Other Faces of Python

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living and working from home in the northeast of Brazil. pcnorton@rbox.co.

I'd like to talk about uses for serialized data this time, looking at them through contrasting language-neutral formats: YAML and protocol buffers. These will be the basis for discussing an interesting Python interpreter, specially built to make working with protocol buffers easier.

Wikipedia (https://en.wikipedia.org/wiki/Serialization) has a really great, straightforward definition of serialization: "the process of translating data structures or object state into a format that can be stored." YAML is a really easy format for serialization/deserialization for simple Python data types since it represents data structures in a way that's really similar to how Python does; in my experience, however, this is not so much the case for defined types.

YAML

So let's talk about YAML. YAML (standing for YAML Ain't Markup Language, or possibly Yet Another Markup Language, or maybe something else) is recognizable in the wild as the prolific format where the whitespace is relevant and indentation is incredibly important, and which breaks if someone naively makes a single whitespace change (like many people's first impression of Python!). Its goal is to be able to serialize and deserialize data in a format that is human-readable (text) and comprehensible (line breaks matter in a way that is similar to written language, indentation guides the structure, etc.).

YAML also has all sorts of interesting features, like the ability to name a structure and reuse it multiple times, and graft that onto various other locations, similar to using variables. (Some interesting discussion about the full range of what it can do is available at http://yaml.org.) YAML is often used as more than just a serialization format since it has the ability to, for example, declare blocks, repeat them, etc. A recent post at https://blog.atomist.com/in-defense-of-yaml/ reminded me of some of the work I've been doing. In short, YAML is hugely useful, but it also has limits that should be respected.

One trivial example of its usefulness is:

```
this:
  is: a mapping with
  different: value types
  here: 3
```

which would look like this in Python:

```
{"this": {"is": "a mapping with", "different": "value types", "here": 3}}
```

Declaring a reusable block (called an anchor) is this simple, and you can see how it's expanded by running this in the Python REPL using the pyyaml module (see http://pyyaml.org for more info):

```
>>> import yaml
>>> yaml.load("""
... this: &use_this_anchor
...   is: cool
...
```

```
... here: *use_this_anchor
... """)
{'this': {'is': 'cool'}, 'here': {'is': 'cool'}}
```

This can greatly reduce size and repetition. It's clear that human-readable and understandable formats like YAML have been a huge positive change. Because of their widespread use and acceptance, people feel less need to create poorly defined ad-hoc configuration formats. The fact that software is shipped using YAML means that they're being configured via plaintext data structures. That's a big win!

YAML and Configuration

These formats make your configuration much easier to comprehend. You almost don't have to do any work. It also means that your configuration often seems to be self-documenting—we can read about specific data types, quantities, etc., and with only a little familiarity with the system you're working with, it's almost obvious what you (or the program) are trying to express. For example, the following is probably going to make sense if I tell you that it's a section of YAML-formatted configuration for the Envoy proxy, a Layer 7 proxy (sometimes called a *service mesh*; see envoyproxy.io for more info):

```
static_resources:
  clusters:
  - circuit_breakers:
      thresholds:
      - max_pending_requests: 8192
        max_requests: 8192
        max_retries: 1000
        priority: DEFAULT
      - max_pending_requests: 8192
        max_requests: 8192
        max_retries: 1000
        priority: HIGH
    connect_timeout: 0.5s
    hosts:
    - socket_address:
        address: foohost-ssl
        port_value: 443
    lb_policy: ROUND_ROBIN
    name: foohost
    per_connection_buffer_limit_bytes: 3100000
    tls_context: {}
    type: STRICT_DNS
```

It doesn't provide the person reading it with the larger picture, but you can use this as a starting point—it's probably configuration that governs the behavior of a listening port and multiple hosts behind a load-balancer .

One limit to YAML's flexibility, though, is that small nested changes prevent the use of anchors. So there are two threshold entries that look almost exactly alike. But the difference in the `priority` key means that the entire structure must be repeated. As you can imagine, this sort of inconsistency can become irritating as the size of the data gets larger.

Using YAML as the representation of the data comes with another weakness: there is no built-in checking that a message has the right shape or the right structure—essentially it doesn't come with any type checking. Let's focus on this, because better type checking is great, especially when it is easily achievable at a low cost.

