
marshmallow

Release 4.3.0

Author name not set

Apr 03, 2026

CONTENTS

1	Get it now	3
2	Upgrading from an older version?	5
3	Why another library?	7
4	Sponsors	9
5	Usage guide	11
6	API reference	47
7	Project info	89
	Python Module Index	185
	Index	187

Object serialization and deserialization, lightweight and fluffy.

Release v4.3.0. (*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 pprint import pprint

from marshmallow import Schema, fields

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, 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.

GET IT NOW

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

Ready to get started? Go on to the [Quickstart](#) or check out some [examples](#).

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.

WHY ANOTHER LIBRARY?

See [this document](#) to learn about what makes marshmallow unique.

SPONSORS

marshmallow is sponsored by [Route4Me](#).



Support this project by becoming a sponsor (or ask your company to support this project by becoming a sponsor). Your logo will be displayed here with a link to your website. [[Become a sponsor](#)]

5.1 Installation

5.1.1 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
```

5.1.2 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.

5.2 Quickstart

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

5.2.1 Declaring schemas

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

```
from dataclasses import dataclass, field
import datetime as dt

@dataclass
class User:
    name: str
    email: str
    created_at: dt.datetime = field(default_factory=dt.datetime.now)
```

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 *fields module documentation*.

📘 Creating schemas from dictionaries

You can also create a schema from a dictionary of fields using the *from_dict* method.

```
from marshmallow import Schema, fields

UserSchema = Schema.from_dict(
    {
        "name": fields.Str(),
        "email": fields.Email(),
        "created_at": fields.DateTime(),
    }
)
```

from_dict is especially useful for generating schemas at runtime.

5.2.2 Serializing objects (“dumping”)

Serialize objects by passing them to your schema’s *dump* method, which returns the formatted result.

```
from pprint import pprint

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

5.2.3 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)
# {"name": "Monty", "email": "monty@python.org"}
```

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

5.2.4 Deserializing objects (“loading”)

The reverse of the `dump` method is `load`, which validates and deserializes an input dictionary to an application-level data structure.

By default, `load` will return a dictionary of field names mapped to deserialized values (or raise a `ValidationError` with a dictionary of validation errors, which we’ll *revisit later*).

```
from pprint import pprint

user_data = {
    "created_at": "2014-08-11T05:26:03.869245",
    "email": "ken@yahoo.com",
    "name": "Ken",
}
schema = UserSchema()
result = schema.load(user_data)
pprint(result)
# {'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.

```
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, **kwargs):
        return User(**data)
```

Now, the `load` method return a `User` instance.

```

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

```

5.2.5 Handling collections of objects

Set `many=True` when dealing with iterable collections of objects.

```

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)
pprint(result)
# [{ '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' } ]

```

5.2.6 Validation

`Schema.load` (and its JSON-decoding counterpart, `Schema.loads`) raises a `ValidationError` error when invalid data are passed in. You can access the dictionary of validation errors from the `ValidationError.messages` attribute. The data that were correctly deserialized are accessible in `ValidationError.valid_data`. Some fields, such as the Email and URL fields, have built-in validation.

```

from marshmallow import ValidationError

try:
    result = UserSchema().load({"name": "John", "email": "foo"})
except ValidationError as err:
    print(err.messages) # => {"email": ["foo" is not a valid email address.]}
    print(err.valid_data) # => {"name": "John"}

```

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

```

from pprint import pprint

from marshmallow import Schema, fields, ValidationError

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"},

```

(continues on next page)

(continued from previous page)

```

    {"email": "charlie@stones.com"}, # missing "name"
]

try:
    BandMemberSchema(many=True).load(user_data)
except ValidationError as err:
    pprint(err.messages)
    # {1: {'email': ['Not a valid email address.']}},
    # 3: {'name': ['Missing data for required field.']}

```

You can perform additional validation for a field by passing the `validate` argument. There are a number of built-in validators in the `marshmallow.validate` module.

```

from pprint import pprint

from marshmallow import Schema, fields, validate, ValidationError

class UserSchema(Schema):
    name = fields.Str(validate=validate.Length(min=1))
    permission = fields.Str(validate=validate.OneOf(["read", "write", "admin"]))
    age = fields.Int(validate=validate.Range(min=18, max=40))

in_data = {"name": "", "permission": "invalid", "age": 71}
try:
    UserSchema().load(in_data)
except ValidationError as err:
    pprint(err.messages)
    # {'age': ['Must be greater than or equal to 18 and less than or equal to 40.'],
    #  'name': ['Shorter than minimum length 1.'],
    #  'permission': ['Must be one of: read, write, admin.']}

```

You may implement your own validators. A validator is a callable that accepts a single argument, the value to validate. If validation fails, the callable should raise a `ValidationError` with an error message.

```

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}
try:

```

(continues on next page)

(continued from previous page)

```
result = ItemSchema().load(in_data)
except ValidationError as err:
    print(err.messages) # => {'quantity': ['Quantity must not be greater than 30.']}
```

You may also pass a collection (list, tuple, generator) of callables to validate.

Warning

Validation occurs on deserialization but not on serialization. To improve serialization performance, data passed to `Schema.dump` are considered valid.

See also

You can register a custom error handler function for a schema by overriding the `handle_error` method. See the [Custom error handling](#) page for more info.

See also

If you need to validate multiple fields within a single validator, see [Schema-level validation](#).

See also

Need to *transform* a field's value? Use the `pre_load` and `post_load` field parameters. See [Field-level pre- and post-processing](#).

Field validators as methods

It is sometimes 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: int, data_key: str) -> None:
        if value < 0:
            raise ValidationError("Quantity must be greater than 0.")
        if value > 30:
            raise ValidationError("Quantity must not be greater than 30.")
```

Note

You can pass multiple field names to the `validates` decorator.

```

from marshmallow import Schema, fields, validates, ValidationError

class UserSchema(Schema):
    name = fields.Str(required=True)
    nickname = fields.Str(required=True)

    @validates("name", "nickname")
    def validate_names(self, value: str, data_key: str) -> None:
        if len(value) < 3:
            raise ValidationError("Too short")

```

5.2.7 Required fields

Make a field required by passing `required=True`. An error will be raised 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.

```

from pprint import pprint

from marshmallow import Schema, fields, ValidationError

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()

try:
    result = UserSchema().load({"email": "foo@bar.com"})
except ValidationError as err:
    pprint(err.messages)
    # {'age': ['Age is required.'],
    # 'city': {'code': 400, 'message': 'City required'},
    # 'name': ['Missing data for required field.']}

```

5.2.8 Partial loading

When using the same schema in multiple places, you may only want to skip required validation by passing `partial`.

```

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

```

(continues on next page)

(continued from previous page)

```
result = UserSchema().load({"age": 42}, partial=("name",))
# OR UserSchema(partial=('name',)).load({'age': 42})
print(result) # => {'age': 42}
```

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

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

result = UserSchema().load({"age": 42}, partial=True)
# OR UserSchema(partial=True).load({'age': 42})
print(result) # => {'age': 42}
```

5.2.9 Specifying defaults

`load_default` specifies the default deserialization value for a field. Likewise, `dump_default` specifies the default serialization value.

```
class UserSchema(Schema):
    id = fields.UUID(load_default=uuid.uuid1)
    birthdate = fields.DateTime(dump_default=dt.datetime(2017, 9, 29))

UserSchema().load({})
# {'id': UUID('337d946c-32cd-11e8-b475-0022192ed31b')}
UserSchema().dump({})
# {'birthdate': '2017-09-29T00:00:00+00:00'}
```

5.2.10 Handling unknown fields

By default, `load` will raise a `ValidationError` if it encounters a key with no matching `Field` in the schema.

```
from marshmallow import Schema, fields

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

UserSchema().load(
    {
        "name": "Monty",
        "email": "monty@python.org",
        "created_at": "2014-08-17T14:54:16.049594+00:00",
        "extra": "Not a field",
    }
)
# raises marshmallow.exceptions.ValidationError: {'extra': ['Unknown field.']}
```

This behavior can be modified with the `unknown` option, which accepts one of the following:

- *RAISE* (default): raise a *ValidationError* if there are any unknown fields
- *EXCLUDE*: exclude unknown fields
- *INCLUDE*: accept and include the unknown fields

You can specify *unknown* in the *class Meta* of your *Schema*,

```
from pprint import pprint
from marshmallow import Schema, fields, INCLUDE

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

    class Meta:
        unknown = INCLUDE

result = UserSchema().load(
    {
        "name": "Monty",
        "email": "monty@python.org",
        "created_at": "2014-08-17T14:54:16.049594+00:00",
        "extra": "Not a field",
    }
)
pprint(result)
# {'created_at': datetime.datetime(2014, 8, 17, 14, 54, 16, 49594, tzinfo=datetime.
↪ timezone(datetime.timedelta(0), '+0000')),
# 'email': 'monty@python.org',
# 'extra': 'Not a field',
# 'name': 'Monty'}
```

at instantiation time,

```
schema = UserSchema(unknown=INCLUDE)
```

or when calling *load*.

```
UserSchema().load(data, unknown=INCLUDE)
```

The *unknown* option value set in *load* will override the value applied at instantiation time, which itself will override the value defined in the *class Meta*.

This order of precedence allows you to change the behavior of a schema for different contexts.

5.2.11 Validation without deserialization

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"})
print(errors) # {'email': ['Not a valid email address.']}
```

5.2.12 “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)
```

Warning

When loading, dump-only fields are considered unknown. If the `unknown` option is set to `INCLUDE`, values with keys corresponding to those fields are therefore loaded with no validation.

5.2.13 Specifying serialization/deserialization keys

Schemas will (de)serialize an input dictionary from/to an output dictionary whose keys are identical to the field names. If you are consuming and producing data that does not match your schema, you can specify the output keys via the `data_key` argument.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.Email(data_key="emailAddress")

s = UserSchema()

data = {"name": "Mike", "email": "foo@bar.com"}
result = s.dump(data)
# {'name': u'Mike',
# 'emailAddress': 'foo@bar.com'}

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

5.2.14 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 more detailed usage examples, check out the [Examples](#) page.

5.3 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. And a User may have many friends, each of which is a User.

```
from __future__ import annotations # Enable newer type annotation syntax

import datetime as dt
from dataclasses import dataclass, field

@dataclass
class User:
    name: str
    email: str
    created_at: dt.datetime = field(default_factory=dt.datetime.now)
    friends: list[User] = field(default_factory=list)
    employer: User | None = None

@dataclass
class Blog:
    title: str
    author: User
```

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

```
from marshmallow import Schema, fields

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.

```
from pprint import pprint

user = User(name="Monty", email="monty@python.org")
blog = Blog(title="Something Completely Different", author=user)
result = 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, pass the *Nested* field to *List*.

```
collaborators = fields.List(fields.Nested(UserSchema))
```

5.3.1 Specifying which fields to nest

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

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

schema = BlogSchema2()
result = schema.dump(blog)
pprint(result)
# {
#   'title': u'Something Completely Different',
#   'author': {'email': u'monty@python.org'}
# }
```

Dotted paths may be passed to `only` and `exclude` to specify nested attributes.

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

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

You can replace nested data with a single value (or flat list of values if `many=True`) using the *Pluck* field.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.Email()
    friends = fields.Pluck("self", "name", many=True)

# ... create `user` ...
serialized_data = UserSchema().dump(user)
pprint(serialized_data)
# {
#   "name": "Steve",
#   "email": "steve@example.com",
```

(continues on next page)

(continued from previous page)

```
#     "friends": ["Mike", "Joe"]
# }
deserialized_data = UserSchema().load(result)
pprint(deserialized_data)
# {
#     "name": "Steve",
#     "email": "steve@example.com",
#     "friends": [{"name": "Mike"}, {"name": "Joe"}]
# }
```

5.3.2 Partial loading

Nested schemas also inherit the `partial` parameter of the parent load call.

```
class UserSchemaStrict(Schema):
    name = fields.String(required=True)
    email = fields.Email()
    created_at = fields.DateTime(required=True)

class BlogSchemaStrict(Schema):
    title = fields.String(required=True)
    author = fields.Nested(UserSchemaStrict, required=True)

schema = BlogSchemaStrict()
blog = {"title": "Something Completely Different", "author": {}}
result = schema.load(blog, partial=True)
pprint(result)
# {'author': {}, 'title': 'Something Completely Different'}
```

You can specify a subset of the fields to allow partial loading using dot delimiters.

```
author = {"name": "Monty"}
blog = {"title": "Something Completely Different", "author": author}
result = schema.load(blog, partial=("title", "author.created_at"))
pprint(result)
# {'author': {'name': 'Monty'}, 'title': 'Something Completely Different'}
```

5.3.3 Two-way nesting

If you have two objects that nest each other, you can pass a callable to `Nested`. This allows you to resolve order-of-declaration issues, such as when one schema nests a schema that is declared below it.

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

```
class BookSchema(Schema):
    id = fields.Int(dump_only=True)
    title = fields.Str()

    # Make sure to use the 'only' or 'exclude'
```

(continues on next page)

(continued from previous page)

```
# to avoid infinite recursion
author = fields.Nested(lambda: AuthorSchema(only=("id", "title")))

class AuthorSchema(Schema):
    id = fields.Int(dump_only=True)
    title = fields.Str()

    books = fields.List(fields.Nested(BookSchema(exclude=("author",))))
```

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

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

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

You can also pass a class name as a string to *Nested*. This is useful for avoiding circular imports when your schemas are located in different modules.

```
# books.py
from marshmallow import Schema, fields

class BookSchema(Schema):
    id = fields.Int(dump_only=True)
    title = fields.Str()

    author = fields.Nested("AuthorSchema", only=("id", "title"))
```

```
# authors.py
from marshmallow import Schema, fields

class AuthorSchema(Schema):
    id = fields.Int(dump_only=True)
    title = fields.Str()

    books = fields.List(fields.Nested("BookSchema", exclude=("author",)))
```

Note

If you have multiple schemas with the same class name, you must pass the full, module-qualified path.

```
author = fields.Nested("authors.BookSchema", only=("id", "title"))
```

5.3.4 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 a callable that returns an instance of the same schema.

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.Email()
    # Use the 'exclude' argument to avoid infinite recursion
    employer = fields.Nested(lambda: UserSchema(exclude=("employer", "friends")))
    friends = fields.List(fields.Nested(lambda: UserSchema("employer", "friends")))

user = User("Steve", "steve@example.com")
user.friends.append(User("Mike", "mike@example.com"))
user.friends.append(User("Joe", "joe@example.com"))
user.employer = User("Dirk", "dirk@example.com")
result = UserSchema().dump(user)
pprint(result, 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
#     }
#   ]
# }
```

(continues on next page)

(continued from previous page)

```
# ],
# "employer": {
#     "name": "Dirk",
#     "email": "dirk@example.com",
#     "friends": []
# }
# }
```

5.3.5 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 more detailed usage examples, check out the *Examples* page.

5.4 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.

5.4.1 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. Field's type argument is the internal type, i.e. the type that the field deserializes to.

```
from marshmallow import fields, ValidationError

class PinCode(fields.Field[list[int]]):
    """Field that serializes to a string of numbers and deserializes
    to a list of numbers.
    """

    def _serialize(self, value, attr, obj, **kwargs):
        if value is None:
            return ""
        return "".join(str(d) for d in value)

    def _deserialize(self, value, attr, data, **kwargs):
        try:
            return [int(c) for c in value]
        except ValueError as error:
            raise ValidationError("Pin codes must contain only digits.") from error
```

(continues on next page)

(continued from previous page)

```
class UserSchema(Schema):
    name = fields.String()
    email = fields.String()
    created_at = fields.DateTime()
    pin_code = PinCode()
```

5.4.2 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
```

5.4.3 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())
```

5.4.4 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):
        return float(value)

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

5.4.5 Using context

A field may need information about its environment to know how to (de)serialize a value.

You can use the experimental `Context` class to set and retrieve context.

Let's say your `UserSchema` needs to output whether or not a `User` is the author of a `Blog` or whether a certain word appears in a `Blog`'s title.

```
import typing
from dataclasses import dataclass

from marshmallow import Schema, fields
from marshmallow.experimental.context import Context

@dataclass
class User:
    name: str

@dataclass
class Blog:
    title: str
    author: User

class ContextDict(typing.TypedDict):
    blog: Blog

class UserSchema(Schema):
    name = fields.String()

    is_author = fields.Function(
        lambda user: user == Context[ContextDict].get()["blog"].author
    )
    likes_bikes = fields.Method("writes_about_bikes")

    def writes_about_bikes(self, user: User) -> bool:
        return "bicycle" in Context[ContextDict].get()["blog"].title.lower()
```

Note

You can use `Context.get` within custom fields, pre-/post-processing methods, and validators.

When (de)serializing, set the context by using `Context` as a context manager.

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

schema = UserSchema()
with Context({"blog": blog}):
    result = schema.dump(user)
```

(continues on next page)

(continued from previous page)

```
print(result["is_author"]) # => True
print(result["likes_bikes"]) # => True
```

5.4.6 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."}
    )
```

5.4.7 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.

5.5 Extending schemas

The guides below demonstrate how to extend schemas in various ways.

5.5.1 Pre-processing and post-processing

Decorator API

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, post_load
```

(continues on next page)

(continued from previous page)

```

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

    @post_load
    def slugify_name(self, in_data, **kwargs):
        in_data["slug"] = in_data["slug"].lower().strip().replace(" ", "-")
        return in_data

schema = UserSchema()
result = 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’s *dump/load* method at runtime.

In cases where your pre- and post-processing methods needs to handle the input collection when processing multiple objects, add `pass_many=True` to the method decorators.

Your method will then receive the input data (which may be a single datum or a collection, depending on the dump/load call).

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, **kwargs):
        key = self.get_envelope_key(many)
        return data[key]

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

```

(continues on next page)

(continued from previous page)

```

@post_load
def make_object(self, data, **kwargs):
    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)
# {'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)
# {'users': [{'email': 'keith@stones.org', 'name': 'Keith'},
#            {'email': 'charlie@stones.org', 'name': 'Charlie'}}

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

```

Field-level pre- and post-processing

For field-level processing, pass `pre_load` and `post_load` callables directly to individual fields. This is useful for simple, field-specific transformations that don't need access to the full schema data.

Each callable receives the field value and returns a transformed value. You can pass a single callable or a list of callables, which are applied in order.

```

from marshmallow import Schema, fields

class UserSchema(Schema):
    name = fields.Str(pre_load=str.strip)
    birthday = fields.Date(post_load=lambda value: value.year)

schema = UserSchema()
result = schema.load({"name": " Steve ", "birthday": "1994-05-12"})
result["name"] # => 'Steve'
result["birthday"] # => 1994

```

`pre_load` callables run before the field's deserialization (and before `allow_none` is checked), while `post_load` callables run after validation and deserialization.

Like validators, `pre_load` and `post_load` callables may raise a `ValidationError`, which will be stored under the

field's key in the errors dictionary.

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, **kwargs):
        if "data" not in data:
            raise ValidationError('Input data must have a "data" key.')
        return data["data"]

sch = BandSchema()
try:
    sch.load({"name": "The Band"})
except ValidationError as err:
    err.messages
# {'_schema': ['Input data must have a "data" key.']}
```

If you want to store an 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, **kwargs):
        if "data" not in data:
            raise ValidationError(
                'Input data must have a "data" key.', "_preprocessing"
            )
        return data["data"]

sch = BandSchema()
try:
    sch.load({"name": "The Band"})
except ValidationError as err:
    err.messages
# {'_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. Field-level `pre_load` callables
4. Field deserialization (`_deserialize`)
5. Field-level `validate` callables and `@validates` methods
6. Field-level `post_load` callables
7. `@validates_schema` methods (schema validators)
8. `@post_load(pass_many=False)` methods
9. `@post_load(pass_many=True)` methods

The pipeline for serialization is similar, except that the `pass_many=True` processors are invoked *after* the `pass_many=False` processors and there are no validators.

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.Raw()

    @pre_load
    def preprocess(self, data, **kwargs):
        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.Raw()

@pre_load
def step1(self, data, **kwargs): ...

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

```

5.5.2 Schema-level validation

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

```

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, **kwargs):
        if data["field_b"] >= data["field_a"]:
            raise ValidationError("field_a must be greater than field_b")

schema = NumberSchema()
try:
    schema.load({"field_a": 1, "field_b": 2})
except ValidationError as err:
    err.messages["_schema"]
# => [{"field_a must be greater than field_b"}]

```

Storing errors on specific fields

It is possible to report errors on fields and subfields using a `dict`.

When multiple schema-level validator return errors, the error structures are merged together in the `ValidationError` raised at the end of the validation.

```

from marshmallow import Schema, fields, validates_schema, ValidationError

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

    @validates_schema
    def validate_lower_bound(self, data, **kwargs):

```

(continues on next page)

(continued from previous page)

```

errors = {}
if data["field_b"] <= data["field_a"]:
    errors["field_b"] = ["field_b must be greater than field_a"]
if data["field_c"] <= data["field_a"]:
    errors["field_c"] = ["field_c must be greater than field_a"]
if errors:
    raise ValidationError(errors)

@validates_schema
def validate_upper_bound(self, data, **kwargs):
    errors = {}
    if data["field_b"] >= data["field_d"]:
        errors["field_b"] = ["field_b must be lower than field_d"]
    if data["field_c"] >= data["field_d"]:
        errors["field_c"] = ["field_c must be lower than field_d"]
    if errors:
        raise ValidationError(errors)

schema = NumberSchema()
try:
    schema.load({"field_a": 3, "field_b": 2, "field_c": 1, "field_d": 0})
except ValidationError as err:
    err.messages
# => {
#   'field_b': [
#     'field_b must be greater than field_a',
#     'field_b must be lower than field_d'
#   ],
#   'field_c': [
#     'field_c must be greater than field_a',
#     'field_c must be lower than field_d'
#   ]
# }

```

5.5.3 Using original input data

If you want to use the original, unprocessed input, you can add `pass_original=True` to `post_load` or `validates_schema`.

```

from marshmallow import Schema, fields, post_load, ValidationError

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

    @post_load(pass_original=True)
    def add_baz_to_bar(self, data, original_data, **kwargs):
        baz = original_data.get("baz")
        if baz:
            data["bar"] = data["bar"] + baz

```

(continues on next page)

(continued from previous page)

```
    return data

schema = MySchema()
schema.load({"foo": 1, "bar": 2, "baz": 3})
# {'foo': 1, 'bar': 5}
```

➔ See also

The default behavior for unspecified fields can be controlled with the `unknown` option, see *Handling Unknown Fields* for more information.

5.5.4 Overriding how attributes are accessed

By default, marshmallow uses `utils.get_value` 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, obj, key, default):
        return obj.get(key, default)
```

5.5.5 Custom error handling

By default, `Schema.load` will raise a `ValidationError` if passed invalid data.

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 input data to be deserialized.

```
import logging
from marshmallow import Schema, fields

class AppError(Exception):
    pass

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

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

(continues on next page)

(continued from previous page)

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

5.5.6 Custom class Meta options

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

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

Example: Enveloping, revisited

Let's build upon the *previous enveloping implementation* 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': {
    '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, **kwargs):
        SchemaOpts.__init__(self, meta, **kwargs)
        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, **kwargs):
        key = self.opts.plural_name if many else self.opts.name
        return data[key]

    @post_dump(pass_many=True)
```

(continues on next page)

(continued from previous page)

```
def wrap_with_envelope(self, data, many, **kwargs):
    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 # {"user": {"name": "Keith", "email": "keith@stones.com"}}
```

5.5.7 Custom error messages

To customize the schema-level error messages that *load* and *loads* use when raising a *ValidationError*, override the *error_messages* class variable:

```
class MySchema(Schema):
    error_messages = {
        "unknown": "Custom unknown field error message.",
        "type": "Custom invalid type error message.",
    }
```

Field-level error message defaults can be set on *Field.default_error_messages*.

```
from marshmallow import Schema, fields

fields.Field.default_error_messages["required"] = "You missed something!"

class ArtistSchema(Schema):
    name = fields.Str(required=True)
    label = fields.Str(required=True, error_messages={"required": "Label missing."})

print(ArtistSchema().validate({}))
# {'label': ['Label missing.'], 'name': ['You missed something!']}
```

5.6 Examples

The below examples demonstrate how to use marshmallow in various contexts. To run each example, you will need to have *uv* installed. The examples use [PEP 723 inline metadata](#) to declare the dependencies of each script. *uv* will install the dependencies automatically when running these scripts.

5.6.1 Validating package.json

marshmallow can be used to validate configuration according to a schema. Below is a schema that could be used to validate package.json files. This example demonstrates the following features:

- Validation and deserialization using *Schema.load*
- *Custom fields*
- Specifying deserialization keys using *data_key*
- Including unknown keys using *unknown = INCLUDE*

```
# /// script
# requires-python = ">=3.10"
# dependencies = [
#     "marshmallow",
#     "packaging>=17.0",
# ]
# ///
import json
import sys
from pprint import pprint

from packaging import version

from marshmallow import INCLUDE, Schema, ValidationError, fields

class Version(fields.Field[version.Version]):
    """Version field that deserializes to a Version object."""

    def _deserialize(self, value, *args, **kwargs):
        try:
            return version.Version(value)
        except version.InvalidVersion as e:
            raise ValidationError("Not a valid version.") from e

    def _serialize(self, value, *args, **kwargs):
        return str(value)

class PackageSchema(Schema):
    name = fields.Str(required=True)
    version = Version(required=True)
    description = fields.Str(required=True)
    main = fields.Str(required=False)
    homepage = fields.URL(required=False)
    scripts = fields.Dict(keys=fields.Str(), values=fields.Str())
    license = fields.Str(required=True)
    dependencies = fields.Dict(keys=fields.Str(), values=fields.Str(), required=False)
    dev_dependencies = fields.Dict(
        keys=fields.Str(),
        values=fields.Str(),
        required=False,
        data_key="devDependencies",
    )
```

(continues on next page)

(continued from previous page)

```

)

class Meta:
    # Include unknown fields in the deserialized output
    unknown = INCLUDE

if __name__ == "__main__":
    pkg = json.load(sys.stdin)
    try:
        pprint(PackageSchema().load(pkg))
    except ValidationError as error:
        print("ERROR: package.json is invalid")
        pprint(error.messages)
    sys.exit(1)

```

Given the following package.json file...

