lambda obj: obj.name.upper())
```

4.4.4 `Method` and `Function` field deserialization

Both *Function* and *Method* receive an optional `deserialize` argument which defines how the field should be deserialized. The method or function passed to `deserialize` receives the input value for the field.

```
class UserSchema(Schema):
    # `Method` takes a method name (str), Function takes a callable
    balance = fields.Method('get_balance', deserialize='load_balance')

    def get_balance(self, obj):
        return obj.income - obj.debt

    def load_balance(self, value):
```

(continues on next page)

(continued from previous page)

```
        return float(value)


schema = UserSchema()
result = schema.load({'balance': '100.00'})
result.data['balance']  # => 100.0
```

4.4.5 Adding Context to `Method` and `Function` Fields

A *`Function`* or *`Method`* field may need information about its environment to know how to serialize a value.

In these cases, you can set the `context` attribute (a dictionary) of a `Schema`. *`Function`* and *`Method`* fields will have access to this dictionary.

As an example, you might want your `UserSchema` to output whether or not a `User` is the author of a `Blog` or whether a certain word appears in a `Blog's` title.

```
class UserSchema(Schema):
    name = fields.String()
    # Function fields optionally receive context argument
    is_author = fields.Function(lambda user, context: user == context['blog'].author)
    likes_bikes = fields.Method('writes_about_bikes')

    # Method fields also optionally receive context argument
    def writes_about_bikes(self, user):
        return 'bicycle' in self.context['blog'].title.lower()

schema = UserSchema()

user = User('Freddie Mercury', 'fred@queen.com')
blog = Blog('Bicycle Blog', author=user)

schema.context = {'blog': blog}
data, errors = schema.dump(user)
data['is_author']  # => True
data['likes_bikes']  # => True
```

4.4.6 Customizing Error Messages

Validation error messages for fields can be configured at the class or instance level.

At the class level, default error messages are defined as a mapping from error codes to error messages.

```
from marshmallow import fields

class MyDate(fields.Date):
    default_error_messages = {
        'invalid': 'Please provide a valid date.',
    }
```

Note: A `Field's default_error_messages` dictionary gets merged with its parent classes' `default_error_messages` dictionaries.

Error messages can also be passed to a `Field`'s constructor.

```
from marshmallow import Schema, fields

class UserSchema(Schema):

    name = fields.Str(
        required=True,
        error_messages={'required': 'Please provide a name.'}
    )
```

4.4.7 Next Steps

- Need to add schema-level validation, post-processing, or error handling behavior? See the *Extending Schemas* page.
- For example applications using marshmallow, check out the *Examples* page.

4.5 Extending Schemas

4.5.1 Pre-processing and Post-processing Methods

Data pre-processing and post-processing methods can be registered using the *pre_load*, *post_load*, *pre_dump*, and *post_dump* decorators.

```
from marshmallow import Schema, fields, pre_load

class UserSchema(Schema):
    name = fields.Str()
    slug = fields.Str()

    @pre_load
    def slugify_name(self, in_data):
        in_data['slug'] = in_data['slug'].lower().strip().replace(' ', '-')
        return in_data

schema = UserSchema()
result, errors = schema.load({'name': 'Steve', 'slug': 'Steve Loria '})
result['slug']  # => 'steve-loria'
```

Passing "many"

By default, pre- and post-processing methods receive one object/datum at a time, transparently handling the `many` parameter passed to the schema at runtime.

In cases where your pre- and post-processing methods need to receive the input collection when `many=True`, add `pass_many=True` to the method decorators. The method will receive the input data (which may be a single datum or a collection) and the boolean value of `many`.

Example: Enveloping

One common use case is to wrap data in a namespace upon serialization and unwrap the data during deserialization.

```
from marshmallow import Schema, fields, pre_load, post_load, post_dump

class BaseSchema(Schema):
    # Custom options
    __envelope__ = {
        'single': None,
        'many': None
    }
    __model__ = User

    def get_envelope_key(self, many):
        """Helper to get the envelope key."""
        key = self.__envelope__['many'] if many else self.__envelope__['single']
        assert key is not None, "Envelope key undefined"
        return key

    @pre_load(pass_many=True)
    def unwrap_envelope(self, data, many):
        key = self.get_envelope_key(many)
        return data[key]

    @post_dump(pass_many=True)
    def wrap_with_envelope(self, data, many):
        key = self.get_envelope_key(many)
        return {key: data}

    @post_load
    def make_object(self, data):
        return self.__model__(**data)

class UserSchema(BaseSchema):
    __envelope__ = {
        'single': 'user',
        'many': 'users',
    }
    __model__ = User
    name = fields.Str()
    email = fields.Email()

user_schema = UserSchema()

user = User('Mick', email='mick@stones.org')
user_data = user_schema.dump(user).data
# {'user': {'email': 'mick@stones.org', 'name': 'Mick'}}

users = [User('Keith', email='keith@stones.org'),
        User('Charlie', email='charlie@stones.org')]
users_data = user_schema.dump(users, many=True).data
# {'users': [{'email': 'keith@stones.org', 'name': 'Keith'},
#            {'email': 'charlie@stones.org', 'name': 'Charlie'}]}

user_objs = user_schema.load(users_data, many=True).data
# [<User(name='Keith Richards')>, <User(name='Charlie Watts')>]
```

Raising Errors in Pre-/Post-processor Methods

Pre- and post-processing methods may raise a *ValidationError*. By default, errors will be stored on the `"_schema"` key in the errors dictionary.

```
from marshmallow import Schema, fields, ValidationError, pre_load

class BandSchema(Schema):
    name = fields.Str()

    @pre_load
    def unwrap_envelope(self, data):
        if 'data' not in data:
            raise ValidationError('Input data must have a "data" key.')
        return data['data']

sch = BandSchema()
sch.load({'name': 'The Band'}).errors
# {'_schema': ['Input data must have a "data" key.']}
```

If you want to store and error on a different key, pass the key name as the second argument to *ValidationError*.

```
from marshmallow import Schema, fields, ValidationError, pre_load

class BandSchema(Schema):
    name = fields.Str()

    @pre_load
    def unwrap_envelope(self, data):
        if 'data' not in data:
            raise ValidationError('Input data must have a "data" key.', '_
↪preprocessing')
        return data['data']

sch = BandSchema()
sch.load({'name': 'The Band'}).errors
# {'_preprocessing': ['Input data must have a "data" key.']}
```

Pre-/Post-processor Invocation Order

In summary, the processing pipeline for deserialization is as follows:

1. `@pre_load(pass_many=True)` methods
2. `@pre_load(pass_many=False)` methods
3. `load(in_data, many)` (validation and deserialization)
4. `@post_load(pass_many=True)` methods
5. `@post_load(pass_many=False)` methods

The pipeline for serialization is similar, except that the "pass_many" processors are invoked *after* the "non-raw" processors.

1. `@pre_dump(pass_many=False)` methods
2. `@pre_dump(pass_many=True)` methods
3. `dump(obj, many)` (serialization)

4. `@post_dump(pass_many=False)` methods
5. `@post_dump(pass_many=True)` methods

Warning: You may register multiple processor methods on a Schema. Keep in mind, however, that **the invocation order of decorated methods of the same type is not guaranteed**. If you need to guarantee order of processing steps, you should put them in the same method.

```
from marshmallow import Schema, fields, pre_load

# YES
class MySchema(Schema):
    field_a = fields.Field()

    @pre_load
    def preprocess(self, data):
        step1_data = self.step1(data)
        step2_data = self.step2(step1_data)
        return step2_data

    def step1(self, data):
        # ...

    # Depends on step1
    def step2(self, data):
        # ...

# NO
class MySchema(Schema):
    field_a = fields.Field()

    @pre_load
    def step1(self, data):
        # ...

    # Depends on step1
    @pre_load
    def step2(self, data):
        # ...
```

4.5.2 Handling Errors

By default, *Schema.dump()* and *Schema.load()* will return validation errors as a dictionary (unless `strict` mode is enabled).

You can specify a custom error-handling function for a *Schema* by overriding the *handle_error* method. The method receives the *ValidationError* and the original object (or input data if deserializing) to be (de)serialized.

```
import logging
from marshmallow import Schema, fields

class AppError(Exception):
    pass

class UserSchema(Schema):
```

(continues on next page)

(continued from previous page)

```
    email = fields.Email()

    def handle_error(self, exc, data):
        """Log and raise our custom exception when (de)serialization fails."""
        logging.error(exc.messages)
        raise AppError('An error occurred with input: {0}'.format(data))

schema = UserSchema()
schema.load({'email': 'invalid-email'})  # raises AppError
```

4.5.3 Schema-level Validation

You can register schema-level validation functions for a *Schema* using the *marshmallow.validates_schema* decorator. Schema-level validation errors will be stored on the `_schema` key of the errors dictonary.

```
from marshmallow import Schema, fields, validates_schema, ValidationError

class NumberSchema(Schema):
    field_a = fields.Integer()
    field_b = fields.Integer()

    @validates_schema
    def validate_numbers(self, data):
        if data['field_b'] >= data['field_a']:
            raise ValidationError('field_a must be greater than field_b')

schema = NumberSchema()
result, errors = schema.load({'field_a': 1, 'field_b': 2})
errors['_schema'] # => ["field_a must be greater than field_b"]
```

Validating Original Input Data

Normally, unspecified field names are ignored by the validator. If you would like access to the original, raw input (e.g. to fail validation if an unknown field name is sent), add `pass_original=True` to your call to *validates_schema*.

```
from marshmallow import Schema, fields, validates_schema, ValidationError

class MySchema(Schema):
    foo = fields.Int()
    bar = fields.Int()

    @validates_schema(pass_original=True)
    def check_unknown_fields(self, data, original_data):
        unknown = set(original_data) - set(self.fields)
        if unknown:
            raise ValidationError('Unknown field', unknown)

schema = MySchema()
errors = schema.load({'foo': 1, 'bar': 2, 'baz': 3, 'bu': 4}).errors
# {'baz': 'Unknown field', 'bu': 'Unknown field'}
```

Storing Errors on Specific Fields

If you want to store schema-level validation errors on a specific field, you can pass a field name (or multiple field names) to the *ValidationError*.

```
class NumberSchema(Schema):
    field_a = fields.Integer()
    field_b = fields.Integer()

    @validates_schema
    def validate_numbers(self, data):
        if data['field_b'] >= data['field_a']:
            raise ValidationError(
                'field_a must be greater than field_b',
                'field_a'
            )

schema = NumberSchema()
result, errors = schema.load({'field_a': 2, 'field_b': 1})
errors['field_a'] # => ["field_a must be greater than field_b"]
```

4.5.4 Overriding how attributes are accessed

By default, marshmallow uses the *utils.get_value* function to pull attributes from various types of objects for serialization. This will work for *most* use cases.

However, if you want to specify how values are accessed from an object, you can override the *get_attribute* method.

```
class UserDictSchema(Schema):
    name = fields.Str()
    email = fields.Email()

    # If we know we're only serializing dictionaries, we can
    # use dict.get for all input objects
    def get_attribute(self, key, obj, default):
        return obj.get(key, default)
```

4.5.5 Custom "class Meta" Options

`class Meta` options are a way to configure and modify a *Schema's* behavior. See the *API docs* for a listing of available options.

You can add custom `class Meta` options by subclassing *SchemaOpts*.

Example: Enveloping, Revisited

Let's build upon the example above for adding an envelope to serialized output. This time, we will allow the envelope key to be customizable with `class Meta` options.

```
# Example outputs
{
    'user': {
```

(continues on next page)

(continued from previous page)

```
        'name': 'Keith',
        'email': 'keith@stones.com'
    }
}
# List output
{
    'users': [{'name': 'Keith'}, {'name': 'Mick'}]
}
```

First, we'll add our namespace configuration to a custom options class.

```
from marshmallow import Schema, SchemaOpts

class NamespaceOpts(SchemaOpts):
    """Same as the default class Meta options, but adds "name" and
    "plural_name" options for enveloping.
    """
    def __init__(self, meta):
        SchemaOpts.__init__(self, meta)
        self.name = getattr(meta, 'name', None)
        self.plural_name = getattr(meta, 'plural_name', self.name)
```

Then we create a custom *Schema* that uses our options class.

```
class NamespacedSchema(Schema):
    OPTIONS_CLASS = NamespaceOpts

    @pre_load(pass_many=True)
    def unwrap_envelope(self, data, many):
        key = self.opts.plural_name if many else self.opts.name
        return data[key]

    @post_dump(pass_many=True)
    def wrap_with_envelope(self, data, many):
        key = self.opts.plural_name if many else self.opts.name
        return {key: data}
```

Our application schemas can now inherit from our custom schema class.

```
class UserSchema(NamespacedSchema):
    name = fields.String()
    email = fields.Email()

    class Meta:
        name = 'user'
        plural_name = 'users'

ser = UserSchema()
user = User('Keith', email='keith@stones.com')
result = ser.dump(user)
result.data  # {"user": {"name": "Keith", "email": "keith@stones.com"}}
```

4.5.6 Using Context

The `context` attribute of a *Schema* is a general-purpose store for extra information that may be needed for (de)serialization. It may be used in both `Schema` and `Field` methods.

```
schema = UserSchema()
# Make current HTTP request available to
# custom fields, schema methods, schema validators, etc.
schema.context['request'] = request
schema.dump(user)
```

4.6 Examples

The examples below will use httpie (a curl-like tool) for testing the APIs.

4.6.1 Text Analysis API (Bottle + TextBlob)

Here is a very simple text analysis API using Bottle and TextBlob that demonstrates how to declare an object serializer.

Assume that `TextBlob` objects have `polarity`, `subjectivity`, `noun_phrase`, `tags`, and `words` properties.

```
from bottle import route, request, run
from textblob import TextBlob
from marshmallow import Schema, fields

class BlobSchema(Schema):
    polarity = fields.Float()
    subjectivity = fields.Float()
    chunks = fields.List(fields.String, attribute="noun_phrases")
    tags = fields.Raw()
    discrete_sentiment = fields.Method("get_discrete_sentiment")
    word_count = fields.Function(lambda obj: len(obj.words))

    def get_discrete_sentiment(self, obj):
        if obj.polarity > 0.1:
            return 'positive'
        elif obj.polarity < -0.1:
            return 'negative'
        else:
            return 'neutral'


blob_schema = BlobSchema()

@route("/api/v1/analyze", method="POST")
def analyze():
    blob = TextBlob(request.json['text'])
    result = blob_schema.dump(blob)
    return result.data


run(reloader=True, port=5000)
```

Using The API

First, run the app.

```
$ python textblob_example.py
```

Then send a POST request with some text.

```
$ http POST :5000/api/v1/analyze text="Simple is better"
HTTP/1.0 200 OK
Content-Length: 189
Content-Type: application/json
Date: Wed, 13 Nov 2013 08:58:40 GMT
Server: WSGIServer/0.1 Python/2.7.5

{
    "chunks": [
        "simple"
    ],
    "discrete_sentiment": "positive",
    "polarity": 0.25,
    "subjectivity": 0.4285714285714286,
    "tags": [
        [
            "Simple",
            "NN"
        ],
        [
            "is",
            "VBZ"
        ],
        [
            "better",
            "JJR"
        ]
    ],
    "word_count": 3
}
```

4.6.2 Quotes API (Flask + SQLAlchemy)

Below is a full example of a REST API for a quotes app using Flask and SQLAlchemy with marshmallow. It demonstrates a number of features, including:

- Validation and deserialization using *`Schema.load()`*.
- Custom validation
- Nesting fields
- Using `dump_only=True` to specify read-only fields
- Output filtering using the `only` parameter
- Using *`@pre_load`* to preprocess input data.

```
import datetime

from flask import Flask, jsonify, request
from flask.ext.sqlalchemy import SQLAlchemy
from sqlalchemy.exc import IntegrityError
from marshmallow import Schema, fields, ValidationError, pre_load

app = Flask(__name__)
```

(continues on next page)

(continued from previous page)

```
app.config["SQLALCHEMY_DATABASE_URI"] = 'sqlite:////tmp/quotes.db'
db = SQLAlchemy(app)

##### MODELS #####

class Author(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first = db.Column(db.String(80))
    last = db.Column(db.String(80))

class Quote(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    content = db.Column(db.String, nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey("author.id"))
    author = db.relationship("Author",
                         backref=db.backref("quotes", lazy="dynamic"))
    posted_at = db.Column(db.DateTime)

##### SCHEMAS #####

class AuthorSchema(Schema):
    id = fields.Int(dump_only=True)
    first = fields.Str()
    last = fields.Str()
    formatted_name = fields.Method("format_name", dump_only=True)

    def format_name(self, author):
        return "{}, {}".format(author.last, author.first)


# Custom validator
def must_not_be_blank(data):
    if not data:
        raise ValidationError('Data not provided.')

class QuoteSchema(Schema):
    id = fields.Int(dump_only=True)
    author = fields.Nested(AuthorSchema, validate=must_not_be_blank)
    content = fields.Str(required=True, validate=must_not_be_blank)
    posted_at = fields.DateTime(dump_only=True)

    # Allow client to pass author's full name in request body
    # e.g. {"author': 'Tim Peters"} rather than {"first": "Tim", "last": "Peters"}
    @pre_load
    def process_author(self, data):
        author_name = data.get('author')
        if author_name:
            first, last = author_name.split(' ')
            author_dict = dict(first=first, last=last)
        else:
            author_dict = {}
        data['author'] = author_dict
        return data


author_schema = AuthorSchema()
authors_schema = AuthorSchema(many=True)
```

(continues on next page)

(continued from previous page)

```
quote_schema = QuoteSchema()
quotes_schema = QuoteSchema(many=True, only=('id', 'content'))

##### API #####

@app.route('/authors')
def get_authors():
    authors = Author.query.all()
    # Serialize the queryset
    result = authors_schema.dump(authors)
    return jsonify({'authors': result.data})

@app.route("/authors/<int:pk>")
def get_author(pk):
    try:
        author = Author.query.get(pk)
    except IntegrityError:
        return jsonify({"message": "Author could not be found."}), 400
    author_result = author_schema.dump(author)
    quotes_result = quotes_schema.dump(author.quotes.all())
    return jsonify({'author': author_result.data, 'quotes': quotes_result.data})

@app.route('/quotes/', methods=['GET'])
def get_quotes():
    quotes = Quote.query.all()
    result = quotes_schema.dump(quotes)
    return jsonify({"quotes": result.data})

@app.route("/quotes/<int:pk>")
def get_quote(pk):
    try:
        quote = Quote.query.get(pk)
    except IntegrityError:
        return jsonify({"message": "Quote could not be found."}), 400
    result = quote_schema.dump(quote)
    return jsonify({"quote": result.data})

@app.route("/quotes/", methods=["POST"])
def new_quote():
    json_data = request.get_json()
    if not json_data:
        return jsonify({'message': 'No input data provided'}), 400
    # Validate and deserialize input
    data, errors = quote_schema.load(json_data)
    if errors:
        return jsonify(errors), 422
    first, last = data['author']['first'], data['author']['last']
    author = Author.query.filter_by(first=first, last=last).first()
    if author is None:
        # Create a new author
        author = Author(first=first, last=last)
        db.session.add(author)
    # Create new quote
    quote = Quote(
        content=data['content'],
        author=author,
        posted_at=datetime.datetime.utcnow()
```

(continues on next page)

(continued from previous page)

```
    )
    db.session.add(quote)
    db.session.commit()
    result = quote_schema.dump(Quote.query.get(quote.id))
    return jsonify({"message": "Created new quote.",
                    "quote": result.data})


if __name__ == '__main__':
    db.create_all()
    app.run(debug=True, port=5000)
```

Using The API

Run the app.

