Proto Plus for Python Documentation

Release 0.1.0

Luke Sneeringer

May 28, 2019

Contents

Beautiful, Pythonic protocol buffers.

This library provides a clean, readable, straightforward pattern for declaraing messages in protocol buffers. It provides a wrapper around the official implementation, so that using messages feels natural while retaining the power and flexibility of protocol buffers.

Warning: This tool is a proof of concept and is being iterated on rapidly. Feedback is welcome, but please do not try to use this in some kind of system where stability is an expectation.

CHAPTER 1

Installing

Install this library using `pip`:

```
$ pip install proto-plus
```

This library carries a dependency on the official implementation (`protobuf`), which may install a C component.

CHAPTER 2

Table of Contents

2.1 Messages

The fundamental building block in protocol buffers are messages. Messages are essentially permissive, strongly-typed structs (dictionaries), which have zero or more fields that may themselves contain primitives or other messages.

```
syntax = "proto3";

message Song {
  Composer composer = 1;
  string title = 2;
  string lyrics = 3;
  int32 year = 4;
}

message Composer {
  string given_name = 1;
  string family_name = 2;
}
```

The most common use case for protocol buffers is to write a `.proto` file, and then use the protocol buffer compiler to generate code for it.

2.1.1 Declaring messages

However, it is possible to declare messages directly. This is the equivalent message declaration in Python, using this library:

```
import proto

class Composer(proto.Message):
    given_name = proto.Field(proto.STRING, number=1)
```

(continues on next page)

(continued from previous page)

```
    family_name = proto.Field(proto.STRING, number=2)

class Song(proto.Message):
    composer = proto.Field(Composer, number=1)
    title = proto.Field(proto.STRING, number=2)
    lyrics = proto.Field(proto.STRING, number=3)
    year = proto.Field(proto.INT32, number=4)
```

A few things to note:

- This library only handles proto3.
- The `number` is really a field ID. It is *not* a value of any kind.
- All fields are optional (as is always the case in proto3). As a general rule, there is no distinction between setting the type's falsy value and not setting it at all (although there are exceptions to this in some cases).

Messages are fundamentally made up of *Fields*. Most messages are nothing more than a name and their set of fields.

2.1.2 Usage

Instantiate messages using either keyword arguments or a `dict` (and mix and matching is acceptable):

```
>>> song = Song(
...     composer={'given_name': 'Johann', 'family_name': 'Pachelbel'},
...     title='Canon in D',
...     year=1680,
... )
>>> song.composer.family_name
'Pachelbel'
>>> song.title
'Canon in D'
```

2.1.3 Enums

Enums are also supported:

```
import proto

class Genre(proto.Enum):
    GENRE_UNSPECIFIED = 0
    CLASSICAL = 1
    JAZZ = 2
    ROCK = 3

class Composer(proto.Message):
    given_name = proto.Field(proto.STRING, number=1)
    family_name = proto.Field(proto.STRING, number=2)

class Song(proto.Message):
    composer = proto.Field(Composer, number=1)
    title = proto.Field(proto.STRING, number=2)
    lyrics = proto.Field(proto.STRING, number=3)
    year = proto.Field(proto.INT32, number=4)
    genre = proto.Field(Genre, number=5)
```

All enums **must** begin with a `0` value, which is always the default in proto3 (and, as above, indistuiguishable from unset).

Enums utilize Python `enum.IntEnum` under the hood:

```
>>> song = Song(
...     composer={'given_name': 'Johann', 'family_name': 'Pachelbel'},
...     title='Canon in D',
...     year=1680,
...     genre=Genre.CLASSICAL,
... )
>>> song.genre
<Genre.CLASSICAL: 1>
>>> song.genre.name
'CLASSICAL'
>>> song.genre.value
1
```

Additionally, it is possible to provide strings or plain integers:

```
>>> song.genre = 2
>>> song.genre
<Genre.JAZZ: 2>
>>> song.genre = 'CLASSICAL'
<Genre.CLASSICAL: 1>
```

2.1.4 Serialization

Serialization and deserialization is available through the *`serialize()`* and *`deserialize()`* class methods.