Skycfg, Protocol Buffers, and YAML

So how can someone do better than YAML? One answer is to use protocol buffers (usually just called *protobufs*). Protocol buffers are also widely used, and one important role they play is in defining APIs. Two examples that have been increasingly adopted over the past few years are the Envoy proxy (mentioned above) and Kubernetes (https://kubernetes.io). In both cases, protocol buffers are used to define the structures used by the API internally, while their external-facing REST API and configuration will accept messages in other formats (e.g., YAML) but translate them and check them against the API definition. This means that a REST API may be used with YAML data, but when this data gets into the system and is deserialized, it'll get checked against the protocol buffer definitions, which are the real source of truth.

In order to make using protobufs easier, the folks at Stripe have created *Skycfg*, which is based on a special-purpose language whose syntax and behavior are derived from Python. While Python is usually considered a "general-purpose" language, Skycfg has an entirely different reason for existing. It is based on a variant of Python whose primary goal is to be as easy to use as the standard CPython but to be limited in a way that focuses on enhancing the process of configuring large software systems. The language Skycfg is based on was once called "Skylark" but was renamed "Starlark" (https://blog.bazel.build/2018/08/17/starlark.html) and released as part of the Bazel build system (http://bazel.build).

With Skycfg, protocol buffer messages are compiled from a neutral format into a Golang-specific library and imported into Skycfg, and your own variation of Skycfg is built for your own use. When your custom interpreter is run, you can create objects using their protocol buffer message definitions, and they maintain their type information per the underlying Golang runtime. The intent is that the protobuf data structures remain strongly typed and will not have implicit conversions done to them. Messages are defined ahead of time; they are created, updated, compared, etc. using the syntax of Python (Skycfg), and doing

things this way maintains a strongly typed, statically checkable configuration.

Some Examples

So let's have some show and tell.

This bit of YAML is pretty easy to comprehend:

```
access_log_path: /var/log/envoy/admin_access_log
address:
 socket_address:
  address: 127.0.0.1
  port_value: 1234
```

This is short and sweet, and as configuration it seems pretty straightforward. As mentioned earlier, the user/operator must make sure to avoid some common mistakes. If I add a tab instead of spaces, it breaks in a way that may not be obvious. If I make the port value >65k, I may not notice it, but it's clearly outside the range of available ports. If I mistype something it's still valid YAML, but it doesn't mean anything to the program that reads it.

By contrast, generating this in Skycfg code has the upfront cost of writing some Python, with a disproportionately large benefit: I can create configuration messages where the type of the message is known and statically checked. So, unlike YAML, this doesn't allow us to graft the wrong message into the wrong place. In addition, the fields of the messages are also type checked, and we can create these messages with proper functions instead of being YAML anchors, in which you can't replace at the granularity of one element of a list or a mapping.

Just in case you are interested in the entire v2 API that Envoy provides, the messages being generated below are documented further at https://www.envoyproxy.io/docs/envoy/latest/api-v2/api.

```
# -*- Python -*-
v2_bootstrap = proto.package("envoy.config.bootstrap.v2")
# Code we write, the "//" is specific to Skycfg/starlark
load("//common_helpers.sky", "to_struct")
load("//common_helpers.sky", "envoy_address")
# this gets code the envoy maintainers wrote,
# built into the main.go
v2_core = proto.package("envoy.api.v2.core")

# Bootstrap message sections
def admin_msg(access_log_path, address, port):
   """This generates the :admin:
   section, including the access log path
   and the listen address of this server.
   """
   admin = v2_bootstrap.Admin(
      access_log_path=access_log_path,
      address=envoy_address(address, port))
   return admin

def node_msg(cluster, node_id):
   """The cluster name should match whatever
   we're using to identify the cluster, the
   node_id should match the IP address or
   hostname.
   """
   return v2_core.Node(
      id=node_id,
      cluster=cluster)

def build_bootstrap_msg(
   admin, node, static_resources, stats_sinks):
   """The core initial config is the
   bootstrap message - this is essentially
   the jumping-off point that we plant in
   `/etc/envoy/envoy.yaml`
   """
   return v2_bootstrap.Bootstrap(
      admin=admin,
      node=node,
      static_resources=static_resources,
      stats_sinks=stats_sinks)
```

To use this, you need to build the interpreter, which is really simple. Look at https://github.com/pcn/followprotocol for the code and try it out.