```
$ python flask_example.py
```

First we'll POST some quotes.

```
$ http POST :5000/quotes/ author="Tim Peters" content="Beautiful is better than ugly."
$ http POST :5000/quotes/ author="Tim Peters" content="Now is better than never."
$ http POST :5000/quotes/ author="Peter Hintjens" content="Simplicity is always␣
↪better than functionality."
```

If we provide invalid input data, we get 400 error response. Let's omit "author" from the input data.

```
$ http POST :5000/quotes/ content="I have no author"
{
    "author": [
        "Data not provided."
    ]
}
```

Now we can GET a list of all the quotes.

```
$ http :5000/quotes/
{
    "quotes": [
        {
            "content": "Beautiful is better than ugly.",
            "id": 1
        },
        {
            "content": "Now is better than never.",
            "id": 2
        },
        {
            "content": "Simplicity is always better than functionality.",
            "id": 3
        }
    ]
}
```

We can also GET the quotes for a single author.

```
$ http :5000/authors/1
{
    "author": {
        "first": "Tim",
        "formatted_name": "Peters, Tim",
        "id": 1,
        "last": "Peters"
    },
    "quotes": [
        {
            "content": "Beautiful is better than ugly.",
            "id": 1
        },
        {
            "content": "Now is better than never.",
            "id": 2
        }
    ]
}
```

4.6.3 ToDo API (Flask + Peewee)

This example uses Flask and the Peewee ORM to create a basic Todo application.

Here, we use *Schema.load* to validate and deserialize input data to model data. Also notice how *pre_load* is used to clean input data and *post_load* is used to add an envelope to response data.

```
import datetime as dt
from functools import wraps

from flask import Flask, request, g, jsonify
import peewee as pw
from marshmallow import Schema, fields, validate, pre_load, post_dump, post_load

app = Flask(__name__)
db = pw.SqliteDatabase('/tmp/todo.db')

###### MODELS #####

class BaseModel(pw.Model):
    """Base model class. All descendants share the same database."""
    class Meta:
        database = db

class User(BaseModel):
    email = pw.CharField(max_length=80, unique=True)
    password = pw.CharField()
    joined_on = pw.DateTimeField()

class Todo(BaseModel):
    content = pw.TextField()
    is_done = pw.BooleanField(default=False)
    user = pw.ForeignKeyField(User)
    posted_on = pw.DateTimeField()

    class Meta:
```

(continues on next page)

(continued from previous page)

```
        order_by = ('-posted_on', )

def create_tables():
    db.connect()
    User.create_table(True)
    Todo.create_table(True)

##### SCHEMAS #####

class UserSchema(Schema):
    id = fields.Int(dump_only=True)
    email = fields.Str(required=True,
                       validate=validate.Email(error='Not a valid email address'))
    password = fields.Str(required=True,
                          validate=[validate.Length(min=6, max=36)],
                          load_only=True)
    joined_on = fields.DateTime(dump_only=True)

    # Clean up data
    @pre_load
    def process_input(self, data):
        data['email'] = data['email'].lower().strip()
        return data

    # We add a post_dump hook to add an envelope to responses
    @post_dump(pass_many=True)
    def wrap(self, data, many):
        key = 'users' if many else 'user'
        return {
            key: data
        }


class TodoSchema(Schema):
    id = fields.Int(dump_only=True)
    done = fields.Boolean(attribute='is_done', missing=False)
    user = fields.Nested(UserSchema, exclude=('joined_on', 'password'), dump_
↪only=True)
    content = fields.Str(required=True)
    posted_on = fields.DateTime(dump_only=True)

    # Again, add an envelope to responses
    @post_dump(pass_many=True)
    def wrap(self, data, many):
        key = 'todos' if many else 'todo'
        return {
            key: data
        }

    # We use make_object to create a new Todo from validated data
    @post_load
    def make_object(self, data):
        if not data:
            return None
        return Todo(content=data['content'],
                    is_done=data['is_done'],
                    posted_on=dt.datetime.utcnow())
```

(continues on next page)

(continued from previous page)

```
user_schema = UserSchema()
todo_schema = TodoSchema()
todos_schema = TodoSchema(many=True)

###### HELPERS ######

def check_auth(email, password):
    """Check if a username/password combination is valid.
    """
    try:
        user = User.get(User.email == email)
    except User.DoesNotExist:
        return False
    return password == user.password

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            resp = jsonify({"message": "Please authenticate."})
            resp.status_code = 401
            resp.headers['WWW-Authenticate'] = 'Basic realm="Example"'
            return resp
        kwargs['user'] = User.get(User.email == auth.username)
        return f(*args, **kwargs)
    return decorated

# Ensure a separate connection for each thread
@app.before_request
def before_request():
    g.db = db
    g.db.connect()

@app.after_request
def after_request(response):
    g.db.close()
    return response

#### API #####

@app.route("/register", methods=["POST"])
def register():
    json_input = request.get_json()
    data, errors = user_schema.load(json_input)
    if errors:
        return jsonify({'errors': errors}), 422
    try:  # Use get to see if user already to exists
        User.get(User.email == data['email'])
    except User.DoesNotExist:
        user = User.create(email=data['email'], joined_on=dt.datetime.now(),
                           password=data['password'])
        message = "Successfully created user: {0}".format(user.email)
    else:
        return jsonify({'errors': 'That email address is already in the database'}),␣
↪400
```

(continues on next page)

(continued from previous page)

```
    data, _ = user_schema.dump(user)
    data['message'] = message
    return jsonify(data), 201

@app.route("/todos/", methods=['GET'])
def get_todos():
    todos = Todo.select().order_by(Todo.posted_on.asc())  # Get all todos
    result = todos_schema.dump(list(todos))
    return jsonify(result.data)

@app.route("/todos/<int:pk>")
def get_todo(pk):
    todo = Todo.get(Todo.id == pk)
    if not todo:
        return jsonify({'errors': 'Todo could not be find'}), 404
    result = todo_schema.dump(todo)
    return jsonify(result.data)

@app.route("/todos/<int:pk>/toggle", methods=["POST", "PUT"])
def toggledone(pk):
    try:
        todo = Todo.get(Todo.id == pk)
    except Todo.DoesNotExist:
        return jsonify({"message": "Todo could not be found"}), 404
    status = not todo.is_done
    update_query = todo.update(is_done=status)
    update_query.execute()
    result = todo_schema.dump(todo)
    return jsonify(result.data)

@app.route('/todos/', methods=["POST"])
@requires_auth
def new_todo(user):
    json_input = request.get_json()
    todo, errors = todo_schema.load(json_input)
    if errors:
        return jsonify({'errors': errors}), 422
    todo.user = user
    todo.save()
    result = todo_schema.dump(todo)
    return jsonify(result.data)


if __name__ == '__main__':
    create_tables()
    app.run(port=5000, debug=True)
```

Using the API

After registering a user and creating some todo items in the database, here is an example response.

```
$ http GET :5000/todos/
{
    "todos": [
        {
            "content": "Install marshmallow",
```

(continues on next page)

(continued from previous page)


```
            "done": false,
            "id": 1,
            "posted_on": "2015-05-05T01:51:12.832232+00:00",
            "user": {
                "user": {
                    "email": "foo@bar.com",
                    "id": 1
                }
            }
        },
        {
            "content": "Learn Python",
            "done": false,
            "id": 2,
            "posted_on": "2015-05-05T01:51:20.728052+00:00",
            "user": {
                "user": {
                    "email": "foo@bar.com",
                    "id": 1
                }
            }
        },
        {
            "content": "Refactor everything",
            "done": false,
            "id": 3,
            "posted_on": "2015-05-05T01:51:25.970153+00:00",
            "user": {
                "user": {
                    "email": "foo@bar.com",
                    "id": 1
                }
            }
        }
    ]
}
```

CHAPTER

FIVE

API REFERENCE

5.1 API Reference

5.1.1 Schema

class marshmallow.**Schema**(*extra=None*, *only=None*, *exclude=()*, *prefix=''*, *strict=None*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*)

Base schema class with which to define custom schemas.

Example usage:

```
import datetime as dt
from marshmallow import Schema, fields

class Album(object):
    def __init__(self, title, release_date):
        self.title = title
        self.release_date = release_date

class AlbumSchema(Schema):
    title = fields.Str()
    release_date = fields.Date()

# Or, equivalently
class AlbumSchema2(Schema):
    class Meta:
        fields = ("title", "release_date")

album = Album("Beggars Banquet", dt.date(1968, 12, 6))
schema = AlbumSchema()
data, errors = schema.dump(album)
data  # {'release_date': '1968-12-06', 'title': 'Beggars Banquet'}
```

Parameters

- **extra** (*dict*) – A dict of extra attributes to bind to the serialized result.
- **only** (*tuple|list*) – Whitelist of fields to select when instantiating the Schema. If None, all fields are used. Nested fields can be represented with dot delimiters.
- **exclude** (*tuple|list*) – Blacklist of fields to exclude when instantiating the Schema. If a field appears in both `only` and `exclude`, it is not used. Nested fields can be represented with dot delimiters.

- **prefix** (*str*) – Optional prefix that will be prepended to all the serialized field names.
- **strict** (*bool*) – If `True`, raise errors if invalid data are passed in instead of failing silently and storing the errors.
- **many** (*bool*) – Should be set to `True` if `obj` is a collection so that the object will be serialized to a list.
- **context** (*dict*) – Optional context passed to *fields.Method* and *fields.Function* fields.
- **load_only** (*tuple|list*) – Fields to skip during serialization (write-only fields)
- **dump_only** (*tuple|list*) – Fields to skip during deserialization (read-only fields)
- **partial** (*bool|tuple*) – Whether to ignore missing fields. If its value is an iterable, only missing fields listed in that iterable will be ignored.

Changed in version 2.0.0: `__validators__`, `__preprocessors__`, and `__data_handlers__` are removed in favor of *marshmallow.decorators.validates_schema*, *marshmallow.decorators.pre_load* and *marshmallow.decorators.post_dump*. `__accessor__` and `__error_handler__` are deprecated. Implement the *handle_error* and *get_attribute* methods instead.

class Meta

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **additional: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **include: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an `OrderedDict`.
- **exclude: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- **dateformat: Date format for all DateTime fields that do not have their** date format explicitly specified.
- **strict: If `True`, raise errors during marshalling rather than** storing them.
- **json_module: JSON module to use for `loads` and `dumps`.** Defaults to the `json` module in the stdlib.
- **ordered: If `True`, order serialization output according to the** order in which fields were declared. Output of *Schema.dump* will be a `collections.OrderedDict`.
- **index_errors: If `True`, errors dictionaries will include the index** of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization

OPTIONS_CLASS
alias of *SchemaOpts*

classmethod accessor(*func*)
Decorator that registers a function for pulling values from an object to serialize. The function receives the *Schema* instance, the `key` of the value to get, the `obj` to serialize, and an optional `default` value.

Deprecated since version 2.0.0: Set the `error_handler` class Meta option instead.

dump(*obj*, *many=None*, *update_fields=True*, ***kwargs*)
Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** – The object to serialize.
- **many** (*bool*) – Whether to serialize `obj` as a collection. If `None`, the value for `self.many` is used.
- **update_fields** (*bool*) – Whether to update the schema's field classes. Typically set to `True`, but may be `False` when serializing a homogenous collection. This parameter is used by *fields.Nested* to avoid multiple updates.

Returns A tuple of the form (`data`, `errors`)

Return type *MarshalResult*, a `collections.namedtuple`

New in version 1.0.0.

dumps(*obj*, *many=None*, *update_fields=True*, **args*, ***kwargs*)
Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** – The object to serialize.
- **many** (*bool*) – Whether to serialize `obj` as a collection. If `None`, the value for `self.many` is used.
- **update_fields** (*bool*) – Whether to update the schema's field classes. Typically set to `True`, but may be `False` when serializing a homogenous collection. This parameter is used by *fields.Nested* to avoid multiple updates.

Returns A tuple of the form (`data`, `errors`)

Return type *MarshalResult*, a `collections.namedtuple`

New in version 1.0.0.

classmethod error_handler(*func*)
Decorator that registers an error handler function for the schema. The function receives the *Schema* instance, a dictionary of errors, and the serialized object (if serializing data) or data dictionary (if deserializing data) as arguments.

Example:

```
class UserSchema(Schema):
    email = fields.Email()

@UserSchema.error_handler
def handle_errors(schema, errors, obj):
    raise ValueError('An error occurred while marshalling {}'.format(obj))

user = User(email='invalid')
```

(continues on next page)

(continued from previous page)

```
UserSchema().dump(user)  # => raises ValueError
UserSchema().load({'email': 'bademail'})  # raises ValueError
```

New in version 0.7.0.

Deprecated since version 2.0.0: Set the `error_handler` class Meta option instead.

get_attribute(*attr*, *obj*, *default*)
Defines how to pull values from an object to serialize.

New in version 2.0.0.

handle_error(*error*, *data*)
Custom error handler function for the schema.

Parameters

- **error** (`ValidationError`) – The `ValidationError` raised during (de)serialization.
- **data** – The original input data.

New in version 2.0.0.

load(*data*, *many=None*, *partial=None*)
Deserialize a data structure to an object defined by this Schema's fields and `make_object()`.

Parameters

- **data** (*dict*) – The data to deserialize.
- **many** (*bool*) – Whether to deserialize `data` as a collection. If `None`, the value for `self.many` is used.
- **partial** (*bool|tuple*) – Whether to ignore missing fields. If `None`, the value for `self.partial` is used. If its value is an iterable, only missing fields listed in that iterable will be ignored.

Returns A tuple of the form (`data`, `errors`)

Return type *UnmarshalResult*, a `collections.namedtuple`

New in version 1.0.0.

loads(*json_data*, *many=None*, **args*, ***kwargs*)
Same as *load()*, except it takes a JSON string as input.

Parameters

- **json_data** (*str*) – A JSON string of the data to deserialize.
- **many** (*bool*) – Whether to deserialize `obj` as a collection. If `None`, the value for `self.many` is used.
- **partial** (*bool|tuple*) – Whether to ignore missing fields. If `None`, the value for `self.partial` is used. If its value is an iterable, only missing fields listed in that iterable will be ignored.

Returns A tuple of the form (`data`, `errors`)

Return type *UnmarshalResult*, a `collections.namedtuple`

New in version 1.0.0.

on_bind_field(*field_name*, *field_obj*)
Hook to modify a field when it is bound to the *Schema*. No-op by default.

validate(*data*, *many=None*, *partial=None*)
Validate `data` against the schema, returning a dictionary of validation errors.

Parameters

- **data** (*dict*) – The data to validate.
- **many** (*bool*) – Whether to validate `data` as a collection. If `None`, the value for `self.many` is used.
- **partial** (*bool|tuple*) – Whether to ignore missing fields. If `None`, the value for `self.partial` is used. If its value is an iterable, only missing fields listed in that iterable will be ignored.

Returns A dictionary of validation errors.

Return type dict

New in version 1.1.0.

class marshmallow.**SchemaOpts**(*meta*)
class Meta options for the *Schema*. Defines defaults.

class marshmallow.**MarshalResult**(*data*, *errors*)
Return type of *Schema.dump()* including serialized data and errors

class marshmallow.**UnmarshalResult**(*data*, *errors*)
Return type of *Schema.load()*, including deserialized data and errors

marshmallow.**pprint**(*obj*, **args*, ***kwargs*)
Pretty-printing function that can pretty-print OrderedDicts like regular dictionaries. Useful for printing the output of *marshmallow.Schema.dump()*.

5.1.2 Fields

Field classes for various types of data.

class marshmallow.fields.**Field**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)
Basic field from which other fields should extend. It applies no formatting by default, and should only be used in cases where data does not need to be formatted before being serialized or deserialized. On error, the name of the field will be returned.

Parameters

- **default** – If set, this value will be used during serialization if the input value is missing. If not set, the field will be excluded from the serialized output if the input value is missing. May be a value or a callable.
- **attribute** (*str*) – The name of the attribute to get the value from. If `None`, assumes the attribute has the same name as the field.
- **load_from** (*str*) – Additional key to look for when deserializing. Will only be checked if the field's name is not found on the input dictionary. If checked, it will return this parameter on error.

- **dump_to** (*str*) – Field name to use as a key when serializing.
- **validate** (*callable*) – Validator or collection of validators that are called during deserialization. Validator takes a field's input value as its only parameter and returns a boolean. If it returns `False`, an `ValidationError` is raised.
- **required** – Raise a `ValidationError` if the field value is not supplied during deserialization.
- **allow_none** – Set this to `True` if `None` should be considered a valid value during validation/deserialization. If `missing=None` and `allow_none` is unset, will default to `True`. Otherwise, the default is `False`.
- **load_only** (*bool*) – If `True` skip this field during serialization, otherwise its value will be present in the serialized data.
- **dump_only** (*bool*) – If `True` skip this field during deserialization, otherwise its value will be present in the deserialized object. In the context of an HTTP API, this effectively marks the field as "read-only".
- **missing** – Default deserialization value for the field if the field is not found in the input data. May be a value or a callable.
- **error_messages** (*dict*) – Overrides for *Field.default_error_messages*.
- **metadata** – Extra arguments to be stored as metadata.

Changed in version 2.0.0: Removed `error` parameter. Use `error_messages` instead.

Changed in version 2.0.0: Added `allow_none` parameter, which makes validation/deserialization of `None` consistent across fields.

Changed in version 2.0.0: Added `load_only` and `dump_only` parameters, which allow field skipping during the (de)serialization process.

Changed in version 2.0.0: Added `missing` parameter, which indicates the value for a field if the field is not found during deserialization.

Changed in version 2.0.0: `default` value is only used if explicitly set. Otherwise, missing values inputs are excluded from serialized output.

_add_to_schema(*field_name*, *schema*)

Update field with values from its parent schema. Called by `__set_field_attrs`.

Parameters

- **field_name** (*str*) – Field name set in schema.
- **schema** (`Schema`) – Parent schema.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)
Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

_validate(*value*)
Perform validation on `value`. Raise a `ValidationError` if validation does not succeed.

_validate_missing(*value*)
Validate missing values. Raise a `ValidationError` if `value` should be considered missing.

property context
The context dictionary for the parent `Schema`.

default_error_messages = {'null': 'Field may not be null.', 'required': 'Missing dat
Default error messages for various kinds of errors. The keys in this dictionary are passed to *Field.fail*. The values are error messages passed to `marshmallow.ValidationError`.

deserialize(*value*, *attr=None*, *data=None*)
Deserialize `value`.

Raises ***ValidationError*** – If an invalid value is passed or if a required value is missing.

fail(*key*, ***kwargs*)
A helper method that simply raises a `ValidationError`.

get_value(*attr*, *obj*, *accessor=None*, *default=<marshmallow.missing>*)
Return the value for a given key from an object.

property root
Reference to the `Schema` that this field belongs to even if it is buried in a *List*. Return `None` for unbound fields.

serialize(*attr*, *obj*, *accessor=None*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

- **attr** (*str*) – The attibute or key to get from the object.
- **obj** (*str*) – The object to pull the key from.
- **accessor** (*callable*) – Function used to pull values from `obj`.

Raises *ValidationError* – In case of formatting problem

class marshmallow.fields.**Raw**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)

Field that applies no formatting or validation.

class marshmallow.fields.**Nested**(*nested*, *default=<marshmallow.missing>*, *exclude=()*, *only=None*, ***kwargs*)

Allows you to nest a *Schema* inside a field.

Examples:

