serpy Documentation

Release 0.3.1

Clark DuVall

Nov 28, 2017

Contents

serpy is a super simple object serialization framework built for speed. **serpy** serializes complex datatypes (Django Models, custom classes, ...) to simple native types (dicts, lists, strings, ...). The native types can easily be converted to JSON or any other format needed.

The goal of **serpy** is to be able to do this *simply*, *reliably*, and *quickly*. Since serializers are class based, they can be combined, extended and customized with very little code duplication. Compared to other popular Python serialization frameworks like marshmallow or Django Rest Framework Serializers **serpy** is at least an order of magnitude faster.

CHAPTER 1

Source

Source at: https://github.com/clarkduvall/serpy

If you want a feature, send a pull request!

CHAPTER 2

Documentation

Full documentation at: http://serpy.readthedocs.org/en/latest/

CHAPTER 3

Installation

```
$ pip install serpy
```

CHAPTER 4

Examples

4.1 Simple Example

```
import serpy

class Foo(object):
    """The object to be serialized."""
    y = 'hello'
    z = 9.5

    def __init__(self, x):
        self.x = x


class FooSerializer(serpy.Serializer):
    """The serializer schema definition."""
    # Use a Field subclass like IntField if you need more validation.
    x = serpy.IntField()
    y = serpy.Field()
    z = serpy.Field()

f = Foo(1)
FooSerializer(f).data
# {'x': 1, 'y': 'hello', 'z': 9.5}

fs = [Foo(i) for i in range(100)]
FooSerializer(fs, many=True).data
# [{'x': 0, 'y': 'hello', 'z': 9.5}, {'x': 1, 'y': 'hello', 'z': 9.5}, ...]
```

4.2 Nested Example

```
import serpy

class Nestee(object):
    """An object nested inside another object."""
    n = 'hi'


class Foo(object):
    x = 1
    nested = Nestee()


class NesteeSerializer(serpy.Serializer):
    n = serpy.Field()


class FooSerializer(serpy.Serializer):
    x = serpy.Field()
    # Use another serializer as a field.
    nested = NesteeSerializer()

f = Foo()
FooSerializer(f).data
# {'x': 1, 'nested': {'n': 'hi'}}
```

4.3 Complex Example

```
import serpy

class Foo(object):
    y = 1
    z = 2
    super_long_thing = 10

    def x(self):
        return 5


class FooSerializer(serpy.Serializer):
    w = serpy.Field(attr='super_long_thing')
    x = serpy.Field(call=True)
    plus = serpy.MethodField()

    def get_plus(self, obj):
        return obj.y + obj.z

f = Foo()
FooSerializer(f).data
# {'w': 10, 'x': 5, 'plus': 3}
```

4.4 Inheritance Example

```
import serpy

class Foo(object):
    a = 1
    b = 2


class ASerializer(serpy.Serializer):
    a = serpy.Field()


class ABSerializer(ASerializer):
    """ABSerializer inherits the 'a' field from ASerializer.

    This also works with multiple inheritance and mixins.
    """
    b = serpy.Field()

f = Foo()
ASerializer(f).data
# {'a': 1}
ABSerializer(f).data
# {'a': 1, 'b': 2}
```

CHAPTER 5

License

serpy is free software distributed under the terms of the MIT license. See the LICENSE file.

Contents:

5.1 API Reference

5.1.1 Serializer

class serpy.**Serializer**(*instance=None*, *many=False*, *data=None*, *context=None*, ***kwargs*)

Serializer is used as a base for custom serializers.

The *Serializer* class is also a subclass of *Field*, and can be used as a *Field* to create nested schemas. A serializer is defined by subclassing *Serializer* and adding each *Field* as a class variable:

Example:

```
class FooSerializer(Serializer):
    foo = Field()
    bar = Field()

foo = Foo(foo='hello', bar=5)
FooSerializer(foo).data
# {'foo': 'hello', 'bar': 5}
```

Parameters

- **instance** – The object or objects to serialize.
- **many** (*bool*) – If `instance` is a collection of objects, set `many` to `True` to serialize to a list.
- **context** – Currently unused parameter for compatability with Django REST Framework serializers.

data
Get the serialized data from the *Serializer*.

