Protocol Buffer Basics: Python

This tutorial provides a basic Python programmer's introduction to working with protocol buffers. By walking through creating a simple example application, it shows you how to

- Define message formats in a `.proto` file.
- Use the protocol buffer compiler.
- Use the Python protocol buffer API to write and read messages.

This isn't a comprehensive guide to using protocol buffers in Python. For more detailed reference information, see the Protocol Buffer Language Guide, the Python API Reference, the Python Generated Code Guide, and the Encoding Reference.

Why Use Protocol Buffers?

The example we're going to use is a very simple "address book" application that can read and write people's contact details to and from a file. Each person in the address book has a name, an ID, an email address, and a contact phone number.

How do you serialize and retrieve structured data like this? There are a few ways to solve this problem:

- Use Python pickling. This is the default approach since it's built into the language, but it doesn't deal well with schema evolution, and also doesn't work very well if you need to share data with application written in C++ or Java.
- You can invent an ad-hoc way to encode the data items into a single string – such as encoding 4 ints as "12:3:-23:67". This is a simple and flexible approach, although it does require writing one-off encoding and parsing code, and the parsing imposes a small run-time cost. This works best for encoding very simple data.
- Serialize the data to XML. This approach can be very attractive since XML is (sort of) human readable and there are binding libraries for lots of languages. This can be a good choice if you want to share data with other applications/projects. However, XML is notoriously space intensive, and encoding/decoding it can impose a huge performance penalty on applications. Also, navigating an XML DOM tree is considerably more complicated than navigating simple fields in a class normally would be.

Protocol buffers are the flexible, efficient, automated solution to solve exactly this problem. With protocol buffers, you write a `.proto` description of the data structure you wish to store. From that, the protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format. The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit. Importantly, the protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.

Where to Find the Example Code

The example code is included in the source code package, under the "examples" directory. Download it here.

Defining Your Protocol Format

To create your address book application, you'll need to start with a `.proto` file. The definitions in a `.proto` file are simple: you add a *message* for each data structure you want to serialize, then specify a name and a type for each field in the message. Here is the `.proto` file that defines your messages, `addressbook.proto`.

```
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;
}
```

As you can see, the syntax is similar to C++ or Java. Let's go through each part of the file and see what it does.

The `.proto` file starts with a package declaration, which helps to prevent naming conflicts between different projects. In Python, packages are normally determined by directory structure, so the `package` you define in your `.proto` file will have no effect on the generated code. However, you should still declare one to avoid name collisions in the Protocol Buffers name space as well as in non-Python languages.

Next, you have your message definitions. A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including `bool`, `int32`, `float`, `double`, and `string`. You can also add further structure to your messages by using other message types as field types – in the above example the `Person` message contains `PhoneNumber` messages, while the `AddressBook` message contains `Person` messages. You can even define message types nested inside other messages – as you can see, the `PhoneNumber` type is defined inside `Person`. You can also define `enum` types if you want one of your fields to have one of a predefined list of values – here you want to specify that a phone number can be one of `MOBILE`, `HOME`, or `WORK`.

The " = 1", " = 2" markers on each element identify the unique "tag" that field uses in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements. Each element in a repeated field requires re-encoding the tag number, so repeated fields are particularly good candidates for this optimization.

Each field must be annotated with one of the following modifiers:

- `required`: a value for the field must be provided, otherwise the message will be considered "uninitialized". Serializing an uninitialized message will raise an exception. Parsing an uninitialized message will fail. Other than this, a required field behaves exactly like an optional field.
- `optional`: the field may or may not be set. If an optional field value isn't set, a default value is used. For simple types, you can specify your own default value, as we've done for the phone number `type` in the example. Otherwise, a system default is used: zero for numeric types, the empty string for strings, false for bools. For embedded messages, the default value is always the "default instance" or "prototype" of the message, which has none of its fields set. Calling the accessor to get the value of an optional (or required) field which has not been explicitly set always returns that field's default value.
- `repeated`: the field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the protocol buffer. Think of repeated fields as dynamically sized arrays.

> **Required Is Forever** You should be very careful about marking fields as `required`. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some engineers at Google have come to the conclusion that using `required` does more harm than good; they prefer to use only `optional` and `repeated`. However, this view is not universal.

You'll find a complete guide to writing `.proto` files – including all the possible field types – in the Protocol Buffer Language Guide. Don't go looking for facilities similar to class inheritance, though – protocol buffers don't do that.