```
user = fields.Nested(UserSchema)
user2 = fields.Nested('UserSchema')  # Equivalent to above
collaborators = fields.Nested(UserSchema, many=True, only='id')
parent = fields.Nested('self')
```

When passing a *Schema* instance as the first argument, the instance's `exclude`, `only`, and `many` attributes will be respected.

Therefore, when passing the `exclude`, `only`, or `many` arguments to `fields.Nested`, you should pass a *Schema* class (not an instance) as the first argument.

```
# Yes
author = fields.Nested(UserSchema, only=('id', 'name'))

# No
author = fields.Nested(UserSchema(), only=('id', 'name'))
```

Parameters

- **nested** (`Schema`) – The Schema class or class name (string) to nest, or `"self"` to nest the `Schema` within itself.
- **exclude** (*tuple*) – A list or tuple of fields to exclude.
- **required** – Raise an `ValidationError` during deserialization if the field, *and* any required field values specified in the `nested` schema, are not found in the data. If not a `bool` (e.g. a `str`), the provided value will be used as the message of the `ValidationError` instead of the default message.
- **only** – A tuple or string of the field(s) to marshal. If `None`, all fields will be marshalled. If a field name (string) is given, only a single value will be returned as output instead of a dictionary. This parameter takes precedence over `exclude`.
- **many** (*bool*) – Whether the field is a collection of objects.
- **kwargs** – The same keyword arguments that *Field* receives.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises *ValidationError* – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*nested_obj*, *attr*, *obj*)
Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises *ValidationError* – In case of formatting or validation failure.

Returns The serialized value

_validate_missing(*value*)
Validate missing values. Raise a `ValidationError` if `value` should be considered missing.

property schema
The nested Schema object.

Changed in version 1.0.0: Renamed from `serializer` to *schema*

class marshmallow.fields.**Dict**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)
A dict field. Supports dicts and dict-like objects.

Note: This field is only appropriate when the structure of nested data is not known. For structured data, use *Nested*.

New in version 2.1.0.

_deserialize(*value*, *attr*, *data*)
Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises *ValidationError* – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

class marshmallow.fields.**List**(*cls_or_instance*, ***kwargs*)

A list field, composed with another *Field* class or instance.

Example:

```
numbers = fields.List(fields.Float())
```

Parameters

- **cls_or_instance** (Field) – A field class or instance.
- **default** (*bool*) – Default value for serialization.
- **kwargs** – The same keyword arguments that *Field* receives.

Changed in version 2.0.0: The `allow_none` parameter now applies to deserialization and has the same semantics as the other fields.

_add_to_schema(*field_name*, *schema*)

Update field with values from its parent schema. Called by `__set_field_attrs`.

Parameters

- **field_name** (*str*) – Field name set in schema.
- **schema** (Schema) – Parent schema.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.

- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

get_value(*attr*, *obj*, *accessor=None*)
Return the value for a given key from an object.

class marshmallow.fields.**String**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)
A string field.

Parameters **kwargs** – The same keyword arguments that *Field* receives.

_deserialize(*value*, *attr*, *data*)
Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)
Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**UUID**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)

A UUID field.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

_validated(*value*)

Format the value or raise a `ValidationError` if an error occurs.

class marshmallow.fields.**Number**(*as_string=False*, ***kwargs*)

Base class for number fields.

Parameters

- **as_string** (*bool*) – If True, format the serialized value as a string.
- **kwargs** – The same keyword arguments that *Field* receives.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_format_num(*value*)
Return the number value for value, given this field's *num_type*.

_serialize(*value*, *attr*, *obj*)
Return a string if `self.as_string=True`, otherwise return this field's *num_type*.

_validated(*value*)
Format the value or raise a `ValidationError` if an error occurs.

num_type
alias of `builtins.float`

class marshmallow.fields.**Integer**(*as_string=False*, ***kwargs*)
An integer field.

Parameters **kwargs** – The same keyword arguments that *Number* receives.

num_type
alias of `builtins.int`

class marshmallow.fields.**Decimal**(*places=None*, *rounding=None*, *allow_nan=False*, *as_string=False*, ***kwargs*)
A field that (de)serializes to the Python `decimal.Decimal` type. It's safe to use when dealing with money values, percentages, ratios or other numbers where precision is critical.

Warning: This field serializes to a `decimal.Decimal` object by default. If you need to render your data as JSON, keep in mind that the `json` module from the standard library does not encode `decimal.Decimal`. Therefore, you must use a JSON library that can handle decimals, such as `simplejson`, or serialize to a string by passing `as_string=True`.

Warning: If a JSON `float` value is passed to this field for deserialization it will first be cast to its corresponding `string` value before being deserialized to a `decimal.Decimal` object. The default `__str__` implementation of the built-in Python `float` type may apply a destructive transformation upon its input data and therefore cannot be relied upon to preserve precision. To avoid this, you can instead pass a JSON `string` to be deserialized directly.

Parameters

- **places** (*int*) – How many decimal places to quantize the value. If `None`, does not quantize the value.
- **rounding** – How to round the value during quantize, for example `decimal.ROUND_UP`. If None, uses the rounding value from the current thread's context.
- **allow_nan** (*bool*) – If `True`, `NaN`, `Infinity` and `-Infinity` are allowed, even though they are illegal according to the JSON specification.

- **as_string** (*bool*) – If True, serialize to a string instead of a Python `decimal.Decimal` type.
- **kwargs** – The same keyword arguments that *Number* receives.

New in version 1.2.0.

_format_num(*value*)
Return the number value for value, given this field's *num_type*.

_validated(*value*)
Format the value or raise a `ValidationError` if an error occurs.

num_type
alias of `decimal.Decimal`

class marshmallow.fields.**Boolean**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)

A boolean field.

Parameters **kwargs** – The same keyword arguments that *Field* receives.

_deserialize(*value*, *attr*, *data*)
Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)
Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

falsy = {0, 'False', 'false', 'F', '0', 'FALSE', 'f'}
Values that will (de)serialize to `False`.

class marshmallow.fields.**FormattedString**(*src_str*, **args*, ***kwargs*)
Interpolate other values from the object into this field. The syntax for the source string is the same as the string `str.format` method from the python stdlib.

```
class UserSchema(Schema):
    name = fields.String()
    greeting = fields.FormattedString('Hello {name}')

ser = UserSchema()
res = ser.dump(user)
res.data  # => {'name': 'Monty', 'greeting': 'Hello Monty'}
```

_serialize(*value*, *attr*, *obj*)
Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**Float**(*as_string=False*, ***kwargs*)
A double as IEEE-754 double precision string.

Parameters

- **as_string** (*bool*) – If True, format the value as a string.
- **kwargs** – The same keyword arguments that *Number* receives.

num_type
alias of `builtins.float`

class marshmallow.fields.**DateTime**(*format=None*, ***kwargs*)
A formatted datetime string in UTC.

Example: `'2014-12-22T03:12:58.019077+00:00'`

Timezone-naive `datetime` objects are converted to UTC (+00:00) by *Schema.dump*. *Schema.load* returns `datetime` objects that are timezone-aware.

Parameters

- **format** (*str*) – Either `"rfc"` (for RFC822), `"iso"` (for ISO8601), or a date format string. If `None`, defaults to "iso".
- **kwargs** – The same keyword arguments that *Field* receives.

_add_to_schema(*field_name*, *schema*)

Update field with values from its parent schema. Called by `__set_field_attrs`.

Parameters

- **field_name** (*str*) – Field name set in schema.
- **schema** (`Schema`) – Parent schema.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**LocalDateTime**(*format=None*, ***kwargs*)

A formatted datetime string in localized time, relative to UTC.

ex. `"Sun, 10 Nov 2013 08:23:45 -0600"`

Takes the same arguments as *DateTime*.

class marshmallow.fields.**Time**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)

ISO8601-formatted time string.

Parameters **kwargs** – The same keyword arguments that *Field* receives.

_deserialize(*value*, *attr*, *data*)

Deserialize an ISO8601-formatted time to a `datetime.time` object.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**Date**(*default=<marshmallow.missing>*, *attribute=None*, *load_from=None*, *dump_to=None*, *error=None*, *validate=None*, *required=False*, *allow_none=None*, *load_only=False*, *dump_only=False*, *missing=<marshmallow.missing>*, *error_messages=None*, ***metadata*)

ISO8601-formatted date string.

Parameters **kwargs** – The same keyword arguments that *Field* receives.

_deserialize(*value*, *attr*, *data*)

Deserialize an ISO8601-formatted date string to a `datetime.date` object.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises *ValidationError* – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**TimeDelta**(*precision='seconds'*, *error=None*, ***kwargs*)

A field that (de)serializes a `datetime.timedelta` object to an integer and vice versa. The integer can represent the number of days, seconds or microseconds.

Parameters

- **precision** (*str*) – Influences how the integer is interpreted during (de)serialization. Must be 'days', 'seconds' or 'microseconds'.
- **error** (*str*) – Error message stored upon validation failure.
- **kwargs** – The same keyword arguments that *Field* receives.

Changed in version 2.0.0: Always serializes to an integer value to avoid rounding errors. Add `precision` parameter.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises *ValidationError* – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises *ValidationError* – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**Url**(*relative=False*, *schemes=None*, ***kwargs*)
A validated URL field. Validation occurs during both serialization and deserialization.

Parameters

- **default** – Default value for the field if the attribute is not set.
- **attribute** (*str*) – The name of the attribute to get the value from. If None, assumes the attribute has the same name as the field.
- **relative** (*bool*) – Allow relative URLs.
- **kwargs** – The same keyword arguments that *String* receives.

marshmallow.fields.**URL**
alias of *marshmallow.fields.Url*

class marshmallow.fields.**Email**(**args*, ***kwargs*)
A validated email field. Validation occurs during both serialization and deserialization.

Parameters

- **args** – The same positional arguments that *String* receives.
- **kwargs** – The same keyword arguments that *String* receives.

class marshmallow.fields.**Method**(*serialize=None*, *deserialize=None*, *method_name=None*, ***kwargs*)
A field that takes the value returned by a Schema method.

Parameters

- **method_name** (*str*) – The name of the Schema method from which to retrieve the value. The method must take an argument obj (in addition to self) that is the object to be serialized.
- **deserialize** (*str*) – Optional name of the Schema method for deserializing a value The method must take a single argument value, which is the value to deserialize.

Changed in version 2.0.0: Removed optional context parameter on methods. Use self.context instead.

Changed in version 2.3.0: Deprecated method_name parameter in favor of serialize and allow serialize to not be passed at all.

_deserialize(*value*, *attr*, *data*)
Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in data to be deserialized.
- **data** (*dict*) – The raw input data passed to the Schema.load.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added attr and data parameters.

_serialize(*value*, *attr*, *obj*)
Serializes value to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

class marshmallow.fields.**Function**(*serialize=None*, *deserialize=None*, *func=None*, ***kwargs*)

A field that takes the value returned by a function.

Parameters

- **serialize** (*callable*) – A callable from which to retrieve the value. The function must take a single argument `obj` which is the object to be serialized. It can also optionally take a `context` argument, which is a dictionary of context variables passed to the serializer. If no callable is provided then the `load_only` flag will be set to True.
- **deserialize** (*callable*) – A callable from which to retrieve the value. The function must take a single argument `value` which is the value to be deserialized. It can also optionally take a `context` argument, which is a dictionary of context variables passed to the deserializer. If no callable is provided then `value` will be passed through unchanged.
- **func** (*callable*) – This argument is to be deprecated. It exists for backwards compatiblity. Use serialize instead.

Changed in version 2.3.0: Deprecated `func` parameter in favor of `serialize`.

_deserialize(*value*, *attr*, *data*)

Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, *attr*, *obj*)

Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

marshmallow.fields.**Str**
alias of *marshmallow.fields.String*

marshmallow.fields.**Bool**
alias of *marshmallow.fields.Boolean*

marshmallow.fields.**Int**
alias of *marshmallow.fields.Integer*

class marshmallow.fields.**Constant**(*constant*, ***kwargs*)
A field that (de)serializes to a preset constant. If you only want the constant added for serialization or deserialization, you should use `dump_only=True` or `load_only=True` respectively.

Parameters **constant** – The constant to return for the field attribute.

New in version 2.0.0.

_deserialize(*value*, **args*, ***kwargs*)
Deserialize value. Concrete *Field* classes should implement this method.

Parameters

- **value** – The value to be deserialized.
- **attr** (*str*) – The attribute/key in `data` to be deserialized.
- **data** (*dict*) – The raw input data passed to the `Schema.load`.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The deserialized value.

Changed in version 2.0.0: Added `attr` and `data` parameters.

_serialize(*value*, **args*, ***kwargs*)
Serializes `value` to a basic Python datatype. Noop by default. Concrete *Field* classes should implement this method.

Example:

```
class TitleCase(Field):
    def _serialize(self, value, attr, obj):
        if not value:
            return ''
        return unicode(value).title()
```

Parameters

- **value** – The value to be serialized.
- **attr** (*str*) – The attribute or key on the object to be serialized.
- **obj** (*object*) – The object the value was pulled from.

Raises ***ValidationError*** – In case of formatting or validation failure.

Returns The serialized value

5.1.3 Decorators

Decorators for registering schema pre-processing and post-processing methods. These should be imported from the top-level *marshmallow* module.

Example:

```
from marshmallow import (
    Schema, pre_load, pre_dump, post_load, validates_schema,
    validates, fields, ValidationError
)

class UserSchema(Schema):

    email = fields.Str(required=True)
    age = fields.Integer(required=True)

    @post_load
    def lowerstrip_email(self, item):
        item['email'] = item['email'].lower().strip()
        return item

    @pre_load(pass_many=True)
    def remove_envelope(self, data, many):
        namespace = 'results' if many else 'result'
        return data[namespace]

    @post_dump(pass_many=True)
    def add_envelope(self, data, many):
        namespace = 'results' if many else 'result'
        return {namespace: data}

    @validates_schema
    def validate_email(self, data):
        if len(data['email']) < 3:
            raise ValidationError('Email must be more than 3 characters', 'email')