```

{
  "name": "dunderscore",
  "version": "1.2.3",
  "description": "The Pythonic JavaScript toolkit",
  "devDependencies": {
    "pest": "^23.4.1"
  },
  "main": "index.js",
  "scripts": {
    "test": "pest"
  },
  "license": "MIT"
}

```

We can validate it using the above script.

```

$ uv run examples/package_json_example.py < examples/package.json
{'description': 'The Pythonic JavaScript toolkit',
 'dev_dependencies': {'pest': '^23.4.1'},
 'license': 'MIT',
 'main': 'index.js',
 'name': 'dunderscore',
 'scripts': {'test': 'pest'},
 'version': <Version('1.2.3')>}

```

Notice that our custom field deserialized the version string to a `Version` object.

But if we pass an invalid package.json file...

```

{
  "name": "dunderscore",
  "version": "INVALID",
  "homepage": "INVALID",
  "description": "The Pythonic JavaScript toolkit",
  "license": "MIT"
}

```

(continues on next page)

(continued from previous page)

}

We see the corresponding error messages.

```
$ uv run examples/package_json_example.py < examples/invalid_package.json
ERROR: package.json is invalid
{'homepage': ['Not a valid URL.'], 'version': ['Not a valid version.']}
```

5.6.2 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:

- 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.

```
# /// script
# requires-python = ">=3.10"
# dependencies = [
#     "flask",
#     "flask-sqlalchemy>=3.1.1",
#     "marshmallow",
#     "sqlalchemy>2.0",
# ]
# ///
from __future__ import annotations

import datetime

from flask import Flask, request
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.exc import NoResultFound
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship

from marshmallow import Schema, ValidationError, fields, pre_load

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///tmp/quotes.db"

class Base(DeclarativeBase):
    pass

db = SQLAlchemy(app, model_class=Base)

##### MODELS #####
```

(continues on next page)

```

class Author(db.Model): # type: ignore[name-defined]
    id: Mapped[int] = mapped_column(primary_key=True)
    first: Mapped[str]
    last: Mapped[str]

class Quote(db.Model): # type: ignore[name-defined]
    id: Mapped[int] = mapped_column(primary_key=True)
    content: Mapped[str] = mapped_column(nullable=False)
    author_id: Mapped[int] = mapped_column(db.ForeignKey(Author.id))
    author: Mapped[Author] = relationship(backref=db.backref("quotes", lazy="dynamic"))
    posted_at: Mapped[datetime.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 f"{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, **kwargs):
        author_name = data.get("author")
        if author_name:
            first, last = author_name.split(" ")
            author_dict = {"first": first, "last": last}
        else:
            author_dict = {}
        data["author"] = author_dict

```

(continues on next page)

(continued from previous page)

```
    return data

author_schema = AuthorSchema()
authors_schema = AuthorSchema(many=True)
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 {"authors": result}

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

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

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

@app.route("/quotes/", methods=["POST"])
def new_quote():
    json_data = request.get_json()
    if not json_data:
        return {"message": "No input data provided"}, 400
```

(continues on next page)

(continued from previous page)

```

# Validate and deserialize input
try:
    data = quote_schema.load(json_data)
except ValidationError as err:
    return err.messages, 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.now(datetime.UTC),
)
db.session.add(quote)
db.session.commit()
result = quote_schema.dump(Quote.query.get(quote.id))
return {"message": "Created new quote.", "quote": result}

if __name__ == "__main__":
    with app.app_context():
        db.create_all()
    app.run(debug=True, port=5000)

```

Using The API

Run the app.

```
$ uv run examples/flask_example.py
```

We'll use the [httpie cli](#) to send requests Install it with uv.

```
$ uv tool install httpie
```

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
    }
  ]
}
```

5.6.3 Inflection (camel-cased keys)

HTTP APIs will often use camel-cased keys for their input and output representations. This example shows how you can use the `Schema.on_bind_field` hook to automatically inflect keys.

```
# /// script
# requires-python = ">=3.9"
# dependencies = [
#     "marshmallow",
# ]
# ///
from marshmallow import Schema, fields
```

(continues on next page)

```
def camelcase(s):
    parts = iter(s.split("_"))
    return next(parts) + "".join(i.title() for i in parts)

class CamelCaseSchema(Schema):
    """Schema that uses camel-case for its external representation
    and snake-case for its internal representation.
    """

    def on_bind_field(self, field_name, field_obj):
        field_obj.data_key = camelcase(field_obj.data_key or field_name)

# -----

class UserSchema(CamelCaseSchema):
    first_name = fields.Str(required=True)
    last_name = fields.Str(required=True)

schema = UserSchema()
loaded = schema.load({"firstName": "David", "lastName": "Bowie"})
print("Loaded data:")
print(loaded)
dumped = schema.dump(loaded)
print("Dumped data:")
print(dumped)
```

To run the example:

```
$ uv run examples/inflection_example.py
Loaded data:
{'first_name': 'David', 'last_name': 'Bowie'}
Dumped data:
{'firstName': 'David', 'lastName': 'Bowie'}
```

API REFERENCE

6.1 Top-level API

Classes:

<i>Schema</i>	Base schema class with which to define schemas.
<i>SchemaOpts</i>	Defines defaults for <i>marshmallow.Schema.Meta</i> .

Exceptions:

<i>ValidationError</i>	Raised when validation fails on a field or schema.
------------------------	--

Functions:

<i>post_dump</i>	Register a method to invoke after serializing an object.
<i>post_load</i>	Register a method to invoke after deserializing an object.
<i>pre_dump</i>	Register a method to invoke before serializing an object.
<i>pre_load</i>	Register a method to invoke before deserializing an object.
<i>validates</i>	Register a validator method for field(s).
<i>validates_schema</i>	Register a schema-level validator.

class `marshmallow.Schema`(* (*Keyword-only parameters separator (PEP 3102)*), *only=None, exclude=()*,
many=None, load_only=(), dump_only=(), partial=None, unknown=None)

Base schema class with which to define schemas.

Example usage:

```
import datetime as dt
from dataclasses import dataclass

from marshmallow import Schema, fields

@dataclass
class Album:
    title: str
    release_date: dt.date
```

(continues on next page)

(continued from previous page)

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

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

Parameters

- **only** (`types.StrSequenceOrSet` / `None`) – Whitelist of the declared fields to select when instantiating the Schema. If `None`, all fields are used. Nested fields can be represented with dot delimiters.
- **exclude** (`types.StrSequenceOrSet`) – Blacklist of the declared 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.
- **many** (`bool` / `None`) – Should be set to `True` if `obj` is a collection so that the object will be serialized to a list.
- **load_only** (`types.StrSequenceOrSet`) – Fields to skip during serialization (write-only fields)
- **dump_only** (`types.StrSequenceOrSet`) – Fields to skip during deserialization (read-only fields)
- **partial** (`bool` / `types.StrSequenceOrSet` / `None`) – Whether to ignore missing fields and not require any fields declared. Propagates down to Nested fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** (`types.UnknownOption` / `None`) – Whether to exclude, include, or raise an error for unknown fields in the data. Use `EXCLUDE`, `INCLUDE` or `RAISE`.

Changed in version 3.0.0: Remove `prefix` parameter.

Changed in version 4.0.0: Remove `context` parameter.

class Meta

Options object for a Schema.

Example usage:

```
from marshmallow import Schema

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

i A note on type checking

Type checkers will only check the attributes of the *Meta* class if you explicitly subclass *marshmallow.Schema.Meta*.

```
from marshmallow import Schema

class MySchema(Schema):
    # Not checked by type checkers
    class Meta:
        additional = True

class MySchema2(Schema):
    # Type checkers will check attributes
    class Meta(Schema.Meta):
        additional = True # Incompatible types in assignment
```

Removed in version 3.0.0b7: Remove `strict`.

Added in version 3.0.0b12: Add `unknown`.

Changed in version 3.0.0b17: Rename `dateformat` to `datetimeformat`.

Added in version 3.9.0: Add `timeformat`.

Changed in version 3.26.0: Deprecate `ordered`. Field order is preserved by default.

Removed in version 4.0.0: Remove `ordered`.

additional: `ClassVar[tuple[str, ...] | list[str]]`

Fields to include in addition to the explicitly declared fields. *additional* and *fields* are mutually-exclusive options.

dateformat: `ClassVar[str]`

Default format for *Date* fields.

datetimeformat: `ClassVar[str]`

Default format for *DateTime* fields.

dump_only: `ClassVar[tuple[str, ...] | list[str]]`

Fields to exclude from deserialized results

exclude: `ClassVar[tuple[str, ...] | list[str]]`

Fields to exclude in the serialized result. Nested fields can be represented with dot delimiters.

fields: `ClassVar[tuple[str, ...] | list[str]]`

Fields to include in the (de)serialized result

include: `ClassVar[dict[str, Field]]`

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.

index_errors: `ClassVar[bool]`

If `True`, errors dictionaries will include the index of invalid items in a collection.

load_only: `ClassVar[tuple[str, ...] | list[str]]`

Fields to exclude from serialized results

many: `ClassVar[bool]`

Whether data should be (de)serialized as a collection by default.

register: `ClassVar[bool]`

Whether to register the *Schema* with marshmallow's internal class registry. Must be `True` if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to `False` when memory usage is critical. Defaults to `True`.

render_module: `Any`

Module to use for *loads* and *dumps*. Defaults to `json` from the standard library.

timeformat: `ClassVar[str]`

Default format for *Time* fields.

unknown: `ClassVar[types.UnknownOption]`

Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.

dict_class

alias of `dict`

dump(*obj*, *, *many=None*)

Serialize an object to native Python data types according to this *Schema*'s fields.

Parameters

- **obj** (*Any*) – The object to serialize.
- **many** (*bool* | *None*) – Whether to serialize *obj* as a collection. If `None`, the value for `self.many` is used.

Returns

Serialized data

Changed in version 3.0.0b7: This method returns the serialized data rather than a (`data`, `errors`) tuple. A *ValidationError* is raised if *obj* is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj*, **args*, *many=None*, ***kwargs*)

Same as `dump()`, except return a JSON-encoded string.

Parameters

- **obj** (*Any*) – The object to serialize.
- **many** (*bool* | *None*) – Whether to serialize *obj* as a collection. If `None`, the value for `self.many` is used.

Returns

A `json` string

Changed in version 3.0.0b7: This method returns the serialized data rather than a (`data`, `errors`) tuple. A *ValidationError* is raised if *obj* is invalid.

error_messages: `dict[str, str] = {}`

Overrides for default schema-level error messages

fields: `dict[str, Field]`

Dictionary mapping field_names -> Field objects

classmethod `from_dict(fields, *, name='GeneratedSchema')`

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"})) # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in Nested fields.

Parameters

- **fields** (`dict[str, Field]`) – Dictionary mapping field names to field instances.
- **name** (`str`) – Optional name for the class, which will appear in the repr for the class.

Return type

`type[Schema]`

Added in version 3.0.0.

get_attribute(`obj, attr, default`)

Defines how to pull values from an object to serialize.

Changed in version 3.0.0a1: Changed position of obj and attr.

Parameters

- **obj** (`Any`)
- **attr** (`str`)
- **default** (`Any`)

handle_error(`error, data, *, many, **kwargs`)

Custom error handler function for the schema.

Parameters

- **error** (`ValidationError`) – The *ValidationError* raised during (de)serialization.
- **data** (`Any`) – The original input data.
- **many** (`bool`) – Value of many on dump or load.
- **partial** – Value of partial on load.

Changed in version 3.0.0rc9: Receives many and partial (on deserialization) as keyword arguments.

load(`data, *, many=None, partial=None, unknown=None`)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** (`Mapping[str, Any] | Sequence[Mapping[str, Any]]`) – The data to deserialize.
- **many** (`bool | None`) – Whether to deserialize data as a collection. If `None`, the value for `self.many` is used.

- **partial** (*bool* | *Sequence[str]* | *AbstractSet[str]* | *None*) – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** (*Literal['exclude', 'include', 'raise']* | *None*) – Whether to exclude, include, or raise an error for unknown fields in the data. Use `EXCLUDE`, `INCLUDE` or `RAISE`. If `None`, the value for `self.unknown` is used.

Returns

Deserialized data

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` tuple. A `ValidationError` is raised if invalid data are passed.

loads(*s*, / (Positional-only parameter separator (PEP 570)), *, *many=None*, *partial=None*, *unknown=None*, ***kwargs*)

Same as `load()`, except it uses `marshmallow.Schema.Meta.render_module` to deserialize the passed string before passing data to `load()`.

Parameters

- **s** (*str* | *bytes* | *bytearray*) – A string of the data to deserialize.
- **many** (*bool* | *None*) – Whether to deserialize `obj` as a collection. If `None`, the value for `self.many` is used.
- **partial** (*bool* | *Sequence[str]* | *AbstractSet[str]* | *None*) – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** (*Literal['exclude', 'include', 'raise']* | *None*) – Whether to exclude, include, or raise an error for unknown fields in the data. Use `EXCLUDE`, `INCLUDE` or `RAISE`. If `None`, the value for `self.unknown` is used.

Returns

Deserialized data

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` tuple. A `ValidationError` is raised if invalid data are passed.

Changed in version 4.0.0: Rename `json_module` parameter to `s`.

on_bind_field(*field_name*, *field_obj*)

Hook to modify a field when it is bound to the `Schema`.

No-op by default.

Parameters

- **field_name** (*str*)
- **field_obj** (`Field`)

Return type

`None`

set_class

alias of `OrderedSet`

validate(*data*, *, *many=None*, *partial=None*)

Validate data against the schema, returning a dictionary of validation errors.

Parameters

- **data** (*Mapping*[*str*, *Any*] | *Sequence*[*Mapping*[*str*, *Any*]]) – The data to validate.
- **many** (*bool* | *None*) – Whether to validate data as a collection. If *None*, the value for `self.many` is used.
- **partial** (*bool* | *Sequence*[*str*] | *AbstractSet*[*str*] | *None*) – Whether to ignore missing fields and not require any fields declared. Propagates down to *Nested* fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns

A dictionary of validation errors.

Return type

`dict[str, list[str]]`

class `marshmallow.SchemaOpts`(*meta*)

Defines defaults for `marshmallow.Schema.Meta`.

Parameters

meta (*type*)

exception `marshmallow.ValidationError`(*message*, *field_name='_schema'*, *data=None*, *valid_data=None*, ***kwargs*)

Raised when validation fails on a field or schema.

Validators and custom fields should raise this exception.

Parameters

- **message** (*str* | *list* | *dict*) – An error message, list of error messages, or dict of error messages. If a dict, the keys are subitems and the values are error messages.
- **field_name** (*str*) – Field name to store the error on.
- **data** (*Mapping*[*str*, *Any*] | *Iterable*[*Mapping*[*str*, *Any*]] | *None*) – Raw input data.
- **valid_data** (*list*[*Any*] | *dict*[*str*, *Any*] | *None*) – Valid (de)serialized data.

`marshmallow.post_dump`(*fn=None*, *, *pass_collection=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 it receives a single object at a time, transparently handling the `many` argument passed to the `Schema`'s `dump()` call. If `pass_collection=True`, the raw data (which may be a collection) is passed.

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

Changed in version 3.0.0: `many` is always passed as a keyword arguments to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection` and `pass_original` are keyword-only arguments.

Parameters

- **fn** (*Callable*[[...], *Any*] | *None*)
- **pass_collection** (*bool*)
- **pass_original** (*bool*)

Return type*Callable*[[...], *Any*]**marshmallow.post_load**(*fn=None*, *, *pass_collection=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 it receives a single object at a time, transparently handling the many argument passed to the *Schema*'s *load()* call. If *pass_collection=True*, the raw data (which may be a collection) is passed.

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

Changed in version 3.0.0: *partial* and *many* are always passed as keyword arguments to the decorated method.

Changed in version 4.0.0: *pass_many* is renamed to *pass_collection*.

Changed in version 4.0.0: *pass_collection* and *pass_original* are keyword-only arguments.

Changed in version 4.0.0: *unknown* is passed as a keyword argument to the decorated method.

Parameters

- **fn** (*Callable*[[...], *Any*] | *None*)
- **pass_collection** (*bool*)
- **pass_original** (*bool*)

Return type*Callable*[[...], *Any*]**marshmallow.pre_dump**(*fn=None*, *, *pass_collection=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 it receives a single object at a time, transparently handling the many argument passed to the *Schema*'s *dump()* call. If *pass_collection=True*, the raw data (which may be a collection) is passed.

Changed in version 3.0.0: *many* is always passed as a keyword arguments to the decorated method.

Changed in version 4.0.0: *pass_many* is renamed to *pass_collection*.

Changed in version 4.0.0: *pass_collection* is a keyword-only argument.

Parameters

- **fn** (*Callable*[[...], *Any*] | *None*)
- **pass_collection** (*bool*)

Return type*Callable*[[...], *Any*]**marshmallow.pre_load**(*fn=None*, *, *pass_collection=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 it receives a single object at a time, transparently handling the many argument passed to the *Schema*'s *load()* call. If *pass_collection=True*, the raw data (which may be a collection) is passed.

Changed in version 3.0.0: `partial` and `many` are always passed as keyword arguments to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection` is a keyword-only argument.

Changed in version 4.0.0: `unknown` is passed as a keyword argument to the decorated method.

Parameters

- `fn` (*Callable*[[...], Any] | None)
- `pass_collection` (*bool*)

Return type

Callable[[...], Any]

`marshmallow.validates(*field_names)`

Register a validator method for field(s).

Parameters

`field_names` (*str*) – Names of the fields that the method validates.

Return type

Callable[[...], Any]

Changed in version 4.0.0: Accepts multiple field names as positional arguments.

Changed in version 4.0.0: Decorated methods receive `data_key` as a keyword argument.

`marshmallow.validates_schema(fn=None, *, pass_collection=False, pass_original=False, skip_on_field_errors=True)`

Register a schema-level validator.

By default it receives a single object at a time, transparently handling the `many` argument passed to the `Schema`'s `validate()` call. If `pass_collection=True`, the raw data (which may be a collection) 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.

Changed in version 3.0.0b1: `skip_on_field_errors` defaults to `True`.

Changed in version 3.0.0: `partial` and `many` are always passed as keyword arguments to the decorated method.

Changed in version 4.0.0: `unknown` is passed as a keyword argument to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection`, `pass_original`, and `skip_on_field_errors` are keyword-only arguments.

Parameters

- `fn` (*Callable*[[...], Any] | None)
- `pass_collection` (*bool*)
- `pass_original` (*bool*)
- `skip_on_field_errors` (*bool*)

Return type

Callable[[...], Any]

marshmallow.missing

Singleton value that indicates that a field's value is missing from input dict passed to `Schema.load`. If the field's value is not required, its default value is used.

6.1.1 Constants for unknown**➔ See also**

Handling unknown fields

marshmallow.EXCLUDE

Indicates that fields that are not explicitly declared on a schema should be excluded from the deserialized result.

marshmallow.INCLUDE

Indicates that fields that are not explicitly declared on a schema should be included from the deserialized result.

marshmallow.RAISE

Indicates that fields that are not explicitly declared on a schema should result in an error.

6.2 Schema

class `marshmallow.schema.Schema`(*, *only=None, exclude=(), many=None, load_only=(), dump_only=(), partial=None, unknown=None*)

Base schema class with which to define schemas.

Example usage:

```
import datetime as dt
from dataclasses import dataclass

from marshmallow import Schema, fields

@dataclass
class Album:
    title: str
    release_date: dt.date

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

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

Parameters

- **only** (`types.StrSequenceOrSet` / `None`) – Whitelist of the declared fields to select when instantiating the Schema. If `None`, all fields are used. Nested fields can be represented with dot delimiters.
- **exclude** (`types.StrSequenceOrSet`) – Blacklist of the declared 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.
- **many** (`bool` / `None`) – Should be set to `True` if `obj` is a collection so that the object will be serialized to a list.
- **load_only** (`types.StrSequenceOrSet`) – Fields to skip during serialization (write-only fields)
- **dump_only** (`types.StrSequenceOrSet`) – Fields to skip during deserialization (read-only fields)
- **partial** (`bool` / `types.StrSequenceOrSet` / `None`) – Whether to ignore missing fields and not require any fields declared. Propagates down to Nested fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** (`types.UnknownOption` / `None`) – Whether to exclude, include, or raise an error for unknown fields in the data. Use `EXCLUDE`, `INCLUDE` or `RAISE`.

Changed in version 3.0.0: Remove `prefix` parameter.

Changed in version 4.0.0: Remove `context` parameter.

Classes:

<code>Meta</code>	Options object for a Schema.
<code>dict_class</code>	<code>dict</code> type to return when serializing.
<code>set_class</code>	

Methods:

<code>dump</code>	Serialize an object to native Python data types according to this Schema's fields.
<code>dumps</code>	Same as <code>dump()</code> , except return a JSON-encoded string.
<code>from_dict</code>	Generate a <code>Schema</code> class given a dictionary of fields.
<code>get_attribute</code>	Defines how to pull values from an object to serialize.
<code>handle_error</code>	Custom error handler function for the schema.
<code>load</code>	Deserialize a data structure to an object defined by this Schema's fields.
<code>loads</code>	Same as <code>load()</code> , except it uses <code>marshmallow.Schema.Meta.render_module</code> to deserialize the passed string before passing data to <code>load()</code> .
<code>on_bind_field</code>	Hook to modify a field when it is bound to the <code>Schema</code> .
<code>validate</code>	Validate data against the schema, returning a dictionary of validation errors.

Attributes:

<code>error_messages</code>	Overrides for default schema-level error messages
<code>fields</code>	Dictionary mapping field_names -> Field objects

class Meta

Options object for a Schema.

Example usage:

```
from marshmallow import Schema

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

i A note on type checking

Type checkers will only check the attributes of the *Meta* class if you explicitly subclass *marshmallow.Schema.Meta*.

