Camel Documentation

Release 0.1.2

Eevee (Alex Munroe)

May 29, 2017

Contents

Camel is a Python serialization library that forces you to explicitly describe how to dump or load your types. It's good for you, just like eating your vegetables.

Contents:

CHAPTER 1

Camel overview

Camel is intended as a replacement for libraries like `pickle` or PyYAML, which automagically serialize any type they come across. That seems convenient at first, but in any large or long-lived application, the benefits are soon outweighed by the costs:

- You can't move, rename, or delete any types that are encoded in a pickle.
- Even private implementation details of your class are encoded in a pickle by default, which means you can't change them either.
- Because pickle's behavior is recursive, it can be difficult to know which types are pickled.
- Because pickle's behavior is recursive, you may inadvertently pickle far more data than necessary, if your objects have caches or reified properties. In extreme cases you may pickle configuration that's no longer correct when the pickle is loaded.
- Since pickles aren't part of your codebase and are rarely covered by tests, you may not know you've broken pickles until your code hits production... or much later.
- Pickle in particular is very opaque, even when using the ASCII format. It's easy to end up with a needlessly large pickle by accident and have no visibility into what's being stored in it, or to break loading a large pickle and be unable to recover gracefully or even tell where the problem is.
- Automagically serialized data is hard enough to load back into your *own* application. Loading it anywhere else is effectively out of the question.

It's certainly possible to whip pickle or PyYAML into shape manually by writing `__reduce__` or representer functions, but their default behavior is still automagic, so you can't be sure you didn't miss something. Also, nobody actually does it, so merely knowing it's possible doesn't help much.

Camel's philosophy

Explicit is better than implicit.

Complex is better than complicated.

Readability counts.

If the implementation is hard to explain, it's a bad idea.

In the face of ambiguity, refuse the temptation to guess.

—The Zen of Python

Serialization is hard. We can't hide that difficulty, only delay it for a while. And it *will* catch up with you.

A few people in the Python community have been rallying against pickle and its ilk for a while, but when asked for alternatives, all we can do is mumble something about writing functions. Well, that's not very helpful.

Camel forces you to write all your own serialization code, then wires it all together for you. It's backed by YAML, which is ostensibly easy for humans to read — and has explicit support for custom types. Hopefully, after using Camel, you'll discover you've been tricked into making a library of every type you serialize, the YAML name you give it, and exactly how it's formatted. All of this lives in your codebase, so someone refactoring a class will easily stumble upon its serialization code. Why, you could even use this knowledge to load your types into an application written in a different language, or turn them into a documented format!

Let's see some code already

Let's!

Here's the Table example from a talk Alex Gaynor gave at PyCon US 2014. Initially we have some square tables.

```python
class Table(object):
    def __init__(self, size):
        self.size = size

    def __repr__(self):
        return "<Table {self.size!r}>".format(self=self)
```

We want to be able to serialize these, so we write a *dumper* and a corresponding *loader* function. We'll also need a *registry* to store these functions:

```python
from camel import CamelRegistry
my_types = CamelRegistry()

@my_types.dumper(Table, 'table', version=1)
def _dump_table(table):
    return dict(
        size=table.size,
    )

@my_types.loader('table', version=1)
def _load_table(data, version):
    return Table(data["size"])
```

Note: This example is intended for Python 3. With Python 2, `dict(size=...)` will create a "size" key that's a `bytes`, which will be serialized as `!!binary`. It will still work, but it'll be ugly, and won't interop with Python 3. If you're still on Python 2, you should definitely use dict literals with `unicode` keys.