The *`serialize()`* method is available on the message *classes* only, and accepts an instance:

```
serialized_song = Song.serialize(song)
```

The *`deserialize()`* method accepts a `bytes`, and returns an instance of the message:

```
song = Song.deserialize(serialized_song)
```

2.2 Fields

Fields are assigned using the *`Field`* class, instantiated within a *`Message`* declaration.

Fields always have a type (either a primitive, a message, or an enum) and a `number`.

```
import proto

class Composer(proto.Message):
    given_name = proto.Field(proto.STRING, number=1)
    family_name = proto.Field(proto.STRING, number=2)

class Song(proto.Message):
    composer = proto.Field(Composer, number=1)
    title = proto.Field(proto.STRING, number=2)
    lyrics = proto.Field(proto.STRING, number=3)
    year = proto.Field(proto.INT32, number=4)
```

For messages and enums, assign the message or enum class directly (as shown in the example above).

Note: For messages declared in the same module, it is also possible to use a string with the message class' name *if* the class is not yet declared, which allows for declaring messages out of order or with circular references.

2.2.1 Repeated fields

Some fields are actually repeated fields. In protocol buffers, repeated fields are generally equivalent to typed lists. In protocol buffers, these are declared using the **repeated** keyword:

```
message Album {
  repeated Song songs = 1;
  string publisher = 2;
}
```

Declare them in Python using the *RepeatedField* class:

```
class Album(proto.Message):
    songs = proto.RepeatedField(Song, number=1)
    publisher = proto.Field(proto.STRING, number=2)
```

2.2.2 Map fields

Similarly, some fields are map fields. In protocol buffers, map fields are equivalent to typed dictionaries, where the keys are either strings or integers, and the values can be any type. In protocol buffers, these use a special `map` syntax:

```
message Album {
  map<uint32, Song> track_list = 1;
  string publisher = 2;
}
```

Declare them in Python using the *MapField* class:

```
class Album(proto.Message):
    track_list = proto.MapField(proto.UINT32, Song, number=1)
    publisher = proto.Field(proto.STRING, number=2)
```

2.2.3 Oneofs (mutually-exclusive fields)

Protocol buffers allows certain fields to be declared as mutually exclusive. This is done by wrapping fields in a `oneof` syntax:

```
import "google/type/postal_address.proto";

message AlbumPurchase {
  Album album = 1;
  oneof delivery {
    google.type.PostalAddress postal_address = 2;
    string download_uri = 3;
  }
}
```

When using this syntax, protocol buffers will enforce that only one of the given fields is set on the message, and setting a field within the oneof will clear any others.

Declare this in Python using the `oneof` keyword-argument, which takes a string (which should match for all fields within the oneof):

```
from google.type.postal_address import PostalAddress

class AlbumPurchase(proto.Message):
    album = proto.Field(Album, number=1)
    postal_address = proto.Field(PostalAddress, number=2, oneof='delivery')
    download_uri = proto.Field(proto.STRING, number=3, oneof='delivery')
```

Warning: `oneof` fields **must** be declared consecutively, otherwise the C implementation of protocol buffers will reject the message. They need not have consecutive field numbers, but they must be declared in consecutive order.

2.3 Type Marshaling

Proto Plus provides a service that converts between protocol buffer objects and native Python types (or the wrapper types provided by this library).

This allows native Python objects to be used in place of protocol buffer messages where appropriate. In all cases, we return the native type, and are liberal on what we accept.