```
from marshmallow import Schema

class MySchema(Schema):
    # Not checked by type checkers
    class Meta:
        additional = True

class MySchema2(Schema):
    # Type checkers will check attributes
    class Meta(Schema.Meta):
        additional = True # Incompatible types in assignment
```

Removed in version 3.0.0b7: Remove `strict`.

Added in version 3.0.0b12: Add `unknown`.

Changed in version 3.0.0b17: Rename `dateformat` to `datetimeformat`.

Added in version 3.9.0: Add `timeformat`.

Changed in version 3.26.0: Deprecate `ordered`. Field order is preserved by default.

Removed in version 4.0.0: Remove `ordered`.

Attributes:

<code>additional</code>	Fields to include in addition to the explicitly declared fields.
<code>dateformat</code>	Default format for <i>Date</i> fields.
<code>datetimeformat</code>	Default format for <i>DateTime</i> fields.

continues on next page

Table 7 – continued from previous page

<i>dump_only</i>	Fields to exclude from deserialized results
<i>exclude</i>	Fields to exclude in the serialized result.
<i>fields</i>	Fields to include in the (de)serialized result
<i>include</i>	Dictionary of additional fields to include in the schema.
<i>index_errors</i>	If <code>True</code> , errors dictionaries will include the index of invalid items in a collection.
<i>load_only</i>	Fields to exclude from serialized results
<i>many</i>	Whether data should be (de)serialized as a collection by default.
<i>register</i>	Whether to register the <i>Schema</i> with marshmallow's internal class registry.
<i>render_module</i>	Module to use for <i>loads</i> and <i>dumps</i> .
<i>timeformat</i>	Default format for <i>Time</i> fields.
<i>unknown</i>	Whether to exclude, include, or raise an error for unknown fields in the data.

additional: `ClassVar[tuple[str, ...] | list[str]]`

Fields to include in addition to the explicitly declared fields. *additional* and *fields* are mutually-exclusive options.

dateformat: `ClassVar[str]`

Default format for *Date* fields.

datetimeformat: `ClassVar[str]`

Default format for *DateTime* fields.

dump_only: `ClassVar[tuple[str, ...] | list[str]]`

Fields to exclude from deserialized results

exclude: `ClassVar[tuple[str, ...] | list[str]]`

Fields to exclude in the serialized result. Nested fields can be represented with dot delimiters.

fields: `ClassVar[tuple[str, ...] | list[str]]`

Fields to include in the (de)serialized result

include: `ClassVar[dict[str, Field]]`

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.

index_errors: `ClassVar[bool]`

If `True`, errors dictionaries will include the index of invalid items in a collection.

load_only: `ClassVar[tuple[str, ...] | list[str]]`

Fields to exclude from serialized results

many: `ClassVar[bool]`

Whether data should be (de)serialized as a collection by default.

register: `ClassVar[bool]`

Whether to register the *Schema* with marshmallow's internal class registry. Must be `True` if you intend to refer to this *Schema* by class name in Nested fields. Only set this to `False` when memory usage is critical. Defaults to `True`.

render_module: *Any*

Module to use for *loads* and *dumps*. Defaults to *json* from the standard library.

timeformat: *ClassVar[str]*

Default format for *Time* fields.

unknown: *ClassVar[types.UnknownOption]*

Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.

dict_class

dict type to return when serializing.

alias of *dict*

dump(*obj*, *, *many=None*)

Serialize an object to native Python data types according to this Schema's fields.

Parameters

- **obj** (*Any*) – The object to serialize.
- **many** (*bool* | *None*) – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

Serialized data

Changed in version 3.0.0b7: This method returns the serialized data rather than a (*data*, *errors*) tuple. A *ValidationError* is raised if *obj* is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

dumps(*obj*, **args*, *many=None*, ***kwargs*)

Same as *dump()*, except return a JSON-encoded string.

Parameters

- **obj** (*Any*) – The object to serialize.
- **many** (*bool* | *None*) – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns

A *json* string

Changed in version 3.0.0b7: This method returns the serialized data rather than a (*data*, *errors*) tuple. A *ValidationError* is raised if *obj* is invalid.

error_messages: *dict[str, str] = {}*

Overrides for default schema-level error messages

fields: *dict[str, Field]*

Dictionary mapping *field_names* -> *Field* objects

classmethod from_dict(*fields*, *, *name='GeneratedSchema'*)

Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"})) # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in Nested fields.

Parameters

- **fields** (*dict*[*str*, *Field*]) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the repr for the class.

Return type

type[*Schema*]

Added in version 3.0.0.

get_attribute(*obj*, *attr*, *default*)

Defines how to pull values from an object to serialize.

Changed in version 3.0.0a1: Changed position of *obj* and *attr*.

Parameters

- **obj** (*Any*)
- **attr** (*str*)
- **default** (*Any*)

handle_error(*error*, *data*, *, *many*, ***kwargs*)

Custom error handler function for the schema.

Parameters

- **error** (*ValidationError*) – The *ValidationError* raised during (de)serialization.
- **data** (*Any*) – The original input data.
- **many** (*bool*) – Value of *many* on dump or load.
- **partial** – Value of *partial* on load.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

load(*data*, *, *many=None*, *partial=None*, *unknown=None*)

Deserialize a data structure to an object defined by this Schema's fields.

Parameters

- **data** (*Mapping*[*str*, *Any*] | *Sequence*[*Mapping*[*str*, *Any*]]) – The data to deserialize.
- **many** (*bool* | *None*) – Whether to deserialize data as a collection. If *None*, the value for *self.many* is used.
- **partial** (*bool* | *Sequence*[*str*] | *AbstractSet*[*str*] | *None*) – Whether to ignore missing fields and not require any fields declared. Propagates down to Nested fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** (*Literal*['exclude', 'include', 'raise'] | *None*) – Whether to exclude, include, or raise an error for unknown fields in the data. Use EXCLUDE, INCLUDE or RAISE. If *None*, the value for *self.unknown* is used.

Returns

Deserialized data

Changed in version 3.0.0b7: This method returns the deserialized data rather than a (data, errors) tuple. A `ValidationError` is raised if invalid data are passed.

loads(*s*, /, *, *many=None*, *partial=None*, *unknown=None*, ***kwargs*)

Same as `load()`, except it uses `marshmallow.Schema.Meta.render_module` to deserialize the passed string before passing data to `load()`.

Parameters

- **s** (*str* | *bytes* | *bytearray*) – A string of the data to deserialize.
- **many** (*bool* | *None*) – Whether to deserialize obj as a collection. If *None*, the value for `self.many` is used.
- **partial** (*bool* | *Sequence[str]* | *AbstractSet[str]* | *None*) – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** (*Literal['exclude', 'include', 'raise']* | *None*) – Whether to exclude, include, or raise an error for unknown fields in the data. Use `EXCLUDE`, `INCLUDE` or `RAISE`. If *None*, the value for `self.unknown` is used.

Returns

Deserialized data

Changed in version 3.0.0b7: This method returns the deserialized data rather than a (data, errors) tuple. A `ValidationError` is raised if invalid data are passed.

Changed in version 4.0.0: Rename `json_module` parameter to `s`.

on_bind_field(*field_name*, *field_obj*)

Hook to modify a field when it is bound to the `Schema`.

No-op by default.

Parameters

- **field_name** (*str*)
- **field_obj** (`Field`)

Return type

None

set_class

alias of `OrderedSet`

validate(*data*, *, *many=None*, *partial=None*)

Validate data against the schema, returning a dictionary of validation errors.

Parameters

- **data** (*Mapping[str, Any]* | *Sequence[Mapping[str, Any]]*) – The data to validate.
- **many** (*bool* | *None*) – Whether to validate data as a collection. If *None*, the value for `self.many` is used.
- **partial** (*bool* | *Sequence[str]* | *AbstractSet[str]* | *None*) – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

Returns

A dictionary of validation errors.

Return type

`dict[str, list[str]]`

`class marshmallow.schema.SchemaOpts`(*meta*)

Defines defaults for `marshmallow.Schema.Meta`.

Parameters

`meta` (*type*)

6.3 Fields

6.3.1 Base Field Class

```
class marshmallow.fields.Field(*, load_default=<marshmallow.missing>,
                               dump_default=<marshmallow.missing>, data_key=None, attribute=None,
                               validate=None, pre_load=None, post_load=None, required=False,
                               allow_none=None, load_only=False, dump_only=False,
                               error_messages=None, metadata=None)
```

Base field from which all other fields inherit. This class should not be used directly within Schemas.

Parameters

- **dump_default** (*Any*) – 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.
- **load_default** (*Any*) – Default deserialization value for the field if the field is not found in the input data. May be a value or a callable.
- **data_key** (*str* | *None*) – The name of the dict key in the external representation, i.e. the input of `load` and the output of `dump`. If `None`, the key will match the name of the field.
- **attribute** (*str* | *None*) – The name of the key/attribute in the internal representation, i.e. the output of `load` and the input of `dump`. If `None`, the key/attribute will match the name of the field. Note: This should only be used for very specific use cases such as outputting multiple fields for a single attribute, or using keys/attributes that are invalid variable names, unsuitable for field names. In most cases, you should use `data_key` instead.
- **validate** (`types.Validator` | `Iterable[types.Validator]` | *None*) – 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`, a `ValidationError` is raised.
- **pre_load** (`types.PreLoadCallable` | `Iterable[types.PreLoadCallable]` | *None*) – Callable or collection of callables that are applied to the raw input value before deserialization. Each callable receives the value and returns a transformed value.
- **post_load** (`types.PostLoadCallable` | `Iterable[types.PostLoadCallable]` | *None*) – Callable or collection of callables that are applied to the deserialized value after validation. Each callable receives the value and returns a transformed value.
- **required** (*bool*) – Raise a `ValidationError` if the field value is not supplied during deserialization.
- **allow_none** (*bool* | *None*) – Set this to `True` if `None` should be considered a valid value during validation/deserialization. If set to `False` (the default), `None` is considered invalid

input. If `load_default` is explicitly set to `None` and `allow_none` is unset, `allow_none` is implicitly set to `True`.

- **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”.
- **error_messages** (`types.ErrorMessages` / `None`) – Overrides `Field.default_error_messages`.
- **metadata** (`Mapping[str, Any]` / `None`) – Extra information to be stored as field metadata.

Changed in version 3.0.0b8: Add `data_key` parameter for the specifying the key in the input and output data. This parameter replaced both `load_from` and `dump_to`.

Changed in version 3.13.0: Replace `missing` and `default` parameters with `load_default` and `dump_default`.

Changed in version 3.24.0: `Field` should no longer be used as a field within a `Schema`. Use `Raw` or another `Field` subclass instead.

Changed in version 4.0.0: Remove `context` property.

Changed in version 4.3.0: Add `pre_load` and `post_load`.

6.3.2 Field subclasses

Classes:

<code>AwareDateTime</code>	A formatted aware datetime string.
<code>Bool</code>	
<code>Boolean</code>	A boolean field.
<code>Constant</code>	A field that (de)serializes to a preset constant.
<code>Date</code>	ISO8601-formatted date string.
<code>DateTime</code>	A formatted datetime string.
<code>Decimal</code>	A field that (de)serializes to the Python <code>decimal.Decimal</code> type.
<code>Dict</code>	A dict field.
<code>Email</code>	An email field.
<code>Enum</code>	An Enum field (de)serializing enum members by symbol (name) or by value.
<code>Float</code>	A double as an IEEE-754 double precision string.
<code>Function</code>	A field that takes the value returned by a function.
<code>IP</code>	A IP address field.
<code>IPInterface</code>	A IPInterface field.
<code>IPv4</code>	A IPv4 address field.
<code>IPv4Interface</code>	A IPv4 Network Interface field.
<code>IPv6</code>	A IPv6 address field.
<code>IPv6Interface</code>	A IPv6 Network Interface field.
<code>Int</code>	
<code>Integer</code>	An integer field.
<code>List</code>	A list field, composed with another <code>Field</code> class or instance.

continues on next page

Table 8 – continued from previous page

<i>Mapping</i>	Abstract base class for objects with key-value pairs.
<i>Method</i>	A field that takes the value returned by a <i>Schema</i> method.
<i>NaiveDateTime</i>	A formatted naive datetime string.
<i>Nested</i>	Allows you to nest a <i>Schema</i> inside a field.
<i>Number</i>	Abstract base class for number fields.
<i>Pluck</i>	Allows you to replace nested data with one of the data's fields.
<i>Raw</i>	Field that applies no formatting.
<i>Str</i>	
<i>String</i>	A string field.
<i>Time</i>	A formatted time string.
<i>TimeDelta</i>	A field that (de)serializes a <code>datetime.timedelta</code> object to a <code>float</code> .
<i>Tuple</i>	A tuple field, composed of a fixed number of other <i>Field</i> classes or instances
<i>URL</i>	
<i>UUID</i>	A UUID field.
<i>Url</i>	An URL field.

class `marshmallow.fields.AwareDateTime`(*format=None*, *, *default_timezone=None*, ***kwargs*)

A formatted aware datetime string.

Parameters

- **format** (*str* | *None*) – See *DateTime*.
- **default_timezone** (*dt.tzinfo* | *None*) – Used on deserialization. If *None*, naive datetimes are rejected. If not *None*, naive datetimes are set this timezone.
- **kwargs** (*Unpack[_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

Added in version 3.0.0rc9.

`marshmallow.fields.Bool`

alias of *Boolean*

class `marshmallow.fields.Boolean`(*, *truthy=None*, *falsy=None*, ***kwargs*)

A boolean field.

Parameters

- **truthy** (*Iterable* | *None*) – Values that will (de)serialize to *True*. If an empty set, any non-falsy value will deserialize to *True*. If *None*, `marshmallow.fields.Boolean.truthy` will be used.
- **falsy** (*Iterable* | *None*) – Values that will (de)serialize to *False*. If *None*, `marshmallow.fields.Boolean.falsy` will be used.
- **kwargs** (*Unpack[_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

```
falsy = {'0', 'F', 'FALSE', 'False', 'N', 'NO', 'No', 'OFF', 'Off', 'f', 'false',
'n', 'no', 'off', 0}
```

Default falsy values.

```
truthy = {'1', 'ON', 'On', 'T', 'TRUE', 'True', 'Y', 'YES', 'Yes', 'on', 't',
         'true', 'y', 'yes', 1}
```

Default truthy values.

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** (*_ContantT*) – The constant to return for the field attribute.
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

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

ISO8601-formatted date string.

Parameters

- **format** (*str* / *None*) – Either "iso" (for ISO8601) or a date format string. If *None*, defaults to "iso".
- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

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

A formatted datetime string.

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

Parameters

- **format** (*str* / *None*) – Either "rfc" (for RFC822), "iso" (for ISO8601), "timestamp", "timestamp_ms" (for a POSIX timestamp) or a date format string. If *None*, defaults to "iso".
- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

Changed in version 3.0.0rc9: Does not modify timezone information on (de)serialization.

Changed in version 3.19: Add timestamp as a format.

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` | `None`) – How many decimal places to quantize the value. If `None`, does not quantize the value.
- **rounding** (`str` | `None`) – 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** (`Unpack [_BaseFieldKwargs]`) – The same keyword arguments that `Number` receives.

```
class marshmallow.fields.Dict(keys=None, values=None, **kwargs)
```

A dict field. Supports dicts and dict-like objects

Example:

```
numbers = fields.Dict(keys=fields.Str(), values=fields.Float())
```

Parameters

- **kwargs** (`Unpack [_BaseFieldKwargs]`) – The same keyword arguments that `Mapping` receives.
- **keys** (`Field` | `type[Field]` | `None`)
- **values** (`Field` | `type[Field]` | `None`)

Added in version 2.1.0.

```
class marshmallow.fields.Email(**kwargs)
```

An email field.

Parameters

- **args** – The same positional arguments that `String` receives.
- **kwargs** (`Unpack [_BaseFieldKwargs]`) – The same keyword arguments that `String` receives.

```
class marshmallow.fields.Enum(enum, *, by_value=False, **kwargs)
```

An Enum field (de)serializing enum members by symbol (name) or by value.

Parameters

- **enum** (`type[_EnumT]`) – Enum class
- **by_value** (`bool` | `Field` | `type[Field]`) – Whether to (de)serialize by value or by name, or Field class or instance to use to (de)serialize by value. Defaults to False.
- **kwargs** (`Unpack [_BaseFieldKwargs]`)

If `by_value` is `False` (default), enum members are (de)serialized by symbol (name). If it is `True`, they are (de)serialized by value using `marshmallow.fields.Raw`. If it is a field instance or class, they are (de)serialized by value using this field.

Added in version 3.18.0.

```
class marshmallow.fields.Float(*, allow_nan=False, as_string=False, **kwargs)
```

A double as an IEEE-754 double precision string.

Parameters

- **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`, format the value as a string.
- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that `Number` receives.

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

A field that takes the value returned by a function.

Parameters

- **serialize** (*Callable[[Any], Any] | Callable[[Any, dict], Any] | None*) – A callable from which to retrieve the value. The function must take a single argument `obj` which is the object to be serialized. If no callable is provided then the `load_only` flag will be set to `True`.
- **deserialize** (*Callable[[Any], Any] | Callable[[Any, dict], Any] | None*) – A callable from which to retrieve the value. The function must take a single argument `value` which is the value to be deserialized. If no callable is provided then `value` will be passed through unchanged.
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

Changed in version 3.0.0a1: Removed `func` parameter.

Changed in version 4.0.0: Don't pass context to serialization and deserialization functions.

```
class marshmallow.fields.IP(*, exploded=False, **kwargs)
```

A IP address field.

Parameters

- **exploded** (*bool*) – If `True`, serialize ipv6 address in long form, ie. with groups consisting entirely of zeros included.
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

Added in version 3.8.0.

```
class marshmallow.fields.IPInterface(*, exploded=False, **kwargs)
```

A IPInterface field.

IP interface is the non-strict form of the IPNetwork type where arbitrary host addresses are always accepted.

IPAddress and mask e.g. '192.168.0.2/24' or '192.168.0.2/255.255.255.0'

see <https://python.readthedocs.io/en/latest/library/ipaddress.html#interface-objects>

Parameters

- **exploded** (*bool*) – If `True`, serialize ipv6 interface in long form, ie. with groups consisting entirely of zeros included.

- **kwargs** (*Unpack [_BaseFieldKwargs]*)

class marshmallow.fields.IPv4(*, exploded=False, **kwargs)

A IPv4 address field.

Added in version 3.8.0.

Parameters

- **exploded** (*bool*)
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

class marshmallow.fields.IPv4Interface(*, exploded=False, **kwargs)

A IPv4 Network Interface field.

Parameters

- **exploded** (*bool*)
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

class marshmallow.fields.IPv6(*, exploded=False, **kwargs)

A IPv6 address field.

Added in version 3.8.0.

Parameters

- **exploded** (*bool*)
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

class marshmallow.fields.IPv6Interface(*, exploded=False, **kwargs)

A IPv6 Network Interface field.

Parameters

- **exploded** (*bool*)
- **kwargs** (*Unpack [_BaseFieldKwargs]*)

marshmallow.fields.Int

alias of *Integer*

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

An integer field.

Parameters

- **strict** (*bool*) – If *True*, only integer types are valid. Otherwise, any value castable to *int* is valid.
- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that *Number* receives.
- **as_string** (*bool*)

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[_InternalT] | type[Field[_InternalT]]`) – A field class or instance.
- **kwargs** (`Unpack[_BaseFieldKwargs]`) – The same keyword arguments that `Field` receives.

Changed in version 3.0.0rc9: Does not serialize scalar values to single-item lists.

class `marshmallow.fields.Mapping`(`keys=None, values=None, **kwargs`)

Abstract base class for objects with key-value pairs.

Use `Dict` within schemas.

Parameters

- **keys** (`Field | type[Field] | None`) – A field class or instance for dict keys.
- **values** (`Field | type[Field] | None`) – A field class or instance for dict values.
- **kwargs** (`Unpack[_BaseFieldKwargs]`) – The same keyword arguments that `Field` receives.

Note

When the structure of nested data is not known, you may omit the `keys` and `values` arguments to prevent content validation.

Added in version 3.0.0rc4.

Changed in version 3.24.0: `Mapping` should no longer be used as a field within a `Schema`. Use `Dict` instead.

class `marshmallow.fields.Method`(`serialize=None, deserialize=None, **kwargs`)

A field that takes the value returned by a `Schema` method.

Parameters

- **serialize** (`str | None`) – 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 | None`) – Optional name of the Schema method for deserializing a value. The method must take a single argument `value`, which is the value to deserialize.
- **kwargs** (`Unpack[_BaseFieldKwargs]`)

Changed in version 3.0.0: Removed `method_name` parameter.

class `marshmallow.fields.NaiveDateTime`(`format=None, *, timezone=None, **kwargs`)

A formatted naive datetime string.

Parameters

- **format** (`str | None`) – See `DateTime`.
- **timezone** (`dt.timezone | None`) – Used on deserialization. If `None`, aware datetimes are rejected. If not `None`, aware datetimes are converted to this timezone before their timezone information is removed.
- **kwargs** (`Unpack[_BaseFieldKwargs]`) – The same keyword arguments that `Field` receives.

Added in version 3.0.0rc9.

```
class marshmallow.fields.Nested(nested, *, only=None, exclude=(), many=None, unknown=None,
                               **kwargs)
```

Allows you to nest a [Schema](#) inside a field.

Examples:

```
class ChildSchema(Schema):
    id = fields.Str()
    name = fields.Str()
    # Use lambda functions when you need two-way nesting or self-nesting
    parent = fields.Nested(lambda: ParentSchema(only=("id",)), dump_only=True)
    siblings = fields.List(
        fields.Nested(lambda: ChildSchema(only=("id", "name"))))
    )

class ParentSchema(Schema):
    id = fields.Str()
    children = fields.List(
        fields.Nested(ChildSchema(only=("id", "parent", "siblings"))))
    )
    spouse = fields.Nested(lambda: ParentSchema(only=("id",)))
```

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](#) | [SchemaMeta](#) | [str](#) | [dict](#)[[str](#), [Field](#)] | [Callable](#)[[], [Schema](#) | [SchemaMeta](#) | [str](#) | [dict](#)[[str](#), [Field](#)]]) – [Schema](#) instance, class, class name (string), dictionary, or callable that returns a [Schema](#) or dictionary. Dictionaries are converted with [Schema.from_dict](#).
- **exclude** ([types.StrSequenceOrSet](#)) – A list or tuple of fields to exclude.
- **only** ([types.StrSequenceOrSet](#) | [None](#)) – A list or tuple of fields to marshal. If [None](#), all fields are marshalled. This parameter takes precedence over `exclude`.
- **many** ([bool](#) | [None](#)) – Whether the field is a collection of objects. If [None](#) (default), and nested [Schema](#) instance is provided, `many` is not overridden. If [None](#) (default), and [Schema](#) subclass is provided, schema instance sets `many` as `False`. If `True` | `False` nested `[nested_field].schema.many` is overridden.
- **unknown** ([types.UnknownOption](#) | [None](#)) – Whether to exclude, include, or raise an error for unknown fields in the data. Use `EXCLUDE`, `INCLUDE` or `RAISE`.

- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

property schema: *Schema*

The nested Schema object.

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

Abstract base class for number fields.

Use *Integer*, *Float*, or *Decimal* within schemas.

Parameters

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

Changed in version 3.24.0: *Number* should no longer be used as a field within a *Schema*. Use *Integer*, *Float*, or *Decimal* instead.

class `marshmallow.fields.Pluck`(*nested*, *field_name*, *, *many=False*, *unknown=None*, **kwargs)

Allows you to replace nested data with one of the data's fields.

Example:

```
from marshmallow import Schema, fields

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

class AlbumSchema(Schema):
    artist = fields.Pluck(ArtistSchema, "id")

in_data = {"artist": 42}
loaded = AlbumSchema().load(in_data) # => {'artist': {'id': 42}}
dumped = AlbumSchema().dump(loaded) # => {'artist': 42}
```

Parameters

- **nested** (*Schema* | *SchemaMeta* | *str* | *Callable[[], Schema]*) – The Schema class or class name (string) to nest
- **field_name** (*str*) – The key to pluck a value from.
- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that *Nested* receives.
- **many** (*bool*)
- **unknown** (*types.UnknownOption* | *None*)

```
class marshmallow.fields.Raw(*, load_default=<marshmallow.missing>,
                             dump_default=<marshmallow.missing>, data_key=None, attribute=None,
                             validate=None, pre_load=None, post_load=None, required=False,
                             allow_none=None, load_only=False, dump_only=False,
                             error_messages=None, metadata=None)
```

Field that applies no formatting.

Parameters

- **load_default** (*Any*)
- **dump_default** (*Any*)
- **data_key** (*str* | *None*)
- **attribute** (*str* | *None*)
- **validate** (*types.Validator* | *Iterable*[*types.Validator*] | *None*)
- **pre_load** (*types.PreLoadCallable* | *Iterable*[*types.PreLoadCallable*] | *None*)
- **post_load** (*types.PostLoadCallable* | *Iterable*[*types.PostLoadCallable*] | *None*)
- **required** (*bool*)
- **allow_none** (*bool* | *None*)
- **load_only** (*bool*)
- **dump_only** (*bool*)
- **error_messages** (*types.ErrorMessages* | *None*)
- **metadata** (*Mapping*[*str*, *Any*] | *None*)

marshmallow.fields.Str

alias of *String*

```
class marshmallow.fields.String(*, load_default=<marshmallow.missing>,
                                 dump_default=<marshmallow.missing>, data_key=None, attribute=None,
                                 validate=None, pre_load=None, post_load=None, required=False,
                                 allow_none=None, load_only=False, dump_only=False,
                                 error_messages=None, metadata=None)
```

A string field.

Parameters

- **kwargs** – The same keyword arguments that *Field* receives.
- **load_default** (*Any*)
- **dump_default** (*Any*)
- **data_key** (*str* | *None*)
- **attribute** (*str* | *None*)
- **validate** (*types.Validator* | *Iterable*[*types.Validator*] | *None*)
- **pre_load** (*types.PreLoadCallable* | *Iterable*[*types.PreLoadCallable*] | *None*)
- **post_load** (*types.PostLoadCallable* | *Iterable*[*types.PostLoadCallable*] | *None*)

- **required** (*bool*)
- **allow_none** (*bool* | *None*)
- **load_only** (*bool*)
- **dump_only** (*bool*)
- **error_messages** (*types.ErrorMessages* | *None*)
- **metadata** (*Mapping[str, Any]* | *None*)

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

A formatted time string.

Example: '03:12:58.019077'

Parameters

- **format** (*str* | *None*) – Either "iso" (for ISO8601) or a date format string. If *None*, defaults to "iso".
- **kwargs** (*Unpack[_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

class marshmallow.fields.**TimeDelta**(*precision='seconds', **kwargs*)

A field that (de)serializes a `datetime.timedelta` object to a `float`. The `float` can represent any time unit that the `datetime.timedelta` constructor supports.

Parameters

- **precision** (*str*) – The time unit used for (de)serialization. Must be one of 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds' or 'microseconds'.
- **kwargs** (*Unpack[_BaseFieldKwargs]*) – The same keyword arguments that *Field* receives.

Float Caveats

Precision loss may occur when serializing a highly precise `datetime.timedelta` object using a big precision unit due to floating point arithmetics.

When necessary, the `datetime.timedelta` constructor rounds `float` inputs to whole microseconds during initialization of the object. As a result, deserializing a `float` might be subject to rounding, regardless of precision. For example, `TimeDelta().deserialize("1.1234567") == timedelta(seconds=1, microseconds=123457)`.

Changed in version 3.17.0: Allow serialization to `float` through use of a new `serialization_type` parameter. Defaults to `int` for backwards compatibility. Also affects deserialization.

Changed in version 4.0.0: Remove `serialization_type` parameter and always serialize to float. Value is cast to a `float` upon deserialization.

class marshmallow.fields.**Tuple**(*tuple_fields, **kwargs*)

A tuple field, composed of a fixed number of other *Field* classes or instances

Example:

```
row = Tuple((fields.String(), fields.Integer(), fields.Float()))
```

Note

Because of the structured nature of `collections.namedtuple` and `typing.NamedTuple`, using a Schema within a Nested field for them is more appropriate than using a `Tuple` field.

Parameters

- **tuple_fields** (*Iterable*[*Field* | *type*[*Field*]]) – An iterable of field classes or instances.
- **kwargs** (*Unpack* [*_BaseFieldKwargs*]) – The same keyword arguments that *Field* receives.

Added in version 3.0.0rc4.

`marshmallow.fields.URL`

alias of *Url*

```
class marshmallow.fields.UUID(*, load_default=<marshmallow.missing>,
                              dump_default=<marshmallow.missing>, data_key=None, attribute=None,
                              validate=None, pre_load=None, post_load=None, required=False,
                              allow_none=None, load_only=False, dump_only=False,
                              error_messages=None, metadata=None)
```

A UUID field.

Parameters

- **load_default** (*Any*)
- **dump_default** (*Any*)
- **data_key** (*str* | *None*)
- **attribute** (*str* | *None*)
- **validate** (*types.Validator* | *Iterable*[*types.Validator*] | *None*)
- **pre_load** (*types.PreLoadCallable* | *Iterable*[*types.PreLoadCallable*] | *None*)
- **post_load** (*types.PostLoadCallable* | *Iterable*[*types.PostLoadCallable*] | *None*)
- **required** (*bool*)
- **allow_none** (*bool* | *None*)
- **load_only** (*bool*)
- **dump_only** (*bool*)
- **error_messages** (*types.ErrorMessages* | *None*)
- **metadata** (*Mapping*[*str*, *Any*] | *None*)

```
class marshmallow.fields.Url(*, relative=False, absolute=True, schemes=None, require_tld=True,
                             **kwargs)
```

An URL field.

Parameters

- **default** – Default value for the field if the attribute is not set.

- **relative** (*bool*) – Whether to allow relative URLs.
- **absolute** (*bool*) – Whether to allow absolute URLs.
- **require_tld** (*bool*) – Whether to reject non-FQDN hostnames.
- **schemes** (*types.StrSequenceOrSet | None*) – Valid schemes. By default, http, https, ftp, and ftps are allowed.
- **kwargs** (*Unpack [_BaseFieldKwargs]*) – The same keyword arguments that *String* receives.

6.4 Decorators

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

Methods decorated with *pre_load*, *post_load*, *pre_dump*, *post_dump*, and *validates_schema* receive many as a keyword argument. In addition, *pre_load*, *post_load*, and *validates_schema* receive *partial*. If you don't need these arguments, add ***kwargs* to your method signature.

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, many, **kwargs):
        item["email"] = item["email"].lower().strip()
        return item

    @pre_load(pass_collection=True)
    def remove_envelope(self, data, many, **kwargs):
        namespace = "results" if many else "result"
        return data[namespace]

    @post_dump(pass_collection=True)
    def add_envelope(self, data, many, **kwargs):
        namespace = "results" if many else "result"
        return {namespace: data}

    @validates_schema
    def validate_email(self, data, **kwargs):
```

(continues on next page)

(continued from previous page)

```

    if len(data["email"]) < 3:
        raise ValidationError("Email must be more than 3 characters", "email")

@validates("age")
def validate_age(self, data, **kwargs):
    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.

Functions:

<code>post_dump</code>	Register a method to invoke after serializing an object.
<code>post_load</code>	Register a method to invoke after deserializing an object.
<code>pre_dump</code>	Register a method to invoke before serializing an object.
<code>pre_load</code>	Register a method to invoke before deserializing an object.
<code>set_hook</code>	Mark decorated function as a hook to be picked up later.
<code>validates</code>	Register a validator method for field(s).
<code>validates_schema</code>	Register a schema-level validator.

`marshmallow.decorators.post_dump(fn=None, *, pass_collection=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 it receives a single object at a time, transparently handling the `many` argument passed to the `Schema`'s `dump()` call. If `pass_collection=True`, the raw data (which may be a collection) is passed.

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

Changed in version 3.0.0: `many` is always passed as a keyword arguments to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection` and `pass_original` are keyword-only arguments.

Parameters

- `fn` (`Callable[[...], Any] | None`)
- `pass_collection` (`bool`)
- `pass_original` (`bool`)

Return type*Callable[[...], Any]*`marshmallow.decorators.post_load(fn=None, *, pass_collection=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 it receives a single object at a time, transparently handling the many argument passed to the *Schema*'s *load()* call. If `pass_collection=True`, the raw data (which may be a collection) is passed.

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

Changed in version 3.0.0: `partial` and `many` are always passed as keyword arguments to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection` and `pass_original` are keyword-only arguments.

Changed in version 4.0.0: `unknown` is passed as a keyword argument to the decorated method.

Parameters

- **fn** (*Callable[[...], Any] | None*)
- **pass_collection** (*bool*)
- **pass_original** (*bool*)

Return type*Callable[[...], Any]*`marshmallow.decorators.pre_dump(fn=None, *, pass_collection=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 it receives a single object at a time, transparently handling the many argument passed to the *Schema*'s *dump()* call. If `pass_collection=True`, the raw data (which may be a collection) is passed.

Changed in version 3.0.0: `many` is always passed as a keyword arguments to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection` is a keyword-only argument.

Parameters

- **fn** (*Callable[[...], Any] | None*)
- **pass_collection** (*bool*)

Return type*Callable[[...], Any]*`marshmallow.decorators.pre_load(fn=None, *, pass_collection=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 it receives a single object at a time, transparently handling the many argument passed to the *Schema*'s *load()* call. If `pass_collection=True`, the raw data (which may be a collection) is passed.

Changed in version 3.0.0: `partial` and `many` are always passed as keyword arguments to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection` is a keyword-only argument.

Changed in version 4.0.0: `unknown` is passed as a keyword argument to the decorated method.

Parameters

- **fn** (*Callable*[[...], Any] | None)
- **pass_collection** (*bool*)

Return type

Callable[[...], Any]

`marshmallow.decorators.set_hook(fn, tag, *, many=False, **kwargs)`

Mark decorated function as a hook to be picked up later. You should not need to use this method directly.

i Note

Currently only 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.

Parameters

- **fn** (*Callable*[[...], Any] | None)
- **tag** (*str*)
- **many** (*bool*)
- **kwargs** (*Any*)

Return type

Callable[[...], Any]

`marshmallow.decorators.validates(*field_names)`

Register a validator method for field(s).

Parameters

field_names (*str*) – Names of the fields that the method validates.

Return type

Callable[[...], Any]

Changed in version 4.0.0: Accepts multiple field names as positional arguments.

Changed in version 4.0.0: Decorated methods receive `data_key` as a keyword argument.

`marshmallow.decorators.validates_schema(fn=None, *, pass_collection=False, pass_original=False, skip_on_field_errors=True)`

Register a schema-level validator.

By default it receives a single object at a time, transparently handling the `many` argument passed to the *Schema*'s `validate()` call. If `pass_collection=True`, the raw data (which may be a collection) 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.

Changed in version 3.0.0b1: `skip_on_field_errors` defaults to `True`.

Changed in version 3.0.0: `partial` and `many` are always passed as keyword arguments to the decorated method.

Changed in version 4.0.0: `unknown` is passed as a keyword argument to the decorated method.

Changed in version 4.0.0: `pass_many` is renamed to `pass_collection`.

Changed in version 4.0.0: `pass_collection`, `pass_original`, and `skip_on_field_errors` are keyword-only arguments.

Parameters

- `fn` (*Callable*[[...], Any] | None)
- `pass_collection` (*bool*)
- `pass_original` (*bool*)
- `skip_on_field_errors` (*bool*)

Return type

Callable[[...], Any]

6.5 Validators

Validation classes for various types of data.

Classes:

<i>And</i>	Compose multiple validators and combine their error messages.
<i>ContainsNoneOf</i>	Validator which fails if <code>value</code> is a sequence and any element in the sequence is a member of the sequence passed as <code>iterable</code> .
<i>ContainsOnly</i>	Validator which succeeds if <code>value</code> is a sequence and each element in the sequence is also in the sequence passed as <code>choices</code> .
<i>Email</i>	Validate an email address.
<i>Equal</i>	Validator which succeeds if the <code>value</code> passed to it is equal to <code>comparable</code> .
<i>Length</i>	Validator which succeeds if the <code>value</code> passed to it has a length between a minimum and maximum.
<i>NoneOf</i>	Validator which fails if <code>value</code> is a member of <code>iterable</code> .
<i>OneOf</i>	Validator which succeeds if <code>value</code> is a member of <code>choices</code> .
<i>Predicate</i>	Call the specified <code>method</code> of the <code>value</code> object.
<i>Range</i>	Validator which succeeds if the <code>value</code> passed to it is within the specified range.
<i>Regexp</i>	Validator which succeeds if the <code>value</code> matches <code>regex</code> .
<i>URL</i>	Validate a URL.
<i>Validator</i>	Abstract base class for validators.

```
class marshmallow.validate.And(*validators)
```

Compose multiple validators and combine their error messages.

Example:

```

from marshmallow import validate, ValidationError

def is_even(value):
    if value % 2 != 0:
        raise ValidationError("Not an even value.")

validator = validate.And(validate.Range(min=0), is_even)
validator(-1)
# ValidationError: ['Must be greater than or equal to 0.', 'Not an even value.']

```

Parameters

validators (*types.Validator*) – Validators to combine.

class marshmallow.validate.**ContainsNoneOf**(*iterable*, *, *error=None*)

Validator which fails if *value* is a sequence and any element in the sequence is a member of the sequence passed as *iterable*. Empty input is considered valid.

Parameters

- **iterable** (*Iterable*) – Same as *NoneOf*.
- **error** (*str* | *None*) – Same as *NoneOf*.

Added in version 3.6.0.

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*. Empty input is considered valid.

Parameters

- **choices** (*Iterable*) – Same as *OneOf*.
- **labels** (*Iterable[str]* | *None*) – Same as *OneOf*.
- **error** (*str* | *None*) – Same as *OneOf*.

Changed in version 3.0.0b2: Duplicate values are considered valid.

Changed in version 3.0.0b2: Empty input is considered valid. Use *validate.Length(min=1)* to validate against empty inputs.

class marshmallow.validate.**Email**(*, *error=None*)

Validate an email address.

Parameters

error (*str* | *None*) – 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* | *None*) – 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, *, equal=None, error=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* | *None*) – The minimum length. If not provided, minimum length will not be checked.
- **max** (*int* | *None*) – The maximum length. If not provided, maximum length will not be checked.
- **equal** (*int* | *None*) – The exact length. If provided, maximum and minimum length will not be checked.
- **error** (*str* | *None*) – 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* | *None*) – 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[str]* | *None*) – Optional sequence of labels to pair with the choices.
- **error** (*str* | *None*) – 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 (*str* | *Callable[[Any], Any]*) – 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().

Return type

Iterable[tuple[Any, str]]

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* | *None*) – 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, *, min_inclusive=True, max_inclusive=True,
                                error=None)
```

Validator which succeeds if the value passed to it is within the specified range. 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. The inclusivity of the bounds (if they exist) is configurable. If `min_inclusive` is not specified, or is specified as `True`, then the `min` bound is included in the range. If `max_inclusive` is not specified, or is specified as `True`, then the `max` bound is included in the range.

Parameters

- **min** (*Any*) – The minimum value (lower bound). If not provided, minimum value will not be checked.
- **max** (*Any*) – The maximum value (upper bound). If not provided, maximum value will not be checked.
- **min_inclusive** (*bool*) – Whether the `min` bound is included in the range.
- **max_inclusive** (*bool*) – Whether the `max` bound is included in the range.
- **error** (*str* / *None*) – 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)
```

Validator which succeeds if the value matches `regex`.

Note

Uses `re.match`, which searches for a match at the beginning of a string.

Parameters

- **regex** (*str* / *bytes* / *re.Pattern[str]* / *re.Pattern[bytes]*) – The regular expression string to use. Can also be a compiled regular expression pattern.
- **flags** (*int*) – The regexp flags to use, for example `re.IGNORECASE`. Ignored if `regex` is not a string.
- **error** (*str* / *None*) – Error message to raise in case of a validation error. Can be interpolated with `{input}` and `{regex}`.

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

Validate a URL.

Parameters

- **relative** (*bool*) – Whether to allow relative URLs.
- **absolute** (*bool*) – Whether to allow absolute URLs.
- **error** (*str* / *None*) – Error message to raise in case of a validation error. Can be interpolated with `{input}`.
- **schemes** (*types.StrSequenceOrSet* / *None*) – Valid schemes. By default, `http`, `https`, `ftp`, and `ftps` are allowed.
- **require_tld** (*bool*) – Whether to reject non-FQDN hostnames.

class `marshmallow.validate.Validator`

Abstract base class for validators.

Note

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

6.6 Utility functions

Utility methods for marshmallow.

`marshmallow.utils.callable_or_raise(obj)`

Check that an object is callable, else raise a `TypeError`.

`marshmallow.utils.get_value(obj, key, default=<marshmallow.missing>)`

Helper for pulling a keyed value off various types of objects. Fields use this method by default to access attributes of the source object. For object `x` and attribute `i`, this method first tries to access `x[i]`, and then falls back to `x.i` if an exception is raised.

Warning

If an object `x` does not raise an exception when `x[i]` does not exist, `get_value` will never check the value `x.i`. Consider overriding `marshmallow.fields.Field.get_value` in this case.

Parameters

key (*int* | *str*)

`marshmallow.utils.is_collection(obj)`

Return True if `obj` is a collection type, e.g list, tuple, queryset.

Return type

TypeGuard[*Iterable*]

`marshmallow.utils.is_generator(obj)`

Return True if `obj` is a generator

Return type

TypeGuard[*Generator*]

`marshmallow.utils.is_iterable_but_not_string(obj)`

Return True if `obj` is an iterable object that isn't a string.

Return type

TypeGuard[*Iterable*]

`marshmallow.utils.is_sequence_but_not_string(obj)`

Return True if `obj` is a sequence that isn't a string.

Return type

TypeGuard[*Sequence*]

`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]
```

Parameters

- **dictlist** (*list[dict[str, Any]]*)
- **key** (*str*)

`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}}
```

Parameters

- **dct** (*dict[str, Any]*)
- **key** (*str*)
- **value** (*Any*)

`marshmallow.utils.timedelta_to_microseconds(value)`

Compute the total microseconds of a timedelta.

https://github.com/python/cpython/blob/v3.13.1/Lib/_pydatetime.py#L805-L807

Parameters

value (*timedelta*)

Return type

int

6.7 Context (experimental)

Helper API for setting serialization/deserialization context.

Example usage:

```
import typing

from marshmallow import Schema, fields
from marshmallow.experimental.context import Context

class UserContext(typing.TypedDict):
    suffix: str
```

(continues on next page)

```
UserSchemaContext = Context[UserContext]

class UserSchema(Schema):
    name_suffixed = fields.Function(
        lambda user: user["name"] + UserSchemaContext.get()["suffix"]
    )

with UserSchemaContext({"suffix": "bar"}):
    print(UserSchema().dump({"name": "foo"}))
    # {'name_suffixed': 'foobar'}
```

```
class marshmallow.experimental.context.Context(context)
```

Context manager for setting and retrieving context.

Parameters

context (*_ContextT*) – The context to use within the context manager scope.

```
classmethod get(default=Ellipsis)
```

Get the current context.

Parameters

default (*_DefaultT | EllipsisType*) – Default value to return if no context is set. If not provided and no context is set, a `LookupError` is raised.

Return type

_ContextT | _DefaultT

6.8 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.StringNotCollectionError
```

Raised when a string is passed when a list of strings is expected.

```
exception marshmallow.exceptions.ValidationError(message, field_name='_schema', data=None,
                                                valid_data=None, **kwargs)
```

Raised when validation fails on a field or schema.

Validators and custom fields should raise this exception.

Parameters

- **message** (*str | list | dict*) – An error message, list of error messages, or dict of error messages. If a dict, the keys are subitems and the values are error messages.
- **field_name** (*str*) – Field name to store the error on.
- **data** (*Mapping[str, Any] | Iterable[Mapping[str, Any]] | None*) – Raw input data.

- `valid_data` (`list[Any]` | `dict[str, Any]` | `None`) – Valid (de)serialized data.

6.9 Private API

6.9.1 Types

Type aliases.

Warning

This module is provisional. Types may be modified, added, and removed between minor releases.

`marshmallow.types.ErrorMessageValue`: `TypeAlias = str | list | dict`

Type for a single error message value, which can be a string, list, or dict

`marshmallow.types.ErrorMessages`(*args, **kwargs)

Type for error_messages dictionaries passed to fields

alias of `dict[str, str | list | dict]`

`marshmallow.types.PostLoadCallable`(*args, **kwargs)

Type for field-level post-load functions

alias of `Callable[[T], T]`

`marshmallow.types.PreLoadCallable`(*args, **kwargs)

Type for field-level pre-load functions

alias of `Callable[[Any], Any]`

`class marshmallow.types.RenderModule`(*args, **kwargs)

`class marshmallow.types.SchemaValidator`(*args, **kwargs)

`marshmallow.types.StrSequenceOrSet`(*args, **kwargs)

A type that can be either a sequence of strings or a set of strings

alias of `Sequence[str] | AbstractSet[str]`

`marshmallow.types.UnknownOption`(*args, **kwargs)

A valid option for the unknown schema option and argument

alias of `Literal['exclude', 'include', 'raise']`

`marshmallow.types.Validator`(*args, **kwargs)

Type for validator functions

alias of `Callable[[Any], Any]`

6.9.2 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: str, *, all: Literal[False] = False) → SchemaType`

`marshmallow.class_registry.get_class(classname: str, *, all: Literal[True] = False) → list[SchemaType]`

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],
# }
```

Parameters

- `classname` (*str*)
- `cls` (*SchemaType*)

Return type

None

6.9.3 Error store

Utilities for storing collections of error messages.

Warning

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

`marshmallow.error_store.merge_errors(errors1, errors2)`

Deeply merge two error messages.

The format of `errors1` and `errors2` matches the message parameter of `marshmallow.exceptions.ValidationError`.

PROJECT INFO

7.1 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.

7.1.1 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](#)).

7.1.2 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 overridden at application runtime by passing arguments to the *Schema* constructor. The *dump* and *load* methods are used for serialization and deserialization (of course!).

7.1.3 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 enables patterns for configuring and extending schemas, such as adding *post-processing and error handling behavior*.

7.1.4 Consistency meets flexibility

marshmallow makes it easy to modify a schema's output at application runtime. A single *Schema* can produce multiple output 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)
```

(continues on next page)

```
score = fields.Nested(ScoreSchema)
players = fields.List(fields.Nested(PlayerSchema))
last_changed = fields.DateTime(format="rfc")

# 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**.

7.1.5 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.

See also

Need help upgrading to newer versions? Check out the [upgrading guide](#).

7.2 Changelog

7.2.1 4.3.0 (2026-04-03)

Features:

- Add `pre_load` and `post_load` parameters to `marshmallow.fields.Field` for field-level pre- and post-processing (#2787).
- Typing: improvements to `marshmallow.validate` (#2940).

7.2.2 4.2.4 (2026-04-02)

Bug fixes:

- `marshmallow.validate.URL` and `marshmallow.validate.Email` accept Internationalized Domain Names (IDNs) (#2821, #2936). `marshmallow.validate.Email` also correctly rejects IDN domains with leading/trailing hyphens. Thanks @touhidurrr for the report.
- Typing: Fix typing of nested in `marshmallow.fields.Nested` (#2935).

7.2.3 4.2.3 (2026-03-25)

Bug fixes:

- Make `marshmallow.fields.Number` and `marshmallow.fields.Mapping` abstract base classes to prevent using them within Schemas (#2924). Thanks @MartingaleCoda for reporting.
- Allow `required` to be set on `marshmallow.fields.Contant` (#2900). Thanks @nosnickid for the report and @worksbyfriday for the PR.
- Fix `marshmallow.validate.OneOf` emitting extra pairs when labels outnumber choices (#2869). Thanks: user:T90REAL for the report and @rstar327 for the PR.
- Fix behavior when passing a dot-delimited attribute name to `partial` for a key with `data_key` set (#2903). Thanks @bysiber for the PR.
- Fix Enum field by-name lookup to only return actual members (#2902). Thanks @bysiber for the PR.
- `marshmallow.fields.DateTime` with `format="timestamp_ms"` properly rejects bool values (#2904). Thanks @bysiber for the PR.
- Fix typing of `error_messages` argument to `marshmallow.fields.Field` (#1636). Thanks @repole for reporting and @dhruvildarji for the PR.

Other changes:

- Add `ipaddress.*` to `marshmallow.Schema.TYPE_MAPPING` (#1695). Thanks @liberforce for the suggestion and @dhruvildarji for the PR.

7.2.4 4.2.2 (2026-02-04)

Bug fixes:

- Fix behavior of `fields.Contant(None)` (#2868). Thanks @T90REAL for reporting and emmanuel-ferdman for the fix.

7.2.5 4.2.1 (2026-01-23)

Bug fixes:

- Fix validation of URLs beginning with uppercase FILE (#2891). Thanks @thanhlecongg for reporting and fixing.

7.2.6 4.2.0 (2026-01-04)

Other changes:

- `many` argument of `Nested` properly overrides schema instance value (#2854). Thanks @jafournier for the PR.

7.2.7 4.1.2 (2025-12-19)

Bug fixes:

- **CVE 2025-68480**: Merge error store messages without rebuilding collections. Thanks for reporting and @deckar01 for the fix.

7.2.8 4.1.1 (2025-11-05)

Bug fixes:

- Ensure URL validator is case-insensitive when using custom schemes (#2874). Thanks @T90REAL for the PR.

7.2.9 4.1.0 (2025-11-01)

Other changes:

- Add `__len__` implementation to `missing` so that it can be used with `validate.Length` (#2861). Thanks @agentgodzilla for the PR.
- Drop support for Python 3.9 (#2363).
- Test against Python 3.14 (#2864).

7.2.10 4.0.1 (2025-08-28)

Bug fixes:

- Fix wildcard import of `from marshmallow import *` (#2823). Thanks @Florian-Laport for the PR.

7.2.11 4.0.0 (2025-04-16)

See *Upgrading to 4.0* for a guide on updating your code.

Features:

- Typing: Add types to all `Field` constructor kwargs (#2285). Thanks @navignaw for the suggestion.
- `DateTime`, `Date`, `Time`, `TimeDelta`, and `Enum` accept their internal value types as valid input (#1415). Thanks @bitdancer for the suggestion.
- `@validates` accepts multiple field names (#1960). *Backwards-incompatible*: Decorated methods now receive `data_key` as a keyword argument. Thanks @dpriskorn for the suggestion and @dharani7998 for the PR.

Other changes:

- Typing: `Field` is now a generic type with a type argument for the internal value type.
- `marshmallow.fields.UUID` no longer subclasses `marshmallow.fields.String`.
- `marshmallow.Schema.load` no longer silently fails to call schema validators when a generator is passed (#1898). The typing of data is also updated to be more accurate. Thanks @ziplokk1 for reporting.
- *Backwards-incompatible*: Use `datetime.date.fromisoformat`, `datetime.time.fromisoformat`, and `datetime.datetime.fromisoformat` from the standard library to deserialize dates, times and datetimes (#2078). As a consequence of this change:
 - Time with time offsets are now supported.
 - YYYY-MM-DD is now accepted as a datetime and deserialized as naive 00:00 AM.
 - `from_iso_date`, `from_iso_time` and `from_iso_datetime` are removed from `marshmallow.utils`.
- Remove `isoformat`, `to_iso_time` and `to_iso_datetime` from `marshmallow.utils` (#2766).
- Remove `from_rfc`, and `rfcformat` from `marshmallow.utils` (#2767).
- Remove `is_keyed_tuple` from `marshmallow.utils` (#2768).
- Remove `get_fixed_timezone` from `marshmallow.utils` (#2773).
- *Backwards-incompatible*: `marshmallow.fields.Boolean` no longer serializes non-boolean values (#2725).
- *Backwards-incompatible*: Rename `schema` parameter to `parent` in `marshmallow.fields.Field._bind_to_schema` (#1360).
- *Backwards-incompatible*: Rename `pass_many` parameter to `pass_collection` in pre/post processing methods (#1369).

- *Backwards-incompatible:* `marshmallow.fields.TimeDelta` no longer truncates float values when deserializing (#2654). This allows microseconds to be preserved, e.g.