Now we just give this registry to a `Camel` object and ask it to dump for us:

```
from camel import Camel
table = Table(25)
print(Camel([my_types]).dump(table))
```

```
!table;1
size: 25
```

Unlike the simple example given in the talk, we can also dump arbitrary structures containing Tables with no extra effort:

```
data = dict(chairs=[], tables=[Table(25), Table(36)])
print(Camel([my_types]).dump(data))
```

```
chairs: []
tables:
- !table;1
  size: 25
- !table;1
  size: 36
```

And load them back in:

```
print(Camel([my_types]).load("[!table;1 {size: 100}]"))
```

```
[<Table 100>]
```

Versioning

As you can see, all serialized Tables are tagged as `!table;1`. The `table` part is the argument we gave to `@dumper` and `@loader`, and the `1` is the version number.

Version numbers mean that when the time comes to change your class, you don't have anything to worry about. Just write a new loader and dumper with a higher version number, and fix the old loader to work with the new code:

```
# Tables can be rectangles now!
class Table(object):
    def __init__(self, height, width):
        self.height = height
        self.width = width

    def __repr__(self):
        return "<Table {self.height!r}x{self.width!r}>".format(self=self)

@my_types.dumper(Table, 'table', version=2)
def _dump_table_v2(table):
    return dict(
        height=table.height,
        width=table.width,
    )

@my_types.loader('table', version=2)
def _load_table_v2(data, version):
    return Table(data["height"], data["width"])

@my_types.loader('table', version=1)
```

```
def _load_table_v1(data, version):
    edge = data["size"] ** 0.5
    return Table(edge, edge)

table = Table(7, 10)
print(Camel([my_types]).dump(table))
```

```
!table;2
height: 7
width: 10
```

More on versions

Versions are expected to be positive integers, presumably starting at 1. Whenever your class changes, you have two options:

1. Fix the dumper and loader to preserve the old format but work with the new internals.
2. Failing that, write new dumpers and loaders and bump the version.

One of the advantages of Camel is that your serialization code is nothing more than functions returning Python structures, so it's very easily tested. Even if you end up with dozens of versions, you can write test cases for each without ever dealing with YAML at all.

You might be wondering whether there's any point to having more than one version of a dumper function. By default, only the dumper with the highest version for a type is used. But it's possible you may want to stay backwards-compatible with other code — perhaps an older version of your application or library — and thus retain the ability to write out older formats. You can do this with `Camel.lock_version()`:

```
@my_types.dumper(Table, 'table', version=1)
def _dump_table_v1(table):
    return dict(
        # not really, but the best we can manage
        size=table.height * table.width,
    )

camel = Camel([my_types])
camel.lock_version(Table, 1)
print(camel.dump(Table(5, 7)))
```

```
!table;1
size: 35
```

Obviously you might lose some information when round-tripping through an old format, but sometimes it's necessary until you can fix old code.

Note that version locking only applies to dumping, not to loading. For loading, there are a couple special versions you can use.

Let's say you delete an old class whose information is no longer useful. While cleaning up all references to it, you discover it has Camel dumpers and loaders. What about all your existing data? No problem! Just use a version of `all` and return a dummy object:

```
class DummyData(object):
    def __init__(self, data):
        self.data = data
```

```
@my_types.loader('deleted-type', version=all)
def _load_deleted_type(data, version):
    return DummyData(data)
```

`all` overrides *all* other loader versions (hence the name). You might instead want to use `any`, which is a fallback for when the version isn't recognized:

```
@my_types.loader('table', version=any)
def _load_table(data, version):
    if 'size' in data:
        # version 1
        edge = data['size'] ** 0.5
        return Table(edge, edge)
    else:
        # version 2?)
        return Table(data['height'], data['width'])
```

Versions must still be integers; a non-integer version will cause an immediate parse error.

Going versionless

You might be thinking that the version numbers everywhere are an eyesore, and your data would be much prettier if it only used `!table`.

Well, yes, it would. But you'd lose your ability to bump the version, so you'd have to be *very very sure* that your chosen format can be adapted to any possible future changes to your class.

If you are, in fact, *very very sure*, then you can use a version of `None`. This is treated like an *infinite* version number, so it will always be used when dumping (unless overridden by a version lock).

