protobuf3

Release 0.2.0

March 08, 2016

Contents

CHAPTER 1

Overview

Protobuf3 is a library for interaction with serialized data encoded with Protocol Buffers This documentation attempts to explain everything you need to know to use **protobuf3**.

1.1 Installing

protobuf3 is in the Python Package Index.

1.1.1 Installing with PIP

To use pip to install protobuf3:

```
$ pip install protobuf3
```

To get a specific version of protobuf3:

```
$ pip install protobuf3==1.0.0
```

To upgrade using pip:

```
$ pip install --upgrade protobuf3
```

1.1.2 Installing from source

If you'd rather install directly from the source (i.e. to stay on the bleeding edge), check out the latest source from github and install the library from the resulting tree:

```
$ git clone git@github.com:Pr0Ger/protobuf3.git
$ cd protobuf3/
$ python setup.py install
```

1.2 Tutorial

This tutorial is intended as an introduction to working with **protobuf3**.

1.2.1 Prerequisites

Before we start, make sure that you have the **PyMongo** distribution installed. In the Python shell, the following should run without raising an exception:

```
>>> import protobuf3
```

This tutorial also assumes that you have installed protobuf compiler. The following command should run and show libprotobuf version:

```
$ protoc --version
```

1.2.2 Defining your protocol format

I don't want to copy-paste official protobuf tutorials, so if you want some explanation for this file, you can find it here.

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

1.2.3 Compiling your protocol buffers

It's very similar with original protobuf implementation. There is only one different thing: use **–python3_out** instead of **–python_out**

1.2.4 Generated code example

Protobuf compiler will generate this code for example .proto file

```
from protobuf3.message import Message
from protobuf3.fields import StringField, EnumField, Int32Field, MessageField
from enum import Enum
```

```
class Person(Message):

    class PhoneType(Enum):
        MOBILE = 0
        HOME = 1
        WORK = 2

    class PhoneNumber(Message):
        pass


class AddressBook(Message):
    pass

Person.PhoneNumber.add_field('number', StringField(field_number=1, required=True))
Person.PhoneNumber.add_field('type', EnumField(field_number=2, optional=True, enum_cls=Person.PhoneTy
Person.add_field('name', StringField(field_number=1, required=True))
Person.add_field('id', Int32Field(field_number=2, required=True))
Person.add_field('email', StringField(field_number=3, optional=True))
Person.add_field('phone', MessageField(field_number=4, repeated=True, message_cls=Person.PhoneNumber)
AddressBook.add_field('person', MessageField(field_number=1, repeated=True, message_cls=Person))
```

But this library also support django-style code for defining data model (this form is more readable). Same code, but hand-written using this style:

```
from protobuf3.message import Message
from protobuf3.fields import StringField, EnumField, Int32Field, MessageField
from enum import Enum


class Person(Message):

    class PhoneType(Enum):
        MOBILE = 0
        HOME = 1
        WORK = 2

    class PhoneNumber(Message):
        number = StringField(field_number=1, required=True)
        type = EnumField(field_number=2, optional=True, enum_cls=Person.PhoneType, default=Person.Pho

    name = StringField(field_number=1, required=True)
    id = Int32Field(field_number=2, required=True)
    email = StringField(field_number=3, optional=True)
    phone = MessageField(field_number=4, repeated=True, message_cls=Person.PhoneNumber)


class AddressBook(Message):
    person = MessageField(field_number=1, repeated=True, message_cls=Person)
```

1.2.5 The Protocol Buffer API

It's very similar to original implementation. Currently there is some difference how repeated field work (probably I make some comparability changes).

```
>>> person = address.Person()
>>> person.id = 1234
>>> person.name = "John Doe"
>>> person.email = "jdoe@example.com"
>>> number = address.Person.PhoneNumber()
>>> number.number = "123"
>>> person.phone.append(number)

>>> person.encode_to_bytes()
b'\n\x08John Doe\x10\xd2\t\x1a\x10jdoe@example.com"\x05\n\x03123'

>>> new_person = address.Person()
>>> new_person.parse_from_bytes(b'\n\x08John Doe\x10\xd2\t\x1a\x10jdoe@example.com"\x05\n\x03123')
>>> assert new_person.id == 1234
```

1.3 Generated code explanation

This page describes exactly what Python definitions the protocol buffer compiler generates for any given protocol definition. Also, this page is very similar to same page from original implementation, so I describe only differences from original implementation.

1.3.1 Compiler invocation

There is two significant differences:

1. **–python3_out** instead of **–python_out**.
2. There is no **_pb2** suffix in generated file names.

1.3.2 Messages

Message can be loaded from serialized form two ways:

1. By calling class-method **create_from_bytes**
2. By creating instance and then calling instance method **parse_from_bytes**

And can be serialized by calling **encode_to_bytes**

1.3.3 Fields

Instead of original implementation, this one doesn't generate any constants with field numbers.

Singular fields

All works very similar to original implementation:

```
message.foo = 123
print message.foo
```

There is some difference how you check fields presence:

```
assert not 'foo' in message
message.foo = 123
assert 'foo' in message
del message.foo
assert not 'foo' in message
```

Singular Message Fields

There is no difference with original implementation

```
message Foo {
    optional Bar bar = 1;
}
message Bar {
    optional int32 i = 1;
}
```

```
foo = Foo()
assert not 'bar' in foo
foo.bar.i = 1
assert 'bar' in foo
assert foo.bar.i == 1
```

Repeated Fields

I copied this section from original documentation.

```
message Foo {
    repeated int32 nums = 1;
}
```

```
foo = Foo()
foo.nums.append(15)         # Appends one value
foo.nums.extend([32, 47]) # Appends an entire list

assert len(foo.nums) == 3
assert foo.nums[0] == 15
assert foo.nums[1] == 32
assert foo.nums == [15, 32, 47]

foo.nums[1] = 56     # Reassigns a value
assert foo.nums[1] == 56
for i in foo.nums:   # Loops and print
  print i
del foo.nums[:]     # Clears list (works just like in a Python list)
```

Repeated Message Fields

It’s very similar to original implementation. Currently **.add()** isn’t supported

1.3.4 Enumerations

In Python 3.4 default **enum** is used, for previous Python version this implementation will require backported implementation enum34.

Some example:

```
message Foo {
    enum SomeEnum {
        VALUE_A = 1;
        VALUE_B = 5;
        VALUE_C = 1234;
    }
    optional SomeEnum bar = 1;
}
```

After generating you will receive following code:

```
from enum import Enum
from protobuf3.message import Message
from protobuf3.fields import EnumField


class Foo(Message):

    class SomeEnum(Enum):
        VALUE_A = 1
        VALUE_B = 5
        VALUE_C = 1234

Foo.add_field('bar', EnumField(field_number=1, optional=True, enum_cls=Foo.SomeEnum))
```

And how this works:

```
foo = Foo()
foo.bar = Foo.SomeEnum.VALUE_A
assert foo.bar.value == 1
assert foo.bar == Foo.SomeEnum.VALUE_A
```

1.3.5 Oneof

Not supported yet.

1.3.6 Extensions

Messages with extension works very similar to messages without extensions. Look at this sample:

```
message Foo {
    extensions 100 to 199;
}

extend Foo {
    optional int32 bar = 123;
}
```

```
from protobuf3.fields import Int32Field
from protobuf3.message import Message


class Foo(Message):
    pass
```

```
Foo.add_field('bar', Int32Field(field_number=123, optional=True))
```

This should work even if message and extension declared in different files