```
from marshmallow import fields

field = fields.TimeDelta()

# Before
field.deserialize(12.9)
datetime.timedelta(seconds=12)
# datetime.timedelta(seconds=12)

# After
field.deserialize(12.9)
# datetime.timedelta(seconds=12, microseconds=900000)
```

- Improve performance and minimize float precision loss of `marshmallow.fields.TimeDelta` serialization (#2654).
- *Backwards-incompatible:* Remove `serialization_type` parameter from `marshmallow.fields.TimeDelta` (#2654).

Thanks @ddelange for the PR.

- *Backwards-incompatible:* Remove `Schema`'s `context` attribute (deprecated since 3.24.0). Passing a context should be done using `contextvars.ContextVar` (#1826). marshmallow 4 provides an experimental `Context` manager class that can be used to both set and retrieve context.

```
import typing

from marshmallow import Schema, fields
from marshmallow.experimental.context import Context

class UserContext(typing.TypedDict):
    suffix: str

class UserSchema(Schema):
    name_suffixed = fields.Function(
        lambda obj: obj["name"] + Context[UserContext].get()["suffix"]
    )

with Context[UserContext]({"suffix": "bar"}):
    UserSchema().dump({"name": "foo"})
# {'name_suffixed': 'foobar'}
```

- Methods decorated with `marshmallow.pre_load`, `marshmallow.post_load`, `marshmallow.validates_schema`, receive `unknown` as a keyword argument (#1632). Thanks @jforand for the PR.
- *Backwards-incompatible:* Arguments to `decorators` are keyword-only arguments.
- *Backwards-incompatible:* Rename `json_data` parameter of `marshmallow.Schema.loads` to `s` for compatibility with most render module implementations (`json`, `simplejson`, etc.) (#2764). Also make it a positional-only argument.

- Incorrectly declaring a field using a field class rather than instance errors at class declaration time (previously happened when the schema was instantiated) (#2772).
- Passing invalid values for `unknown` will cause an error in type checkers (#2771).

Deprecations/Removals:

- *Backwards-incompatible*: Remove implicit field creation, i.e. using the `fields` or `additional` class Meta options with undeclared fields (#1356).
- The `ordered` class Meta option is removed (#2146). Field order is already preserved by default. Set `Schema.dict_class` to `OrderedDict` to maintain the previous behavior.
- The `marshmallow.base` module is removed (#2722).

Previously-deprecated APIs have been removed, including:

- The `ordered class Meta` option is removed (#2146) (deprecated in 3.26.0).
- *Backwards-incompatible*: `marshmallow.fields.Number` is no longer usable as a field in a schema (deprecated in 3.24.0). Use `marshmallow.fields.Integer`, `marshmallow.fields.Float`, or `marshmallow.fields.Decimal` instead.
- *Backwards-incompatible*: `marshmallow.fields.Mapping` is no longer usable as a field in a schema (deprecated in 3.24.0).
- *Backwards-incompatible*: Custom validators must raise a `ValidationError` for invalid values (deprecated in 3.24.0). Returning `False` is no longer supported (#1775). Use `marshmallow.fields.Dict` instead.
- Remove `__version__`, `__parsed_version__`, and `__version_info__` attributes (deprecated in 3.21.0).
- `default` and `missing` parameters, which were replaced by `dump_default` and `load_default` in 3.13.0 (#1742, #2700).
- Passing field metadata via keyword arguments (deprecated in 3.10.0). Use the explicit `metadata=...` argument instead (#1350).
- `marshmallow.utils.pprint` (deprecated in 3.7.0). Use `pprint.pprint` instead.
- Passing `"self"` to `fields.Nested` (deprecated in 3.3.0). Use a callable instead.
- `Field.fail`, which was replaced by `Field.make_error` in 3.0.0.
- `json_module` class Meta option (deprecated in 3.0.0b3). Use `render_module` instead.

7.2.12 3.26.2 (2025-12-19)

Bug fixes:

- **CVE 2025-68480**: Merge error store messages without rebuilding collections. Thanks for reporting and @deckar01 for the fix.

7.2.13 3.26.1 (2025-02-03)

Bug fixes:

- Typing: Fix type annotations for `class Meta` options (#2804). Thanks @lawrence-law for reporting.

Other changes:

- Remove default value for the `data` param of `Nested._deserialize` (#2802). Thanks @gbenson for reporting.

7.2.14 3.26.0 (2025-01-22)

Features:

- Typing: Add type annotations and improved documentation for `class Meta` options (#2760).
- Typing: Improve type coverage of `marshmallow.Schema.SchemaMeta` (#2761).
- Typing: `marshmallow.Schema.loads` parameter allows `bytes` and `bytesarray` (#2769).

Bug fixes:

- Respect `data_key` when schema validators raise a `ValidationError` with a `field_name` argument (#2170). Thanks @matejsp for reporting.
- Correctly handle multiple `@post_load` methods where one method appends to the data and another passes `pass_original=True` (#1755). Thanks @ghostwheel42 for reporting.
- URL fields now properly validate file paths (#2249). Thanks @0xDEC0DE for reporting and fixing.

Documentation:

- Add *upgrading guides* for 3.24 and 3.26 (#2780).
- Various documentation improvements (#2757, #2759, #2765, #2774, #2778, #2783, #2796).

Deprecations:

- The ordered `class Meta` option is deprecated (#2146, #2762). Field order is already preserved by default. Set `marshmallow.Schema.dict_class` to `collections.OrderedDict` to maintain the previous behavior.

7.2.15 3.25.1 (2025-01-11)

Bug fixes:

- Typing: Fix type annotations for `Tuple`, `Boolean`, and `Pluck` constructors (#2756).
- Typing: Fix overload for `marshmallow.class_registry.get_class` (#2756).

Documentation:

- Various documentation improvements (#2746, #2747, #2748, #2749, #2750, #2751).

7.2.16 3.25.0 (2025-01-09)

Features:

- Typing: Improve type annotations for `SchemaMeta.get_declared_fields` (#2742).

Bug fixes:

- Typing: Relax type annotation for `Schema.opts` to allow subclasses to define their own options classes (#2744).

Other changes:

- Restore `marshmallow.base.SchemaABC` for backwards-compatibility (#2743). Note that this class is deprecated and will be removed in marshmallow 4. Use `marshmallow.schema.Schema` as a base class for type-checking instead.

7.2.17 3.24.2 (2025-01-08)

Changes:

- Don't override `__new__` to avoid breaking usages of `inspect.signature` with `Field` classes. This allows `marshmallow-sqlalchemy` users to upgrade `marshmallow` without upgrading to `marshmallow-sqlalchemy>=1.1.1`.

Documentation:

- Add top-level API back to docs (#2739). Thanks @llucax for reporting.

7.2.18 3.24.1 (2025-01-06)

Bug fixes:

- Typing: Fix typing for `class_registry.get_class` (#2735).

7.2.19 3.24.0 (2025-01-06)

Features:

- Typing: Improve typings in `marshmallow.fields` (#2723).
- Typing: Replace type comments with inline typings (#2718).

Bug fixes:

- Typing: Fix type hint for nested parameter of `Nested` (#2721).

Deprecations:

- Custom validators should raise a `ValidationError` for invalid values. Returning `False` is no longer supported.
- Deprecate `context` parameter of `Schema` (#1826). Use `contextVars.ContextVar` to pass context data instead.
- `Field`, `Mapping`, and `Number` should no longer be used as fields within schemas. Use their subclasses instead.

7.2.20 3.23.3 (2025-01-03)

Bug fixes:

- Typing: Fix typing for `Schema.from_dict` (#1653). Thanks @SteadBytes for reporting.

Support:

- Documentation: Various documentation cleanups, including more concise docs in the `marshmallow.fields` API reference (#2307). Thanks @AbdealiLoKo for reporting.

7.2.21 3.23.2 (2024-12-18)

Bug fixes:

- Improve type hint formatting for `Field`, `Nested`, and `Function` fields to resolve PyCharm warnings (#2268). Thanks @Fares-Abubaker for reporting and fixing.

7.2.22 3.23.1 (2024-11-01)

Support:

- Document `absolute` parameter of URL field (#2327).
- Documentation: Remove (outdated) minimum Python 3 minor version in documentation and README (#2323).

7.2.23 3.23.0 (2024-10-17)

Features:

- Typing: replace “type” with specific metaclass for Schema and Field.

Other changes:

- Officially support Python 3.13 (#2319).
- Drop support for Python 3.8 (#2318).

7.2.24 3.22.0 (2024-08-20)

Features:

- Add many Meta option to Schema so it expects a collection by default (#2270). Thanks @himalczyk for reporting and @deckar01 for the PR.
- Refactor hooks (#2279). Thanks @deckar01 for the PR.

7.2.25 3.21.3 (2024-06-05)

Bug fixes:

- Fix memory leak that prevented schema instances from getting GC'd (#2277). Thanks @mrcljx for the PR.

7.2.26 3.21.2 (2024-05-01)

Bug fixes:

- Allow timestamp 0 in fields.DateTimeField (#2133). Thanks @flydzen for reporting.

7.2.27 3.21.1 (2024-03-04)

Bug fixes:

- Fix error message when field is declared as a class and not an instance (#2245). Thanks @travnick for reporting.

7.2.28 3.21.0 (2024-02-26)

Bug fixes:

- Fix validation of URL fields to allow missing user field, per NWG RFC 3986 (#2232). Thanks @ddennerline3 for reporting and @deckar01 for the PR.

Other changes:

- *Backwards-incompatible*: `__version__`, `__parsed_version__`, and `__version_info__` attributes are deprecated (#2227). Use feature detection or `importlib.metadata.version("marshmallow")` instead.

7.2.29 3.20.2 (2024-01-09)

Bug fixes:

- Fix Nested field type hint for lambda Schema types (#2164). Thanks @somethingnew2-0 for the PR.

Other changes:

- Officially support Python 3.12 (#2188). Thanks @hugovk for the PR.

7.2.30 3.20.1 (2023-07-20)

Bug fixes:

- Fix call to `get_declared_fields`: pass `dict_cls` again (#2152). Thanks @Cheaterman for reporting.

7.2.31 3.20.0 (2023-07-20)

Features:

- Add `absolute` parameter to URL validator and `Url` field (#2123). Thanks @sirosen for the PR.
- Use Abstract Base Classes to define `FieldABC` and `SchemaABC` (#1449). Thanks @aditkumar72 for the PR.
- Use `OrderedSet` as default `set_class`. Schemas are now ordered by default. (#1744)

Bug fixes:

- Handle `OSError` and `OverflowError` in `utils.from_timestamp` (#2102). Thanks @TheBigRoomXXL for the PR.
- Fix the default inheritance of nested partial schemas (#2149). Thanks @matejsp for reporting.

Other changes:

- Officially support Python 3.11 (#2067).
- Drop support for Python 3.7 (#2135).

7.2.32 3.19.0 (2022-11-11)

Features:

- Add `timestamp` and `timestamp_ms` formats to `fields.DateTime` (#612). Thanks @vgavro for the suggestion and thanks @vanHoi for the PR.

7.2.33 3.18.0 (2022-09-15)

Features:

- Add `Enum` field (#2017) and (#2044).

Bug fixes:

- Fix typing in `Field._serialize` signature (#2046).

7.2.34 3.17.1 (2022-08-22)

Bug fixes:

- Add return type to `fields.Email.__init__` (#2018). Thanks @kkirsche for the PR.
- Add missing type hint to `IPInterface.__init__` (#2036).

7.2.35 3.17.0 (2022-06-26)

Features:

- Support serialization as float in `TimeDelta` field (#1998). Thanks @marcosatti for the PR.
- Add `messages_dict` property to `ValidationError` to facilitate type checking (#1976). Thanks @sirosen for the PR.

7.2.36 3.16.0 (2022-05-29)

Features:

- Raise `ValueError` if an invalid value is passed to the `unknown` argument (#1721, #1732). Thanks @sirosen for the PR.

Other changes:

- Set lower bound for packaging requirement (#1957). Thanks @MatthewNicolTR for reporting and thanks @sirosen for the PR.
- Improve warning messages by passing `stacklevel` (#1986). Thanks @tirkarthi for the PR.

7.2.37 3.15.0 (2022-03-12)

Features:

- Allow passing a dict to `fields.Nested` (#1935). Thanks @sirosen for the PR.

Other changes:

- Address `distutils` deprecation warning in Python 3.10 (#1903). Thanks @kkirsche for the PR.
- Add `py310` to `black` target-version (#1921).
- Drop support for Python 3.6 (#1923).
- Use postponed evaluation of annotations (#1932). Thanks @Isira-Seneviratne for the PR.

7.2.38 3.14.1 (2021-11-13)

Bug fixes:

- Fix publishing type hints per `PEP-561` (#1905). Thanks @bwindor for the catch and patch.

7.2.39 3.14.0 (2021-10-17)

Bug fixes:

- Fix `fields.TimeDelta` serialization precision (#1865). Thanks @yarsanich for reporting.

Other changes:

- Fix type-hints for `data` arg in `Schema.validate` to accept list of dictionaries (#1790, #1868). Thanks @yourun-proger for PR.
- Improve warning when passing metadata as keyword arguments (#1882). Thanks @traherom for the PR.
- Don't build universal wheels. We don't support Python 2 anymore. (#1860) Thanks @YKdvd for reporting.
- Make the build reproducible (#1862).
- Drop support for Python 3.5 (#1863).
- Test against Python 3.10 (#1888).

7.2.40 3.13.0 (2021-07-21)

Features:

- Replace `missing/default` field parameters with `load_default/dump_default` (#1742). Thanks @sirosen for the PR.

Deprecations:

- The use of `missing/default` field parameters is deprecated and will be removed in marshmallow 4. `load_default/dump_default` should be used instead.

7.2.41 3.12.2 (2021-07-06)

Bug fixes:

- Don't expose `Fields` as `Schema` attributes. This reverts a change introduced in 3.12.0 that causes issues when field names conflict with `Schema` attributes or methods. `Fields` are still accessible on a `Schema` instance through the `fields` attribute. (#1843)

7.2.42 3.12.1 (2021-05-10)

Bug fixes:

- Fix bug that raised an `AttributeError` when instantiating a `Schema` with a field named `parent` (#1808). Thanks @flying-sheep for reporting and helping with the fix.

7.2.43 3.12.0 (2021-05-09)

Features:

- Add `validate.And` (#1768). Thanks @rugleb for the suggestion.
- Add type annotations to `marshmallow.decorators` (#1788, #1789). Thanks @michaeldimchuk for the PR.
- Let `Fields` be accessed by name as `Schema` attributes (#1631).

Other changes:

- Improve types in `marshmallow.validate` (#1786).
- Make `marshmallow.validate.Validator` an abstract base class (#1786).
- Remove unnecessary list cast (#1785).

7.2.44 3.11.1 (2021-03-29)

Bug fixes:

- Fix treatment of dotted keys when `unknown=INCLUDE` (#1506). Thanks @rbu for reporting and thanks @sirosen for the fix (#1745).

7.2.45 3.11.0 (2021-03-28)

Features:

- Add `fields.IPInterface`, `fields.IPv4Interface`, and `IPv6Interface` (#1733). Thanks @madeinoz67 for the suggestion and the PR.
- Raise `AttributeError` for missing methods when using `fields.Method` (#1675). Thanks @lassandroan.

Other changes:

- Remove unnecessary `hasattr` and `getattr` checks in `Field` (#1770).

7.2.46 3.10.0 (2020-12-19)

Deprecations:

- Passing field metadata via keyword arguments is deprecated and will be removed in marshmallow 4 (#1350). Use the explicit `metadata=...` argument instead. Thanks @sirosen.

7.2.47 3.9.1 (2020-11-07)

Bug fixes:

- Cast to mapping type in `Mapping.serialize` and `Mapping.deserialize` (#1685).
- Fix bug letting `Dict` pass invalid dict on deserialization when no key or value `Field` is specified (#1685).

7.2.48 3.9.0 (2020-10-31)

Features:

- Add `format` argument to `fields.Time` and `timeformat` class `Meta` option (#686). Thanks @BennyAlex for the suggestion and thanks @infinityxxx for the PR.

Other changes:

- Remove usage of `implicit typing.Optional` (#1663). Thanks @nadega for the PR.

7.2.49 3.8.0 (2020-09-16)

Features:

- Add `fields.IP`, `fields.IPv4` and `fields.IPv6` (#1485). Thanks @mgetka for the PR.

Bug fixes:

- Fix typing in `AwareDateTime` (#1658). Thanks @adithyabsk for reporting.

7.2.50 3.7.1 (2020-07-20)

Bug fixes:

- `fields.Boolean` correctly serializes non-hashable types (#1633). Thanks @jun0jang for the PR.

7.2.51 3.7.0 (2020-07-08)

Deprecations:

- `marshmallow.pprint` is deprecated and will be removed in marshmallow 4 (#1588).

Support:

- Document `default_error_messages` on field classes (#1619). Thanks @weeix.

Bug fixes:

- Fix passing `only` and `exclude` to `Nested` with an ordered `Schema` (#1627). Thanks @juannorris for the PR.

7.2.52 3.6.1 (2020-06-02)

No code changes—only docs and contributor-facing updates in this release.

Support:

- Documentation: improve custom fields example (#1538). Thanks @pablospizzamiglio for reporting the problem with the old example and thanks @Resinderate for the PR.
- Documentation: Split up API reference into multiple pages and add summary tables (#1587). Thanks @EpicWink for the PR.

7.2.53 3.6.0 (2020-05-08)

Features:

- Add `validate.ContainsNoneOf` (#1528). Thanks @Resinderate for the suggestion and the PR.

7.2.54 3.5.2 (2020-04-30)

Bug fixes:

- Fix typing in `class_registry` (#1574). Thanks @mahenzon.

7.2.55 3.5.1 (2020-03-05)

Bug fixes:

- Includes bug fix from 2.21.0.

7.2.56 3.5.0 (2020-02-19)

Bug fixes:

- Fix list of nullable nested fields `List(Nested(Field, allow_none=True))` (#1497). Because this fix reverts an optimization introduced to speed-up serialization and deserialization of lists of nested fields, a negative impact on performance in this specific case is expected.

7.2.57 3.4.0 (2020-02-02)

Features:

- Improve type coverage (#1479). Thanks @Reskov.

Bug fixes:

- Fix typing for `data` param of `Schema.load` and `ValidationError` (#1492). Thanks @mehdigmira for reporting and thanks @dfirst for the PR.

Other changes:

- Remove unnecessary typecasts (#1500). Thanks @hukkinj1.
- Remove useless `_serialize` override in `UUID` field (#1489).

7.2.58 3.3.0 (2019-12-05)

Features:

- `fields.Nested` may take a callable that returns a schema instance. Use this to resolve order-of-declaration issues when schemas nest each other (#1146).

```
# <3.3
class AlbumSchema(Schema):
    title = fields.Str()
    artist = fields.Nested("ArtistSchema", only=("name",))

class ArtistSchema(Schema):
    name = fields.Str()
    albums = fields.List(fields.Nested(AlbumSchema))
```

(continues on next page)

(continued from previous page)

```
# >=3.3
class AlbumSchema(Schema):
    title = fields.Str()
    artist = fields.Nested(lambda: ArtistSchema(only=("name",)))

class ArtistSchema(Schema):
    name = fields.Str()
    albums = fields.List(fields.Nested(AlbumSchema))
```

Deprecations:

- Passing the string "self" to `fields.Nested` is deprecated. Use a callable instead.

```
from marshmallow import Schema, fields

# <3.3
class PersonSchema(Schema):
    partner = fields.Nested("self", exclude=("partner",))
    friends = fields.List(fields.Nested("self"))