    @validates('age')
    def validate_age(self, data):
        if data < 14:
            raise ValidationError('Too young!')
```

Note: These decorators only work with instance methods. Class and static methods are not supported.

Warning: The invocation order of decorated methods of the same type is not guaranteed. If you need to guarantee order of different processing steps, you should put them in the same processing method.

marshmallow.decorators.**post_dump**(*fn=None*, *pass_many=False*, *pass_original=False*)
Register a method to invoke after serializing an object. The method receives the serialized object and returns the processed object.

By default, receives a single object at a time, transparently handling the `many` argument passed to the Schema. If `pass_many=True`, the raw data (which may be a collection) and the value for `many` is passed.

marshmallow.decorators.**post_load**(*fn=None*, *pass_many=False*, *pass_original=False*)
Register a method to invoke after deserializing an object. The method receives the deserialized data and returns the processed data.

By default, receives a single datum at a time, transparently handling the `many` argument passed to the Schema. If `pass_many=True`, the raw data (which may be a collection) and the value for `many` is passed.

marshmallow.decorators.**pre_dump**(*fn=None*, *pass_many=False*)
Register a method to invoke before serializing an object. The method receives the object to be serialized and returns the processed object.

By default, receives a single object at a time, regardless of whether `many=True` is passed to the `Schema`. If `pass_many=True`, the raw data (which may be a collection) and the value for `many` is passed.

marshmallow.decorators.**pre_load**(*fn=None*, *pass_many=False*)
Register a method to invoke before deserializing an object. The method receives the data to be deserialized and returns the processed data.

By default, receives a single datum at a time, transparently handling the `many` argument passed to the Schema. If `pass_many=True`, the raw data (which may be a collection) and the value for `many` is passed.

marshmallow.decorators.**tag_processor**(*tag_name*, *fn*, *pass_many*, ***kwargs*)
Tags decorated processor function to be picked up later.

Note: Currently ony works with functions and instance methods. Class and static methods are not supported.

Returns Decorated function if supplied, else this decorator with its args bound.

marshmallow.decorators.**validates**(*field_name*)
Register a field validator.

Parameters **field_name** (*str*) – Name of the field that the method validates.

marshmallow.decorators.**validates_schema**(*fn=None*, *pass_many=False*, *pass_original=False*, *skip_on_field_errors=False*)
Register a schema-level validator.

By default, receives a single object at a time, regardless of whether `many=True` is passed to the `Schema`. If `pass_many=True`, the raw data (which may be a collection) and the value for `many` is passed.

If `pass_original=True`, the original data (before unmarshalling) will be passed as an additional argument to the method.

If `skip_on_field_errors=True`, this validation method will be skipped whenever validation errors have been detected when validating fields.

5.1.4 Validators

Validation classes for various types of data.

class marshmallow.validate.**ContainsOnly**(*choices*, *labels=None*, *error=None*)
Validator which succeeds if `value` is a sequence and each element in the sequence is also in the sequence passed as `choices`.

Parameters

- **choices** (*iterable*) – Same as *OneOf*.
- **labels** (*iterable*) – Same as *OneOf*.
- **error** (*str*) – Same as *OneOf*.

class marshmallow.validate.**Email**(*error=None*)
Validate an email address.

Parameters **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}`.

class marshmallow.validate.**Equal**(*comparable*, *error=None*)
Validator which succeeds if the `value` passed to it is equal to `comparable`.

Parameters

- **comparable** – The object to compare to.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}` and `{other}`.

class marshmallow.validate.**Length**(*min=None*, *max=None*, *error=None*, *equal=None*)
Validator which succeeds if the value passed to it has a length between a minimum and maximum. Uses len(), so it can work for strings, lists, or anything with length.

Parameters

- **min** (*int*) – The minimum length. If not provided, minimum length will not be checked.
- **max** (*int*) – The maximum length. If not provided, maximum length will not be checked.
- **equal** (*int*) – The exact length. If provided, maximum and minimum length will not be checked.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}`, `{min}` and `{max}`.

class marshmallow.validate.**NoneOf**(*iterable*, *error=None*)
Validator which fails if `value` is a member of `iterable`.

Parameters

- **iterable** (*iterable*) – A sequence of invalid values.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated using `{input}` and `{values}`.

class marshmallow.validate.**OneOf**(*choices*, *labels=None*, *error=None*)
Validator which succeeds if `value` is a member of `choices`.

Parameters

- **choices** (*iterable*) – A sequence of valid values.
- **labels** (*iterable*) – Optional sequence of labels to pair with the choices.

- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}`, `{choices}` and `{labels}`.

options(*valuegetter=<class 'str'>*)
Return a generator over the (value, label) pairs, where value is a string associated with each choice. This convenience method is useful to populate, for instance, a form select field.

Parameters **valuegetter** – Can be a callable or a string. In the former case, it must be a one-argument callable which returns the value of a choice. In the latter case, the string specifies the name of an attribute of the choice objects. Defaults to `str()` or `unicode()`.

class marshmallow.validate.**Predicate**(*method*, *error=None*, ***kwargs*)
Call the specified `method` of the `value` object. The validator succeeds if the invoked method returns an object that evaluates to True in a Boolean context. Any additional keyword argument will be passed to the method.

Parameters

- **method** (*str*) – The name of the method to invoke.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}` and `{method}`.
- **kwargs** – Additional keyword arguments to pass to the method.

class marshmallow.validate.**Range**(*min=None*, *max=None*, *error=None*)
Validator which succeeds if the value it is passed is greater or equal to `min` and less than or equal to `max`. If `min` is not specified, or is specified as `None`, no lower bound exists. If `max` is not specified, or is specified as `None`, no upper bound exists.

Parameters

- **min** – The minimum value (lower bound). If not provided, minimum value will not be checked.
- **max** – The maximum value (upper bound). If not provided, maximum value will not be checked.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}`, `{min}` and `{max}`.

class marshmallow.validate.**Regexp**(*regex*, *flags=0*, *error=None*)
Validate `value` against the provided regex.

Parameters

- **regex** – The regular expression string to use. Can also be a compiled regular expression pattern.
- **flags** – The regexp flags to use, for example re.IGNORECASE. Ignored if `regex` is not a string.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}` and `{regex}`.

class marshmallow.validate.**URL**(*relative=False*, *error=None*, *schemes=None*, *require_tld=True*)
Validate a URL.

Parameters

- **relative** (*bool*) – Whether to allow relative URLs.
- **error** (*str*) – Error message to raise in case of a validation error. Can be interpolated with `{input}`.
- **schemes** (*set*) – Valid schemes. By default, `http`, `https`, `ftp`, and `ftps` are allowed.

- **require_tld** (*bool*) – Whether to reject non-FQDN hostnames

class marshmallow.validate.**Validator**
Base abstract class for validators.

Note: This class does not provide any behavior. It is only used to add a useful `__repr__` implementation for validators.

5.1.5 Utility Functions

Utility methods for marshmallow.

marshmallow.utils.**callable_or_raise**(*obj*)
Check that an object is callable, else raise a `ValueError`.

marshmallow.utils.**decimal_to_fixed**(*value*, *precision*)
Convert a `Decimal` to a fixed-precision number as a string.

marshmallow.utils.**float_to_decimal**(*f*)
Convert a floating point number to a Decimal with no loss of information. See: http://docs.python.org/release/2.6.7/library/decimal.html#decimal-faq

marshmallow.utils.**from_datestring**(*datestring*)
Parse an arbitrary datestring and return a datetime object using dateutils' parser.

marshmallow.utils.**from_iso**(*datestring*, *use_dateutil=True*)
Parse an ISO8601-formatted datetime string and return a datetime object.

Use dateutil's parser if possible and return a timezone-aware datetime.

marshmallow.utils.**from_iso_time**(*timestring*, *use_dateutil=True*)
Parse an ISO8601-formatted datetime string and return a datetime.time object.

marshmallow.utils.**from_rfc**(*datestring*, *use_dateutil=True*)
Parse a RFC822-formatted datetime string and return a datetime object.

Use dateutil's parser if possible.

https://stackoverflow.com/questions/885015/how-to-parse-a-rfc-2822-date-time-into-a-python-datetime

marshmallow.utils.**get_func_args**(*func*)
Given a callable, return a list of argument names. Handles `functools.partial` objects and callable objects.

marshmallow.utils.**get_value**(*key*, *obj*, *default=<marshmallow.missing>*)
Helper for pulling a keyed value off various types of objects

marshmallow.utils.**is_collection**(*obj*)
Return True if `obj` is a collection type, e.g list, tuple, queryset.

marshmallow.utils.**is_generator**(*obj*)
Return True if `obj` is a generator

marshmallow.utils.**is_indexable_but_not_string**(*obj*)
Return True if `obj` is indexable but isn't a string.

marshmallow.utils.**is_instance_or_subclass**(*val*, *class_*)
Return True if `val` is either a subclass or instance of `class_`.

marshmallow.utils.**is_iterable_but_not_string**(*obj*)
Return True if `obj` is an iterable object that isn't a string.

marshmallow.utils.**is_keyed_tuple**(*obj*)
Return True if `obj` has keyed tuple behavior, such as namedtuples or SQLAlchemy's KeyedTuples.

marshmallow.utils.**isoformat**(*dt*, *localtime=False*, **args*, ***kwargs*)
Return the ISO8601-formatted UTC representation of a datetime object.

marshmallow.utils.**local_rfcformat**(*dt*)
Return the RFC822-formatted representation of a timezone-aware datetime with the UTC offset.

marshmallow.utils.**pluck**(*dictlist*, *key*)
Extracts a list of dictionary values from a list of dictionaries.

```
>>> dlist = [{'id': 1, 'name': 'foo'}, {'id': 2, 'name': 'bar'}]
>>> pluck(dlist, 'id')
[1, 2]
```

marshmallow.utils.**pprint**(*obj*, **args*, ***kwargs*)
Pretty-printing function that can pretty-print OrderedDicts like regular dictionaries. Useful for printing the output of *marshmallow.Schema.dump()*.

marshmallow.utils.**rfcformat**(*dt*, *localtime=False*)
Return the RFC822-formatted representation of a datetime object.

Parameters

- **dt** (*datetime*) – The datetime.
- **localtime** (*bool*) – If `True`, return the date relative to the local timezone instead of UTC, displaying the proper offset, e.g. "Sun, 10 Nov 2013 08:23:45 -0600"

marshmallow.utils.**set_value**(*dct*, *key*, *value*)
Set a value in a dict. If `key` contains a '.', it is assumed be a path (i.e. dot-delimited string) to the value's location.

```
>>> d = {}
>>> set_value(d, 'foo.bar', 42)
>>> d
{'foo': {'bar': 42}}
```

marshmallow.utils.**to_marshallable_type**(*obj*, *field_names=None*)
Helper for converting an object to a dictionary only if it is not dictionary already or an indexable object nor a simple type

5.1.6 Marshalling

Utility classes and values used for marshalling and unmarshalling objects to and from primitive types.

Warning: This module is treated as private API. Users should not need to use this module directly.

class marshmallow.marshalling.**Marshaller**(*prefix=''*)
Callable class responsible for serializing data and storing errors.

Parameters **prefix** (*str*) – Optional prefix that will be prepended to all the serialized field names.

serialize(*obj*, *fields_dict*, *many=False*, *accessor=None*, *dict_class=<class 'dict'>*, *index_errors=True*, *index=None*)

Takes raw data (a dict, list, or other object) and a dict of fields to output and serializes the data based on those fields.

Parameters

- **obj** – The actual object(s) from which the fields are taken from
- **fields_dict** (*dict*) – Mapping of field names to `Field` objects.
- **many** (*bool*) – Set to `True` if `data` should be serialized as a collection.
- **accessor** (*callable*) – Function to use for getting values from `obj`.
- **dict_class** (*type*) – Dictionary class used to construct the output.
- **index_errors** (*bool*) – Whether to store the index of invalid items in `self.errors` when `many=True`.
- **index** (*int*) – Index of the item being serialized (for storing errors) if serializing a collection, otherwise `None`.

Returns A dictionary of the marshalled data

Changed in version 1.0.0: Renamed from `marshal`.

class marshmallow.marshalling.**Unmarshaller**

Callable class responsible for deserializing data and storing errors.

New in version 1.0.0.

deserialize(*data*, *fields_dict*, *many=False*, *partial=False*, *dict_class=<class 'dict'>*, *index_errors=True*, *index=None*)

Deserialize `data` based on the schema defined by `fields_dict`.

Parameters

- **data** (*dict*) – The data to deserialize.
- **fields_dict** (*dict*) – Mapping of field names to `Field` objects.
- **many** (*bool*) – Set to `True` if `data` should be deserialized as a collection.
- **partial** (*bool | tuple*) – Whether to ignore missing fields. If its value is an iterable, only missing fields listed in that iterable will be ignored.
- **dict_class** (*type*) – Dictionary class used to construct the output.
- **index_errors** (*bool*) – Whether to store the index of invalid items in `self.errors` when `many=True`.
- **index** (*int*) – Index of the item being serialized (for storing errors) if serializing a collection, otherwise `None`.

Returns A dictionary of the deserialized data.

5.1.7 Class Registry

A registry of *Schema* classes. This allows for string lookup of schemas, which may be used with class:*fields.Nested*.

Warning: This module is treated as private API. Users should not need to use this module directly.

marshmallow.class_registry.**get_class**(*classname*, *all=False*)
Retrieve a class from the registry.

Raises marshmallow.exceptions.RegistryError if the class cannot be found or if there are multiple entries for the given class name.

marshmallow.class_registry.**register**(*classname*, *cls*)
Add a class to the registry of serializer classes. When a class is registered, an entry for both its classname and its full, module-qualified path are added to the registry.

Example:

```
class MyClass:
    pass

register('MyClass', MyClass)
# Registry:
# {
#   'MyClass': [path.to.MyClass],
#   'path.to.MyClass': [path.to.MyClass],
# }
```

5.1.8 Exceptions

Exception classes for marshmallow-related errors.

exception marshmallow.exceptions.**MarshmallowError**
Base class for all marshmallow-related errors.

exception marshmallow.exceptions.**RegistryError**
Raised when an invalid operation is performed on the serializer class registry.

exception marshmallow.exceptions.**ValidationError**(*message*, *field_names=None*, *fields=None*, *data=None*, ***kwargs*)
Raised when validation fails on a field. Validators and custom fields should raise this exception.

Parameters

- **message** – An error message, list of error messages, or dict of error messages.
- **field_names** (*list*) – Field names to store the error on. If None, the error is stored in its default location.
- **fields** (*list*) – Field objects to which the error applies.

field_names = None
List of field_names which failed validation.

fields = None
List of field objects which failed validation.

messages = None
String, list, or dictionary of error messages. If a dict, the keys will be field names and the values will be lists of messages.

CHAPTER

SIX

PROJECT INFO

6.1 Why marshmallow?

The Python ecosystem has many great libraries for data formatting and schema validation.

In fact, marshmallow was influenced by a number of these libraries. Marshmallow is inspired by Django REST Framework, Flask-RESTful, and colander. It borrows a number of implementation and design ideas from these libraries to create a flexible and productive solution for marshalling, unmarshalling, and validating data.

Here are just a few reasons why you might use marshmallow.

6.1.1 Agnostic.

Marshmallow makes no assumption about web frameworks or database layers. It will work with just about any ORM, ODM, or no ORM at all. This gives you the freedom to choose the components that fit your application's needs without having to change your data formatting code. If you wish, you can build integration layers to make marshmallow work more closely with your frameworks and libraries of choice (for examples, see Flask-Marshmallow and Django REST Marshmallow).

6.1.2 Concise, familiar syntax.

If you have used Django REST Framework or WTForms, marshmallow's *`Schema`* syntax will feel familiar to you. Class-level field attributes define the schema for formatting your data. Configuration is added using the `class Meta` paradigm. Configuration options can be overriden at application runtime by passing arguments to the *`Schema`* constructor. The *`dump`* and *`load`* methods are used for serialization and deserialization (of course!).

6.1.3 Class-based schemas allow for code reuse and configuration.

Unlike Flask-RESTful, which uses dictionaries to define output schemas, marshmallow uses classes. This allows for easy code reuse and configuration. It also allows for powerful means for configuring and extending schemas, such as adding *post-processing and error handling behavior*.

6.1.4 Consistency meets flexibility.

Marshmallow makes it easy to modify a schema's output at application runtime. A single *`Schema`* can produce multiple outputs formats while keeping the individual field outputs consistent.

As an example, you might have a JSON endpoint for retrieving all information about a video game's state. You then add a low-latency endpoint that only returns a minimal subset of information about game state. Both endpoints can be handled by the same *Schema*.

```
class GameStateSchema(Schema):
    _id = fields.UUID(required=True)
    players = fields.Nested(PlayerSchema, many=True)
    score = fields.Nested(ScoreSchema)
    last_changed = fields.DateTime(format='rfc')

    class Meta:
        additional = ('title', 'date_created', 'type', 'is_active')

# Serializes full game state
full_serializer = GameStateSchema()
# Serializes a subset of information, for a low-latency endpoint
summary_serializer = GameStateSchema(only=('_id', 'last_changed'))
# Also filter the fields when serializing multiple games
gamelist_serializer = GameStateSchema(many=True,
                                      only=('_id', 'players', 'last_changed'))
```

In this example, a single schema produced three different outputs! The dynamic nature of a `Schema` leads to **less code** and **more consistent formatting**.

6.1.5 Context-aware serialization.

Marshmallow schemas can modify their output based on the context in which they are used. Field objects have access to a `context` dictionary that can be changed at runtime.

Here's a simple example that shows how a *Schema* can anonymize a person's name when a boolean is set on the context.

