Proto Plus for Python Documentation

Luke Sneeringer

May 19, 2022

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.

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). The only general way to determine whether a field was explicitly set to its falsy value or not set all is to mark it `optional`.
- Because all fields are optional, it is the responsibility of application logic to determine whether a necessary field has been set.

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 Assigning to Fields

One of the goals of proto-plus is to make protobufs feel as much like regular python objects as possible. It is possible to update a message's field by assigning to it, just as if it were a regular python object.

```
song = Song()
song.composer = Composer(given_name="Johann", family_name="Bach")

# Can also assign from a dictionary as a convenience.
song.composer = {"given_name": "Claude", "family_name": "Debussy"}

# Repeated fields can also be assigned
class Album(proto.Message):
    songs = proto.RepeatedField(Song, number=1)

a = Album()
songs = [Song(title="Canon in D"), Song(title="Little Fugue")]
a.songs = songs
```

Note: Assigning to a proto-plus message field works by making copies, not by updating references. This is necessary because of memory layout requirements of protocol buffers. These memory constraints are maintained by the protocol buffers runtime. This behavior can be surprising under certain circumstances, e.g. trying to save an alias to a nested field.

`proto.Message` defines a helper message, *`copy_from()`* to help make the distinction clear when reading code. The semantics of *`copy_from()`* are identical to the field assignment behavior described above.

```
composer = Composer(given_name="Johann", family_name="Bach")
song = Song(title="Tocatta and Fugue in D Minor", composer=composer)
composer.given_name = "Wilhelm"

# 'composer' is NOT a reference to song.composer
assert song.composer.given_name == "Johann"

# We CAN update the song's composer by assignment.
song.composer = composer
composer.given_name = "Carl"

# 'composer' is STILL not a reference to song.composer.
assert song.composer.given_name == "Wilhelm"

# It does work in reverse, though,
# if we want a reference we can access then update.
composer = song.composer
composer.given_name = "Gottfried"

assert song.composer.given_name == "Gottfried"

# We can use 'copy_from' if we're concerned that the code
# implies that assignment involves references.
composer = Composer(given_name="Elisabeth", family_name="Bach")
# We could also do Message.copy_from(song.composer, composer) instead.
Composer.copy_from(song.composer, composer)

assert song.composer.given_name == "Elisabeth"
```

2.1.4 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):
```

(continues on next page)

(continued from previous page)

```
    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>
```

2.1.5 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)
```

JSON serialization and deserialization are also available from message *classes* via the *`to_json()`* and *`from_json()`* methods.

```
json = Song.to_json(song)

new_song = Song.from_json(json)
```

Similarly, messages can be converted into dictionaries via the *`to_dict()`* helper method. There is no `from_dict()` method because the Message constructor already allows construction from mapping types.

```
song_dict = Song.to_dict(song)

new_song = Song(song_dict)
```

Note: Although Python's pickling protocol has known issues when used with untrusted collaborators, some frameworks do use it for communication between trusted hosts. To support such frameworks, protobuf messages **can** be pickled and unpickled, although the preferred mechanism for serializing proto messages is *`serialize()`*.

Multiprocessing example:

```
import proto
from multiprocessing import Pool

class Composer(proto.Message):
   name = proto.Field(proto.STRING, number=1)
   genre = proto.Field(proto.STRING, number=2)

composers = [Composer(name=n) for n in ["Bach", "Mozart", "Brahms", "Strauss"]]

with multiprocessing.Pool(2) as p:
   def add_genre(comp_bytes):
       composer = Composer.deserialize(comp_bytes)
       composer.genre = "classical"
       return Composer.serialize(composer)

updated_composers = [
   Composer.deserialize(comp_bytes)
   for comp_bytes in p.map(add_genre, (Composer.serialize(comp) for comp in␣
↪composers))
]
```

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)
```

Note: Elements **must** be appended individually for repeated fields of *struct.Value*.