The data will be cached for future accesses.

default_getter
The default getter used if *Field.as_getter()* returns None.

alias of `attrgetter`

class `serpy.`**DictSerializer**(*instance=None*, *many=False*, *data=None*, *context=None*, ***kwargs*)
DictSerializer serializes python `dicts` instead of objects.

Instead of the serializer's fields fetching data using `operator.attrgetter`, *DictSerializer* uses `operator.itemgetter`.

Example:

```
class FooSerializer(DictSerializer):
    foo = IntField()
    bar = FloatField()

foo = {'foo': '5', 'bar': '2.2'}
FooSerializer(foo).data
# {'foo': 5, 'bar': 2.2}
```

default_getter
alias of `itemgetter`

5.1.2 Fields

If none of these fields fit your needs, **serpy** makes it simple to create custom fields. See the Custom Fields documentation.

class `serpy.`**Field**(*attr=None*, *call=False*, *label=None*, *required=True*)
Field is used to define what attributes will be serialized.

A *Field* maps a property or function on an object to a value in the serialized result. Subclass this to make custom fields. For most simple cases, overriding *Field.to_value()* should give enough flexibility. If more control is needed, override *Field.as_getter()*.

Parameters

- **attr** (*str*) – The attribute to get on the object, using the same format as `operator.attrgetter`. If this is not supplied, the name this field was assigned to on the serializer will be used.
- **call** (*bool*) – Whether the value should be called after it is retrieved from the object. Useful if an object has a method to be serialized.
- **label** (*str*) – A label to use as the name of the serialized field instead of using the attribute name of the field.
- **required** (*bool*) – Whether the field is required. If set to `False`, *Field.to_value()* will not be called if the value is `None`.

as_getter(*serializer_field_name*, *serializer_cls*)
Returns a function that fetches an attribute from an object.

Return `None` to use the default getter for the serializer defined in *Serializer.default_getter*.

When a *Serializer* is defined, each *Field* will be converted into a getter function using this method. During serialization, each getter will be called with the object being serialized, and the return value will be passed through *Field.to_value()*.

If a *Field* has `getter_takes_serializer = True`, then the getter returned from this method will be called with the *Serializer* instance as the first argument, and the object being serialized as the second.

Parameters

- **serializer_field_name** (*str*) – The name this field was assigned to on the serializer.
- **serializer_cls** – The *Serializer* this field is a part of.

getter_takes_serializer = False
Set to `True` if the value function returned from *Field.as_getter()* requires the serializer to be passed in as the first argument. Otherwise, the object will be the only parameter.

to_value(*value*)
Transform the serialized value.

Override this method to clean and validate values serialized by this field. For example to implement an `int` field:

```
def to_value(self, value):
    return int(value)
```

Parameters **value** – The value fetched from the object being serialized.

class serpy.**StrField**(*attr=None*, *call=False*, *label=None*, *required=True*)
A *Field* that converts the value to a string.

to_value
alias of `unicode`

class serpy.**IntField**(*attr=None*, *call=False*, *label=None*, *required=True*)
A *Field* that converts the value to an integer.

to_value
alias of `int`

class serpy.**FloatField**(*attr=None*, *call=False*, *label=None*, *required=True*)
A *Field* that converts the value to a float.

to_value
alias of `float`

class serpy.**BoolField**(*attr=None*, *call=False*, *label=None*, *required=True*)
A *Field* that converts the value to a boolean.

to_value
alias of `bool`

class serpy.**MethodField**(*method=None*, ***kwargs*)
A *Field* that calls a method on the *Serializer*.

This is useful if a *Field* needs to serialize a value that may come from multiple attributes on an object. For example:

```
class FooSerializer(Serializer):
    plus = MethodField()
    minus = MethodField('do_minus')

    def get_plus(self, foo_obj):
        return foo_obj.bar + foo_obj.baz

    def do_minus(self, foo_obj):
        return foo_obj.bar - foo_obj.baz

foo = Foo(bar=5, baz=10)
FooSerializer(foo).data
# {'plus': 15, 'minus': -5}
```

Parameters **method** (*str*) – The method on the serializer to call. Defaults to `'get_<field name>'`.