Similarly, an unversioned tag will look for a loader with a `None` version, then fall back to `all` or `any`. The order versions are checked for is thus:

- `None`, if appropriate
- `all`
- Numeric version, if appropriate
- `any`

There are deliberately no examples of unversioned tags here. Designing an unversioned format requires some care, and a trivial documentation example can't do it justice.

Supported types

By default, Camel knows how to load and dump all types in the YAML type registry to their Python equivalents, which are as follows.

YAML tag	Python type
`!!binary`	`bytes`
`!!bool`	`bool`
`!!float`	`float`
`!!int`	`int` (or `long` on Python 2)
`!!map`	`dict`
`!!merge`	—
`!!null`	`NoneType`
`!!omap`	`collections.OrderedDict`
`!!seq`	`list` or `tuple` (dump only)
`!!set`	`set`
`!!str`	`str` (`unicode` on Python 2)
`!!timestamp`	`datetime.date` or `datetime.datetime` as appropriate

Note: PyYAML tries to guess whether a bytestring is "really" a string on Python 2, but Camel does not. Serializing *any* bytestring produces an ugly base64-encoded `!!binary` representation.

This is a **feature**.

The following additional types are loaded by default, but **not dumped**. If you want to dump these types, you can use the existing `camel.PYTHON_TYPES` registry.

YAML tag	Python type
`!!python/complex`	`complex`
`!!python/frozenset`	`frozenset`
`!!python/namespace`	`types.SimpleNamespace` (Python 3.3+)
`!!python/tuple`	`tuple`

Other design notes

- Camel will automatically use the C extension if available, and fall back to a Python implementation otherwise. The PyYAML documentation says it doesn't have this behavior because there are some slight differences between the implementations, but fails to explain what they are.
- `Camel.load()` is safe by default. There is no calling of arbitrary functions or execution of arbitrary code just from loading data. There is no "dangerous" mode. PyYAML's `!!python/object` and similar tags are not supported. (Unless you write your own loaders for them, of course.)
- There is no "OO" interface, where dumpers or loaders can be written as methods with special names. That approach forces a class to have only a single representation, and more importantly litters your class with junk unrelated to the class itself. Consider this a cheap implementation of traits. You can fairly easily build support for this in your application if you really *really* want it.
- Yes, you may have to write a lot of boring code like this:

```
@my_types.dumper(SomeType, 'sometype')
def _dump_sometype(data):
    return dict(
        foo=data.foo,
        bar=data.bar,
        baz=data.baz,
        ...
    )
```

I strongly encourage you *not* to do this automatically using introspection, which would defeat the point of using Camel. If it's painful, step back and consider whether you really need to be serializing as much as you are, or whether your classes need to be so large.

- There's no guarantee that the data you get will actually be in the correct format for that version. YAML is meant for human beings, after all, and human beings make mistakes. If you're concerned about this, you could combine Camel with something like the Colander library.

Known issues

Camel is a fairly simple wrapper around PyYAML, and inherits many of its problems. Only YAML 1.1 is supported, not 1.2, so a handful of syntactic edge cases may not parse correctly. Loading and dumping are certainly slower and more memory-intensive than pickle or JSON. Unicode handling is slightly clumsy. Python-specific types use tags starting with `!!`, which is supposed for be for YAML's types only.

Formatting and comments are not preserved during a round-trip load and dump. The ruamel.yaml library is a fork of PyYAML that solves this problem, but it only works when using the pure-Python implementation, which would hurt Camel's performance even more. Opinions welcome.

PyYAML has several features that aren't exposed in Camel yet: dumpers that work on subclasses, loaders that work on all tags with a given prefix, and parsers for plain scalars in custom formats.

CHAPTER 2

Brief YAML reference

There is no official YAML reference guide. The YAML website only offers the YAML specification, which is a dense and thorny tome clearly aimed at implementers. I suspect this has greatly hampered YAML's popularity.