# >=3.3
class PersonSchema(Schema):
    partner = fields.Nested(lambda: PersonSchema(exclude=("partner",)))
    friends = fields.List(fields.Nested(lambda: PersonSchema()))
```

Other changes:

- Fix typing for `Number._format_num` (#1466). Thanks @hukkinj1.
- Make mypy stricter and remove dead code (#1467). Thanks again, @hukkinj1.

7.2.59 3.2.2 (2019-11-04)

Bug fixes:

- Don't load fields for which `load_only` and `dump_only` are both `True` (#1448).
- Fix types in `marshmallow.validate` (#1446).

Support:

- Test against Python 3.8 (#1431).

7.2.60 3.2.1 (2019-09-30)

Bug fixes:

- Fix typing for `Schema.dump[s]` (#1416).

7.2.61 3.2.0 (2019-09-17)

Features:

- Add type annotations to `marshmallow.schema` and `marshmallow.validate` (#1407, #663).

Bug fixes:

- Fix compatibility with Python < 3.5.3 (#1409). Thanks @lukaszdudek-silvair for reporting.

Refactoring:

- Remove unnecessary `BaseSchema` superclass (#1406).

7.2.62 3.1.1 (2019-09-16)

Bug fixes:

- Restore inheritance hierarchy of `Number` fields (#1403). `fields.Integer` and `fields.Decimal` inherit from `fields.Number`.
- Fix bug that raised an uncaught error when a nested schema instance had an unpickleable object in its context (#1404). Thanks @metheoryt for reporting.

7.2.63 3.1.0 (2019-09-15)

Features:

- Add more type annotations (#663). Type information is distributed per PEP 561 . Thanks @fuhrysteve for helping with this.

Bug fixes:

- Includes bug fix from 2.20.5.

7.2.64 3.0.5 (2019-09-12)

Bug fixes:

- Fix bug that raised an uncaught error when passing both a schema instance and `only` to `Nested` (#1395). This bug also affected passing a schema instance to `fields.Pluck`.

7.2.65 3.0.4 (2019-09-11)

Bug fixes:

- Fix propagating `dot-delimited only` and `exclude` parameters to nested schema instances (#1384).
- Includes bug fix from 2.20.4 (#1160).

7.2.66 3.0.3 (2019-09-04)

Bug fixes:

- Handle when `data_key` is an empty string (#1378). Thanks @jtrakk for reporting.

7.2.67 3.0.2 (2019-09-04)

Bug fixes:

- Includes bug fix from 2.20.3 (#1376).
- Fix incorrect `super()` call in `SchemaMeta.__init__` (#1362).

7.2.68 3.0.1 (2019-08-21)

Bug fixes:

- Fix bug when nesting `fields.DateTimeField` within `fields.List` or `fields.Tuple` (#1357). This bug was introduced in 3.0.0rc9. Thanks @zblz for reporting.

7.2.69 3.0.0 (2019-08-18)

Features:

- Optimize `List(Nested(...))` (#779).
- Minor performance improvements and cleanup (#1328).
- Add `Schema.from_dict` (#1312).

Deprecations/Removals:

- `Field.fail` is deprecated. Use `Field.make_error` instead.
- Remove UUID validation from `fields.UUID`, for consistency with other fields (#1132).

Support:

- Various docs improvements (#1329).

7.2.70 3.0.0rc9 (2019-07-31)

Features:

- *Backwards-incompatible*: Validation does not occur on serialization (#1132). This significantly improves serialization performance.
- *Backwards-incompatible*: `DateTime` does not affect timezone information on serialization and deserialization (#1234, #1278).
- Add `NaiveDateTime` and `AwareDateTime` to enforce timezone awareness (#1234, #1287).
- *Backwards-incompatible*: `List` does not wrap single values in a list on serialization (#1307).
- *Backwards-incompatible*: `Schema.handle_error` receives `many` and `partial` as keyword arguments (#1321).
- Use `raise` from more uniformly to improve stack traces (#1313).
- Rename `Nested.__schema` to `Nested._schema` to prevent name mangling (#1289).
- Performance improvements (#1309).

Deprecations/Removals:

- `LocalDateTime` is removed (#1234).
- `marshmallow.utils.utc` is removed. Use `datetime.timezone.utc` instead.

Bug fixes:

- Fix behavior of `List(Nested("self"))` (#779 (comment)).

Support:

- Document usage of `validate.Regexp`'s usage re. `search` (#1285). Thanks @macdonaldezra.

7.2.71 3.0.0rc8 (2019-07-04)

Features:

- Propagate `only` and `exclude` parameters to Nested fields within `List` and `Dict` (#779, #946).
- Use `email.utils.parsedate_to_datetime` instead of conditionally using `dateutil` for parsing RFC dates (#1246).
- Use internal util functions instead of conditionally using `dateutil` for parsing ISO 8601 datetimes, dates and times. Timezone info is now correctly deserialized whether or not `dateutil` is installed. (#1265)
- Improve error messages for `validate.Range`.
- Use `raise from error` for better stack traces (#1254). Thanks @fuhrysteve.
- `python-dateutil` is no longer used. This resolves the inconsistent behavior based on the presence of `python-dateutil` (#497, #1234).

Bug fixes:

- Fix method resolution for `__init__` method of `fields.Email` and `fields.URL` (#1268). Thanks @dursk for the catch and patch.
- Includes bug fixes from 2.19.4 and 2.19.5.

Other changes:

- *Backwards-incompatible:* Rename `fields.List.container` to `fields.List.inner`, `fields.Dict.key_container` to `fields.Dict.key_field`, and `fields.Dict.value_container` to `fields.Dict.value_field`.
- Switch to Azure Pipelines for CI (#1261).

7.2.72 3.0.0rc7 (2019-06-15)

Features:

- *Backwards-incompatible:* `many` is passed as a keyword argument to methods decorated with `pre_load`, `post_load`, `pre_dump`, `post_dump`, and `validates_schema`. `partial` is passed as a keyword argument to methods decorated with `pre_load`, `post_load` and `validates_schema`. `**kwargs` should be added to all decorated methods.
- Add `min_inclusive` and `max_exclusive` parameters to `validate.Range` (#1221). Thanks @kdop for the PR.

Bug fixes:

- Fix propagation of `partial` to Nested containers (part of #779).
- Includes bug fix from 2.19.3.

Other changes:

- *Backwards-incompatible:* Use keyword-only arguments (#1216).

7.2.73 3.0.0rc6 (2019-05-05)

Support:

- *Backwards-incompatible:* Remove support for Python 2 (#1120). Only Python >= 3.5 is supported. Thank you @rooterkyberian for the suggestion and the PR.
- *Backwards-incompatible:* Remove special-casing in `fields.List` and `fields.Tuple` for accessing nested attributes (#1188). Use `fields.List(fields.Pluck(...))` instead.

- Add `python_requires` to `setup.py` (#1194). Thanks @hugovk.
- Upgrade syntax with `pyupgrade` in pre-commit (#1195). Thanks again @hugovk.

7.2.74 3.0.0rc5 (2019-03-30)

Features:

- Allow input value to be included in error messages for a number of fields (#1129). Thanks @hdoupe for the PR.
- Improve default error messages for `OneOf` and `ContainsOnly` (#885). Thanks @mcgfeller for the suggestion and @maxalbert for the PR.

Deprecations/Removals:

- Remove `fields.FormattedString` (#1141). Use `fields.Function` or `fields.Method` instead.

Bug fixes:

- Includes bug fix from 2.19.2.

7.2.75 3.0.0rc4 (2019-02-08)

Features:

- Add `fields.Tuple` (#1103) Thanks @zblz for the PR.
- Add `fields.Mapping`, which makes it easier to support other mapping types (e.g. `OrderedDict`) (#1092). Thank @sayanarjit for the suggestion and the PR.

7.2.76 3.0.0rc3 (2019-01-13)

Features:

- Make the error messages for “unknown fields” and “invalid data type” configurable (#852). Thanks @Dunstrom for the PR.
- `fields.Boolean` parses “yes”/“no” values (#1081). Thanks @r1b.

Other changes:

- *Backwards-incompatible with previous 3.x versions*: Change ordering of keys and values arguments to `fields.Dict`.
- Remove unused code in `marshmallow.utils`: `is_indexable_but_not_string`, `float_to_decimal`, `decimal_to_fixed`, `from_iso` (#1088).
- Remove unused `marshmallow.compat.string_types`.

Bug fixes:

- Includes bug fix from 2.18.0.

7.2.77 3.0.0rc2 (2019-01-03)

Features:

- Add `register class Meta` option to allow bypassing marshmallow’s internal class registry when memory usage is critical (#660).

Bug fixes:

- Fix serializing dict-like objects with properties (#1060). Thanks @taion for the fix.
- Fix populating `ValidationError.valid_data` for `List` and `Dict` fields (#766).

Other changes:

- Add `marshmallow.__version_info__` (#1074).
- Remove the `marshmallow.marshalling` internal module (#1070).
- A `ValueError` is raised when the missing parameter is passed for required fields (#1040).
- Extra keyword arguments passed to `ValidationError` in validators are no longer passed to the final `ValidationError` raised upon validation completion (#996).

7.2.78 3.0.0rc1 (2018-11-29)

Features:

- *Backwards-incompatible*: Rework `ValidationError` API. It now expects a single field name, and error structures are merged in the final `ValidationError` raised when validation completes. This allows schema-level validators to raise errors for individual fields (#441). Thanks @maximkulkin for writing the original `merge_errors` implementation in #442 and thanks @lafrech for completing the implementation in #1026.

Bug fixes:

- Fix `TypeError` when serializing `None` with `Pluck` (#1049). Thanks @toffan for the catch and patch.

7.2.79 3.0.0b20 (2018-11-01)

Bug fixes:

- Includes bug fixes from 2.16.2 and 2.16.3.

7.2.80 3.0.0b19 (2018-10-24)

Features:

- Support partial loading of nested fields (#438). Thanks @arbor-dwatson for the PR. *Note*: Subclasses of `fields`. Nested now take an additional `partial` parameter in the `_deserialize` method.

Bug fixes:

- Restore `Schema.TYPE_MAPPING`, which was removed in 3.0.0b17 (#1012).

Other changes:

- *Backwards-incompatible*: `_serialize` and `_deserialize` methods of all `fields.Field` subclasses must accept `**kwargs` (#1007).

7.2.81 3.0.0b18 (2018-10-15)

Bug fixes:

- Fix `Date` deserialization when using custom format (#1001). Thanks @Ondkloss for reporting.

Deprecations/Removals:

- `prefix` parameter or `Schema` class is removed (#991). The same can be achieved using a `@post_dump` method.

7.2.82 3.0.0b17 (2018-10-13)

Features:

- Add `format` option to `Date` field (#869).
- *Backwards-incompatible*: Rename `DateTime`'s `dateformat` Meta option to `datetimeformat`. `dateformat` now applies to `Date` (#869). Thanks @knagra for implementing these changes.

- Enforce ISO 8601 when deserializing date and time (#899). Thanks @dushr for the report and the work on the PR.
- *Backwards-incompatible*: Raise `ValueError` on Schema instantiation in case of `attribute` or `data_key` collision (#992).

Bug fixes:

- Fix inconsistencies in field inference by refactoring the inference feature into a dedicated field (#809). Thanks @taion for the PR.
- When `unknown` is not passed to `Nested`, default to nested Schema `unknown` meta option rather than `RAISE` (#963). Thanks @vgavro for the PR.
- Fix loading behavior of `fields.Pluck` (#990).
- Includes bug fix from 2.16.0.

7.2.83 3.0.0b16 (2018-09-20)

Bug fixes:

- Fix root attribute for nested container fields on inheriting schemas (#956). Thanks @bmcbu for reporting.

7.2.84 3.0.0b15 (2018-09-18)

Bug fixes:

- Raise `ValidationError` instead of `TypeError` when non-iterable types are validated with `many=True` (#851).
- `many=True` no longer iterates over `str` and `collections.abc.Mapping` objects and instead raises a `ValidationError` with `{'_schema': ['Invalid input type.']}` (#930).
- Return `[]` as `ValidationError.valid_data` instead of `{}` when `many=True` (#907).

Thanks @tuukkamustonen for implementing these changes.

7.2.85 3.0.0b14 (2018-09-15)

Features:

- Add `fields.Pluck` for serializing a single field from a nested object (#800). Thanks @time13 for the feedback and @deckar01 for the implementation.
- *Backwards-incompatible*: Passing a string argument as `only` to `fields.Nested` is no longer supported. Use `fields.Pluck` instead (#800).
- Raise a `StringNotCollectionError` if `only` or `exclude` is passed as a string to `fields.Nested` (#931).
- *Backwards-incompatible*: `Float` takes an `allow_nan` parameter to explicitly allow serializing and deserializing special values (`nan`, `inf` and `-inf`). `allow_nan` defaults to `False`.

Other changes:

- *Backwards-incompatible*: `Nested` field now defaults to `unknown=RAISE` instead of `EXCLUDE`. This harmonizes behavior with `Schema` that already defaults to `RAISE` (#908). Thanks @tuukkamustonen.
- Tested against Python 3.7.

7.2.86 3.0.0b13 (2018-08-04)

Bug fixes:

- Errors reported by a schema-level validator for a field in a `Nested` field are stored under corresponding field name, not `_schema` key (#862).
- Includes bug fix from 2.15.4.

Other changes:

- *Backwards-incompatible*: The `unknown` option now defaults to `RAISE` (#524 (comment), #851).
- *Backwards-incompatible*: When a schema error is raised with a `dict` as payload, the `dict` overwrites any existing error list. Before this change, it would be appended to the list.
- Raise a `StringNotCollectionError` if `only` or `exclude` is passed as a string (#316). Thanks @paulocheque for reporting.

7.2.87 3.0.0b12 (2018-07-04)

Features:

- The behavior to apply when encountering unknown fields while deserializing can be controlled with the `unknown` option (#524, #747, #127). It makes it possible to either “include”, “exclude”, or “raise”. Thanks @tuukkamus-tonen for the suggestion and thanks @ramnes for the PR.

Warning

The default for `unknown` will be changed to `RAISE` in the next release.

Other changes:

- *Backwards-incompatible*: Pre/Post-processors **MUST** return modified data. Returning `None` does not imply data were mutated (#347). Thanks @tdevelioglu for reporting.
- *Backwards-incompatible*: `only` and `exclude` are bound by declared and additional fields. A `ValueError` is raised if invalid fields are passed (#636). Thanks @jan-23 for reporting. Thanks @ikilledthecat and @deckar01 for the PRs.
- Format code using pre-commit (#855).

Deprecations/Removals:

- `ValidationError.fields` is removed (#840). Access field instances from `Schema.fields`.

7.2.88 3.0.0b11 (2018-05-20)

Features:

- Clean up code for schema hooks (#814). Thanks @taion.
- Minor performance improvement from simplifying `utils.get_value` (#811). Thanks again @taion.
- Add `require_tld` argument to `fields.URL` (#749). Thanks @DenerKup for reporting and thanks @surik00 for the PR.
- `fields.UUID` deserializes bytes strings using `UUID(bytes=b'...')` (#625). Thanks @JeffBerger for the suggestion and the PR.

Bug fixes:

- Fields nested within `Dict` correctly inherit context from their parent schema (#820). Thanks @RosanneZe for reporting and @deckar01 for the PR.
- Includes bug fix from 2.15.3.

7.2.89 3.0.0b10 (2018-05-10)

Bug fixes:

- Includes bugfixes from 2.15.2.

7.2.90 3.0.0b9 (2018-04-25)

Features:

- *Backwards-incompatible*: missing and default values are passed in deserialized form (#378). Thanks @chadrik for the suggestion and thanks @lafrech for the PR.

Bug fixes:

- Includes the bugfix from 2.15.1.

7.2.91 3.0.0b8 (2018-03-24)

Features:

- *Backwards-incompatible*: Add `data_key` parameter to fields for specifying the key in the input and output data dict. This parameter replaces both `load_from` and `dump_to` (#717). Thanks @lafrech.
- *Backwards-incompatible*: When `pass_original=True` is passed to one of the decorators and a collection is being (de)serialized, the `original_data` argument will be a single object unless `pass_many=True` is also passed to the decorator (#315, #743). Thanks @stj for the PR.
- *Backwards-incompatible*: Don't recursively check nested required fields when the `Nested` field's key is missing (#319). This reverts #235. Thanks @chekunkov reporting and thanks @lafrech for the PR.
- *Backwards-incompatible*: Change error message collection for `Dict` field (#730). Note: this is backwards-incompatible with previous 3.0.0bX versions. Thanks @shabble for the report and thanks @lafrech for the PR.

7.2.92 3.0.0b7 (2018-02-03)

Features:

- *Backwards-incompatible*: Schemas are always strict (#377). The `strict` parameter is removed.
- *Backwards-incompatible*: `Schema().load` and `Schema().dump` return data instead of a `(data, errors)` tuple (#598).
- *Backwards-incompatible*: `Schema().load(None)` raises a `ValidationError` (#511).

See *Upgrading to 3.0* for a guide on updating your code.

Thanks @lafrech for implementing these changes. Special thanks to @MichalKononenko, @douglas-treadwell, and @maximkulkin for the discussions on these changes.

Other changes:

- *Backwards-incompatible*: Field name is not checked when `load_from` is specified (#714). Thanks @lafrech.

Support:

- Add Code of Conduct.

7.2.93 3.0.0b6 (2018-01-02)

Bug fixes:

- Fixes `ValidationError.valid_data` when a nested field contains errors (#710). This bug was introduced in 3.0.0b3. Thanks @lafrech.

Other changes:

- *Backwards-incompatible*: `Email` and `URL` fields don't validate on serialization (#608). This makes them more consistent with the other fields and improves serialization performance. Thanks again @lafrech.
- `validate.URL` requires square brackets around IPv6 URLs (#707). Thanks @harlov.

7.2.94 3.0.0b5 (2017-12-30)

Features:

- Add support for structured dictionaries by providing values and keys arguments to the `Dict` field's constructor. This mirrors the `List` field's ability to validate its items (#483). Thanks @deckar01.

Other changes:

- *Backwards-incompatible*: `utils.from_iso` is deprecated in favor of `utils.from_iso_datetime` (#694). Thanks @sklarsa.

7.2.95 3.0.0b4 (2017-10-23)

Features:

- Add support for millisecond, minute, hour, and week precisions to `fields.TimeDelta` (#537). Thanks @Fedalto for the suggestion and the PR.
- Includes features from release 2.14.0.

Support:

- Copyright year in docs uses `CHANGELOG.rst`'s modified date for reproducible builds (#679). Thanks @bmwiedemann.
- Test against Python 3.6 in tox. Thanks @Fedalto.
- Fix typo in exception message (#659). Thanks @wonderbeyond for reporting and thanks @yoichi for the PR.

7.2.96 3.0.0b3 (2017-08-20)

Features:

- Add `valid_data` attribute to `ValidationError`.
- Add `strict` parameter to `Integer` (#667). Thanks @yoichi.

Deprecations/Removals:

- Deprecate `json_module` option in favor of `render_module` (#364, #130). Thanks @justanr for the suggestion.

Bug fixes:

- Includes bug fixes from releases 2.13.5 and 2.13.6.
- *Backwards-incompatible*: `Number` fields don't accept booleans as valid input (#623). Thanks @tuukkamustonen for the suggestion and thanks @rowillia for the PR.

Support:

- Add benchmark script. Thanks @rowillia.

7.2.97 3.0.0b2 (2017-03-19)

Features:

- Add `truthy` and `falsy` params to `fields.Boolean` (#580). Thanks @zwack for the PR. Note: This is potentially a breaking change if your code passes the default parameter positionally. Pass `default` as a keyword argument instead, e.g. `fields.Boolean(default=True)`.

Other changes:

- *Backwards-incompatible*: `validate.ContainsOnly` allows empty and duplicate values (#516, #603). Thanks @maximkulkin for the suggestion and thanks @lafrech for the PR.

Bug fixes:

- Includes bug fixes from release 2.13.4.

7.2.98 3.0.0b1 (2017-03-10)

Features:

- `fields.Nested` respects `only='field'` when deserializing (#307). Thanks @erlingbo for the suggestion and the PR.
- `fields.Boolean` parses "on"/"off" (#580). Thanks @marcellarius for the suggestion.

Other changes:

- Includes changes from release 2.13.2.
- *Backwards-incompatible*: `skip_on_field_errors` defaults to `True` for `validates_schema` (#352).

7.2.99 3.0.0a1 (2017-02-26)

Features:

- `dump_only` and `load_only` for `Function` and `Method` are set based on `serialize` and `deserialize` arguments (#328).

Other changes:

- *Backwards-incompatible*: `fields.Method` and `fields.Function` no longer swallow `AttributeErrors` (#395). Thanks @bereal for the suggestion.
- *Backwards-incompatible*: `validators.Length` is no longer a subclass of `validators.Range` (#458). Thanks @deckar01 for the catch and patch.
- *Backwards-incompatible*: `utils.get_func_args` no longer returns bound arguments. This is consistent with the behavior of `inspect.signature`. This change prevents a `DeprecationWarning` on Python 3.5 (#415, #479). Thanks @deckar01 for the PR.
- *Backwards-incompatible*: Change the signature of `utils.get_value` and `Schema.get_attribute` for consistency with Python builtins (e.g. `getattr`) (#341). Thanks @stas for reporting and thanks @deckar01 for the PR.
- *Backwards-incompatible*: Don't unconditionally call callable attributes (#430, reverts #242). Thanks @mirko for the suggestion.
- Drop support for Python 2.6 and 3.3.

Deprecation/Removals:

- Remove `__error_handler__`, `__accessor__`, `@Schema.error_handler`, and `@Schema.accessor`. Override `Schema.handle_error` and `Schema.get_attribute` instead.

- Remove `func` parameter of `fields.Function`. Remove `method_name` parameter of `fields.Method` (issue:325). Use the `serialize` parameter instead.
- Remove extra parameter from `Schema`. Use a `@post_dump` method to add additional data.

7.2.100 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).

7.2.101 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).

7.2.102 2.20.4 (2019-09-11)

Bug fixes:

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

7.2.103 2.20.3 (2019-09-04)

Bug fixes:

- Don't swallow `TypeError` exceptions raised by `Field._bind_to_schema` or `Schema.on_bind_field` (#1376).

7.2.104 2.20.2 (2019-08-20)

Bug fixes:

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

7.2.105 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).

7.2.106 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/Removals:

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

7.2.107 2.19.5 (2019-06-18)

Bug fixes:

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

7.2.108 2.19.4 (2019-06-16)

Bug fixes:

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

7.2.109 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.

7.2.110 2.19.2 (2019-03-30)

Bug fixes:

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

7.2.111 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.

7.2.112 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).

7.2.113 2.18.1 (2019-02-15)

Bug fixes:

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

7.2.114 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).

7.2.115 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).

7.2.116 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.

7.2.117 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.

7.2.118 2.16.1 (2018-10-17)

Bug fixes:

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

7.2.119 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).

7.2.120 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 @bmcibu for reporting.

These fixes were backported from 3.0.0b15 and 3.0.0b16.

7.2.121 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.

7.2.122 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.

7.2.123 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.

7.2.124 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.

7.2.125 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.

7.2.126 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.

7.2.127 2.14.0 (2017-10-23)

Features:

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

7.2.128 2.13.6 (2017-08-16)

Bug fixes:

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

7.2.129 2.13.5 (2017-04-12)

Bug fixes:

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

7.2.130 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.

7.2.131 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.

7.2.132 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.

7.2.133 2.13.1 (2017-03-04)

Bug fixes:

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

7.2.134 2.13.0 (2017-02-18)

Features:

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

7.2.135 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.

7.2.136 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.

7.2.137 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.

7.2.138 2.11.1 (2017-01-08)

Bug fixes:

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

7.2.139 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 @mindjo-victor.

Support:

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

7.2.140 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.

7.2.141 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.

7.2.142 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.

7.2.143 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.

7.2.144 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.

7.2.145 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.

7.2.146 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.

7.2.147 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.

7.2.148 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.

7.2.149 2.7.3 (2016-05-05)

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

7.2.150 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.

7.2.151 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).

7.2.152 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.

7.2.153 2.6.1 (2016-03-17)

Bug fixes:

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

7.2.154 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.

7.2.155 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).

7.2.156 2.4.2 (2015-12-08)

Bug fixes:

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

7.2.157 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.

7.2.158 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.

7.2.159 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.

7.2.160 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.

7.2.161 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.

7.2.162 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.

7.2.163 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.

7.2.164 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.

7.2.165 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.

7.2.166 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.

7.2.167 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.

7.2.168 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.

7.2.169 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 @lex-toumbourou for reporting and @mwstobo for sending the patch.

7.2.170 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.

7.2.171 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.

7.2.172 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.

7.2.173 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`.

7.2.174 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 @nickreilack 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 @nickreilack for the suggestion.

7.2.175 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).

7.2.176 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.

7.2.177 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.

7.2.178 1.2.3 (2015-03-15)

Bug fixes:

- Fix inheritance of `ordered` class Meta option (#162). Thanks @stephenfin for reporting.

7.2.179 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.

7.2.180 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.

7.2.181 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 @amikhola 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.

7.2.182 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.

7.2.183 1.0.1 (2014-11-18)

Hotfix release.

- Ensure that errors dictionary is correctly cleared on each call to `Schema.dump` and `Schema.load`.

7.2.184 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.

7.2.185 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.

7.2.186 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.

7.2.187 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.

7.2.188 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.

7.2.189 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.

7.2.190 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.

7.2.191 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.

7.2.192 0.5.1 (2014-02-02)

- Avoid unnecessary attribute access in `utils.to_marshallable_type` for improved performance.
- Fix RFC822 formatting for localized datetimes.

7.2.193 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.

7.2.194 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.

7.2.195 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.

7.2.196 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 + SQLAlchemy example in docs.

7.2.197 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`.

7.2.198 0.2.1 (2013-11-12)

- Allow prefixing field names.
- Fix storing errors on Nested `Serializers`.
- Python 2.6 support.

7.2.199 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` parameter to `fields.Url` that allows for relative URLs.

7.2.200 0.1.0 (2013-11-10)

- First release.

7.3 Upgrading to newer releases

This section documents migration paths to new releases.

7.3.1 Upgrading to 4.0

Field as a generic class

`Field` is a generic class with a type argument. When defining a custom field, the type argument should be used to specify the internal type.

```

from marshmallow import fields

class PinCode(fields.Field[list[int]]):
    """Field that serializes to a string of numbers and deserializes
    to a list of numbers.
    """

    def _serialize(self, value, attr, obj, **kwargs):
        if value is None:
            return ""
        return "".join(str(d) for d in value)

    # The inferred return type is list[int]
    def _deserialize(self, value, attr, data, **kwargs):
        try:
            return [int(c) for c in value]
        except ValueError as error:
            raise ValidationError("Pin codes must contain only digits.") from error

```

New context API

Passing context to *Schema* classes is removed. Use `contextvars.ContextVar` for passing context to fields, pre/post-processing methods, and validators instead.

marshmallow 4 provides an experimental *Context* wrapper around `contextvars.ContextVar` that can be used to both set and retrieve context.

```

# 3.x
from marshmallow import Schema, fields

class UserSchema(Schema):
    name_suffixed = fields.Function(
        lambda obj, context: obj["name"] + context["suffix"]
    )

user_schema = UserSchema()
user_schema.context = {"suffix": "bar"}
user_schema.dump({"name": "foo"})
# {'name_suffixed': 'foobar'}

# 4.x
import typing

from marshmallow import Schema, fields
from marshmallow.experimental.context import Context

class UserContext(typing.TypedDict):
    suffix: str

```

(continues on next page)

(continued from previous page)

```

UserSchemaContext = Context[UserContext]

class UserSchema(Schema):
    name_suffixed = fields.Function(
        lambda obj: obj["name"] + UserSchemaContext.get()["suffix"]
    )

with UserSchemaContext({"suffix": "bar"}):
    UserSchema().dump({"name": "foo"})
    # {'name_suffixed': 'foobar'}

```

See *Using context* for more information.

Implicit field creation is removed

In marshmallow 3, the `fields` and additional class `Meta` options allowed fields to be implicitly created via introspection of the data being serialized.

In marshmallow 4.0, implicit field creation is removed to prevent conflicts with libraries that generate fields dynamically.

```

import datetime as dt
import dataclasses

from marshmallow import Schema, fields

@dataclasses.dataclass
class User:
    name: str
    birthdate: dt.date

# 3.x
class UserSchema(Schema):
    class Meta:
        fields = ("name", "birthdate")

# 4.x
class UserSchema(Schema):
    name = fields.String()
    email = fields.Date()

```

To automatically generate schema fields from model classes, consider using a separate library, e.g. `marshmallow-sqlalchemy` for SQLAlchemy models.

```

from marshmallow_sqlalchemy import SQLAlchemySchema, auto_field

class UserSchema(SQLAlchemySchema):

```

(continues on next page)

(continued from previous page)

```
class Meta:
    model = Author

    name = auto_field()
    birthdate = auto_field()
```

@validates accepts multiple field names

The `@validates` decorator now accepts multiple field names as arguments. Decorated methods receive `data_key` as a keyword argument.

```
from marshmallow import fields, Schema, validates

# 3.x
class UserSchema(Schema):
    name = fields.Str(required=True)
    nickname = fields.Str(required=True)

    @validates("name")
    def validate_name(self, value: str) -> None:
        if len(value) < 3:
            raise ValidationError("name" too short')

    @validates("nickname")
    def validate_nickname(self, value: str) -> None:
        if len(value) < 3:
            raise ValidationError("nickname" too short')

# 4.x
class UserSchema(Schema):
    name = fields.Str(required=True)
    nickname = fields.Str(required=True)

    @validates("name", "nickname")
    def validate_names(self, value: str, data_key: str) -> None:
        if len(value) < 3:
            raise ValidationError(f'"{data_key}" too short')
```

Remove ordered from the SchemaOpts constructor

Subclasses of `marshmallow.SchemaOpts` should remove the `ordered` argument from the constructor.

```
# 3.x
class CustomOpts(SchemaOpts):
    def __init__(self, meta, ordered=False):
        super().__init__(meta, ordered=ordered)
        self.custom_option = getattr(meta, "custom_option", False)

# 4.x
```

(continues on next page)

(continued from previous page)

```
class CustomOpts(SchemaOpts):
    def __init__(self, meta):
        super().__init__(meta)
        self.custom_option = getattr(meta, "custom_option", False)
```

TimeDelta changes

The `TimeDelta` field now preserves float values such that microseconds are included in the resulting `datetime.datetime.timedelta` object.