5.2 Custom Fields

The most common way to create a custom field with **serpy** is to override *serpy.Field.to_value()*. This method is called on the value retrieved from the object being serialized. For example, to create a field that adds 5 to every value it serializes, do:

```
class Add5Field(serpy.Field):
   def to_value(self, value):
      return value + 5
```

Then to use it:

```
class Obj(object):
   pass

class ObjSerializer(serpy.Serializer):
   foo = Add5Field()

f = Obj()
f.foo = 9
ObjSerializer(f).data
# {'foo': 14}
```

Another use for custom fields is data validation. For example, to validate that every serialized value has a `'.'` in it:

```
class ValidateDotField(serpy.Field):
   def to_value(self, value):
      if '.' not in value:
         raise ValidationError('no dot!')
      return value
```

For more control over the behavior of the field, override *serpy.Field.as_getter()*. When the *serpy.Serializer* class is created, each field will be compiled to a getter, that will be called to get its associated attribute from the object. For an example of this, see the *serpy.MethodField()* implementation.

5.3 Performance Benchmarks

serpy was compared against two other popular serializer frameworks:

- marshmallow
- Django Rest Framework Serializers

These graphs show the results. The benchmark scripts are found in the benchmarks directory in the **serpy** GitHub repository. Run these benchmarks yourself with:

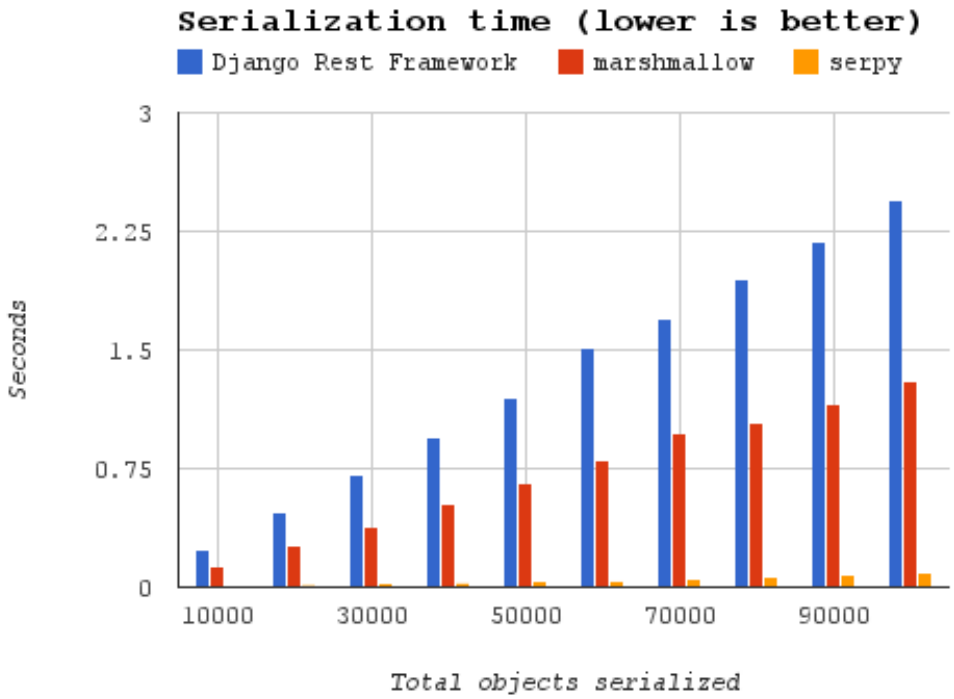
```
$ git clone https://github.com/clarkduvall/serpy.git && cd serpy
$ tox -e benchmarks
```