```
class PersonSchema(Schema):
    id = fields.Integer()
    name = fields.Method('get_name')

    def get_name(self, person, context):
        if context.get('anonymize'):
            return '<anonymized>'
        return person.name

person = Person(name='Monty')
schema = PersonSchema()
schema.dump(person)  # {'id': 143, 'name': 'Monty'}

# In a different context, anonymize the name
schema.context['anonymize'] = True
schema.dump(person)  # {'id': 143, 'name': '<anonymized>'}
```

See also:

See the relevant section of the *usage guide* to learn more about context-aware serialization.

6.1.6 Advanced schema nesting.

Most serialization libraries provide some means for nesting schemas within each other, but they often fail to meet common use cases in clean way. Marshmallow aims to fill these gaps by adding a few nice features for *nesting schemas*:

- You can specify which *subset of fields* to include on nested schemas.
- *Two-way nesting*. Two different schemas can nest each other.
- *Self-nesting*. A schema can be nested within itself.

6.2 Changelog

6.2.1 2.21.0 (2020-03-05)

Bug fixes:

- Don't match string-ending newlines in `URL` and `Email` fields (#1522). Thanks @nbanmp for the PR.

Other changes:

- Drop support for Python 3.4 (#1525).

6.2.2 2.20.5 (2019-09-15)

Bug fixes:

- Fix behavior when a non-list collection is passed to the `validate` argument of `fields.Email` and `fields.URL` (#1400).

6.2.3 2.20.4 (2019-09-11)

Bug fixes:

- Respect the `many` value on `Schema` instances passed to `Nested` (#1160). Thanks @Kamforka for reporting.

6.2.4 2.20.3 (2019-09-04)

Bug fixes:

- Don't swallow `TypeError` exceptions raised by `Field._bind_to_schema` or `Schema.on_bind_field`.

6.2.5 2.20.2 (2019-08-20)

Bug fixes:

- Prevent warning about importing from `collections` on Python 3.7 (#1354). Thanks @nicktimko for the PR.

6.2.6 2.20.1 (2019-08-13)

Bug fixes:

- Fix bug that raised `TypeError` when invalid data type is passed to a nested schema with `@validates` (#1342).

6.2.7 2.20.0 (2019-08-10)

Bug fixes:

- Fix deprecated functions' compatibility with Python 2 (#1337). Thanks @airstandley for the catch and patch.
- Fix error message consistency for invalid input types on nested fields (#1303). This is a backport of the fix in #857. Thanks @cristi23 for the thorough bug report and the PR.

Deprecation/Removal:

- Python 2.6 is no longer officially supported (#1274).

6.2.8 2.19.5 (2019-06-18)

Bug fixes:

- Fix deserializing ISO8601-formatted datetimes with less than 6-digit miroseconds (#1251). Thanks @diego-plan9 for reporting.

6.2.9 2.19.4 (2019-06-16)

Bug fixes:

- Microseconds no longer gets lost when deserializing datetimes without dateutil installed (#1147).

6.2.10 2.19.3 (2019-06-15)

Bug fixes:

- Fix bug where nested fields in `Meta.exclude` would not work on multiple instantiations (#1212). Thanks @MHannila for reporting.

6.2.11 2.19.2 (2019-03-30)

Bug fixes:

- Handle `OverflowError` when (de)serializing large integers with `fields.Float` (#1177). Thanks @brycedrennan for the PR.

6.2.12 2.19.1 (2019-03-16)

Bug fixes:

- Fix bug where `Nested(many=True)` would skip first element when serializing a generator (#1163). Thanks @khvn26 for the catch and patch.

6.2.13 2.19.0 (2019-03-07)

Deprecation/Removal:

- A `RemovedInMarshmallow3` warning is raised when using `fields.FormattedString`. Use `fields.Method` or `fields.Function` instead (#1141).

6.2.14 2.18.1 (2019-02-15)

Bug fixes:

- A `ChangedInMarshmallow3Warning` is no longer raised when `strict=False` (#1108). Thanks @Aegdesil for reporting.

6.2.15 2.18.0 (2019-01-13)

Features:

- Add warnings for functions in `marshmallow.utils` that are removed in marshmallow 3.

Bug fixes:

- Copying `missing` with `copy.copy` or `copy.deepcopy` will not duplicate it (#1099).

6.2.16 2.17.0 (2018-12-26)

Features:

- Add `marshmallow.__version_info__` (#1074).
- Add warnings for API that is deprecated or changed to help users prepare for marshmallow 3 (#1075).

6.2.17 2.16.3 (2018-11-01)

Bug fixes:

- Prevent memory leak when dynamically creating classes with `type()` (#732). Thanks @asmodehn for writing the tests to reproduce this issue.

6.2.18 2.16.2 (2018-10-30)

Bug fixes:

- Prevent warning about importing from `collections` on Python 3.7 (#1027). Thanks @nkonin for reporting and @jmargeta for the PR.

6.2.19 2.16.1 (2018-10-17)

Bug fixes:

- Remove spurious warning about implicit collection handling (#998). Thanks @lalvarezguillen for reporting.

6.2.20 2.16.0 (2018-10-10)

Bug fixes:

- Allow username without password in basic auth part of the url in `fields.Url` (#982). Thanks user:`alefnula` for the PR.

Other changes:

- Drop support for Python 3.3 (#987).

6.2.21 2.15.6 (2018-09-20)

Bug fixes:

- Prevent `TypeError` when a non-collection is passed to a `Schema` with `many=True`. Instead, raise `ValidationError` with `{'_schema': ['Invalid input type.']}` (#906).
- Fix `root` attribute for nested container fields on list on inheriting schemas (#956). Thanks @bmcbu for reporting.

These fixes were backported from 3.0.0b15 and 3.0.0b16.

6.2.22 2.15.5 (2018-09-15)

Bug fixes:

- Handle empty SQLAlchemy lazy lists gracefully when dumping (#948). Thanks @vke-code for the catch and @YuriHeupa for the patch.

6.2.23 2.15.4 (2018-08-04)

Bug fixes:

- Respect `load_from` when reporting errors for `@validates('field_name')` (#748). Thanks @m-novikov for the catch and patch.

6.2.24 2.15.3 (2018-05-20)

Bug fixes:

- Fix passing `only` as a string to `nested` when the passed field defines `dump_to` (#800, #822). Thanks @deckar01 for the catch and patch.

6.2.25 2.15.2 (2018-05-10)

Bug fixes:

- Fix a race condition in validation when concurrent threads use the same `Schema` instance (#783). Thanks @yupeng0921 and @lafrech for the fix.
- Fix serialization behavior of `fields.List(fields.Integer(as_string=True))` (#788). Thanks @cactus for reporting and @lafrech for the fix.
- Fix behavior of `exclude` parameter when passed from parent to nested schemas (#728). Thanks @timc13 for reporting and @deckar01 for the fix.

6.2.26 2.15.1 (2018-04-25)

Bug fixes:

- CVE-2018-17175: Fix behavior when an empty list is passed as the `only` argument (#772). Thanks @deckar01 for reporting and thanks @lafrech for the fix.

6.2.27 2.15.0 (2017-12-02)

Bug fixes:

- Handle `UnicodeDecodeError` when deserializing `bytes` with a `String` field (#650). Thanks @dan-blanchard for the suggestion and thanks @4lissonsilveira for the PR.

6.2.28 2.14.0 (2017-10-23)

Features:

- Add `require_tld` parameter to `validate.URL` (#664). Thanks @sduthil for the suggestion and the PR.

6.2.29 2.13.6 (2017-08-16)

Bug fixes:

- Fix serialization of types that implement `__getitem__` (#669). Thanks @MichalKononenko.

6.2.30 2.13.5 (2017-04-12)

Bug fixes:

- Fix validation of iso8601-formatted dates (#556). Thanks @lafrech for reporting.

6.2.31 2.13.4 (2017-03-19)

Bug fixes:

- Fix symmetry of serialization and deserialization behavior when passing a dot-delimited path to the `attribute` parameter of fields (#450). Thanks @itajaja for reporting.

6.2.32 2.13.3 (2017-03-11)

Bug fixes:

- Restore backwards-compatibility of `SchemaOpts` constructor (#597). Thanks @Wesmania for reporting and thanks @frol for the fix.

6.2.33 2.13.2 (2017-03-10)

Bug fixes:

- Fix inheritance of `ordered` option when `Schema` subclasses define `class Meta` (#593). Thanks @frol.

Support:

- Update contributing docs.

6.2.34 2.13.1 (2017-03-04)

Bug fixes:

- Fix sorting on Schema subclasses when `ordered=True` (#592). Thanks @frol.

6.2.35 2.13.0 (2017-02-18)

Features:

- Minor optimizations (#577). Thanks @rowillia for the PR.

6.2.36 2.12.2 (2017-01-30)

Bug fixes:

- Unbound fields return `None` rather returning the field itself. This fixes a corner case introduced in #572. Thanks @touilleMan for reporting and @YuriHeupa for the fix.

6.2.37 2.12.1 (2017-01-23)

Bug fixes:

- Fix behavior when a `Nested` field is composed within a `List` field (#572). Thanks @avish for reporting and @YuriHeupa for the PR.

6.2.38 2.12.0 (2017-01-22)

Features:

- Allow passing nested attributes (e.g. `'child.field'`) to the `dump_only` and `load_only` parameters of `Schema` (#572). Thanks @YuriHeupa for the PR.
- Add `schemes` parameter to `fields.URL` (#574). Thanks @mosquito for the PR.

6.2.39 2.11.1 (2017-01-08)

Bug fixes:

- Allow `strict` class Meta option to be overriden by constructor (#550). Thanks @douglas-treadwell for reporting and thanks @podhmo for the PR.

6.2.40 2.11.0 (2017-01-08)

Features:

- Import `marshmallow.fields` in `marshmallow/__init__.py` to save an import when importing the `marshmallow` module (#557). Thanks @mindojo-victor.

Support:

- Documentation: Improve example in "Validating Original Input Data" (#558). Thanks @altaurog.
- Test against Python 3.6.

6.2.41 2.10.5 (2016-12-19)

Bug fixes:

- Reset user-defined kwargs passed to `ValidationError` on each `Schema.load` call (#565). Thanks @jbasko for the catch and patch.

Support:

- Tests: Fix redefinition of `test_utils.test_get_value()` (#562). Thanks @nelfin.

6.2.42 2.10.4 (2016-11-18)

Bug fixes:

- `Function` field works with callables that use Python 3 type annotations (#540). Thanks @martinstein for reporting and thanks @sabinem, @lafrech, and @maximkulkin for the work on the PR.

6.2.43 2.10.3 (2016-10-02)

Bug fixes:

- Fix behavior for serializing missing data with `Number` fields when `as_string=True` is passed (#538). Thanks @jessemyers for reporting.

6.2.44 2.10.2 (2016-09-25)

Bug fixes:

- Use fixed-point notation rather than engineering notation when serializing with `Decimal` (#534). Thanks @gdub.
- Fix UUID validation on serialization and deserialization of `uuid.UUID` objects (#532). Thanks @pauljz.

6.2.45 2.10.1 (2016-09-14)

Bug fixes:

- Fix behavior when using `validate.Equal(False)` (#484). Thanks @pktangyue for reporting and thanks @tuukkamustonen for the fix.
- Fix `strict` behavior when errors are raised in `pre_dump`/`post_dump` processors (#521). Thanks @tvuotila for the catch and patch.

- Fix validation of nested fields on dumping (#528). Thanks again @tvuotila.

6.2.46 2.10.0 (2016-09-05)

Features:

- Errors raised by pre/post-load/dump methods will be added to a schema's errors dictionary (#472). Thanks @dbertouille for the suggestion and for the PR.

6.2.47 2.9.1 (2016-07-21)

Bug fixes:

- Fix serialization of `datetime.time` objects with microseconds (#464). Thanks @Tim-Erwin for reporting and thanks @vuonghv for the fix.
- Make `@validates` consistent with field validator behavior: if validation fails, the field will not be included in the deserialized output (#391). Thanks @martinstein for reporting and thanks @vuonghv for the fix.

6.2.48 2.9.0 (2016-07-06)

- `Decimal` field coerces input values to a string before deserializing to a `decimal.Decimal` object in order to avoid transformation of float values under 12 significant digits (#434, #435). Thanks @davidthornton for the PR.

6.2.49 2.8.0 (2016-06-23)

Features:

- Allow `only` and `exclude` parameters to take nested fields, using dot-delimited syntax (e.g. `only=['blog.author.email']`) (#402). Thanks @Tim-Erwin and @deckar01 for the discussion and implementation.

Support:

- Update tasks.py for compatibility with invoke>=0.13.0. Thanks @deckar01.

6.2.50 2.7.3 (2016-05-05)

- Make `field.parent` and `field.name` accessible to `on_bind_field` (#449). Thanks @immerrr.

6.2.51 2.7.2 (2016-04-27)

No code changes in this release. This is a reupload in order to distribute an sdist for the last hotfix release. See #443.

Support:

- Update license entry in setup.py to fix RPM distributions (#433). Thanks @rrajaravi for reporting.

6.2.52 2.7.1 (2016-04-08)

Bug fixes:

- Only add Schemas to class registry if a class name is provided. This allows Schemas to be constructed dynamically using the `type` constructor without getting added to the class registry (which is useful for saving memory).

6.2.53 2.7.0 (2016-04-04)

Features:

- Make context available to `Nested` field's `on_bind_field` method (#408). Thanks @immerrr for the PR.
- Pass through user `ValidationError` kwargs (#418). Thanks @russelldavies for helping implement this.

Other changes:

- Remove unused attributes `root`, `parent`, and `name` from `SchemaABC` (#410). Thanks @Tim-Erwin for the PR.

6.2.54 2.6.1 (2016-03-17)

Bug fixes:

- Respect `load_from` when reporting errors for nested required fields (#414). Thanks @yumike.

6.2.55 2.6.0 (2016-02-01)

Features:

- Add `partial` argument to `Schema.validate` (#379). Thanks @tdevelioglu for the PR.
- Add `equal` argument to `validate.Length`. Thanks @daniloakamine.
- Collect all validation errors for each item deserialized by a `List` field (#345). Thanks @maximkulkin for the report and the PR.

6.2.56 2.5.0 (2016-01-16)

Features:

- Allow a tuple of field names to be passed as the `partial` argument to `Schema.load` (#369). Thanks @tdevelioglu for the PR.
- Add `schemes` argument to `validate.URL` (#356).

6.2.57 2.4.2 (2015-12-08)

Bug fixes:

- Prevent duplicate error messages when validating nested collections (#360). Thanks @alexmorken for the catch and patch.

6.2.58 2.4.1 (2015-12-07)

Bug fixes:

- Serializing an iterator will not drop the first item (#343, #353). Thanks @jmcarp for the patch. Thanks @edgar-allang and @jmcarp for reporting.

6.2.59 2.4.0 (2015-12-06)

Features:

- Add `skip_on_field_errors` parameter to `validates_schema` (#323). Thanks @jjvattamattom for the suggestion and @d-sutherland for the PR.

Bug fixes:

- Fix `FormattedString` serialization (#348). Thanks @acaird for reporting.
- Fix `@validates` behavior when used when `attribute` is specified and `strict=True` (#350). Thanks @density for reporting.

6.2.60 2.3.0 (2015-11-22)

Features:

- Add `dump_to` parameter to fields (#310). Thanks @ShayanArmanPercolate for the suggestion. Thanks @franciscod and @ewang for the PRs.
- The `deserialize` function passed to `fields.Function` can optionally receive a `context` argument (#324). Thanks @DamianHeard.
- The `serialize` function passed to `fields.Function` is optional (#325). Thanks again @DamianHeard.
- The `serialize` function passed to `fields.Method` is optional (#329). Thanks @justanr.

Deprecation/Removal:

- The `func` argument of `fields.Function` has been renamed to `serialize`.
- The `method_name` argument of `fields.Method` has been renamed to `serialize`.

`func` and `method_name` are still present for backwards-compatibility, but they will both be removed in marshmallow 3.0.

6.2.61 2.2.1 (2015-11-11)

Bug fixes:

- Skip field validators for fields that aren't included in `only` (#320). Thanks @carlos-alberto for reporting and @eprikazc for the PR.

6.2.62 2.2.0 (2015-10-26)

Features:

- Add support for partial deserialization with the `partial` argument to `Schema` and `Schema.load` (#290). Thanks @taion.

Deprecation/Removals:

- `Query` and `QuerySelect` fields are removed.
- Passing of strings to `required` and `allow_none` is removed. Pass the `error_messages` argument instead.

Support:

- Add example of Schema inheritance in docs (#225). Thanks @martinstein for the suggestion and @juanrossi for the PR.
- Add "Customizing Error Messages" section to custom fields docs.

6.2.63 2.1.3 (2015-10-18)

Bug fixes:

- Fix serialization of collections for which `iter` will modify position, e.g. Pymongo cursors (#303). Thanks @Mise for the catch and patch.

6.2.64 2.1.2 (2015-10-14)

Bug fixes:

- Fix passing data to schema validator when using `@validates_schema(many=True)` (#297). Thanks @d-sutherland for reporting.
- Fix usage of `@validates` with a nested field when `many=True` (#298). Thanks @nelfin for the catch and patch.

6.2.65 2.1.1 (2015-10-07)

Bug fixes:

- `Constant` field deserializes to its value regardless of whether its field name is present in input data (#291). Thanks @fayazkhan for reporting.

6.2.66 2.1.0 (2015-09-30)

Features:

- Add `Dict` field for arbitrary mapping data (#251). Thanks @dwieeb for adding this and @Dowwie for the suggestion.
- Add `Field.root` property, which references the field's Schema.

Deprecation/Removals:

- The `extra` param of `Schema` is deprecated. Add extra data in a `post_load` method instead.
- `UnmarshallingError` and `MarshallingError` are removed.

Bug fixes:

- Fix storing multiple schema-level validation errors (#287). Thanks @evgeny-sureev for the patch.
- If `missing=None` on a field, `allow_none` will be set to `True`.

Other changes:

- A `List's` inner field will have the list field set as its parent. Use `root` to access the `Schema`.

6.2.67 2.0.0 (2015-09-25)

Features:

- Make error messages configurable at the class level and instance level (`Field.default_error_messages` attribute and `error_messages` parameter, respectively).

Deprecation/Removals:

- Remove `make_object`. Use a `post_load` method instead (#277).
- Remove the `error` parameter and attribute of `Field`.
- Passing string arguments to `required` and `allow_none` is deprecated. Pass the `error_messages` argument instead. **This API will be removed in version 2.2**.
- Remove `Arbitrary`, `Fixed`, and `Price` fields (#86). Use `Decimal` instead.
- Remove `Select` / `Enum` fields (#135). Use the `OneOf` validator instead.

Bug fixes:

- Fix error format for `Nested` fields when `many=True`. Thanks @alexmorken.
- `pre_dump` methods are invoked before implicit field creation. Thanks @makmanalp for reporting.
- Return correct "required" error message for `Nested` field.
- The `only` argument passed to a `Schema` is bounded by the `fields` option (#183). Thanks @lustdante for the suggestion.

Changes from 2.0.0rc2:

- `error_handler` and `accessor` options are replaced with the `handle_error` and `get_attribute` methods #284.
- Remove `marshmallow.compat.plain_function` since it is no longer used.
- Non-collection values are invalid input for `List` field (#231). Thanks @density for reporting.
- Bug fix: Prevent infinite loop when validating a required, self-nested field. Thanks @Bachmann1234 for the fix.

6.2.68 2.0.0rc2 (2015-09-16)

Deprecation/Removals:

- `make_object` is deprecated. Use a `post_load` method instead (#277). **This method will be removed in the final 2.0 release**.
- `Schema.accessor` and `Schema.error_handler` decorators are deprecated. Define the `accessor` and `error_handler` class Meta options instead.

Bug fixes:

- Allow non-field names to be passed to `ValidationError` (#273). Thanks @evgeny-sureev for the catch and patch.

Changes from 2.0.0rc1:

- The `raw` parameter of the `pre_*`, `post_*`, `validates_schema` decorators was renamed to `pass_many` (#276).
- Add `pass_original` parameter to `post_load` and `post_dump` (#216).
- Methods decorated with the `pre_*`, `post_*`, and `validates_*` decorators must be instance methods. Class methods and instance methods are not supported at this time.

6.2.69 2.0.0rc1 (2015-09-13)

Features:

- *Backwards-incompatible*: `fields.Field._deserialize` now takes `attr` and `data` as arguments (#172). Thanks @alexmic and @kevinastone for the suggestion.
- Allow a `Field's attribute` to be modified during deserialization (#266). Thanks @floqqi.
- Allow partially-valid data to be returned for `Nested` fields (#269). Thanks @jomag for the suggestion.
- Add `Schema.on_bind_field` hook which allows a `Schema` to modify its fields when they are bound.
- Stricter validation of string, boolean, and number fields (#231). Thanks @touilleMan for the suggestion.
- Improve consistency of error messages.

Deprecation/Removals:

- `Schema.validator`, `Schema.preprocessor`, and `Schema.data_handler` are removed. Use `validates_schema`, `pre_load`, and `post_dump` instead.
- `QuerySelect` and `QuerySelectList` are deprecated (#227). **These fields will be removed in version 2.1.**
- `utils.get_callable_name` is removed.

Bug fixes:

- If a date format string is passed to a `DateTime` field, it is always used for deserialization (#248). Thanks @bartaelterman and @praveen-p.

Support:

- Documentation: Add "Using Context" section to "Extending Schemas" page (#224).
- Include tests and docs in release tarballs (#201).
- Test against Python 3.5.

6.2.70 2.0.0b5 (2015-08-23)

Features:

- If a field corresponds to a callable attribute, it will be called upon serialization. Thanks @alexmorken.
- Add `load_only` and `dump_only` class Meta options. Thanks @kelvinhammond.
- If a `Nested` field is required, recursively validate any required fields in the nested schema (#235). Thanks @max-orhai.
- Improve error message if a list of dicts is not passed to a `Nested` field for which `many=True`. Thanks again @max-orhai.

Bug fixes:

- `make_object` is only called after all validators and postprocessors have finished (#253). Thanks @sunsongxp for reporting.
- If an invalid type is passed to `Schema` and `strict=False`, store a `_schema` error in the errors dict rather than raise an exception (#261). Thanks @density for reporting.