```
from marshmallow import fields

field = fields.TimeDelta()
value = field.deserialize(12.9)

# 3.x
print(value) # => datetime.timedelta(seconds=12)

# 4.x
print(value) # => datetime.timedelta(seconds=12, microseconds=9000000)
```

The `serialization_type` parameter has been removed. Use a custom field or cast the serialized value if you need to change the final output type.

pass_many is renamed to pass_collection in decorators

The `pass_many` argument to `pre_load`, `post_load`, `pre_dump`, and `post_dump` is renamed to `pass_collection`.

The behavior is unchanged.

```
from marshmallow import Schema, fields, post_load

# 3.x
class MySchema(Schema):
    name = fields.Str()

    @post_dump(pass_many=True)
    def post_dump(self, data, many, **kwargs): ...

# 4.x
class MySchema(Schema):
    name = fields.Str()

    @post_dump(pass_collection=True)
    def post_dump(self, data, many, **kwargs): ...
```

Rename schema to parent in _bind_to_schema

Custom fields that define a `_bind_to_schema` method should rename the `schema` argument to `parent`.

```

from marshmallow import fields

# 3.x
class MyField(fields.Field):
    def _bind_to_schema(self, field_name: str, schema: Schema | Field): ...

# 4.x
class MyField(fields.Field[int]):
    def _bind_to_schema(self, field_name: str, parent: Schema | Field): ...

```

Use standard library functions to handle RFC 822 and ISO 8601 dates, times, and datetimes

ISO 8601 and RFC 822 utilities are removed from *marshmallow.utils* in favor of using the standard library implementations.

```

# 3.x
import datetime as dt
from marshmallow.utils import (
    from_iso_date,
    from_iso_time,
    from_iso_datetime,
    to_iso_date,
    to_iso_time,
    isoformat,
    from_rfc,
    rfc_format,
)

from_iso_date("2013-11-10")
from_iso_time("01:23:45")
from_iso_datetime("2013-11-10T01:23:45")
to_iso_date(dt.date(2013, 11, 10))
to_iso_time(dt.time(1, 23, 45))
isoformat(dt.datetime(2013, 11, 10, 1, 23, 45))
from_rfc("Sun, 10 Nov 2013 01:23:45 -0000")
rfc_format(dt.datetime(2013, 11, 10, 1, 23, 45))

# 4.x
import datetime as dt
import email.utils

dt.date.fromisoformat("2013-11-10")
dt.time.fromisoformat("01:23:45")
dt.datetime.fromisoformat("2013-11-10T01:23:45")
dt.date(2013, 11, 10).isoformat()
dt.time(1, 23, 45).isoformat()
dt.datetime(2013, 11, 10, 1, 23, 45).isoformat()
email.utils.parsedate_to_datetime("Sun, 10 Nov 2013 01:23:45 -0000")
email.utils.format_datetime(dt.datetime(2013, 11, 10, 1, 23, 45))

```

7.3.2 Upgrading to 3.26

ordered is deprecated

The *ordered* class Meta option is removed, since order is already preserved by default.

```
from marshmallow import Schema, fields

# <3.26
class MySchema(Schema):
    id = fields.Integer()

    class Meta:
        ordered = True

# >=3.26
class MySchema(Schema):
    id = fields.Integer()
```

Note

You can set `marshmallow.Schema.dict_class` to `collections.OrderedDict` to force the output type of `marshmallow.Schema.dump` to be an `OrderedDict`.

7.3.3 Upgrading to 3.24

Field usage

Field is the base class for all fields and should not be used directly within schemas. Only use subclasses of *Field* in your schemas. Instantiating *Field* will raise a warning in marshmallow>=3.24 and an error in marshmallow 4.0.

```
from marshmallow import Schema, fields

# <3.24
class UserSchema(Schema):
    name = fields.Field()

# >=3.24
class UserSchema(Schema):
    name = fields.String()
```

Number and Mapping fields as base classes

Number and *Mapping* are base classes that should not be used within schemas. Use their subclasses instead. Instantiating *Number* or *Mapping* will raise a warning in marshmallow>=3.24 and an error in marshmallow 4.0.

```
from marshmallow import Schema, fields
```

(continues on next page)

(continued from previous page)

```
# <3.24
class PackageSchema(Schema):
    revision = fields.Number()
    dependencies = fields.Mapping()

# >=3.24
class PackageSchema(Schema):
    revision = fields.Integer()
    dependencies = fields.Dict()
```

Validators must raise a *ValidationError*

Validators must raise a *ValidationError* when the value is invalid. Returning `False` from a validator is deprecated and will be removed in marshmallow 4.0.

```
from marshmallow import Schema, fields

# <3.24
class UserSchema(Schema):
    password = fields.String(validate=lambda x: x == "password")

# >=3.24
def validate_password(val):
    if val != "password":
        raise ValidationError("Invalid password.")

class UserSchema(Schema):
    password = fields.String(validate=validate_password)
```

If you want to use anonymous functions, you can use this helper function in your code.

```
from typing import Any, Callable

from marshmallow import Schema, fields

def predicate(
    func: Callable[[Any], bool],
) -> Callable[[Any], None]:
    def validate(value: Any) -> None:
        if func(value) is False:
            raise ValidationError("Invalid value.")

    return validate

# Usage
class UserSchema(Schema):
    password = fields.String(validate=predicate(lambda x: x == "password"))
```

context is deprecated

Passing context to *Schema* classes will raise a warning in marshmallow \geq 3.24 and will be removed in marshmallow 4.0. Use `contextvars.ContextVar` for passing context to fields, *pre-/post-processing methods*, and *validators* instead.

7.3.4 Upgrading to 3.13

7.3.5 load_default and dump_default

The missing and default parameters of fields are renamed to `load_default` and `dump_default`, respectively.

```
from marshmallow import Schema, fields

# < 3.13
class MySchema(Schema):
    name = fields.Str(missing="Monty")
    age = fields.Int(default=42)

# >=3.13
class MySchema(Schema):
    name = fields.Str(load_default="Monty")
    age = fields.Int(dump_default=42)
```

`load_default` and `dump_default` are passed to the field constructor as keyword arguments.

7.3.6 Upgrading to 3.3

In 3.3, *fields.Nested* may take a callable that returns a schema instance. Use this to resolve order-of-declaration issues when schemas nest each other.

```
from marshmallow import Schema, fields

# <3.3
class AlbumSchema(Schema):
    title = fields.Str()
    artist = fields.Nested("ArtistSchema", only=("name",))

class ArtistSchema(Schema):
    name = fields.Str()
    albums = fields.List(fields.Nested(AlbumSchema))

# >=3.3
class AlbumSchema(Schema):
    title = fields.Str()
    artist = fields.Nested(lambda: ArtistSchema(only=("name",)))

class ArtistSchema(Schema):
```

(continues on next page)

(continued from previous page)

```
name = fields.Str()
albums = fields.List(fields.Nested(AlbumSchema))
```

A callable should also be used when nesting a schema within itself. Passing "self" is deprecated.

```
from marshmallow import Schema, fields

# <3.3
class PersonSchema(Schema):
    partner = fields.Nested("self", exclude=("partner",))
    friends = fields.List(fields.Nested("self"))

# >=3.3
class PersonSchema(Schema):
    partner = fields.Nested(lambda: PersonSchema(exclude=("partner",)))
    friends = fields.List(fields.Nested(lambda: PersonSchema()))
```

7.3.7 Upgrading to 3.0

Python compatibility

The marshmallow 3.x series requires Python 3.

Schemas are always strict

Two major changes were made to (de)serialization behavior:

- The `strict` parameter was removed. Schemas are always strict.
- `Schema().load` and `Schema().dump` don't return a `(data, errors)` tuple any more. Only data is returned.

If invalid data are passed, a `ValidationError` is raised. The dictionary of validation errors is accessible from the `ValidationError.messages` attribute, along with the valid data from the `ValidationError.valid_data` attribute.

```
from marshmallow import ValidationError

# 2.x
schema = UserSchema()
data, errors = schema.load({"name": "Monty", "email": "monty@python.org"})
# OR
schema = UserSchema(strict=True)
try:
    data, _ = schema.load({"name": "Monty", "email": "monty@python.org"})
except ValidationError as err:
    errors = err.messages
    valid_data = err.valid_data

# 3.x
schema = UserSchema()
# There is only one right way
try:
```

(continues on next page)

(continued from previous page)

```

data = schema.load({"name": "Monty", "email": "monty@python.org"})
except ValidationError as err:
    errors = err.messages
    valid_data = err.valid_data

```

`Schema.validate` always returns a dictionary of validation errors (same as 2.x with `strict=False`).

```

schema.validate({"email": "invalid"})
# {'email': ['Not a valid email address.']}

```

Setting the `strict` option on `class Meta` has no effect on `Schema` behavior. Passing `strict=True` or `strict=False` to the `Schema` constructor will raise a `TypeError`.

```

# 3.x
UserSchema(strict=True)
# TypeError: __init__() got an unexpected keyword argument 'strict'

```

➔ See also

See GitHub issues [#377](#) and [#598](#) for the discussions on this change.

Decorated methods and `handle_error` receive many and partial

Methods decorated with `pre_load`, `post_load`, `pre_dump`, `post_dump`, and `validates_schema` receive many as a keyword argument. In addition, `pre_load`, `post_load`, and `validates_schema` receive `partial`. To account for these additional arguments, add `**kwargs` to your methods.

```

# 2.x
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

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

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

```

`Schema.handle_error` also receives many and `partial` as keyword arguments.

```

# 2.x
class UserSchema(Schema):

```

(continues on next page)

(continued from previous page)

```

def handle_error(self, exc, data):
    raise AppError("An error occurred with input: {0}".format(data))

# 3.x
class UserSchema(Schema):
    def handle_error(self, exc, data, **kwargs):
        raise AppError("An error occurred with input: {0}".format(data))

```

Validation does not occur on serialization

`Schema.dump` will no longer validate and collect error messages. You *must* validate your data before serializing it.

```

from marshmallow import Schema, fields, ValidationError

invalid_data = dict(created_at="invalid")

class WidgetSchema(Schema):
    created_at = fields.DateTime()

# 2.x
WidgetSchema(strict=True).dump(invalid_data)
# marshmallow.exceptions.ValidationError: {'created_at': ['"invalid" cannot be formatted
↳ as a datetime.']}

# 3.x
WidgetSchema().dump(invalid_data)
# AttributeError: 'str' object has no attribute 'isoformat'

# Instead, validate before dumping
schema = WidgetSchema()
try:
    widget = schema.load(invalid_data)
except ValidationError:
    print("handle errors...")
else:
    dumped = schema.dump(widget)

```

Deserializing invalid types raises a ValidationError

Numbers, booleans, strings, and None are considered invalid input to `Schema.load`.

```

# 2.x
# Passes silently
schema.load(None)
schema.load(False)
schema.load("pass")

# 3.x
# marshmallow.exceptions.ValidationError: {'_schema': ['Invalid input type.']}

```

(continues on next page)

(continued from previous page)

```
schema.load(None)
schema.load(False)
schema.load("nope")
```

When `many=True`, non-collection types are also considered invalid.

```
# 2.x
# Passes silently
schema.load(None, many=True)
schema.load({}, many=True)
schema.load("pass", many=True)

# 3.x
# marshmallow.exceptions.ValidationError: {'_schema': ['Invalid input type.']}
schema.load(None, many=True)
schema.load({}, many=True)
schema.load("invalid", many=True)
```

ValidationError.fields is removed

`ValidationError` no longer stores a list of `Field` instances associated with the validation errors.

If you need field instances associated with an error, you can access them from `schema.fields`.

```
from marshmallow import Schema, fields, ValidationError

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

schema = MySchema()

try:
    schema.load({"foo": "invalid"})
except ValidationError as error:
    field = schema.fields["foo"]
    # ...
```

ValidationError expects a single field name

`ValidationError` no longer accepts a list of field names. It expects a single field name. If none is passed, the error refers to the schema.

To return an error for several fields at once, a `dict` must be used.

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

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

(continues on next page)

(continued from previous page)

```

# 2.x
@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", "field_b"]
        )

# 3.x
@validates_schema
def validate_numbers(self, data):
    if data["field_b"] >= data["field_a"]:
        raise ValidationError(
            {
                "field_a": ["field_a must be greater than field_b"],
                "field_b": ["field_a must be greater than field_b"],
            }
        )

```

ValidationError error messages are deep-merged

When multiple *ValidationError* are raised, the error structures are merged in the final *ValidationError* raised at the end of the process.

When reporting error messages as *dict*, the keys should refer to subitems of the item the message refers to, and the values should be error messages.

See *Schema-level validation* for an example. page for an example.

Schemas raise ValidationError when deserializing data with unknown keys

marshmallow 3.x schemas can deal with unknown keys in three different ways, configurable with the *unknown* option:

- EXCLUDE: drop those keys (same as marshmallow 2)
- INCLUDE: pass those keys/values as is, with no validation performed
- RAISE (default): raise a *ValidationError*

The *unknown* option can be passed as a Meta option, on Schema instantiation, or at load time.

```

from marshmallow import Schema, fields, EXCLUDE, INCLUDE, RAISE

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

    class Meta:
        # Pass EXCLUDE as Meta option to keep marshmallow 2 behavior
        unknown = EXCLUDE

MySchema().load({"foo": 42, "bar": "whatever"}) # => ['foo': 42]

# Value passed on instantiation overrides Meta option

```

(continues on next page)

(continued from previous page)

```

schema = MySchema(unknown=INCLUDE)
schema.load({"foo": 42, "bar": "whatever"}) # => ['foo': 42, 'bar': 'whatever']

# Value passed on load overrides instance attribute
schema.load({"foo": 42, "bar": "whatever"}, unknown=RAISE) # => ValidationError

```

Overriding `get_attribute`

If your *Schema* overrides `get_attribute`, you will need to update the method's signature. The positions of the `attr` and `obj` arguments were switched for consistency with Python builtins, e.g. `getattr`.

```

from marshmallow import Schema

# 2.x
class MySchema(Schema):
    def get_attribute(self, attr, obj, default):
        return getattr(obj, attr, default)

# 3.x
class MySchema(Schema):
    def get_attribute(self, obj, attr, default):
        return getattr(obj, attr, default)

```

`pass_original=True` passes individual items when `many=True`

When `pass_original=True` is passed to `validates_schema`, `post_load`, or `post_dump`, the `original_data` argument will be a single item corresponding to the (de)serialized datum.

```

from marshmallow import Schema, fields, post_load, EXCLUDE

class ShoeSchema(Schema):
    size = fields.Int()

    class Meta:
        unknown = EXCLUDE

    @post_load(pass_original=True)
    def post_load(self, data, original_data, **kwargs):
        # original_data has 'width' but
        # data does not because it's not
        # in the schema
        assert "width" in original_data
        assert "width" not in data
        return data

input_data = [{"size": 10, "width": "M"}, {"size": 6, "width": "W"}]

print(ShoeSchema(many=True).load(input_data))

```

(continues on next page)

(continued from previous page)

```
# [{'size': 10}, {'size': 6}]
```

utils.get_func_args no longer returns bound arguments

The `utils.get_func_args` function will no longer return bound arguments, e.g. `'self'`.

```
from marshmallow.utils import get_func_args

class MyCallable:
    def __call__(self, foo, bar):
        return 42

callable_obj = MyCallable()

# 2.x
get_func_args(callable_obj) # => ['self', 'foo', 'bar']

# 3.x
get_func_args(callable_obj) # => ['foo', 'bar']
```

Handling AttributeError in Method and Function fields

The `Method` and `Function` fields no longer swallow `AttributeErrors`. Therefore, your methods and functions are responsible for handling inputs such as `None`.

```
from marshmallow import Schema, fields, missing

# 2.x
class ShapeSchema(Schema):
    area = fields.Method("get_area")

    def get_area(self, obj):
        return obj.height * obj.length

schema = ShapeSchema()
# In 2.x, the following would pass without errors
# In 3.x, and AttributeError would be raised
result = schema.dump(None)
result # => {}

# 3.x
class ShapeSchema(Schema):
    area = fields.Method("get_area")

    def get_area(self, obj):
        if obj is None:
            # 'area' will not appear in serialized output
```

(continues on next page)

```
        return missing
    return obj.height * obj.length

schema = ShapeSchema()
result = schema.dump(None)
result # => {}
```

Adding additional data to serialized output

Use a `post_dump` to add additional data on serialization. The extra argument on `Schema` was removed.

```
from marshmallow import Schema, fields, post_dump

# 2.x
class MySchema(Schema):
    x = fields.Int()
    y = fields.Int()

schema = MySchema(extra={"z": 123})
schema.dump({"x": 1, "y": 2})
# => {'z': 123, 'y': 2, 'x': 1}

# 3.x
class MySchema(Schema):
    x = fields.Int()
    y = fields.Int()

    @post_dump
    def add_z(self, output):
        output["z"] = 123
        return output

schema = MySchema()
schema.dump({"x": 1, "y": 2})
# => {'z': 123, 'y': 2, 'x': 1}
```

Schema-level validators are skipped when field validation fails

By default, schema validator methods decorated by `validates_schema` won't execute if any of the field validators fails (including `required=True` validation).

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

class MySchema(Schema):
    x = fields.Int(required=True)
    y = fields.Int(required=True)
```

(continues on next page)

(continued from previous page)

```

@validates_schema
def validate_schema(self, data):
    if data["x"] <= data["y"]:
        raise ValidationError("x must be greater than y")

schema = MySchema()

# 2.x
# A KeyError is raised in validate_schema
schema.load({"x": 2})

# 3.x
# marshmallow.exceptions.ValidationError: {'y': ['Missing data for required field.']}
# validate_schema is not run
schema.load({"x": 2})

```

If you want a schema validator to run even if a field validator fails, pass `skip_on_field_errors=False`. Make sure your code handles cases where fields are missing from the deserialized data (due to validation errors).

```

from marshmallow import Schema, fields, validates_schema, ValidationError

class MySchema(Schema):
    x = fields.Int(required=True)
    y = fields.Int(required=True)

    @validates_schema(skip_on_field_errors=False)
    def validate_schema(self, data):
        if "x" in data and "y" in data:
            if data["x"] <= data["y"]:
                raise ValidationError("x must be greater than y")

schema = MySchema()
schema.load({"x": 2})
# marshmallow.exceptions.ValidationError: {'y': ['Missing data for required field.']}

```

SchemaOpts constructor receives ordered argument

Subclasses of `SchemaOpts` receive an additional argument, `ordered`, which is `True` if the `ordered` option is set to `True` on a `Schema` or one of its parent classes.

```

from marshmallow import SchemaOpts

# 2.x
class CustomOpts(SchemaOpts):
    def __init__(self, meta):
        super().__init__(meta)
        self.custom_option = getattr(meta, "meta", False)

```

(continues on next page)

(continued from previous page)

```
# 3.x
class CustomOpts(SchemaOpts):
    def __init__(self, meta, ordered=False):
        super().__init__(meta, ordered)
        self.custom_option = getattr(meta, "meta", False)
```

ContainsOnly accepts empty and duplicate values

`validate.ContainsOnly` now accepts duplicate values in the input value.

```
from marshmallow import validate

validator = validate.ContainsOnly(["red", "blue"])

# in 2.x the following raises a ValidationError
# in 3.x, no error is raised
validator(["red", "red", "blue"])
```

If you don't want to accept duplicates, use a custom validator, like the following.

```
from marshmallow import ValidationError
from marshmallow.validate import ContainsOnly

class ContainsOnlyNoDuplicates(ContainsOnly):
    def __call__(self, value):
        ret = super(ContainsOnlyNoDuplicates, self).__call__(value)
        if len(set(value)) != len(value):
            raise ValidationError("Duplicate values not allowed")
        return ret
```

Note

If you need to handle unhashable types, you can use the implementation of `ContainsOnly` from marshmallow 2.x.

`validate.ContainsOnly` also accepts empty values as valid input.

```
from marshmallow import validate

validator = validate.ContainsOnly(["red", "blue"])

# in 2.x the following raises a ValidationError
# in 3.x, no error is raised
validator([])
```

To validate against empty inputs, use `validate.Length(min=1)`.

json_module option is renamed to render_module

The `json_module` *class Meta* option is deprecated in favor of `render_module`.

```
import ujson

# 2.x
class MySchema(Schema):
    class Meta:
        json_module = ujson

# 3.x
class MySchema(Schema):
    class Meta:
        render_module = ujson
```

missing and default Field parameters are passed in deserialized form

```
# 2.x
class UserSchema(Schema):
    id = fields.UUID(missing=lambda: str(uuid.uuid1()))
    birthdate = fields.DateTime(default=lambda: dt.datetime(2017, 9, 19).isoformat())

# 3.x
class UserSchema(Schema):
    id = fields.UUID(missing=uuid.uuid1)
    birthdate = fields.DateTime(default=dt.datetime(2017, 9, 19))
```

Pass default as a keyword argument

`fields.Boolean` now receives additional truthy and falsy parameters. Consequently, the default parameter should always be passed as a keyword argument.

```
# 2.x
fields.Boolean(True)

# 3.x
fields.Boolean(default=True)
```

Email and URL fields do not validate on serialization

`fields.Email` and `fields.URL` only validate input upon deserialization. They do not validate on serialization. This makes them more consistent with the other fields and improves serialization performance.

load_from and dump_to are merged into data_key

The same key is used for serialization and deserialization.

```
# 2.x
class UserSchema(Schema):
    email = fields.Email(load_from="CamelCasedEmail", dump_to="CamelCasedEmail")
```

(continues on next page)

(continued from previous page)

```
# 3.x
class UserSchema(Schema):
    email = fields.Email(data_key="CamelCasedEmail")
```

It is not possible to specify a different key for serialization and deserialization on the same field. This use case is covered by using two different *Schema*.

```
from marshmallow import Schema, fields

# 2.x
class UserSchema(Schema):
    id = fields.Str()
    email = fields.Email(load_from="CamelCasedEmail", dump_to="snake_case_email")

# 3.x
class BaseUserSchema(Schema):
    id = fields.Str()

class LoadUserSchema(BaseUserSchema):
    email = fields.Email(data_key="CamelCasedEmail")

class DumpUserSchema(BaseUserSchema):
    email = fields.Email(data_key="snake_case_email")
```

Also, when `data_key` is specified on a field, only `data_key` is checked in the input data. In marshmallow 2.x the field name is checked if `load_from` is missing from the input data.

Pre/Post-processors must return modified data

In marshmallow 2.x, `None` returned by a pre or post-processor is interpreted as “the data was mutated”. In marshmallow 3.x, the return value is considered as processed data even if it is `None`.

Processors that mutate the data should be updated to also return it.

```
# 2.x
class UserSchema(Schema):
    name = fields.Str()
    slug = fields.Str()

    @pre_load
    def slugify_name(self, in_data):
        # In 2.x, implicitly returning None implied that data were mutated
        in_data["slug"] = in_data["slug"].lower().strip().replace(" ", "-")

# 3.x
class UserSchema(Schema):
```

(continues on next page)

(continued from previous page)

```

name = fields.Str()
slug = fields.Str()

@pre_load
def slugify_name(self, in_data, **kwargs):
    # In 3.x, always return the processed data
    in_data["slug"] = in_data["slug"].lower().strip().replace(" ", "-")
    return in_data

```

Nested field no longer supports plucking

In marshmallow 2.x, when a string was passed to a Nested field's `only` parameter, the field would be plucked. In marshmallow 3.x, the `Pluck` field must be used instead.

```

# 2.x
class UserSchema(Schema):
    name = fields.Str()
    friends = fields.Nested("self", many=True, only="name")

# 3.x
class UserSchema(Schema):
    name = fields.Str()
    friends = fields.Pluck("self", "name", many=True)

```

Accessing attributes on objects within a list

In order to serialize attributes on inner objects within a list, use the `Pluck` field.

```

# 2.x
class FactorySchema(Schema):
    widget_ids = fields.List(fields.Int(attribute="id"))

# 3.x
class FactorySchema(Schema):
    widget_ids = fields.List(fields.Pluck(WidgetSchema, "id"))

```

List does not wrap single values in a list on serialization

In marshmallow 2.x, `List` serializes a single object as a list with a single element. In marshmallow 3.x, the object is assumed to be iterable and passing a non-iterable element results in an error.

```

class UserSchema(Schema):
    numbers = fields.List(fields.Int())

user = {"numbers": 1}
UserSchema().dump(user)

# 2.x
# => {'numbers': [1]}

```

(continues on next page)

(continued from previous page)

```
# 3.x
# => TypeError: 'int' object is not iterable
```

Float field takes a new `allow_nan` parameter

In marshmallow 2.x, Float field would serialize and deserialize special values such as `nan`, `inf` or `-inf`. In marshmallow 3, those values trigger a `ValidationError` unless `allow_nan` is `True`. `allow_nan` defaults to `False`.

```
# 2.x
class MySchema(Schema):
    x = fields.Float()

MySchema().load({"x": "nan"})
# => {'x': nan}

# 3.x
class MySchema(Schema):
    x = fields.Float()
    y = fields.Float(allow_nan=True)

MySchema().load({"x": 12, "y": "nan"})
# => {'x': 12.0, 'y': nan}

MySchema().load({"x": "nan"})
# marshmallow.exceptions.ValidationError: {'x': ['Special numeric values (nan or infinity)
↳ are not permitted.']}
```

`DateTime` field `dateformat` Meta option is renamed `datetimeformat`

The Meta option `dateformat` used to pass format to `DateTime` field is renamed as `datetimeformat`.