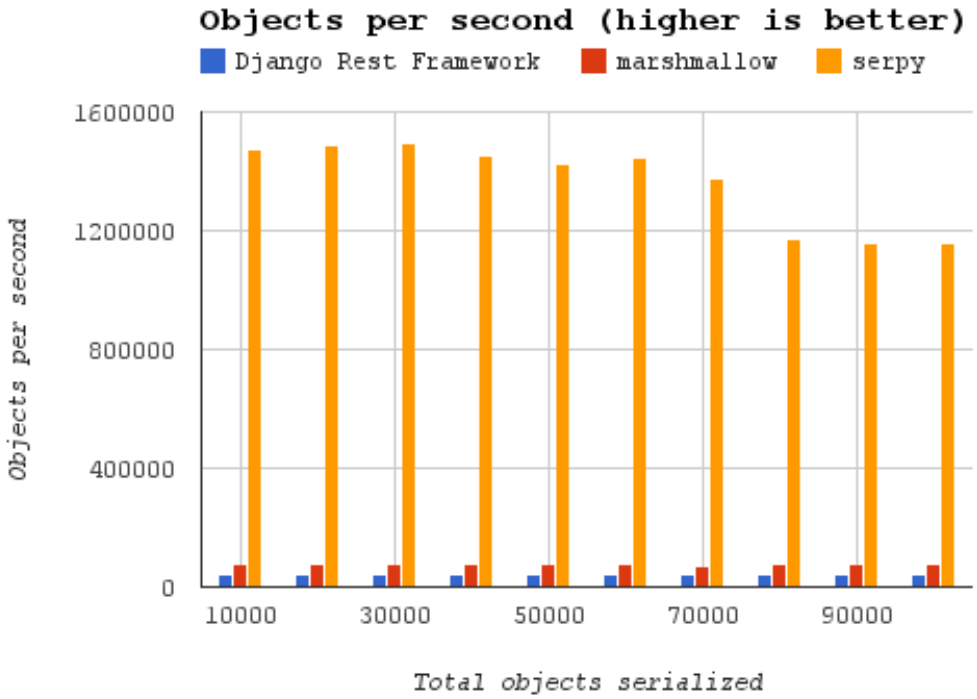
5.3.1 Results

These benchmarks were run on a Lenovo T530 with a 2-core 2.5 GHz i5 processor and 8G of memory.

Simple Benchmark

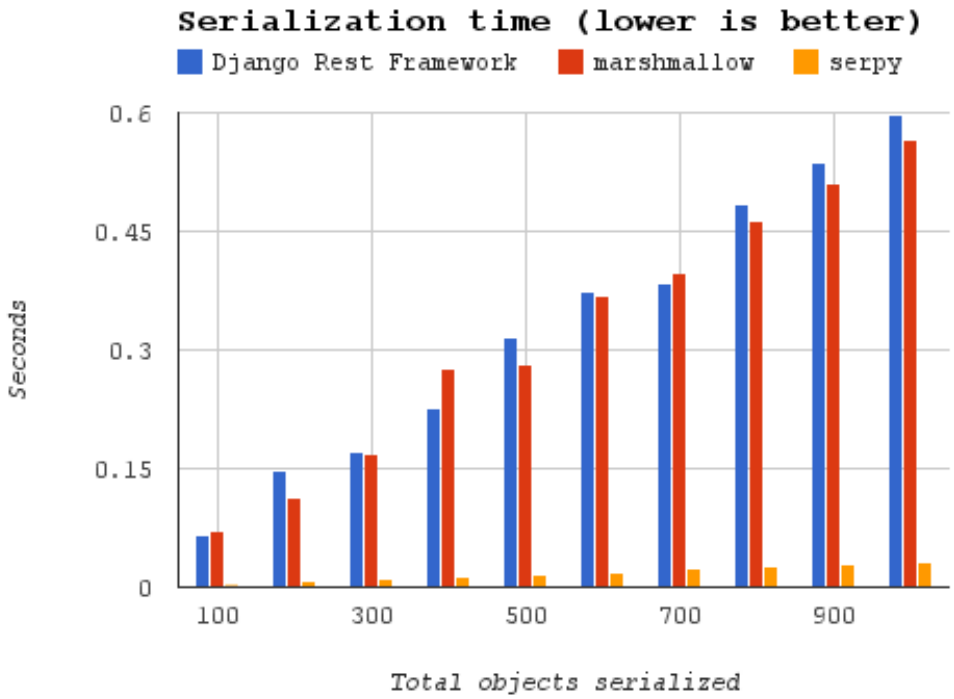
This benchmark serializes simple objects that have a single field.