```
class Row(proto.Message):
    values = proto.RepeatedField(proto.MESSAGE, number=1, message=struct.Value,)

>>> row = Row()
>>> values = [struct_pb2.Value(string_value="hello")]
>>> for v in values:
>>>     row.values.append(v)
```

Direct assignment will result in an error.

```
class Row(proto.Message):
    values = proto.RepeatedField(proto.MESSAGE, number=1, message=struct.Value,)

>>> row = Row()
>>> row.values = [struct_pb2.Value(string_value="hello")]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/local/google/home/busunkim/github/python-automl/.nox/unit-3-8/lib/python3.
↪8/site-packages/proto/message.py", line 543, in __setattr__
    self._pb.MergeFrom(self._meta.pb(**{key: pb_value}))
TypeError: Value must be iterable
```

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.

Warning: If a message is constructed with multiple variants of a single `oneof` passed to its constructor, the **last** keyword/value pair passed will be the final value set.

This is consistent with PEP-468, which specifies the order that keyword args are seen by called functions, and with the regular protocol buffers runtime, which exhibits the same behavior.

Example:

```
import proto

class Song(proto.Message):
    name = proto.Field(proto.STRING, number=1, oneof="identifier")
    database_id = proto.Field(proto.STRING, number=2, oneof="identifier")

s = Song(name="Canon in D minor", database_id="b5a37aad3")
assert "database_id" in s and "name" not in s

s = Song(database_id="e6aa708c7e", name="Little Fugue")
assert "name" in s and "database_id" not in s
```

2.2.4 Optional fields

All fields in protocol buffers are optional, but it is often necessary to check for field presence. Sometimes legitimate values for fields can be falsy, so checking for truthiness is not sufficient. Proto3 v3.12.0 added the `optional` keyword to field descriptions, which enables a mechanism for checking field presence.

In proto plus, fields can be marked as optional by passing `optional=True` in the constructor. The message *class* then gains a field of the same name that can be used to detect whether the field is present in message *instances*.

```
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)
    performer = proto.Field(proto.STRING, number=5, optional=True)

>>> s = Song(
...     composer={'given_name': 'Johann', 'family_name': 'Pachelbel'},
...     title='Canon in D',
...     year=1680,
...     genre=Genre.CLASSICAL,
... )
>>> Song.performer in s
False
>>> s.performer = 'Brahms'
>>> Song.performer in s
True
>>> del s.performer
>>> Song.performer in s
False
>>> s.performer = ""    # The mysterious, unnamed composer
>>> Song.performer in s
True
```

Under the hood, fields marked as optional are implemented via a synthetic one-variant `oneof`. See the protocolbuffers documentation for more information.

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`	Yes
`google.protobuf.StringValue`	`str`	Yes
`google.protobuf.Struct`	`MutableMapping`	Yes
`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. This library converts timestamps to an implementation of `datetime.datetime`, DatetimeWithNanoseconds, that includes nanosecond precision.

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

Note: Setting a `bytes` field from a string value will first base64 decode the string. This is necessary to preserve the original protobuf semantics when converting between Python dicts and proto messages. Converting a message containing a bytes field to a dict will base64 encode the bytes field and yield a value of type str.

```
import proto
from google.protobuf.json_format import ParseDict

class MyMessage(proto.Message):
    data = proto.Field(proto.BYTES, number=1)

msg = MyMessage(data=b"this is a message")
msg_dict = MyMessage.to_dict(msg)

# Note: the value is the base64 encoded string of the bytes field.
# It has a type of str, NOT bytes.
assert type(msg_dict['data']) == str

msg_pb = ParseDict(msg_dict, MyMessage.pb())
msg_two = MyMessage(msg_dict)