`Date` field gets a new `format` parameter to specify the format to use for serialization. `dateformat` Meta option now applies to `Date` field.

```
# 2.x
class MySchema(Schema):
    x = fields.DateTime()

    class Meta:
        dateformat = "%Y-%m"

MySchema().dump({"x": dt.datetime(2017, 9, 19)})
# => {'x': '2017-09'}

# 3.x
class MySchema(Schema):
    x = fields.DateTime()
```

(continues on next page)

(continued from previous page)

```

y = fields.Date()

class Meta:
    datetimeformat = "%Y-%m"
    dateformat = "%m-%d"

MySchema().dump({"x": dt.datetime(2017, 9, 19), "y": dt.date(2017, 9, 19)})
# => {'x': '2017-09', 'y': '09-19'}

```

DateTime leaves timezone information untouched during serialization

DateTime does not convert naive datetimes to UTC on serialization and LocalDateTime is removed.

```

# 2.x
class MySchema(Schema):
    x = fields.DateTime()
    y = fields.DateTime()
    z = fields.LocalDateTime()

MySchema().dump(
    {
        "x": dt.datetime(2017, 9, 19),
        "y": dt.datetime(2017, 9, 19, tzinfo=dt.timezone(dt.timedelta(hours=2))),
        "z": dt.datetime(2017, 9, 19, tzinfo=dt.timezone(dt.timedelta(hours=2))),
    }
)
# => {'x': '2017-09-19T00:00:00+00:00', 'y': '2017-09-18T22:00:00+00:00', 'z': '2017-09-
↳ 19T00:00:00+02:00'}

# 3.x
class MySchema(Schema):
    x = fields.DateTime()
    y = fields.DateTime()

MySchema().dump(
    {
        "x": dt.datetime(2017, 9, 19),
        "y": dt.datetime(2017, 9, 19, tzinfo=dt.timezone(dt.timedelta(hours=2))),
    }
)
# => {'x': '2017-09-19T00:00:00', 'y': '2017-09-19T00:00:00+02:00'}

```

The prefix Schema parameter is removed

The prefix parameter of Schema is removed. The same feature can be achieved using a `post_dump <marshmallow.decorators.post_dump>` method.

```
# 2.x
class MySchema(Schema):
    f1 = fields.Raw()
    f2 = fields.Raw()

MySchema(prefix="pre_").dump({"f1": "one", "f2": "two"})
# {'pre_f1': 'one', '_pre_f2': 'two'}

# 3.x
class MySchema(Schema):
    f1 = fields.Raw()
    f2 = fields.Raw()

    @post_dump
    def prefix_usr(self, data):
        return {"usr_{}".format(k): v for k, v in iteritems(data)}

MySchema().dump({"f1": "one", "f2": "two"})
# {'pre_f1': 'one', '_pre_f2': 'two'}
```

fields.FormattedString is removed

fields.FormattedString field is removed. Use *fields.Function* or *fields.Method* instead.

```
# 2.x
class MySchema(Schema):
    full_name = fields.FormattedString("{first_name} {last_name}")

# 3.x
class MySchema(Schema):
    full_name = fields.Function(lambda u: f"{u.first_name} {u.last_name}")
```

attribute or data_key collision triggers an exception

When a *Schema* is instantiated, a check is performed and a *ValueError* is triggered if

- several fields have the same attribute value (or field name if attribute is not passed), excluding *dump_only* fields, or
- several fields have the same *data_key* value (or field name if *data_key* is not passed), excluding *load_only* fields

In marshmallow 2, it was possible to have multiple fields with the same attribute. It would work provided the *Schema* was only used for dumping. When loading, the behaviour was undefined. In marshmallow 3, all but one of those fields must be marked as *dump_only*. Likewise for *data_key* (formerly *dump_to*) for fields that are not *load_only*.

```
# 2.x
class MySchema(Schema):
    f1 = fields.Raw()
    f2 = fields.Raw(attribute="f1")
```

(continues on next page)

(continued from previous page)

```

f3 = fields.Raw(attribute="f5")
f4 = fields.Raw(attribute="f5")

MySchema()
# No error

# 3.x
class MySchema(Schema):
    f1 = fields.Raw()
    f2 = fields.Raw(attribute="f1")
    f3 = fields.Raw(attribute="f5")
    f4 = fields.Raw(attribute="f5")

MySchema()
# ValueError: 'Duplicate attributes: ['f1', 'f5']'

class MySchema(Schema):
    f1 = fields.Raw()
    f2 = fields.Raw(attribute="f1", dump_only=True)
    f3 = fields.Raw(attribute="f5")
    f4 = fields.Raw(attribute="f5", dump_only=True)

MySchema()
# No error

```

Field.fail is deprecated in favor of Field.make_error

Field.fail is deprecated. Use Field.make_error. This allows you to re-raise exceptions using `raise ... from ...`.

```

from marshmallow import fields, ValidationError
from packaging import version

# 2.x
class Version(fields.Field):
    default_error_messages = {"invalid": "Not a valid version."}

    def _deserialize(self, value, *args, **kwargs):
        try:
            return version.Version(value)
        except version.InvalidVersion:
            self.fail("invalid")

# 3.x
class Version(fields.Field):

```

(continues on next page)

(continued from previous page)

```

default_error_messages = {"invalid": "Not a valid version."}

def _deserialize(self, value, *args, **kwargs):
    try:
        return version.Version(value)
    except version.InvalidVersion as error:
        raise self.make_error("invalid") from error

```

python-dateutil recommended dependency is removed

In marshmallow 2, `python-dateutil` was used to deserialize RFC or ISO 8601 datetimes if it was installed. In marshmallow 3, datetime deserialization is done with no additional dependency.

`python-dateutil` is no longer used by marshmallow.

Custom Fields

To make your custom fields compatible with marshmallow 3, `_deserialize` should accept `**kwargs`:

```

from marshmallow import fields, ValidationError
from packaging import version

# 2.x
class MyCustomField(fields.Field):
    def _deserialize(self, value, attr, obj): ...

# 3.x
class MyCustomField(fields.Field):
    def _deserialize(self, value, attr, obj, **kwargs): ...

```

7.3.8 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())

```

Similarly, 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")
```

7.3.9 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}
```

(continues on next page)

(continued from previous page)

```

).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()

```

(continues on next page)

(continued from previous page)

```

@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 *Pre-processing and post-processing* 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"]:
        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

```

(continues on next page)

(continued from previous page)

```

# 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):
    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()

```

(continues on next page)

(continued from previous page)

```
@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
```

(continues on next page)

(continued from previous page)

```

try:
    schema.load({"email": "invalid-email"})
except exc.UnmarshallingError as err:
    handle_error(err)

# 2.0
try:
    schema.load({"email": "invalid-email"})
except exc.ValidationError as err:
    handle_error(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:
            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):

```

(continues on next page)

(continued from previous page)

```

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

# 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.'],
#     '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*.

7.3.10 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 of the `Decimal` field.

7.3.11 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()

```

(continues on next page)

(continued from previous page)

```

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):

```

(continues on next page)

(continued from previous page)

```

if len(val) < 8:
    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.

7.4 Who's using marshmallow?

Visit the link below to see a list of companies using marshmallow.

<https://github.com/marshmallow-code/marshmallow/wiki/Who's-using-marshmallow%3F>

Is your company or organization using marshmallow? Add it to the wiki.

7.5 License

Copyright Steven Loria **and** contributors

Permission **is** hereby granted, free of charge, to **any** person obtaining a copy of this software **and** associated documentation files (the "**Software**"), to deal **in** the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, **and/or** sell copies of the Software, **and** to permit persons to whom the Software **is** furnished to do so, subject to the following conditions:

The above copyright notice **and** this permission notice shall be included **in** **all** copies **or** substantial portions of the Software.

(continues on next page)

(continued from previous page)

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

7.6 Authors

7.6.1 Leads

- Steven Loria @sloria
- Jérôme Lafréchoux @lafrech
- Jared Deckard @deckar01

7.6.2 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
- @amikhlap
- 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 Suşcov @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
- Jared Deckard @deckar01
- 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 Varvayuk @mindoyo-victor
- Jāzeps Baško @jbasko
- @podhmo
- Dmitry Orlov @mosquito
- Yuri Heupa @YuriHeupa
- Roy Williams @rowillia
- Vlad Frolov @frol
- Erling Børresen @erlingbo
- Jérôme Lafréchoux @lafrech
- Roy Williams @rowillia
- @dradetsky
- Michal Kononenko @MichalKononenko
- Yoichi NAKAYAMA @yoichi
- Bernhard M. Wiedemann @bmwiedemann
- Scott Werner @scottwernervt
- Leonardo Fedalto @Fedalto
- @sduthil
- Steven Sklar @sklarsa
- Alisson Silveira @4lissonsilveira
- Harlov Nikita @harlov
- @stj
- Tomasz Magulski @magul
- Suren Khorenyan @mahenzon
- Jeffrey Berger @JeffBerger
- Felix Yan @felixonmars
- Prasanjit Prakash @ikilledthecat
- Guillaume Gelin @ramnes
- Maxim Novikov @m-novikov
- James Remeika @remeika

- Karandeep Singh Nagra @knagra
- Dushyant Rijhwani @dushr
- Viktor Kerkez @alefnula
- Victor Gavro @vgavro
- Kamil Galuszka @galuszkak
- David Watson @arbor-dwatson
- Jan Margeta @jmargeta
- AlexV @asmodehn
- @toffan
- Hampus Dunström @Dunstrom
- Robert Jensen @r1b
- Arijit Basu @sayanarijit
- Sanjay P @snjypl
- Víctor Zabalza @zblz
- Riley Gibbs @rileyjohngibbs
- Henry Doupe @hdoupe
- @miniscruff
- @maxalbert
- Kim Gustyr @khvn26
- Bryce Drennan @brycedrennan
- Tim Shaffer @timster
- Hugo van Kemenade @hugovk
- Maciej Urbański @rooterkyberian
- Kostas Konstantopoulos @kdop
- Stephen J. Fuhry @fuhrysteve
- @dursk
- Ezra MacDonald @macdonaldezra
- Stanislav Rogovskiy @atmo
- Cristi Scoarta @cristi23
- Anthony Sottile @asottile
- Charles-Axel Dein @charlax
- @phrfpeixoto
- @jceresini
- Nikolay Shebanov @killthekitten
- Taneli Hukkinen @hukkinj1
- @Reskov

- Albert Tugushev @atugushev
- @dfirst
- Tim Gates @timgates42
- Nathan @nbanmp
- Ronan Murphy @Resinderate
- Laurie Opperman @EpicWink
- Ram Rachum @cool-RR
- @weeix
- Juan Norris @juannorris
- @jun0jang
- @ebargtuo
- Michał Getka @mgetka
- Nadège Michel @nadege
- Tamara @infinityxxx
- Stephen Rosen @sirosen
- Vladimir Mikhaylov @vemikhaylov
- Stephen Eaton @madeinoz67
- Antonio Lassandro @lassandroan
- Javier Fernández @jfernandz
- Michael Dimchuk @michaeldimchuk
- Jochen Kupperschmidt @homeworkprod
- @yourun-proger
- Ryan Morehart @traherom
- Ben Windsor @bwindsor
- Kevin Kirsche @kkirsche
- Isira Seneviratne @Isira-Seneviratne
- Karthikeyan Singaravelan @tirkarthi
- Marco Satti @marcosatti
- Ivo Reumkens @vanHoi
- Aditya Tewary @aditkumar72
- Sebastien Lovergne @TheBigRoomXXL
- Peter C @somethingnew2-0
- Marcel Jackwerth @mrcljx
- Fares Abubaker @Fares-Abubaker
- Dharanikumar Sekar @dharani7998
- Nicolas Simonds @0xDECODE

- Florian Laport @Florian-Laport
- @agentgodzilla
- Xiao @T90REAL
- jfo @jafournier
- thanhlecong @thanhlecong
- Emmanuel Ferdman @emmanuel-ferdman
- Fridayworks @worksbyfriday
- @rstar327
- Kadir Can Ozden @bysiber
- Dhruvil Darji @dhruvildarji

7.7 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*).

7.7.1 Security contact information

To report a security vulnerability, please use the [Tidelift security contact](#). Tidelift will coordinate the fix and disclosure.

7.7.2 Questions, feature requests, bug reports, and feedback...

...should all be reported on the [Github Issue Tracker](#).

7.7.3 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 edit button in the top-right corner of the page. 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.

7.7.4 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 [uv](#).
3. Install development requirements.

```
$ uv sync
```

- (Optional but recommended) Install the pre-commit hooks, which will format and lint your git staged files.

```
$ uv run pre-commit install --allow-missing-config
```

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.** A 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

- Create a new local branch.

For a new feature:

```
$ git checkout -b name-of-feature dev
```

For a bugfix:

```
$ git checkout -b fix-something 3.x-line
```

- Commit your changes. Write [good commit messages](#).

```
$ git commit -m "Detailed commit message"
$ git push origin name-of-feature
```

- 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`.
- 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:

```
$ uv run pytest
```

To run formatting and syntax checks:

```
$ uv run tox -e lint
```

(Optional) To run tests in all supported Python versions in their own virtual environments (must have each interpreter installed):

```
$ uv run tox
```

Documentation

Contributions to the documentation are welcome. Documentation is written in [reStructuredText](#) (rST). A quick rST reference can be found [here](#). Builds are powered by [Sphinx](#).

To build and serve the docs in “watch” mode:

```
$ uv run tox -e docs-serve
```

Changes to documentation 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.

7.8 Code of Conduct

This code of conduct applies to the marshmallow project and all associated projects in the [marshmallow-code](#) organization.

7.8.1 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*.

7.8.2 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.

7.8.3 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 harassment.
- 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-actullies"
- Other conduct which could reasonably be considered inappropriate in a professional or community setting.

7.8.4 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.

7.8.5 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 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 @sloria, 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.

7.8.6 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.

7.8.7 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 <https://contributor-covenant.org/version/1/4>, and the [LGBTQ in Technology Slack Code of Conduct](#).

7.9 Kudos

A hat tip to [Django Rest Framework](#) , [Flask-RESTful](#), and [colander](#) for ideas and API design.

7.10 Donate

If you find marshmallow useful, please consider supporting the team with a donation:



Your donation keeps marshmallow healthy and maintained.

PYTHON MODULE INDEX

m

- `marshmallow`, 47
- `marshmallow.class_registry`, 87
- `marshmallow.decorators`, 76
- `marshmallow.error_store`, 88
- `marshmallow.exceptions`, 86
- `marshmallow.experimental.context`, 85
- `marshmallow.fields`, 64
- `marshmallow.types`, 87
- `marshmallow.utils`, 84
- `marshmallow.validate`, 80

A

additional (*marshmallow.Schema.Meta* attribute), 49
 additional (*marshmallow.schema.Schema.Meta* attribute), 59
 And (*class in marshmallow.validate*), 80
 AwareDateTime (*class in marshmallow.fields*), 65

B

Bool (*in module marshmallow.fields*), 65
 Boolean (*class in marshmallow.fields*), 65

C

callable_or_raise() (*in module marshmallow.utils*), 84
 Common Vulnerabilities and Exposures
 CVE 2018-17175, 117
 CVE 2025-68480, 91, 94
 Constant (*class in marshmallow.fields*), 66
 ContainsNoneOf (*class in marshmallow.validate*), 81
 ContainsOnly (*class in marshmallow.validate*), 81
 Context (*class in marshmallow.experimental.context*), 86

D

Date (*class in marshmallow.fields*), 66
 dateformat (*marshmallow.Schema.Meta* attribute), 49
 dateformat (*marshmallow.schema.Schema.Meta* attribute), 59
 DateTime (*class in marshmallow.fields*), 66
 datetimeformat (*marshmallow.Schema.Meta* attribute), 49
 datetimeformat (*marshmallow.schema.Schema.Meta* attribute), 59
 Decimal (*class in marshmallow.fields*), 66
 Dict (*class in marshmallow.fields*), 67
 dict_class (*marshmallow.Schema* attribute), 50
 dict_class (*marshmallow.schema.Schema* attribute), 60
 dump() (*marshmallow.Schema* method), 50
 dump() (*marshmallow.schema.Schema* method), 60
 dump_only (*marshmallow.Schema.Meta* attribute), 49

dump_only (*marshmallow.schema.Schema.Meta* attribute), 59
 dumps() (*marshmallow.Schema* method), 50
 dumps() (*marshmallow.schema.Schema* method), 60

E

Email (*class in marshmallow.fields*), 67
 Email (*class in marshmallow.validate*), 81
 Enum (*class in marshmallow.fields*), 67
 Equal (*class in marshmallow.validate*), 81
 error_messages (*marshmallow.Schema* attribute), 50
 error_messages (*marshmallow.schema.Schema* attribute), 60
 ErrorMessageValue (*in module marshmallow.types*), 87
 ErrorMessageValue (*in module marshmallow.types*), 87
 EXCLUDE (*in module marshmallow*), 56
 exclude (*marshmallow.Schema.Meta* attribute), 49
 exclude (*marshmallow.schema.Schema.Meta* attribute), 59

F

falsy (*marshmallow.fields.Boolean* attribute), 65
 Field (*class in marshmallow.fields*), 63
 fields (*marshmallow.Schema* attribute), 50
 fields (*marshmallow.Schema.Meta* attribute), 49
 fields (*marshmallow.schema.Schema* attribute), 60
 fields (*marshmallow.schema.Schema.Meta* attribute), 59
 Float (*class in marshmallow.fields*), 68
 from_dict() (*marshmallow.Schema* class method), 51
 from_dict() (*marshmallow.schema.Schema* class method), 60
 Function (*class in marshmallow.fields*), 68

G

get() (*marshmallow.experimental.context.Context* class method), 86
 get_attribute() (*marshmallow.Schema* method), 51
 get_attribute() (*marshmallow.schema.Schema* method), 61
 get_class() (*in module marshmallow.class_registry*), 88

get_value() (in module *marshmallow.utils*), 84

H

handle_error() (*marshmallow.Schema* method), 51
 handle_error() (*marshmallow.schema.Schema* method), 61

I

INCLUDE (in module *marshmallow*), 56
 include (*marshmallow.Schema.Meta* attribute), 49
 include (*marshmallow.schema.Schema.Meta* attribute), 59
 index_errors (*marshmallow.Schema.Meta* attribute), 49
 index_errors (*marshmallow.schema.Schema.Meta* attribute), 59
 Int (in module *marshmallow.fields*), 69
 Integer (class in *marshmallow.fields*), 69
 IP (class in *marshmallow.fields*), 68
 IPInterface (class in *marshmallow.fields*), 68
 IPv4 (class in *marshmallow.fields*), 69
 IPv4Interface (class in *marshmallow.fields*), 69
 IPv6 (class in *marshmallow.fields*), 69
 IPv6Interface (class in *marshmallow.fields*), 69
 is_collection() (in module *marshmallow.utils*), 84
 is_generator() (in module *marshmallow.utils*), 84
 is_iterable_but_not_string() (in module *marshmallow.utils*), 84
 is_sequence_but_not_string() (in module *marshmallow.utils*), 84

L

Length (class in *marshmallow.validate*), 81
 List (class in *marshmallow.fields*), 69
 load() (*marshmallow.Schema* method), 51
 load() (*marshmallow.schema.Schema* method), 61
 load_only (*marshmallow.Schema.Meta* attribute), 49
 load_only (*marshmallow.schema.Schema.Meta* attribute), 59
 loads() (*marshmallow.Schema* method), 52
 loads() (*marshmallow.schema.Schema* method), 62

M

many (*marshmallow.Schema.Meta* attribute), 50
 many (*marshmallow.schema.Schema.Meta* attribute), 59
 Mapping (class in *marshmallow.fields*), 70
 marshmallow
 module, 47
 marshmallow.class_registry
 module, 87
 marshmallow.decorators
 module, 76
 marshmallow.error_store

 module, 88
 marshmallow.exceptions
 module, 86
 marshmallow.experimental.context
 module, 85
 marshmallow.fields
 module, 64
 marshmallow.types
 module, 87
 marshmallow.utils
 module, 84
 marshmallow.validate
 module, 80
 MarshmallowError, 86
 merge_errors() (in module *marshmallow.error_store*), 88
 Method (class in *marshmallow.fields*), 70
 missing (in module *marshmallow*), 55
 module
 marshmallow, 47
 marshmallow.class_registry, 87
 marshmallow.decorators, 76
 marshmallow.error_store, 88
 marshmallow.exceptions, 86
 marshmallow.experimental.context, 85
 marshmallow.fields, 64
 marshmallow.types, 87
 marshmallow.utils, 84
 marshmallow.validate, 80

N

NaiveDateTime (class in *marshmallow.fields*), 70
 Nested (class in *marshmallow.fields*), 71
 NoneOf (class in *marshmallow.validate*), 82
 Number (class in *marshmallow.fields*), 72

O

on_bind_field() (*marshmallow.Schema* method), 52
 on_bind_field() (*marshmallow.schema.Schema* method), 62
 OneOf (class in *marshmallow.validate*), 82
 options() (*marshmallow.validate.OneOf* method), 82

P

Pluck (class in *marshmallow.fields*), 72
 pluck() (in module *marshmallow.utils*), 84
 post_dump() (in module *marshmallow*), 53
 post_dump() (in module *marshmallow.decorators*), 77
 post_load() (in module *marshmallow*), 54
 post_load() (in module *marshmallow.decorators*), 78
 PostLoadCallable (in module *marshmallow.types*), 87
 pre_dump() (in module *marshmallow*), 54
 pre_dump() (in module *marshmallow.decorators*), 78
 pre_load() (in module *marshmallow*), 54

`pre_load()` (in module `marshmallow.decorators`), 78
 Predicate (class in `marshmallow.validate`), 82
 PreLoadCallable (in module `marshmallow.types`), 87

R

RAISE (in module `marshmallow`), 56
 Range (class in `marshmallow.validate`), 83
 Raw (class in `marshmallow.fields`), 72
 Regexp (class in `marshmallow.validate`), 83
 register (`marshmallow.Schema.Meta` attribute), 50
 register (`marshmallow.schema.Schema.Meta` attribute), 59
 register() (in module `marshmallow.class_registry`), 88
 RegistryError, 86
 render_module (`marshmallow.Schema.Meta` attribute), 50
 render_module (`marshmallow.schema.Schema.Meta` attribute), 59
 RenderModule (class in `marshmallow.types`), 87

S

Schema (class in `marshmallow`), 47
 Schema (class in `marshmallow.schema`), 56
 schema (`marshmallow.fields.Nested` property), 72
 Schema.Meta (class in `marshmallow`), 48
 Schema.Meta (class in `marshmallow.schema`), 58
 SchemaOpts (class in `marshmallow`), 53
 SchemaOpts (class in `marshmallow.schema`), 63
 SchemaValidator (class in `marshmallow.types`), 87
 set_class (`marshmallow.Schema` attribute), 52
 set_class (`marshmallow.schema.Schema` attribute), 62
 set_hook() (in module `marshmallow.decorators`), 79
 set_value() (in module `marshmallow.utils`), 85
 Str (in module `marshmallow.fields`), 73
 String (class in `marshmallow.fields`), 73
 StringNotCollectionError, 86
 StrSequenceOrSet (in module `marshmallow.types`), 87

T

Time (class in `marshmallow.fields`), 74
 TimeDelta (class in `marshmallow.fields`), 74
 timedelta_to_microseconds() (in module `marshmallow.utils`), 85
 timeformat (`marshmallow.Schema.Meta` attribute), 50
 timeformat (`marshmallow.schema.Schema.Meta` attribute), 60
 truthful (`marshmallow.fields.Boolean` attribute), 65
 Tuple (class in `marshmallow.fields`), 74

U

unknown (`marshmallow.Schema.Meta` attribute), 50
 unknown (`marshmallow.schema.Schema.Meta` attribute), 60

UnknownOption (in module `marshmallow.types`), 87
 Url (class in `marshmallow.fields`), 75
 URL (class in `marshmallow.validate`), 83
 URL (in module `marshmallow.fields`), 75
 UUID (class in `marshmallow.fields`), 75

V

validate() (`marshmallow.Schema` method), 52
 validate() (`marshmallow.schema.Schema` method), 62
 validates() (in module `marshmallow`), 55
 validates() (in module `marshmallow.decorators`), 79
 validates_schema() (in module `marshmallow`), 55
 validates_schema() (in module `marshmallow.decorators`), 79
 ValidationError, 53, 86
 Validator (class in `marshmallow.validate`), 83
 Validator (in module `marshmallow.types`), 87