In the hopes of improving this situation, here is a very quick YAML overview that should describe the language almost entirely. Hopefully it's useful whether or not you use Camel.

Overall structure and design

As I see it, YAML has two primary goals: to support encoding any arbitrary data structure; and to be easily read and written by humans. If only the spec shared that last goal.

Human-readability means that much of YAML's syntax is optional, wherever it would be unambiguous and easier on a human. The trade-off is more complexity in parsers and emitters.

Here's an example document, configuration for some hypothetical app:

```
database:
    username: admin
    password: foobar  # TODO get prod passwords out of config
    socket: /var/tmp/database.sock
    options: {use_utf8: true}
memcached:
    host: 10.0.0.99
workers:
  - host: 10.0.0.101
    port: 2301
  - host: 10.0.0.102
    port: 2302
```

YAML often has more than one way to express the same data, leaving a human free to use whichever is most convenient. More convenient syntax tends to be more contextual or whitespace-sensitive. In the above document, you can see that indenting is enough to make a nested mapping. Integers and booleans are automatically distinguished from unquoted strings, as well.

General syntax

As of 1.2, YAML is a strict superset of JSON. Any valid JSON can be parsed in the same structure with a YAML 1.2 parser.

YAML is designed around Unicode, not bytes, and its syntax assumes Unicode input. There is no syntactic mechanism for giving a character encoding; the parser is expected to recognize BOMs for UTF-8, UTF-16, and UTF-32, but otherwise a byte stream is assumed to be UTF-8.

The only vertical whitespace characters are U+000A LINE FEED and U+000D CARRIAGE RETURN. The only horizontal whitespace characters are U+0009 TAB and U+0020 SPACE. Other control characters are not allowed anywhere. Otherwise, anything goes.

YAML operates on *streams*, which can contain multiple distinct structures, each parsed individually. Each structure is called a *document*.

A document begins with `---` and ends with `...`. Both are optional, though a `...` can only be followed by directives or `---`. You don't see multiple documents very often, but it's a very useful feature for sending intermittent chunks of data over a single network connection. With JSON you'd usually put each chunk on its own line and delimit with newlines; YAML has support built in.

Documents may be preceded by *directives*, in which case the `---` is required to indicate the end of the directives. Directives are a `%` followed by an identifier and some parameters. (This is how directives are distinguished from a bare document without `---`, so the first non-blank non-comment line of a document can't start with a `%`.)

There are only two directives at the moment: `%YAML` specifies the YAML version of the document, and `%TAG` is used for tag shorthand, described in *More on tags*. Use of directives is, again, fairly uncommon.

Comments may appear anywhere. `#` begins a comment, and it runs until the end of the line. In most cases, comments are whitespace: they don't affect indentation level, they can appear between any two tokens, and a comment on its own line is the same as a blank line. The few exceptions are not too surprising; for example, you can't have a comment between the key and colon in `key:`.

A YAML document is a graph of values, called *nodes*. See *Kinds of value*.

Nodes may be prefixed with up to two properties: a *tag* and an *anchor*. Order doesn't matter, and both are optional. Properties can be given to any value, regardless of kind or style.

Tags

Tags are prefixed with `!` and describe the *type* of a node. This allows for adding new types without having to extend the syntax or mingle type information with data. Omitting the tag leaves the type to the parser's discretion; usually that means you'll get lists, dicts, strings, numbers, and other simple types.

You'll probably only see tags in two forms:

- `!foo` is a "local" tag, used for some custom type that's specific to the document.
- `!!bar` is a built-in YAML type from the YAML tag repository. Most of these are inferred from plain data — `!!seq` for sequences, `!!int` for numbers, and so on — but a few don't have dedicated syntax and have to be given explicitly.

  For example, `!!binary` is used for representing arbitrary binary data encoded as base64. So `!!binary aGVsbG8=` would parse as the bytestring `hello`.