Other changes:

- `make_object` is only called when input data are completely valid (#243). Thanks @kissgyorgy for reporting.
- Change default error messages for `URL` and `Email` validators so that they don't include user input (#255).
- `Email` validator permits email addresses with non-ASCII characters, as per RFC 6530 (#221). Thanks @lextoumbourou for reporting and @mwstobo for sending the patch.

6.2.71 2.0.0b4 (2015-07-07)

Features:

- `List` field respects the `attribute` argument of the inner field. Thanks @jmcarp.
- The `container` field `List` field has access to its parent `Schema` via its `parent` attribute. Thanks again @jmcarp.

Deprecation/Removals:

- Legacy validator functions have been removed (#73). Use the class-based validators in `marshmallow.validate` instead.

Bug fixes:

- `fields.Nested` correctly serializes nested `sets` (#233). Thanks @traut.

Changes from 2.0.0b3:

- If `load_from` is used on deserialization, the value of `load_from` is used as the key in the errors dict (#232). Thanks @alexmorken.

6.2.72 2.0.0b3 (2015-06-14)

Features:

- Add `marshmallow.validates_schema` decorator for defining schema-level validators (#116).
- Add `marshmallow.validates` decorator for defining field validators as Schema methods (#116). Thanks @philtay.
- Performance improvements.
- Defining `__marshallable__` on complex objects is no longer necessary.
- Add `fields.Constant`. Thanks @kevinastone.

Deprecation/Removals:

- Remove `skip_missing` class Meta option. By default, missing inputs are excluded from serialized output (#211).
- Remove optional `context` parameter that gets passed to methods for `Method` fields.
- `Schema.validator` is deprecated. Use `marshmallow.validates_schema` instead.
- `utils.get_func_name` is removed. Use `utils.get_callable_name` instead.

Bug fixes:

- Fix serializing values from keyed tuple types (regression of #28). Thanks @makmanalp for reporting.

Other changes:

- Remove unnecessary call to `utils.get_value` for `Function` and `Method` fields (#208). Thanks @jmcarp.
- Serializing a collection without passing `many=True` will not result in an error. Be very careful to pass the `many` argument when necessary.

Support:

- Documentation: Update Flask and Peewee examples. Update Quickstart.

Changes from 2.0.0b2:

- `Boolean` field serializes `None` to `None`, for consistency with other fields (#213). Thanks @cmanallen for reporting.
- Bug fix: `load_only` fields do not get validated during serialization.
- Implicit passing of original, raw data to Schema validators is removed. Use `@marshmallow.validates_schema(pass_original=True)` instead.

6.2.73 2.0.0b2 (2015-05-03)

Features:

- Add useful `__repr__` methods to validators (#204). Thanks @philtay.
- *Backwards-incompatible*: By default, `NaN`, `Infinity`, and `-Infinity` are invalid values for `fields.Decimal`. Pass `allow_nan=True` to allow these values. Thanks @philtay.

Changes from 2.0.0b1:

- Fix serialization of `None` for `Time`, `TimeDelta`, and `Date` fields (a regression introduced in 2.0.0a1).

Includes bug fixes from 1.2.6.

6.2.74 2.0.0b1 (2015-04-26)

Features:

- Errored fields will not appear in (de)serialized output dictionaries (#153, #202).
- Instantiate `OPTIONS_CLASS` in `SchemaMeta`. This makes `Schema.opts` available in metaclass methods. It also causes validation to occur earlier (upon `Schema` class declaration rather than instantiation).
- Add `SchemaMeta.get_declared_fields` class method to support adding additional declared fields.

Deprecation/Removals:

- Remove `allow_null` parameter of `fields.Nested` (#203).

Changes from 2.0.0a1:

- Fix serialization of `None` for `fields.Email`.

6.2.75 2.0.0a1 (2015-04-25)

Features:

- *Backwards-incompatible*: When `many=True`, the errors dictionary returned by `dump` and `load` will be keyed on the indices of invalid items in the (de)serialized collection (#75). Add `index_errors=False` on a Schema's `class Meta` options to disable this behavior.
- *Backwards-incompatible*: By default, fields will raise a ValidationError if the input is `None`. The `allow_none` parameter can override this behavior.
- *Backwards-incompatible*: A `Field's default` parameter is only used if explicitly set and the field's value is missing in the input to `Schema.dump`. If not set, the key will not be present in the serialized output for missing values . This is the behavior for *all* fields. `fields.Str` no longer defaults to `''`, `fields.Int` no longer defaults to `0`, etc. (#199). Thanks @jmcarp for the feedback.
- In `strict` mode, a `ValidationError` is raised. Error messages are accessed via the `ValidationError's messages` attribute (#128).
- Add `allow_none` parameter to `fields.Field`. If `False` (the default), validation fails when the field's value is `None` (#76, #111). If `allow_none` is `True`, `None` is considered valid and will deserialize to `None`.
- Schema-level validators can store error messages for multiple fields (#118). Thanks @ksesong for the suggestion.
- Add `pre_load`, `post_load`, `pre_dump`, and `post_dump` Schema method decorators for defining pre- and post- processing routines (#153, #179). Thanks @davidism, @taion, and @jmcarp for the suggestions and feedback. Thanks @taion for the implementation.
- Error message for `required` validation is configurable. (#78). Thanks @svenstaro for the suggestion. Thanks @0xDCA for the implementation.
- Add `load_from` parameter to fields (#125). Thanks @hakjoon.
- Add `load_only` and `dump_only` parameters to fields (#61, #87). Thanks @philtay.
- Add `missing` parameter to fields (#115). Thanks @philtay.
- Schema validators can take an optional `raw_data` argument which contains raw input data, incl. data not specified in the schema (#127). Thanks @ryanlowe0.
- Add `validate.OneOf` (#135) and `validate.ContainsOnly` (#149) validators. Thanks @philtay.
- Error messages for validators can be interpolated with `{input}` and other values (depending on the validator).
- `fields.TimeDelta` always serializes to an integer value in order to avoid rounding errors (#105). Thanks @philtay.
- Add `include` class Meta option to support field names which are Python keywords (#139). Thanks @nickretallack for the suggestion.
- `exclude` parameter is respected when used together with `only` parameter (#165). Thanks @lustdante for the catch and patch.
- `fields.List` works as expected with generators and sets (#185). Thanks @sergey-aganezov-jr.

Deprecation/Removals:

- `MarshallingError` and `UnmarshallingError` error are deprecated in favor of a single `ValidationError` (#160).
- `context` argument passed to Method fields is deprecated. Use `self.context` instead (#184).
- Remove `ForcedError`.

- Remove support for generator functions that yield validators (#74). Plain generators of validators are still supported.
- The `Select/Enum` field is deprecated in favor of using `validate.OneOf` validator (#135).
- Remove legacy, pre-1.0 API (`Schema.data` and `Schema.errors` properties) (#73).
- Remove `null` value.

Other changes:

- `Marshaller`, `Unmarshaller` were moved to `marshmallow.marshalling`. These should be considered private API (#129).
- Make `allow_null=True` the default for `Nested` fields. This will make `None` serialize to `None` rather than a dictionary with empty values (#132). Thanks @nickrellack for the suggestion.

6.2.76 1.2.6 (2015-05-03)

Bug fixes:

- Fix validation error message for `fields.Decimal`.
- Allow error message for `fields.Boolean` to be customized with the `error` parameter (like other fields).

6.2.77 1.2.5 (2015-04-25)

Bug fixes:

- Fix validation of invalid types passed to a `Nested` field when `many=True` (#188). Thanks @juanrossi for reporting.

Support:

- Fix pep8 dev dependency for flake8. Thanks @taion.

6.2.78 1.2.4 (2015-03-22)

Bug fixes:

- Fix behavior of `as_string` on `fields.Integer` (#173). Thanks @taion for the catch and patch.

Other changes:

- Remove dead code from `fields.Field`. Thanks @taion.

Support:

- Correction to `_postprocess` method in docs. Thanks again @taion.

6.2.79 1.2.3 (2015-03-15)

Bug fixes:

- Fix inheritance of `ordered` class Meta option (#162). Thanks @stephenfin for reporting.

6.2.80 1.2.2 (2015-02-23)

Bug fixes:

- Fix behavior of `skip_missing` and `accessor` options when `many=True` (#137). Thanks @3rdcycle.
- Fix bug that could cause an `AttributeError` when nesting schemas with schema-level validators (#144). Thanks @vovanbo for reporting.

6.2.81 1.2.1 (2015-01-11)

Bug fixes:

- A `Schema's error_handler`–if defined–will execute if `Schema.validate` returns validation errors (#121).
- Deserializing `None` returns `None` rather than raising an `AttributeError` (#123). Thanks @RealSalmon for the catch and patch.

6.2.82 1.2.0 (2014-12-22)

Features:

- Add `QuerySelect` and `QuerySelectList` fields (#84).
- Convert validators in `marshmallow.validate` into class-based callables to make them easier to use when declaring fields (#85).
- Add `Decimal` field which is safe to use when dealing with precise numbers (#86).

Thanks @philtay for these contributions.

Bug fixes:

- `Date` fields correctly deserializes to a `datetime.date` object when `python-dateutil` is not installed (#79). Thanks @malexer for the catch and patch.
- Fix bug that raised an `AttributeError` when using a class-based validator.
- Fix `as_string` behavior of Number fields when serializing to default value.
- Deserializing `None` or the empty string with either a `DateTime`, `Date`, `Time` or `TimeDelta` results in the correct unmarshalling errors (#96). Thanks @svenstaro for reporting and helping with this.
- Fix error handling when deserializing invalid UUIDs (#106). Thanks @vesauimonen for the catch and patch.
- `Schema.loads` correctly defaults to use the value of `self.many` rather than defaulting to `False` (#108). Thanks @davidism for the catch and patch.
- Validators, data handlers, and preprocessors are no longer shared between schema subclasses (#88). Thanks @amikholap for reporting.
- Fix error handling when passing a `dict` or `list` to a `ValidationError` (#110). Thanks @ksesong for reporting.

Deprecation:

- The validator functions in the `validate` module are deprecated in favor of the class-based validators (#85).
- The `Arbitrary`, `Price`, and `Fixed` fields are deprecated in favor of the `Decimal` field (#86).

Support:

- Update docs theme.
- Update contributing docs (#77).
- Fix namespacing example in "Extending Schema" docs. Thanks @Ch00k.
- Exclude virtualenv directories from syntax checking (#99). Thanks @svenstaro.

6.2.83 1.1.0 (2014-12-02)

Features:

- Add `Schema.validate` method which validates input data against a schema. Similar to `Schema.load`, but does not call `make_object` and only returns the errors dictionary.
- Add several validation functions to the `validate` module. Thanks @philtay.
- Store field name and instance on exceptions raised in `strict` mode.

Bug fixes:

- Fix serializing dictionaries when field names are methods of `dict` (e.g. `"items"`). Thanks @rozenm for reporting.
- If a Nested field is passed `many=True`, `None` serializes to an empty list. Thanks @nickretallack for reporting.
- Fix behavior of `many` argument passed to `dump` and `load`. Thanks @svenstaro for reporting and helping with this.
- Fix `skip_missing` behavior for `String` and `List` fields. Thanks @malexer for reporting.
- Fix compatibility with python-dateutil 2.3.
- More consistent error messages across DateTime, TimeDelta, Date, and Time fields.

Support:

- Update Flask and Peewee examples.

6.2.84 1.0.1 (2014-11-18)

Hotfix release.

- Ensure that errors dictionary is correctly cleared on each call to Schema.dump and Schema.load.

6.2.85 1.0.0 (2014-11-16)

Adds new features, speed improvements, better error handling, and updated documentation.

- Add `skip_missing` `class Meta` option.
- A field's `default` may be a callable.
- Allow accessor function to be configured via the `Schema.accessor` decorator or the `__accessor__` class member.
- `URL` and `Email` fields are validated upon serialization.
- `dump` and `load` can receive the `many` argument.
- Move a number of utility functions from fields.py to utils.py.
- More useful `repr` for `Field` classes.

- If a field's default is `fields.missing` and its serialized value is `None`, it will not be included in the final serialized result.
- Schema.dumps no longer coerces its result to a binary string on Python 3.
- *Backwards-incompatible*: Schema output is no longer an `OrderedDict` by default. If you want ordered field output, you must explicitly set the `ordered` option to `True`.
- *Backwards-incompatible*: `error` parameter of the `Field` constructor is deprecated. Raise a `ValidationError` instead.
- Expanded test coverage.
- Updated docs.

6.2.86 1.0.0-a (2014-10-19)

Major reworking and simplification of the public API, centered around support for deserialization, improved validation, and a less stateful `Schema` class.

- Rename `Serializer` to `Schema`.
- Support for deserialization.
- Use the `Schema.dump` and `Schema.load` methods for serializing and deserializing, respectively.
- *Backwards-incompatible*: Remove `Serializer.json` and `Serializer.to_json`. Use `Schema.dumps` instead.
- Reworked fields interface.
- *Backwards-incompatible*: `Field` classes implement `_serialize` and `_deserialize` methods. `serialize` and `deserialize` comprise the public API for a `Field`. `Field.format` and `Field.output` have been removed.
- Add `exceptions.ForcedError` which allows errors to be raised during serialization (instead of storing errors in the `errors` dict).
- *Backwards-incompatible*: `DateTime` field serializes to ISO8601 format by default (instead of RFC822).
- *Backwards-incompatible*: Remove `Serializer.factory` method. It is no longer necessary with the `dump` method.
- *Backwards-incompatible*: Allow nesting a serializer within itself recursively. Use `exclude` or `only` to prevent infinite recursion.
- *Backwards-incompatible*: Multiple errors can be stored for a single field. The errors dictionary returned by `load` and `dump` have lists of error messages keyed by field name.
- Remove `validated` decorator. Validation occurs within `Field` methods.
- `Function` field raises a `ValueError` if an uncallable object is passed to its constructor.
- `Nested` fields inherit context from their parent.
- Add `Schema.preprocessor` and `Schema.validator` decorators for registering preprocessing and schema-level validation functions respectively.
- Custom error messages can be specified by raising a `ValidationError` within a validation function.
- Extra keyword arguments passed to a Field are stored as metadata.
- Fix ordering of field output.
- Fix behavior of the `required` parameter on `Nested` fields.

- Fix serializing keyed tuple types (e.g. `namedtuple`) with `class Meta` options.
- Fix default value for `Fixed` and `Price` fields.
- Fix serialization of binary strings.
- `Schemas` can inherit fields from non-`Schema` base classes (e.g. mixins). Also, fields are inherited according to the MRO (rather than recursing over base classes). Thanks @jmcarp.
- Add `Str`, `Bool`, and `Int` field class aliases.

6.2.87 0.7.0 (2014-06-22)

- Add `Serializer.error_handler` decorator that registers a custom error handler.
- Add `Serializer.data_handler` decorator that registers data post-processing callbacks.
- *Backwards-incompatible*: `process_data` method is deprecated. Use the `data_handler` decorator instead.
- Fix bug that raised error when passing `extra` data together with `many=True`. Thanks @buttsicles for reporting.
- If `required=True` validation is violated for a given `Field`, it will raise an error message that is different from the message specified by the `error` argument. Thanks @asteinlein.
- More generic error message raised when required field is missing.
- `validated` decorator should only wrap a `Field` class's `output` method.

6.2.88 0.6.0 (2014-06-03)

- Fix bug in serializing keyed tuple types, e.g. `namedtuple` and `KeyedTuple`.
- Nested field can load a serializer by its class name as a string. This makes it easier to implement 2-way nesting.
- Make Serializer.data override-able.

6.2.89 0.5.5 (2014-05-02)

- Add `Serializer.factory` for creating a factory function that returns a Serializer instance.
- `MarshallingError` stores its underlying exception as an instance variable. This is useful for inspecting errors.
- `fields.Select` is aliased to `fields.Enum`.
- Add `fields.__all__` and `marshmallow.__all__` so that the modules can be more easily extended.
- Expose `Serializer.OPTIONS_CLASS` as a class variable so that options defaults can be overridden.
- Add `Serializer.process_data` hook that allows subclasses to manipulate the final output data.

6.2.90 0.5.4 (2014-04-17)

- Add `json_module` class Meta option.
- Add `required` option to fields . Thanks @DeaconDesperado.
- Tested on Python 3.4 and PyPy.

6.2.91 0.5.3 (2014-03-02)

- Fix `Integer` field default. It is now `0` instead of `0.0`. Thanks @kalasjocke.
- Add `context` param to `Serializer`. Allows accessing arbitrary objects in `Function` and `Method` fields.
- `Function` and `Method` fields raise `MarshallingError` if their argument is uncallable.

6.2.92 0.5.2 (2014-02-10)

- Enable custom field validation via the `validate` parameter.
- Add `utils.from_rfc` for parsing RFC datestring to Python datetime object.

6.2.93 0.5.1 (2014-02-02)

- Avoid unnecessary attribute access in `utils.to_marshallable_type` for improved performance.
- Fix RFC822 formatting for localized datetimes.

6.2.94 0.5.0 (2013-12-29)

- Can customize validation error messages by passing the `error` parameter to a field.
- *Backwards-incompatible*: Rename `fields.NumberField` -> `fields.Number`.
- Add `fields.Select`. Thanks @ecarreras.
- Support nesting a Serializer within itself by passing `"self"` into `fields.Nested` (only up to depth=1).
- *Backwards-incompatible*: No implicit serializing of collections. Must set `many=True` if serializing to a list. This ensures that marshmallow handles singular objects correctly, even if they are iterable.
- If Nested field `only` parameter is a field name, only return a single value for the nested object (instead of a dict) or a flat list of values.
- Improved performance and stability.

6.2.95 0.4.1 (2013-12-01)

- An object's `__marshallable__` method, if defined, takes precedence over `__getitem__`.
- Generator expressions can be passed to a serializer.
- Better support for serializing list-like collections (e.g. ORM querysets).
- Other minor bugfixes.

6.2.96 0.4.0 (2013-11-24)

- Add `additional` `class Meta` option.
- Add `dateformat` `class Meta` option.
- Support for serializing UUID, date, time, and timedelta objects.
- Remove `Serializer.to_data` method. Just use `Serialize.data` property.

- String field defaults to empty string instead of `None`.
- *Backwards-incompatible*: `isoformat` and `rfcformat` functions moved to utils.py.
- *Backwards-incompatible*: Validation functions moved to validate.py.
- *Backwards-incompatible*: Remove types.py.
- Reorder parameters to `DateTime` field (first parameter is dateformat).
- Ensure that `to_json` returns bytestrings.
- Fix bug with including an object property in `fields` Meta option.
- Fix bug with passing `None` to a serializer.

6.2.97 0.3.1 (2013-11-16)

- Fix bug with serializing dictionaries.
- Fix error raised when serializing empty list.
- Add `only` and `exclude` parameters to Serializer constructor.
- Add `strict` parameter and option: causes Serializer to raise an error if invalid data are passed in, rather than storing errors.
- Updated Flask + SQLA example in docs.

6.2.98 0.3.0 (2013-11-14)

- Declaring Serializers just got easier. The *class Meta* paradigm allows you to specify fields more concisely. Can specify `fields` and `exclude` options.
- Allow date formats to be changed by passing `format` parameter to `DateTime` field constructor. Can either be `"rfc"` (default), `"iso"`, or a date format string.
- More useful error message when declaring fields as classes (instead of an instance, which is the correct usage).
- Rename MarshallingException -> MarshallingError.
- Rename marshmallow.core -> marshmallow.serializer.

6.2.99 0.2.1 (2013-11-12)

- Allow prefixing field names.
- Fix storing errors on Nested Serializers.
- Python 2.6 support.

6.2.100 0.2.0 (2013-11-11)

- Field-level validation.
- Add `fields.Method`.
- Add `fields.Function`.
- Allow binding of extra data to a serialized object by passing the `extra` param when initializing a `Serializer`.

- Add `relative` paramater to `fields.Url` that allows for relative URLs.

6.2.101 0.1.0 (2013-11-10)

- First release.

6.3 Upgrading to Newer Releases

This section documents migration paths to new releases.

6.3.1 Upgrading to 2.3

The `func` parameter of *`fields.Function`* was renamed to `serialize`.