assert msg == msg_pb == msg_two
```

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.

- 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.
- The *Datetime Helpers* has datetime related helpers to maintain nanosecond precision.

2.5.1 Message and Field

class proto.message.**Message**(*mapping=None*, *, *ignore_unknown_fields=False*, ***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.
- **ignore_unknown_fields** (*Optional(bool)*) – If True, do not raise errors for unknown fields. Only applied if *mapping* is a mapping type or there are keyword parameters.
- **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 wrap(*pb*)

Return a Message object that shallowly wraps the descriptor.

Parameters **pb** – A protocol buffer object, such as would be returned by `pb()`.

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

classmethod to_json(*instance*, *, *use_integers_for_enums=True*, *including_default_value_fields=True*, *preserving_proto_field_name=False*) → str
Given a message instance, serialize it to json

Parameters

- **instance** – An instance of this message type, or something compatible (accepted by the type's constructor).
- **use_integers_for_enums** (*Optional(bool)*) – An option that determines whether enum values should be represented by strings (False) or integers (True). Default is True.
- **preserving_proto_field_name** (*Optional(bool)*) – An option that determines whether field name representations preserve proto case (snake_case) or use lowerCamelCase. Default is False.

Returns The json string representation of the protocol buffer.

Return type str

classmethod from_json(*payload*, *, *ignore_unknown_fields=False*) → proto.message.Message
Given a json string representing an instance, parse it into a message.

Parameters

- **paylod** – A json string representing a message.
- **ignore_unknown_fields** (*Optional(bool)*) – If True, do not raise errors for unknown fields.

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

Return type Message

classmethod to_dict(*instance*, *, *use_integers_for_enums=True*, *preserving_proto_field_name=True*, *including_default_value_fields=True*) → proto.message.Message
Given a message instance, return its representation as a python dict.

Parameters

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

- **use_integers_for_enums** (*Optional(bool)*) – An option that determines whether enum values should be represented by strings (False) or integers (True). Default is True.
- **preserving_proto_field_name** (*Optional(bool)*) – An option that determines whether field name representations preserve proto case (snake_case) or use lowerCamelCase. Default is True.
- **including_default_value_fields** (*Optional(bool)*) – An option that determines whether the default field values should be included in the results. Default is True.

Returns

A representation of the protocol buffer using pythonic data structures. Messages and map fields are represented as dicts, repeated fields are represented as lists.

Return type dict

classmethod copy_from(*instance*, *other*)
Equivalent for protobuf.Message.CopyFrom

Parameters

- **instance** – An instance of this message type
- **other** – (Union[dict, ~.Message): A dictionary or message to reinitialize the values for this message.

class proto.fields.**Field**(*proto_type*, ***, *number: int*, *message=None*, *enum=None*, *oneof: str = None*, *json_name: str = None*, *optional: bool = False*)
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 the primitive type if a primitive.

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*, *oneof: str = None*, *json_name: str = None*, *optional: bool = False*)
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.

2.5.3 Datetime Helpers

Helpers for `datetime`.

class proto.datetime_helpers.**DatetimeWithNanoseconds**

Track nanosecond in addition to normal datetime attrs.

Nanosecond can be passed only as a keyword argument.

classmethod from_rfc3339(*stamp*)

Parse RFC3339-compliant timestamp, preserving nanoseconds.

Parameters **stamp** (*str*) – RFC3339 stamp, with up to nanosecond precision

Returns an instance matching the timestamp string

Return type *DatetimeWithNanoseconds*

Raises `ValueError` – if *stamp* does not match the expected format

classmethod from_timestamp_pb(*stamp*)

Parse RFC3339-compliant timestamp, preserving nanoseconds.

Parameters **stamp** (`Timestamp`) – timestamp message

Returns an instance matching the timestamp message

Return type *DatetimeWithNanoseconds*

nanosecond

nanosecond precision.

Type Read-only

replace(**args*, ***kw*)

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if d == date(2002, 12, 31), then d.replace(day=26) == date(2002, 12, 26). NOTE: nanosecond and microsecond are mutually exclusive arguments.

rfc3339()

Return an RFC3339-compliant timestamp.

Returns Timestamp string according to RFC3339 spec.

Return type (str)

timestamp_pb()

Return a timestamp message.

Returns Timestamp message

Return type (`Timestamp`)

Python Module Index

p

Index