There's much more to tags, most of which is rarely used in practice. See *More on tags*.

Anchors

The other node property is the *anchor*, which is how YAML can store recursive data structures. Anchor names are prefixed with `&` and can't contain whitespace, brackets, braces, or commas.

An *alias node* is an anchor name prefixed with `*`, and indicates that the node with that anchor name should occur in both places. (Alias nodes can't have properties themselves; the properties of the anchored node are used.) For example, you might share configuration:

```
host1:
    &common-host
    os: linux
    arch: x86_64
host2: *common-host
```

Or serialize a list that contains itself:

```
&me [*me]
```

Note: This is **not** a copy. The exact same value is reused.

Anchor names act somewhat like variable assignments: at any point in the document, the parser only knows about the anchors it's seen so far, and a second anchor with the same name takes precedence. This means that aliases cannot refer to anchors that appear later in the document.

Anchor names aren't intended to carry information, which unfortunately means that most YAML parsers throw them away, and re-serializing a document will get you anchor names like `ANCHOR1`.

Kinds of value

Values come in one of three *kinds*, which reflect the general "shape" of the data. Scalars are individual values; sequences are ordered collections; mappings are unordered associations. Each can be written in either a whitespace-sensitive *block style* or a more compact and explicit *flow style*.

Scalars

Most values in a YAML document will be *plain scalars*. They're defined by exclusion: if it's not anything else, it's a plain scalar. Technically, they can only be flow style, so they're really "plain flow scalar style" scalars.

Plain scalars are the most flexible kind of value, and may resolve to a variety of types from the YAML tag repository:

- Integers become, well, integers (`!!int`). Leading `0`, `0b`, and `0x` are recognized as octal, binary, and hexadecimal. `_` is allowed, and ignored. Curiously, `:` is allowed and treated as a base 60 delimiter, so you can write a time as `1:59` and it'll be loaded as the number of seconds, 119.
- Floats become floats (`!!float`). Scientific notation using `e` is also recognized. As with integers, `_` is ignored and `:` indicates base 60, though only the last component can have a fractional part. Positive infinity, negative infinity, and not-a-number are recognized with a leading dot: `.inf`, `-.inf`, and `.nan`.
- `true` and `false` become booleans (`!!bool`). `y`, `n`, `yes`, `no`, `on`, and `off` are allowed as synonyms. Uppercase and title case are also recognized.
- `~` and `null` become nulls (`!!null`), which is `None` in Python. A completely empty value also becomes null.

- ISO8601 dates are recognized (`!!timestamp`), with whitespace allowed between the date and time. The time is also optional, and defaults to midnight UTC.
- `=` is a special value (`!!value`) used as a key in mappings. I've never seen it actually used, and the thing it does is nonsense in many languages anyway, so don't worry about it. Just remember you can't use = as a plain string.
- `<<` is another special value (`!!merge`) used as a key in mappings. This one is actually kind of useful; it's described below in *Merge keys*.

Note: The YAML spec has a notion of *schemas*, sets of types which are recognized. The recommended schema is "core", which doesn't actually require `!!timestamp` support. I think the idea is to avoid requiring support for types that may not exist natively — a Perl YAML parser can't reasonably handle `!!timestamp` out of the box, because Perl has no built-in timestamp type. So while you could technically run into a parser that doesn't support floats (the "failsafe" schema only does strings!), it probably won't come as a surprise.

Otherwise, it's a string. Well. Probably. As part of tag resolution (see *More on tags*), an application is allowed to parse plain scalars however it wants; you might add logic that parses `1..5` as a range type, or you might recognize keywords and replace them with special objects. But if you're doing any of that, you're hopefully aware of it.