```
# YES
lowername = fields.Function(serialize=lambda obj: obj.name.lower())
# or
lowername = fields.Function(lambda obj: obj.name.lower())

# NO
lowername = fields.Function(func=lambda obj: obj.name.lower())
```

Similiarly, the `method_name` of *`fields.Method`* was also renamed to `serialize`.

```
# YES
lowername = fields.Method(serialize='lowercase')
# or
lowername = fields.Method('lowercase')

# NO
lowername = fields.Method(method_name='lowercase')
```

The `func` parameter is still available for backwards-compatibility. It will be removed in marshmallow 3.0.

Both *`fields.Function`* and *`fields.Method`* will allow the serialize parameter to not be passed, in this case use the `deserialize` parameter by name.

```
lowername = fields.Function(deserialize=lambda name: name.lower())
# or
lowername = fields.Method(deserialize='lowername')
```

6.3.2 Upgrading to 2.0

Deserializing `None`

In 2.0, validation/deserialization of `None` is consistent across field types. If `allow_none` is `False` (the default), validation fails when the field's value is `None`. If `allow_none` is `True`, `None` is considered valid, and the field deserializes to `None`.

```
from marshmallow import fields

# In 1.0, deserialization of None was inconsistent
fields.Int().deserialize(None)  # 0
fields.Str().deserialize(None)  # ''
fields.DateTime().deserialize(None)  # error: Could not deserialize None to a␣
↪datetime.


# In 2.0, validation/deserialization of None is consistent
fields.Int().deserialize(None)  # error: Field may not be null.
fields.Str().deserialize(None)  # error: Field may not be null.
fields.DateTime().deserialize(None)  # error: Field may not be null.

# allow_none makes None a valid value
fields.Int(allow_none=True).deserialize(None)  # None
```

Default Values

Before version 2.0, certain fields (including *String*, *List*, *Nested*, and number fields) had implicit default values that would be used if their corresponding input value was None or missing.

In 2.0, these implicit defaults are removed. A *Field's* default parameter is only used if you explicitly set it. Otherwise, missing inputs will be excluded from the serialized output.

```
from marshmallow import Schema, fields

class MySchema(Schema):
    str_no_default = fields.Str()
    int_no_default = fields.Int()
    list_no_default = fields.List(fields.Str)

schema = MySchema()

# In 1.0, None was treated as a missing input, so implicit default values were used
schema.dump({'str_no_default': None,
             'int_no_default': None,
             'list_no_default': None}).data
# {'str_no_default': '', 'int_no_default': 0, 'list_no_default': []}

# In 2.0, None serializes to None. No more implicit defaults.
schema.dump({'str_no_default': None,
             'int_no_default': None,
             'list_no_default': None}).data
# {'str_no_default': None, 'int_no_default': None, 'list_no_default': None}
```

```
# In 1.0, implicit default values were used for missing inputs
schema.dump({}).data
# {'int_no_default': 0, 'str_no_default': '', 'list_no_default': []}

# In 2.0, missing inputs are excluded from the serialized output
# if no defaults are specified
schema.dump({}).data
# {}
```

As a consequence of this new behavior, the skip_missing class Meta option has been removed.

Pre-processing and Post-processing Methods

The pre- and post-processing API was significantly improved for better consistency and flexibility. The *pre_load*, *post_load*, *pre_dump*, and *post_dump* should be used to define processing hooks. `Schema.preprocessor` and `Schema.data_handler` are removed.

```
# 1.0 API
from marshmallow import Schema, fields

class ExampleSchema(Schema):
    field_a = fields.Int()

@ExampleSchema.preprocessor
def increment(schema, data):
    data['field_a'] += 1
    return data

@ExampleSchema.data_handler
def decrement(schema, data, obj):
    data['field_a'] -= 1
    return data


# 2.0 API
from marshmallow import Schema, fields, pre_load, post_dump

class ExampleSchema(Schema):
    field_a = fields.Int()

    @pre_load
    def increment(self, data):
        data['field_a'] += 1
        return data

    @post_dump
    def decrement(self, data):
        data['field_a'] -= 1
        return data
```

See the *Extending Schemas* page for more information on the `pre_*` and `post_*` decorators.

Schema Validators

Similar to pre-processing and post-processing methods, schema validators are now defined as methods. Decorate schema validators with *validates_schema*. `Schema.validator` is removed.

```
# 1.0 API
from marshmallow import Schema, fields, ValidationError

class MySchema(Schema):
    field_a = fields.Int(required=True)
    field_b = fields.Int(required=True)

@ExampleSchema.validator
def validate_schema(schema, data):
    if data['field_a'] < data['field_b']:
```

(continues on next page)

(continued from previous page)

```
        raise ValidationError('field_a must be greater than field_b')

# 2.0 API
from marshmallow import Schema, fields, validates_schema, ValidationError

class MySchema(Schema):
    field_a = fields.Int(required=True)
    field_b = fields.Int(required=True)

    @validates_schema
    def validate_schema(self, data):
        if data['field_a'] < data['field_b']:
            raise ValidationError('field_a must be greater than field_b')
```

Custom Accessors and Error Handlers

Custom accessors and error handlers are now defined as methods. `Schema.accessor` and `Schema.error_handler` are deprecated.

```
from marshmallow import Schema, fields

# 1.0 Deprecated API
class ExampleSchema(Schema):
    field_a = fields.Int()

@ExampleSchema.accessor
def get_from_dict(schema, attr, obj, default=None):
    return obj.get(attr, default)

@ExampleSchema.error_handler
def handle_errors(schema, errors, obj):
    raise CustomError('Something bad happened', messages=errors)

# 2.0 API
class ExampleSchema(Schema):
    field_a = fields.Int()

    def get_attribute(self, attr, obj, default):
        return obj.get(attr, default)

    # handle_error gets passed a ValidationError
    def handle_error(self, exc, data):
        raise CustomError('Something bad happened', messages=exc.messages)
```

Use `post_load` instead of `make_object`

The `make_object` method was deprecated from the *`Schema`* API (see #277 for the rationale). In order to deserialize to an object, use a *`post_load`* method.

```
# 1.0
from marshmallow import Schema, fields, post_load

class UserSchema(Schema):
```

(continues on next page)

(continued from previous page)

```
    name = fields.Str()
    created_at = fields.DateTime()

    def make_object(self, data):
        return User(**data)

# 2.0
from marshmallow import Schema, fields, post_load

class UserSchema(Schema):
    name = fields.Str()
    created_at = fields.DateTime()

    @post_load
    def make_user(self, data):
        return User(**data)
```

Error Format when `many=True`

When validating a collection (i.e. when calling `load` or `dump` with `many=True`), the errors dictionary will be keyed on the indices of invalid items.

```
from marshmallow import Schema, fields

class BandMemberSchema(Schema):
    name = fields.String(required=True)
    email = fields.Email()

user_data = [
    {'email': 'mick@stones.com', 'name': 'Mick'},
    {'email': 'invalid', 'name': 'Invalid'},  # invalid email
    {'email': 'keith@stones.com', 'name': 'Keith'},
    {'email': 'charlie@stones.com'},  # missing "name"
]

result = BandMemberSchema(many=True).load(user_data)

# 1.0
result.errors
# {'email': ['"invalid" is not a valid email address.'],
#  'name': ['Missing data for required field.']}

# 2.0
result.errors
# {1: {'email': ['"invalid" is not a valid email address.']},
#  3: {'name': ['Missing data for required field.']}}
```

You can still get the pre-2.0 behavior by setting `index_errors = False` in a `Schema`'s *class Meta* options.

Use `ValidationError` instead of `MarshallingError` and `UnmarshallingError`

The `MarshallingError` and `UnmarshallingError` exceptions are deprecated in favor of a single *`ValidationError`*. Users who have written custom fields or are using `strict` mode will need to change their code accordingly.

Handle `ValidationError` in strict mode

When using `strict` mode, you should handle `ValidationErrors` when calling `Schema.dump` and `Schema.load`.

```
from marshmallow import exceptions as exc

schema = BandMemberSchema(strict=True)

# 1.0
try:
    schema.load({'email': 'invalid-email'})
except exc.UnmarshallingError as err:
    # ...

# 2.0
try:
    schema.load({'email': 'invalid-email'})
except exc.ValidationError as err:
    # ...
```

Accessing error messages in strict mode

In 2.0, `strict` mode was improved so that you can access all error messages for a schema (rather than failing early) by accessing a `ValidationError's messages` attribute.

```
schema = BandMemberSchema(strict=True)

try:
    result = schema.load({'email': 'invalid'})
except ValidationMessage as err:
    print(err.messages)
# {
#     'email': ['"invalid" is not a valid email address.'],
#     'name': ['Missing data for required field.']
# }
```

Custom Fields

Two changes must be made to make your custom fields compatible with version 2.0.

- The *`_deserialize`* method of custom fields now receives `attr` (the key corresponding to the value to be deserialized) and the raw input `data` as arguments.
- Custom fields should raise *`ValidationError`* in their `_deserialize` and `_serialize` methods when a validation error occurs.

```
from marshmallow import fields, ValidationError
from marshmallow.exceptions import UnmarshallingError

# In 1.0, an UnmarshallingError was raised
class PasswordField(fields.Field):

    def _deserialize(self, val):
        if not len(val) >= 6:
```

(continues on next page)

(continued from previous page)

```
            raise UnmarshallingError('Password too short.')
        return val

# In 2.0, _deserialize receives attr and data,
# and a ValidationError is raised
class PasswordField(fields.Field):

    def _deserialize(self, val, attr, data):
        if not len(val) >= 6:
            raise ValidationError('Password too short.')
        return val
```

To make a field compatible with both marshmallow 1.x and 2.x, you can pass `*args` and `**kwargs` to the signature.

```
class PasswordField(fields.Field):

    def _deserialize(self, val, *args, **kwargs):
        if not len(val) >= 6:
            raise ValidationError('Password too short.')
        return val
```

Custom Error Messages

Error messages can be customized at the `Field` class or instance level.

```
# 1.0
field = fields.Number(error='You passed a bad number')

# 2.0
# Instance-level
field = fields.Number(error_messages={'invalid': 'You passed a bad number.'})


# Class-level
class MyNumberField(fields.Number):
    default_error_messages = {
        'invalid': 'You passed a bad number.'
    }
```

Passing a string to `required` is deprecated.

```
# 1.0
field = fields.Str(required='Missing required argument.')

# 2.0
field = fields.Str(error_messages={'required': 'Missing required argument.'})
```

Use `OneOf` instead of `fields.Select`

The `fields.Select` field is deprecated in favor of the newly-added `OneOf` validator.

```
from marshmallow import fields
from marshmallow.validate import OneOf
```

(continues on next page)

(continued from previous page)

```
# 1.0
fields.Select(['red', 'blue'])

# 2.0
fields.Str(validate=OneOf(['red', 'blue']))
```

Accessing Context from Method fields

Use `self.context` to access a schema's context within a `Method` field.

```
class UserSchema(Schema):
    name = fields.String()
    likes_bikes = fields.Method('writes_about_bikes')

    def writes_about_bikes(self, user):
        return 'bicycle' in self.context['blog'].title.lower()
```

Validation Error Messages

The default error messages for many fields and validators have been changed for better consistency.

```
from marshmallow import Schema, fields, validate

class ValidatingSchema(Schema):
    foo = fields.Str()
    bar = fields.Bool()
    baz = fields.Int()
    qux = fields.Float()
    spam = fields.Decimal(2, 2)
    eggs = fields.DateTime()
    email = fields.Str(validate=validate.Email())
    homepage = fields.Str(validate=validate.URL())
    nums = fields.List(fields.Int())

schema = ValidatingSchema()
invalid_data = {
    'foo': 42,
    'bar': 24,
    'baz': 'invalid-integer',
    'qux': 'invalid-float',
    'spam': 'invalid-decimal',
    'eggs': 'invalid-datetime',
    'email': 'invalid-email',
    'homepage': 'invalid-url',
    'nums': 'invalid-list',
}
errors = schema.validate(invalid_data)
# {
#     'foo': ['Not a valid string.'],
#     'bar': ['Not a valid boolean.'],
#     'baz': ['Not a valid integer.'],
#     'qux': ['Not a valid number.'],
#     'spam': ['Not a valid number.']
```

(continues on next page)

(continued from previous page)

```
#     'eggs': ['Not a valid datetime.'],
#     'email': ['Not a valid email address.'],
#     'homepage': ['Not a valid URL.'],
#     'nums': ['Not a valid list.'],
# }
```

More

For a full list of changes in 2.0, see the *Changelog*.

6.3.3 Upgrading to 1.2

Validators

Validators were rewritten as class-based callables, making them easier to use when declaring fields.