Compiling Your Protocol Buffers

Now that you have a `.proto`, the next thing you need to do is generate the classes you'll need to read and write `AddressBook` (and hence `Person` and `PhoneNumber`) messages. To do this, you need to run the protocol buffer compiler `protoc` on your `.proto`:

1. If you haven't installed the compiler, download the package and follow the instructions in the README.
2. Now run the compiler, specifying the source directory (where your application's source code lives – the current directory is used if you don't provide a value), the destination directory (where you want the generated code to go; often the same as `$SRC_DIR`), and the path to your `.proto`. In this case, you...:

   ```
   protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/addressbook.proto
   ```

   Because you want Python classes, you use the `--python_out` option – similar options are provided for other supported languages.

This generates `addressbook_pb2.py` in your specified destination directory.

The Protocol Buffer API

Unlike when you generate Java and C++ protocol buffer code, the Python protocol buffer compiler doesn't generate your data access code for you directly. Instead (as you'll see if you look at `addressbook_pb2.py`) it generates special descriptors for all your messages, enums, and fields, and some mysteriously empty classes, one for each message type:

```
class Person(message.Message):
  __metaclass__ = reflection.GeneratedProtocolMessageType

  class PhoneNumber(message.Message):
    __metaclass__ = reflection.GeneratedProtocolMessageType
    DESCRIPTOR = _PERSON_PHONENUMBER
  DESCRIPTOR = _PERSON

class AddressBook(message.Message):
  __metaclass__ = reflection.GeneratedProtocolMessageType
  DESCRIPTOR = _ADDRESSBOOK
```

The important line in each class is `__metaclass__ = reflection.GeneratedProtocolMessageType`. While the details of how Python metaclasses work is beyond the scope of this tutorial, you can think of them as like a template for creating classes. At load time, the `GeneratedProtocolMessageType` metaclass uses the specified descriptors to create all the Python methods you need to work with each message type and adds them to the relevant classes. You can then use the fully-populated classes in your code.

The end effect of all this is that you can use the `Person` class as if it defined each field of the `Message` base class as a regular field. For example, you could write:

```
import addressbook_pb2
person = addressbook_pb2.Person()
person.id = 1234
person.name = "John Doe"
person.email = "jdoe@example.com"
phone = person.phone.add()
phone.number = "555-4321"
phone.type = addressbook_pb2.Person.HOME
```

Note that these assignments are not just adding arbitrary new fields to a generic Python object. If you were to try to assign a field that isn't defined in the `.proto` file, an `AttributeError` would be raised. If you assign a field to a value of the wrong type, a `TypeError` will be raised. Also, reading the value of a field before it has been set returns the default value.

```
person.no_such_field = 1  # raises AttributeError
person.id = "1234"        # raises TypeError
```

For more information on exactly what members the protocol compiler generates for any particular field definition, see the Python generated code reference.

Enums

Enums are expanded by the metaclass into a set of symbolic constants with integer values. So, for example, the constant `addressbook_pb2.Person.WORK` has the value 2.

Standard Message Methods

Each message class also contains a number of other methods that let you check or manipulate the entire message, including:

- `IsInitialized()`: checks if all the required fields have been set.

- `__str__()`: returns a human-readable representation of the message, particularly useful for debugging. (Usually invoked as `str(message)` or `print message`.)
- `CopyFrom(other_msg)`: overwrites the message with the given message's values.
- `Clear()`: clears all the elements back to the empty state.

These methods implement the `Message` interface. For more information, see the complete API documentation for `Message`.

Parsing and Serialization

Finally, each protocol buffer class has methods for writing and reading messages of your chosen type using the protocol buffer binary format. These include:

- `SerializeToString()`: serializes the message and returns it as a string. Note that the bytes are binary, not text; we only use the `str` type as a convenient container.
- `ParseFromString(data)`: parses a message from the given string.

These are just a couple of the options provided for parsing and serialization. Again, see the `Message` API reference for a complete list.

> **Protocol Buffers and O-O Design** Protocol buffer classes are basically dumb data holders (like structs in C++); they don't make good first class citizens in an object model. If you want to add richer behaviour to a generated class, the best way to do this is to wrap the generated protocol buffer class in an application-specific class. Wrapping protocol buffers is also a good idea if you don't have control over the design of the `.proto` file (if, say, you're reusing one from another project). In that case, you can use the wrapper class to craft an interface better suited to the unique environment of your application: hiding some data and methods, exposing convenience functions, etc. **You should never add behaviour to the generated classes by inheriting from them**. This will break internal mechanisms and is not good object-oriented practice anyway.

Writing A Message

Now let's try using your protocol buffer classes. The first thing you want your address book application to be able to do is write personal details to your address book file. To do this, you need to create and populate instances of your protocol buffer classes and then write them to an output stream.

