Apache Avro# 1.10.0 Getting Started (Python)

Table of contents

This is a short guide for getting started with Apache Avro# using Python. This guide only covers using Avro for data serialization; see Patrick Hunt's [Avro RPC Quick Start](#) for a good introduction to using Avro for RPC.

1 Notice for Python 3 users

A package called "avro-python3" had been provided to support Python 3 previously, but the codebase was consolidated into the "avro" package and that supports both Python 2 and 3 now. The avro-python3 package will be removed in the near future, so users should use the "avro" package instead. They are mostly API compatible, but there's a few minor difference (e.g., function name capitalization, such as avro.schema.Parse vs avro.schema.parse).

2 Download

For Python, the easiest way to get started is to install it from PyPI. Python's Avro API is available over [PyPi](#).

```
$ python3 -m pip install avro
```

The official releases of the Avro implementations for C, C++, C#, Java, PHP, Python, and Ruby can be downloaded from the [Apache Avro# Releases](#) page. This guide uses Avro 1.10.0, the latest version at the time of writing. Download and unzip *avro-1.10.0.tar.gz*, and install via `python setup.py` (this will probably require root privileges). Ensure that you can `import avro` from a Python prompt.

```
$ tar xvf avro-1.10.0.tar.gz
$ cd avro-1.10.0
$ python setup.py install
$ python
>>> import avro # should not raise ImportError
```

Alternatively, you may build the Avro Python library from source. From your the root Avro directory, run the commands

```
$ cd lang/py/
$ python3 -m pip install -e .
$ python
```

3 Defining a schema

Avro schemas are defined using JSON. Schemas are composed of [primitive types](#) (`null`, `boolean`, `int`, `long`, `float`, `double`, `bytes`, and `string`) and [complex types](#) (`record`, `enum`, `array`, `map`, `union`, and `fixed`). You can learn more about Avro schemas and types from the specification, but for now let's start with a simple schema example, *user.avsc*:

```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

This schema defines a record representing a hypothetical user. (Note that a schema file can only contain a single schema definition.) At minimum, a record definition must include its type (`"type": "record"`), a name (`"name": "User"`), and fields, in this case `name`, `favorite_number`, and `favorite_color`. We also define a namespace (`"namespace": "example.avro"`), which together with the name attribute defines the "full name" of the schema (`example.avro.User` in this case).

Fields are defined via an array of objects, each of which defines a name and type (other attributes are optional, see the [record specification](#) for more details). The type attribute of a field is another schema object, which can be either a primitive or complex type. For example, the `name` field of our User schema is the primitive type `string`, whereas the `favorite_number` and `favorite_color` fields are both `unions`, represented by JSON arrays. `unions` are a complex type that can be any of the types listed in the array; e.g., `favorite_number` can either be an `int` or `null`, essentially making it an optional field.

4 Serializing and deserializing without code generation

Data in Avro is always stored with its corresponding schema, meaning we can always read a serialized item, regardless of whether we know the schema ahead of time. This allows us to perform serialization and deserialization without code generation. Note that the Avro Python library does not support code generation.

Try running the following code snippet, which serializes two users to a data file on disk, and then reads back and deserializes the data file:

```
import avro.schema
from avro.datafile import DataFileReader, DataFileWriter
```

```
from avro.io import DatumReader, DatumWriter

schema = avro.schema.parse(open("user.avsc", "rb").read())

writer = DataFileWriter(open("users.avro", "wb"), DatumWriter(), schema)
writer.append({"name": "Alyssa", "favorite_number": 256})
writer.append({"name": "Ben", "favorite_number": 7, "favorite_color": "red"})
writer.close()

reader = DataFileReader(open("users.avro", "rb"), DatumReader())
for user in reader:
    print user
reader.close()
```

This outputs:

```
{u'favorite_color': None, u'favorite_number': 256, u'name': u'Alyssa'}
{u'favorite_color': u'red', u'favorite_number': 7, u'name': u'Ben'}
```

Do make sure that you open your files in binary mode (i.e. using the modes wb or rb respectively). Otherwise you might generate corrupt files due to automatic replacement of newline characters with the platform-specific representations.

Let's take a closer look at what's going on here.

```
schema = avro.schema.parse(open("user.avsc", "rb").read())
```

avro.schema.parse takes a string containing a JSON schema definition as input and outputs a avro.schema.Schema object (specifically a subclass of Schema, in this case RecordSchema). We're passing in the contents of our user.avsc schema file here.

```
writer = DataFileWriter(open("users.avro", "wb"), DatumWriter(), schema)
```

We create a DataFileWriter, which we'll use to write serialized items to a data file on disk. The DataFileWriter constructor takes three arguments:

- The file we'll serialize to
- A DatumWriter, which is responsible for actually serializing the items to Avro's binary format (DatumWriters can be used separately from DataFileWriters, e.g., to perform IPC with Avro).
- The schema we're using. The DataFileWriter needs the schema both to write the schema to the data file, and to verify that the items we write are valid items and write the appropriate fields.

```
writer.append({"name": "Alyssa", "favorite_number": 256})
writer.append({"name": "Ben", "favorite_number": 7, "favorite_color": "red"})
```

We use `DataFileWriter.append` to add items to our data file. Avro records are represented as Python `dicts`. Since the field `favorite_color` has type `["int", "null"]`, we are not required to specify this field, as shown in the first append. Were we to omit the required `name` field, an exception would be raised. Any extra entries not corresponding to a field are present in the `dict` are ignored.

```
reader = DataFileReader(open("users.avro", "rb"), DatumReader())
```

We open the file again, this time for reading back from disk. We use a `DataFileReader` and `DatumReader` analagous to the `DataFileWriter` and `DatumWriter` above.

```
for user in reader:
    print user
```

The `DataFileReader` is an iterator that returns `dicts` corresponding to the serialized items.

Page 5