This is a pretty neat trick, and the benefits become clearer once you consider the power of the combination of Python's syntax to easily and dynamically script up the configuration and then add strong type checking, where the definitions are supplied by the authors of the server side, so you don't have to track changes. So, for instance, when a breaking change appears in a newer API version, it will be made clear to you just by generating the configuration. The server doesn't have to try the bad configuration and reject it.

What's more, protocol buffers provide a way to update message formats by adding to the end of a structure. This allows for compatibility as you change your messages.

Also, notice that the above snippets do the right thing when type bounds are violated. So if I change the port number in the `bootstrap.sky` to something far outside of the bounds of a TCP port, the following would happen:

```
followprotocol$ ./followprotocol envoy.sky
panic: ValueError: value 12345678910 overflows type 'uint32'.
goroutine 1 [running]:
main.main()
            /home/spacey/go/src/github.com/pcn/
followprotocol/main.go:94 +0x7c6
```

Full disclosure: the definition of the message specifies that this uint32 must be <= 65536, but as of this writing, there seems to be

an issue with this, so I overflowed with a larger number for this example to contrast it with CPython behavior.

Any time this sort of check catches an error, it is like free time being given back to you! One of the most common problems with configuring programs is that in order to know whether a configuration is even valid—things that are supposed to be strings look and act like strings, numbers are numbers, etc.—you need to pass them into a running process where that process validates it (e.g., in the best case with a `--check-config` flag or something along those lines). But even a checker often won't be able to tell you that you've violated a constraint that has to do with the type and not the format. Some things that a strongly typed checker can know is that, for example, you've configured a number value to be larger than an unsigned 8-bit integer, and it will only accept a signed 8-bit integer. Or you create a list that contains strings and numbers, but for a situation where the required list is only able to accept strings. These, in addition to the actual syntax errors, are much, much more difficult to catch, and the goal is that the process of creating the configuration makes it clear that these errors are present. It also turns the problem of perhaps indenting YAML a bit oddly into a problem of Python indentation. Since the syntax is Python, you can use Python syntax checkers to your advantage, though they're imperfect. In any case, the fact that these messages are declared and dealt with by the Skycfg protocol buffer handling means a whole class of checks is largely done for you.

Another effect of this is that since Skycfg isn't a general-purpose language, once a message is created, handling it is done outside of Python syntax. With only a little bit of experience with Golang, you can take the messages that are generated by Skycfg and do something with them there. They could also be saved to a file or shipped out over a network socket—you do need to add this in for yourself. Oddly, you may find that after doing all this work, you end up writing everything out as YAML, per my example repository. So it's always a good idea to keep that option in mind.

Templating

Lastly, let's discuss the Skycfg approach as compared to another method that's often used to generate configuration: use a templating language/macro language like Jinja2, Mustache, or maybe Erb if you're using Ruby. Configuration syntax for simpler things tend to be quite comprehensible at first, but some parts can grow and change to the point where you end up gaining domain-specific knowledge about particular sections of configuration that because of their irregularity have nothing to do with anything else—they sort of make up their own rules as they go along. The configurations for Apache and Nginx are very expressive, but they also make it very challenging to just confirm that they are acceptably formatted. Using a templating approach works very, very well when the problem is simple, and, fortunately, most configurations can be made to be simple and can work by fitting them into a pretty simple template.

Unfortunately, generating YAML with templates, even with regular, simple YAML, gets tricky as soon as you attempt to graft new data structures onto the existing text of a partial message via appending templates. It doesn't seem like it should be so hard, but it turns out that it often is.

By defining a service in terms of protocol buffers, and by using that to make systems that are meant to be operated programmatically via an API, the authors of Envoy and Kubernetes (among others) are inviting the use of a solution like Skycfg in order to generate the desired configuration faster and more safely. I recommend taking a look at Skycfg if you find yourself working with a system that defines itself via protobuf messages.

Note about the last column: In the last column, I mentioned that I'd look to see whether there's a way to make a change to something like the `zip` built-in work throughout a codebase. So far, I haven't found a way to do that well (the idea I had in my head failed so hard...), so I'll look a bit more to see if it does, in fact, seem possible.