Here is a program which reads an `AddressBook` from a file, adds one new `Person` to it based on user input, and writes the new `AddressBook` back out to the file again. The parts which directly call or reference code generated by the protocol compiler are highlighted.

```
#! /usr/bin/python

import addressbook_pb2
import sys

# This function fills in a Person message based on user input.
def PromptForAddress(person):
  person.id = int(raw_input("Enter person ID number: "))
  person.name = raw_input("Enter name: ")

  email = raw_input("Enter email address (blank for none): ")
  if email != "":
    person.email = email
```

```
  while True:
    number = raw_input("Enter a phone number (or leave blank to finish): ")
    if number == "":
      break

    phone_number = person.phone.add()
    phone_number.number = number

    type = raw_input("Is this a mobile, home, or work phone? ")
    if type == "mobile":
      phone_number.type = addressbook_pb2.Person.MOBILE
    elif type == "home":
      phone_number.type = addressbook_pb2.Person.HOME
    elif type == "work":
      phone_number.type = addressbook_pb2.Person.WORK
    else:
      print "Unknown phone type; leaving as default value."

# Main procedure:  Reads the entire address book from a file,
#   adds one person based on user input, then writes it back out to the same
#   file.
if len(sys.argv) != 2:
  print "Usage:", sys.argv[0], "ADDRESS_BOOK_FILE"
  sys.exit(-1)

address_book = addressbook_pb2.AddressBook()

# Read the existing address book.
try:
  f = open(sys.argv[1], "rb")
  address_book.ParseFromString(f.read())
  f.close()
except IOError:
  print sys.argv[1] + ": Could not open file.  Creating a new one."

# Add an address.
PromptForAddress(address_book.person.add())

# Write the new address book back to disk.
f = open(sys.argv[1], "wb")
f.write(address_book.SerializeToString())
f.close()
```

Reading A Message

Of course, an address book wouldn't be much use if you couldn't get any information out of it! This example reads the file created by the above example and prints all the information in it.

```
#! /usr/bin/python

import addressbook_pb2
import sys

# Iterates though all people in the AddressBook and prints info about them.
def ListPeople(address_book):
```

```
  for person in address_book.person:
    print "Person ID:", person.id
    print "  Name:", person.name
    if person.HasField('email'):
      print "  E-mail address:", person.email

    for phone_number in person.phone:
      if phone_number.type == addressbook_pb2.Person.MOBILE:
        print "  Mobile phone #: ",
      elif phone_number.type == addressbook_pb2.Person.HOME:
        print "  Home phone #: ",
      elif phone_number.type == addressbook_pb2.Person.WORK:
        print "  Work phone #: ",
      print phone_number.number

# Main procedure:  Reads the entire address book from a file and prints all
#   the information inside.
if len(sys.argv) != 2:
  print "Usage:", sys.argv[0], "ADDRESS_BOOK_FILE"
  sys.exit(-1)

address_book = addressbook_pb2.AddressBook()

# Read the existing address book.
f = open(sys.argv[1], "rb")
address_book.ParseFromString(f.read())
f.close()

ListPeople(address_book)
```

Extending a Protocol Buffer

Sooner or later after you release the code that uses your protocol buffer, you will undoubtedly want to "improve" the protocol buffer's definition. If you want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible – and you almost certainly do want this – then there are some rules you need to follow. In the new version of the protocol buffer:

- you *must not* change the tag numbers of any existing fields.
- you *must not* add or delete any required fields.
- you *may* delete optional or repeated fields.
- you *may* add new optional or repeated fields but you must use fresh tag numbers (i.e. tag numbers that were never used in this protocol buffer, not even by deleted fields).

(There are some exceptions to these rules, but they are rarely used.)

If you follow these rules, old code will happily read new messages and simply ignore any new fields. To the old code, optional fields that were deleted will simply have their default value, and deleted repeated fields will be empty. New code will also transparently read old messages. However, keep in mind that new optional fields will not be present in old messages, so you will need to either check explicitly whether they're set with `has_`, or provide a reasonable default value in your `.proto` file with `[default = value]` after the tag number. If the default value is not specified for an optional element, a type-specific default value is used instead: for strings, the default value is the empty string. For booleans, the default value is false. For numeric types, the default value is zero. Note also that if you added a new repeated field, your new code will not be able to tell whether it was left empty (by new code) or never set at all (by old code) since there is no `has_` flag for it.

Advanced Usage

Protocol buffers have uses that go beyond simple accessors and serialization. Be sure to explore the Python API reference to see what else you can do with them.

One key feature provided by protocol message classes is *reflection*. You can iterate over the fields of a message and manipulate their values without writing your code against any specific message type. One very useful way to use reflection is for converting protocol messages to and from other encodings, such as XML or JSON. A more advanced use of reflection might be to find differences between two messages of the same type, or to develop a sort of "regular expressions for protocol messages" in which you can write expressions that match certain message contents. If you use your imagination, it's possible to apply Protocol Buffers to a much wider range of problems than you might initially expect!

Reflection is provided as part of the `Message` interface.

Last updated 四月 *2, 2012.*