2.3.1 Well-known types

The following types are currently handled by Proto Plus:

Protocol buffer type	Python type	Nullable
`google.protobuf.BoolValue`	`bool`	Yes
`google.protobuf.BytesValue`	`bytes`	Yes
`google.protobuf.DoubleValue`	`float`	Yes
`google.protobuf.Duration`	`datetime.timedelta`	–
`google.protobuf.FloatValue`	`float`	Yes
`google.protobuf.Int32Value`	`int`	Yes
`google.protobuf.Int64Value`	`int`	Yes
`google.protobuf.ListValue`	`MutableSequence`	–
`google.protobuf.StringValue`	`str`	Yes
`google.protobuf.Struct`	`MutableMapping`	–
`google.protobuf.Timestamp`	`datetime.datetime`	Yes
`google.protobuf.UInt32Value`	`int`	Yes
`google.protobuf.UInt64Value`	`int`	Yes
`google.protobuf.Value`	JSON-encodable values	Yes

Note: Protocol buffers include well-known types for `Timestamp` and `Duration`, both of which have nanosecond precision. However, the Python `datetime` and `timedelta` objects have only microsecond precision.

If you *write* a timestamp field using a Python `datetime` value, any existing nanosecond precision will be overwritten.

2.3.2 Wrapper types

Additionally, every *Message* subclass is a wrapper class. The creation of the class also creates the underlying protocol buffer class, and this is registered to the marshal.

The underlying protocol buffer message class is accessible with the *pb()* class method.

2.4 Status

2.4.1 Features and Limitations

Nice things this library does:

- Idiomatic protocol buffer message representation and usage.
- Wraps the official protocol buffers implementation, and exposes its objects in the public API so that they are available where needed.

2.4.2 Upcoming work

- Specialized behavior for `google.protobuf.FieldMask` objects.

2.5 Reference

Below is a reference for the major classes and functions within this module.

It is split into two main sections:

- The *Message and Field* section (which uses the `message` and `fields` modules) handles constructing messages.
- The *Marshal* module handles translating between internal protocol buffer instances and idiomatic equivalents.

2.5.1 Message and Field

class proto.message.**Message**(*mapping=None*, ***kwargs*)

The abstract base class for a message.

Parameters

- **mapping** (*Union[dict, Message]*) – A dictionary or message to be used to determine the values for this message.
- **kwargs** (*dict*) – Keys and values corresponding to the fields of the message.

classmethod pb(*obj=None*, ***, *coerce: bool = False*)

Return the underlying protobuf Message class or instance.

Parameters

- **obj** – If provided, and an instance of `cls`, return the underlying protobuf instance.
- **coerce** (*bool*) – If provided, will attempt to coerce `obj` to `cls` if it is not already an instance.

classmethod serialize(*instance*) → bytes
Return the serialized proto.

Parameters **instance** – An instance of this message type, or something compatible (accepted by the type's constructor).

Returns The serialized representation of the protocol buffer.

Return type bytes

classmethod deserialize(*payload: bytes*) → proto.message.Message
Given a serialized proto, deserialize it into a Message instance.

Parameters **payload** (*bytes*) – The serialized proto.

Returns An instance of the message class against which this method was called.

Return type Message

class proto.fields.**Field**(*proto_type*, *, *number: int*, *message=None*, *enum=None*, *oneof: str = None*, *json_name: str = None*)
A representation of a type of field in protocol buffers.

descriptor
Return the descriptor for the field.

name
Return the name of the field.

package
Return the package of the field.

pb_type
Return the composite type of the field, or None for primitives.

class proto.fields.**MapField**(*key_type*, *value_type*, *, *number: int*, *message=None*, *enum=None*)
A representation of a map field in protocol buffers.

class proto.fields.**RepeatedField**(*proto_type*, *, *number: int*, *message=None*, *enum=None*)
A representation of a repeated field in protocol buffers.

class proto.enums.**Enum**
A enum object that also builds a protobuf enum descriptor.

class proto.enums.**ProtoEnumMeta**
A metaclass for building and registering protobuf enums.

2.5.2 Marshal

class proto.marshal.**Marshal**(*, *name: str*)
The translator between protocol buffer and Python instances.

The bulk of the implementation is in `BaseMarshal`. This class adds identity tracking: multiple instantiations of *Marshal* with the same name will provide the same instance.

Python Module Index

p

Index