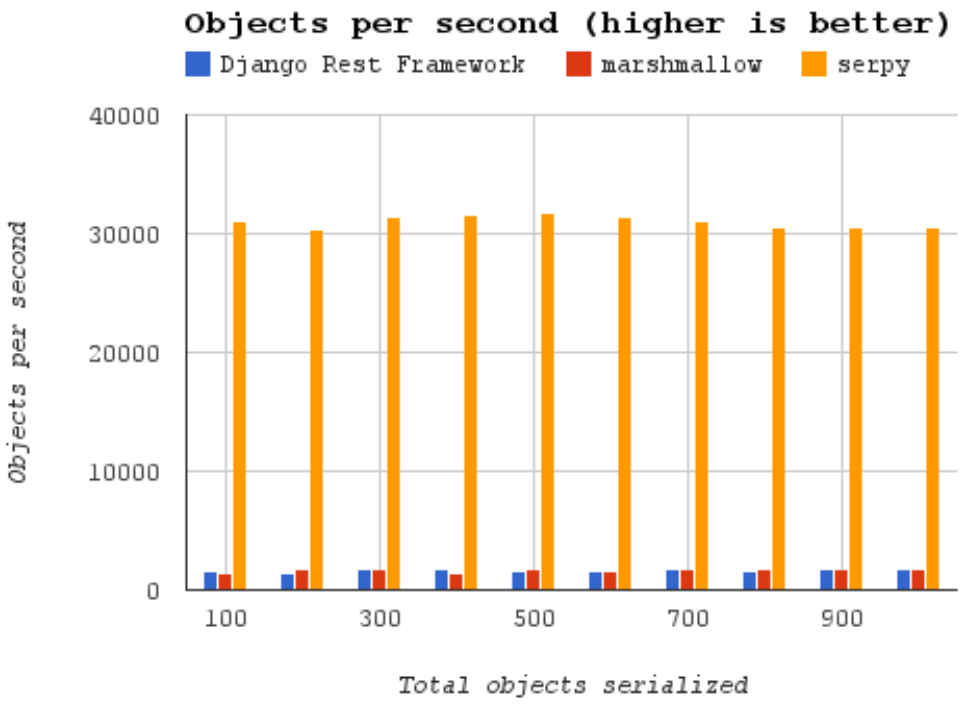




Complex Benchmark

This benchmark serializes nested objects with multiple fields of different types.





Data

Table 5.1: bm_simple.py time taken (in seconds)

# objects	Django Rest Framework	marshmallow	serpy
10000	0.2414798737	0.1281440258	0.006773948669
20000	0.4704430103	0.2609949112	0.01343297958
30000	0.7049410343	0.3850591183	0.02005600929
40000	0.9448800087	0.5248949528	0.02748799324
50000	1.196242809	0.6589410305	0.03510689735
60000	1.513856888	0.8019201756	0.04155898094
70000	1.695443153	0.9703800678	0.05080986023
80000	1.943806887	1.04428792	0.06843280792
90000	2.189687967	1.16334486	0.07787084579
100000	2.445794821	1.302541018	0.0864470005

Table 5.2: bm_simple.py objects per second

# objects	Django Rest Framework	marshmallow	serpy
10000	41411.31867	78037.19243	1476243.841
20000	42513.11968	76629.84657	1488872.954
30000	42556.7509	77910.11452	1495811.034
40000	42333.41761	76205.72419	1455180.8
50000	41797.53442	75879.32408	1424221.557
60000	39633.86532	74820.41459	1443731.262
70000	41287.14069	72136.68368	1377685.349
80000	41156.35177	76607.22533	1169029.92
90000	41101.74662	77363.1303	1155759.888
100000	40886.50411	76773.01417	1156778.135

Table 5.3: bm_complex.py time taken (in seconds)

# objects	Django Rest Framework	marshmallow	serpy
100	0.06559991837	0.07014703751	0.003219127655
200	0.1476380825	0.1144611835	0.006608009338
300	0.171423912	0.169506073	0.009553909302
400	0.2272388935	0.2767920494	0.01268196106
500	0.3147311211	0.2825651169	0.0157828331
600	0.3746049404	0.3694860935	0.01907610893
700	0.3846490383	0.3978009224	0.02250695229
800	0.4846799374	0.4635269642	0.02613210678
900	0.5376219749	0.5094399452	0.02945303917
1000	0.5961399078	0.5659701824	0.03282499313

Table 5.4: bm_complex.py objects per second

# objects	Django Rest Framework	marshmallow	serpy
100	1524.392141	1425.576953	31064.3164
200	1354.664031	1747.317246	30266.30105
300	1750.047566	1769.848093	31400.75863
400	1760.262048	1445.128214	31540.86329
500	1588.65764	1769.50363	31679.99033
600	1601.687365	1623.877084	31452.95522
700	1819.840765	1759.67415	31101.50104
800	1650.573788	1725.897438	30613.68173
900	1674.038715	1766.645919	30557.11823
1000	1677.458574	1766.877534	30464.59129

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index