```
from marshmallow import fields

# 1.2
from marshmallow.validate import Range

age = fields.Int(validate=[Range(min=0, max=999)])

# Pre-1.2
from marshmallow.validate import ranging

age = fields.Int(validate=[lambda val: ranging(val, min=0, max=999)])
```

The validator functions from 1.1 are deprecated and will be removed in 2.0.

Deserializing the Empty String

In version 1.2, deserialization of the empty string (`''`) with `DateTime`, `Date`, `Time`, or `TimeDelta` fields results in consistent error messages, regardless of whether or not `python-dateutil` is installed.

```
from marshmallow import fields

fields.Date().deserialize('')
# UnmarshallingError: Could not deserialize '' to a date object.
```

Decimal

The `Decimal` field was added to support serialization/deserialization of `decimal.Decimal` numbers. You should use this field when dealing with numbers where precision is critical. The `Fixed`, `Price`, and `Arbitrary` fields are deprecated in favor the `Decimal` field.

6.3.4 Upgrading to 1.0

Version 1.0 marks the first major release of marshmallow. Many big changes were made from the pre-1.0 releases in order to provide a cleaner API, support object deserialization, and improve field validation.

Perhaps the largest change is in how objects get serialized. Serialization occurs by invoking the `Schema.dump()` method rather than passing the object to the constructor. Because only configuration options (e.g. the `many`, `strict`, and `only` parameters) are passed to the constructor, you can more easily reuse serializer instances. The `dump` method also forms a nice symmetry with the `Schema.load()` method, which is used for deserialization.

```
from marshmallow import Schema, fields

class UserSchema(Schema):
    email = fields.Email()
    name = fields.String()

user= User(email='monty@python.org', name='Monty Python')

# 1.0
serializer = UserSchema()
data, errors = serializer.dump(user)
# OR
result = serializer.dump(user)
result.data  # => serialized result
result.errors  # => errors

# Pre-1.0
serialized = UserSchema(user)
data = serialized.data
errors = serialized.errors
```

Note: Some crucial parts of the pre-1.0 API have been retained to ease the transition. You can still pass an object to a `Schema` constructor and access the `Schema.data` and `Schema.errors` properties. The `is_valid` method, however, has been completely removed. It is recommended that you migrate to the new API to prevent future releases from breaking your code.

The Fields interface was also reworked in 1.0 to make it easier to define custom fields with their own serialization and deserialization behavior. Custom fields now implement `Field._serialize()` and `Field._deserialize()`.

```
from marshmallow import fields, MarshallingError

class PasswordField(fields.Field):
    def _serialize(self, value, attr, obj):
        if not value or len(value) < 6:
            raise MarshallingError('Password must be greater than 6 characters.')
        return str(value).strip()

    # Similarly, you can override the _deserialize method
```

Another major change in 1.0 is that multiple validation errors can be stored for a single field. The `errors` dictionary returned by `Schema.dump()` and `Schema.load()` is a list of error messages keyed by field name.

```
from marshmallow import Schema, fields, ValidationError

def must_have_number(val):
    if not any(ch.isdigit() for ch in val):
        raise ValidationError('Value must have an number.')

def validate_length(val):
    if len(val) < 8:
```

(continues on next page)

(continued from previous page)

```
        raise ValidationError('Value must have 8 or more characters.')

class ValidatingSchema(Schema):
    password = fields.String(validate=[must_have_number, validate_length])

result, errors = ValidatingSchema().load({'password': 'secure'})
print(errors)
# {'password': ['Value must have an number.',
#               'Value must have 8 or more characters.']}
```

Other notable changes:

- Serialized output is no longer an `OrderedDict` by default. You must explicitly set the `ordered` class Meta option to `True`.
- `Serializer` has been renamed to `Schema`, but you can still import `marshmallow.Serializer` (which is aliased to `Schema`).
- `datetime` objects serialize to ISO8601-formatted strings by default (instead of RFC821 format).
- The `fields.validated` decorator was removed, as it is no longer necessary given the new Fields interface.
- `Schema.factory` class method was removed.

See also:

See the *Changelog* for a more complete listing of added features, bugfixes and breaking changes.

6.4 Ecosystem

A list of marshmallow-related libraries can be found at the GitHub wiki here:

https://github.com/marshmallow-code/marshmallow/wiki/Ecosystem

6.5 License

6.6 Authors

6.6.1 Leads

- Steven Loria @sloria

6.6.2 Contributors (chronological)

- Sebastian Vetter @elbaschid
- Eduard Carreras @ecarreras
- Joakim Ekberg @kalasjocke
- Mark Grey @DeaconDesperado
- Anders Steinlein @asteinlein
- Cyril Thomas @Ketouem
- Austin Macdonald @asmacdo
- Josh Carp @jmcarp
- @amikholap
- Sven-Hendrik Haase @svenstaro
- Eric Wang @ewang
- @philtay
- @malexer
- Andriy Yurchuk @Ch00k
- Vesa Uimonen @vesauimonen
- David Lord @davidism
- Daniel Castro @0xDCA
- Ben Jones @RealSalmon
- Patrick Woods @hakjoon
- Lukas Heiniger @3rdcycle
- Ryan Lowe @ryanlowe0
- Jimmy Jia @taion
- @lustdante
- Sergey Aganezov, Jr. @sergey-aganezov-jr
- Kevin Stone @kevinastone
- Alex Morken @alexmorken
- Sergey Polzunov @traut
- Kelvin Hammond @kelvinhammond
- Matt Stobo @mwstobo
- Max Orhai @max-orhai

- Praveen @praveen-p
- Stas Sușcov @stas
- Florian @floqqi
- Evgeny Sureev @evgeny-sureev
- Matt Bachmann @Bachmann1234
- Daniel Imhoff @dwieeb
- Juan Rossi @juanrossi
- Andrew Haigh @nelfin
- @Mise
- Taylor Edmiston @tedmiston
- Francisco Demartino @franciscod
- Eric Wang @ewang
- Eugene Prikazchikov @eprikazc
- Damian Heard @DamianHeard
- Alec Reiter @justanr
- Dan Sutherland @d-sutherland
- Jeff Widman @jeffwidman
- Simeon Visser @svisser
- Taylan Develioglu @tdevelioglu
- Danilo Akamine @daniloakamine
- Maxim Kulkin @maximkulkin
- @immerrr
- Mike Yumatov @yumike
- Tim Mundt @Tim-Erwin
- Russell Davies @russelldavies
- David Thornton @davidthornton
- Vuong Hoang @vuonghv
- David Bertouille @dbertouille
- Alexandre Bonnetain @Shir0kamii
- Tuukka Mustonen @tuukkamustonen
- Tero Vuotila @tvuotila
- Paul Zumbrun @pauljz
- Gary Wilson Jr. @gdub
- Sabine Maennel @sabinem
- Victor Varvaryuk @mindojo-victor
- Jāzeps Baško @jbasko

- @podhmo
- Dmitry Orlov @mosquito
- Yuri Heupa @YuriHeupa
- Roy Williams @rowillia
- Vlad Frolov @frol
- Michal Kononenko @MichalKononenko
- @sduthil
- Alisson Silveira @4lissonsilveira
- Maxim Novikov @m-novikov
- Viktor Kerkez @alefnula
- Jan Margeta @jmargeta
- AlexV @asmodehn
- @miniscruff
- Kim Gustyr @khvn26
- Bryce Drennan @brycedrennan
- Cristi Scoarta @cristi23
- Nathan @nbanmp

6.7 Contributing Guidelines

So you're interested in contributing to marshmallow or one of our associated projects? That's awesome! We welcome contributions from anyone willing to work in good faith with other contributors and the community (see also our *Code of Conduct*).

6.7.1 Security Contact Information

To report a security vulnerability, please use the Tidelift security contact. Tidelift will coordinate the fix and disclosure.

6.7.2 Questions, Feature Requests, Bug Reports, and Feedback...

...should all be reported on the Github Issue Tracker .

6.7.3 Ways to Contribute

- Comment on some of marshmallow's open issues (especially those labeled "feedback welcome"). Share a solution or workaround. Make a suggestion for how a feature can be made better. Opinions are welcome!
- Improve the docs. For straightforward edits, click the ReadTheDocs menu button in the bottom-right corner of the page and click "Edit". See the *Documentation* section of this page if you want to build the docs locally.
- If you think you've found a bug, open an issue.
- Contribute an *example usage* of marshmallow.

- Send a PR for an open issue (especially one labeled "help wanted"). The next section details how to contribute code.

6.7.4 Contributing Code

Setting Up for Local Development

1. Fork marshmallow on Github.

```
$ git clone https://github.com/marshmallow-code/marshmallow.git
$ cd marshmallow
```

2. Install development requirements. **It is highly recommended that you use a virtualenv.** Use the following command to install an editable version of marshmallow along with its development requirements.

```
# After activating your virtualenv
$ pip install -e '.[dev]'
```

Git Branch Structure

Marshmallow abides by the following branching model:

`dev` Current development branch. **New features should branch off here**.

`X.Y-line` Maintenance branch for release `X.Y`. **Bug fixes should be sent to the most recent release branch.** The maintainer will forward-port the fix to `dev`. Note: exceptions may be made for bug fixes that introduce large code changes.

Always make a new branch for your work, no matter how small. Also, **do not put unrelated changes in the same branch or pull request**. This makes it more difficult to merge your changes.

Pull Requests

1. Create a new local branch.

```
# For a new feature
$ git checkout -b name-of-feature dev

# For a bugfix
$ git checkout -b fix-something 2.x-line
```

2. Commit your changes. Write good commit messages.

```
$ git commit -m "Detailed commit message"
$ git push origin name-of-feature
```

3. Before submitting a pull request, check the following:

- If the pull request adds functionality, it is tested and the docs are updated.
- You've added yourself to `AUTHORS.rst`.

4. Submit a pull request to `marshmallow-code:dev` or the appropriate maintenance branch. The CI build must be passing before your pull request is merged.

Running tests

To run all tests:

```
$ pytest
```

To run syntax checks:

```
$ tox -e lint
```

(Optional) To run tests on Python 2.7, 3.4, 3.5, and 3.6 virtual environments (must have each interpreter installed):

```
$ tox
```

Documentation

Contributions to the documentation are welcome. Documentation is written in reStructured Text (rST). A quick rST reference can be found here. Builds are powered by Sphinx.

To build the docs in "watch" mode:

```
$ tox -e watch-docs
```

Changes in the `docs/` directory will automatically trigger a rebuild.

Contributing Examples

Have a usage example you'd like to share? A custom `Field` that others might find useful? Feel free to add it to the examples directory and send a pull request.

6.8 Code of Conduct

This code of conduct applies to the marshmallow project and all associated projects in the marshmallow-code organization.

6.8.1 When Something Happens

If you see a Code of Conduct violation, follow these steps:

1. Let the person know that what they did is not appropriate and ask them to stop and/or edit their message(s) or commits.
2. That person should immediately stop the behavior and correct the issue.
3. If this doesn't happen, or if you're uncomfortable speaking up, *contact the maintainers*.
4. As soon as possible, a maintainer will look into the issue, and take *further action (see below)*, starting with a warning, then temporary block, then long-term repo or organization ban.

When reporting, please include any relevant details, links, screenshots, context, or other information that may be used to better understand and resolve the situation.

The maintainer team will prioritize the well-being and comfort of the recipients of the violation over the comfort of the violator. See *some examples below*.

6.8.2 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers of this project pledge to making participation in our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, technical preferences, nationality, personal appearance, race, religion, or sexual identity and orientation.

6.8.3 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language.
- Being respectful of differing viewpoints and experiences.
- Gracefully accepting constructive feedback.
- Focusing on what is best for the community.
- Showing empathy and kindness towards other community members.
- Encouraging and raising up your peers in the project so you can all bask in hacks and glory.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances, including when simulated online. The only exception to sexual topics is channels/spaces specifically for topics of sexual identity.
- Casual mention of slavery or indentured servitude and/or false comparisons of one's occupation or situation to slavery. Please consider using or asking about alternate terminology when referring to such metaphors in technology.
- Making light of/making mocking comments about trigger warnings and content warnings.
- Trolling, insulting/derogatory comments, and personal or political attacks.
- Public or private harassment, deliberate intimidation, or threats.
- Publishing others' private information, such as a physical or electronic address, without explicit permission. This includes any sort of "outing" of any aspect of someone's identity without their consent.
- Publishing private screenshots or quotes of interactions in the context of this project without all quoted users' *explicit* consent.
- Publishing of private communication that doesn't have to do with reporting harrassment.
- Any of the above even when presented as "ironic" or "joking".
- Any attempt to present "reverse-ism" versions of the above as violations. Examples of reverse-isms are "reverse racism", "reverse sexism", "heterophobia", and "cisphobia".
- Unsolicited explanations under the assumption that someone doesn't already know it. Ask before you teach! Don't assume what people's knowledge gaps are.
- Feigning or exaggerating surprise when someone admits to not knowing something.
- "Well-actuallies"
- Other conduct which could reasonably be considered inappropriate in a professional or community setting.

6.8.4 Scope

This Code of Conduct applies both within spaces involving this project and in other spaces involving community members. This includes the repository, its Pull Requests and Issue tracker, its Twitter community, private email communications in the context of the project, and any events where members of the project are participating, as well as adjacent communities and venues affecting the project's members.

Depending on the violation, the maintainers may decide that violations of this code of conduct that have happened outside of the scope of the community may deem an individual unwelcome, and take appropriate action to maintain the comfort and safety of its members.

Other Community Standards

As a project on GitHub, this project is additionally covered by the GitHub Community Guidelines.

Enforcement of those guidelines after violations overlapping with the above are the responsibility of the entities, and enforcement may happen in any or all of the services/communities.

6.8.5 Maintainer Enforcement Process

Once the maintainers get involved, they will follow a documented series of steps and do their best to preserve the well-being of project members. This section covers actual concrete steps.

Contacting Maintainers

As a small and young project, we don't yet have a Code of Conduct enforcement team. Hopefully that will be addressed as we grow, but for now, any issues should be addressed to Steven Loria, via email or any other medium that you feel comfortable with. Using words like "marshmallow code of conduct" in your subject will help make sure your message is noticed quickly.

Further Enforcement

If you've already followed the *initial enforcement steps*, these are the steps maintainers will take for further enforcement, as needed:

1. Repeat the request to stop.
2. If the person doubles down, they will be given an official warning. The PR or Issue may be locked.
3. If the behavior continues or is repeated later, the person will be blocked from participating for 24 hours.
4. If the behavior continues or is repeated after the temporary block, a long-term (6-12mo) ban will be used.
5. If after this the behavior still continues, a permanent ban may be enforced.

On top of this, maintainers may remove any offending messages, images, contributions, etc, as they deem necessary.

Maintainers reserve full rights to skip any of these steps, at their discretion, if the violation is considered to be a serious and/or immediate threat to the health and well-being of members of the community. These include any threats, serious physical or verbal attacks, and other such behavior that would be completely unacceptable in any social setting that puts our members at risk.

Members expelled from events or venues with any sort of paid attendance will not be refunded.

Who Watches the Watchers?

Maintainers and other leaders who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership. These may include anything from removal from the maintainer team to a permanent ban from the community.

Additionally, as a project hosted on GitHub, *their Code of Conduct may be applied against maintainers of this project*, externally of this project's procedures.

6.8.6 Enforcement Examples

The Best Case

The vast majority of situations work out like this. This interaction is common, and generally positive.

> Alex: "Yeah I used X and it was really crazy!"
>
> Patt (not a maintainer): "Hey, could you not use that word? What about 'ridiculous' instead?"
>
> Alex: "oh sorry, sure." -> edits old comment to say "it was really confusing!"

The Maintainer Case

Sometimes, though, you need to get maintainers involved. Maintainers will do their best to resolve conflicts, but people who were harmed by something **will take priority**.

> Patt: "Honestly, sometimes I just really hate using $library and anyone who uses it probably sucks at their job."
>
> Alex: "Whoa there, could you dial it back a bit? There's a CoC thing about attacking folks' tech use like that."
>
> Patt: "I'm not attacking anyone, what's your problem?"
>
> Alex: "@maintainers hey uh. Can someone look at this issue? Patt is getting a bit aggro. I tried to nudge them about it, but nope."
>
> KeeperOfCommitBits: (on issue) "Hey Patt, maintainer here. Could you tone it down? This sort of attack is really not okay in this space."
>
> Patt: "Leave me alone I haven't said anything bad wtf is wrong with you."
>
> KeeperOfCommitBits: (deletes user's comment), "@patt I mean it. Please refer to the CoC over at (URL to this CoC) if you have questions, but you can consider this an actual warning. I'd appreciate it if you reworded your messages in this thread, since they made folks there uncomfortable. Let's try and be kind, yeah?"
>
> Patt: "@KeeperOfCommitBits Okay sorry. I'm just frustrated and I'm kinda burnt out and I guess I got carried away. I'll DM Alex a note apologizing and edit my messages. Sorry for the trouble."
>
> KeeperOfCommitBits: "@patt Thanks for that. I hear you on the stress. Burnout sucks :/. Have a good one!"

The Nope Case

> PepeTheFrog: "Hi, I am a literal actual nazi and I think white supremacists are quite fashionable."
>
> Patt: "NOOOOPE. OH NOPE NOPE."

Alex: “JFC NO. NOPE. @KeeperOfCommitBits NOPE NOPE LOOK HERE”

KeeperOfCommitBits: “ Nope. NOPE NOPE NOPE. ”

PepeTheFrog has been banned from all organization or user repositories belonging to KeeperOfCommitBits.

6.8.7 Attribution

This Code of Conduct is based on Trio’s Code of Conduct, which is based on the WeAllJS Code of Conduct, which is itself based on Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4, and the LGBTQ in Technology Slack Code of Conduct.

6.9 Kudos

A hat tip to Django Rest Framework , Flask-RESTful, and colander for ideas and API design.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

N

O

P

R

S

T

U

V