Between the above parsing and conflicts with the rest of YAML's syntax, for a plain scalar to be a string, it must meet these restrictions:

- It must not be `true`, `false`, `yes`, `no`, `y`, `n`, `on`, `off`, `null`, or any of those words in uppercase or title case, which would all be parsed as booleans or nulls.
- It must not be `~`, which is null. If it's a mapping key, it must not be `=` or `<<`, which are special key values.
- It must not be something that looks like a number or timestamp. I wouldn't bet on anything that consists exclusively of digits, dashes, underscores, and colons.
- The first character must not be any of: ``[ ] { } , # & * ! | > ' " % @ ` ``. All of these are YAML syntax for some other kind of construct.
- If the first character is `?`, `:`, or `-`, the next character must not be whitespace. Otherwise it'll be parsed as a block mapping or sequence.
- It must not contain ` #` or `: `, which would be parsed as a comment or a key. A hash not preceded by space or a colon not followed by space is fine.
- If the string is inside a flow collection (i.e., inside `[...]` or `{...}`), it must not contain any of `[ ] { } ,`, which would all be parsed as part of the collection syntax.
- Leading and trailing whitespace are ignored.
- If the string is broken across lines, then the newline and any adjacent whitespace are collapsed into a single space.

That actually leaves you fairly wide open; the biggest restriction is on the first character. You can have spaces, you can wrap across lines, you can include whatever (non-control) Unicode you want.

If you need explicit strings, you have some other options.

Strings

YAML has lots of ways to write explicit strings. Aside from plain scalars, there are two other *flow scalar styles*.

Single-quoted strings are surrounded by `'`. Single quotes may be escaped as `''`, but otherwise no escaping is done at all. You may wrap over multiple lines, but the newline and any surrounding whitespace becomes a single space. A line containing only whitespace becomes a newline.

Double-quoted strings are surrounded by `"`. Backslash escapes are recognized:

Sequence	Result
`\0`	U+0000 NULL
`\a`	U+0007 BELL
`\b`	U+0008 BACKSPACE
`\t`	U+0009 CHARACTER TABULATION
`\n`	U+000A LINE FEED
`\v`	U+000B LINE TABULATION
`\f`	U+000C FORM FEED
`\r`	U+000D CARRIAGE RETURN
`\e`	U+001B ESCAPE
`\"`	U+0022 QUOTATION MARK
`\/`	U+002F SOLIDUS
`\\`	U+005C REVERSE SOLIDUS
`\N`	U+0085 NEXT LINE
`\_`	U+00A0 NO-BREAK SPACE
`\L`	U+2028 LINE SEPARATOR
`\P`	U+2029 PARAGRAPH SEPARATOR
`\xNN`	Unicode character `NN`
`\uNNNN`	Unicode character `NNNN`
`\UNNNNNNNN`	Unicode character `NNNNNNNN`

As usual, you may wrap a double-quoted string across multiple lines, but the newline and any surrounding whitespace becomes a single space. As with single-quoted strings, a line containing only whitespace becomes a newline. You can escape spaces and tabs to protect them from being thrown away. You can also escape a newline to preserve any trailing whitespace on that line, but throw away the newline and any leading whitespace on the next line.

These rules are weird, so here's a contrived example:

```
"line  \
    one

    line two\n\
\ \ line three\nline four\n
line five
"
```

Which becomes:

```
line  one
line two
  line three
line four
 line five
```

Right, well, I hope that clears that up.

There are also two *block scalar styles*, both consisting of a header followed by an indented block. The header is usually just a single character, indicating which block style to use.

`|` indicates *literal style*, which preserves all newlines in the indented block. `>` indicates *folded style*, which performs the same line folding as with quoted strings. Escaped characters are not recognized in either style. Indentation, the initial newline, and any leading blank lines are always ignored.

So to represent this string:

```
This is paragraph one.

This is paragraph two.
```

You could use either literal style:

```
|
    This is paragraph one.

    This is paragraph two.
```

Or folded style:

```
>
    This is
    paragraph one.


    This
    is paragraph
    two.
```

Obviously folded style is more useful if you have paragraphs with longer lines. Note that there are two blank lines between paragraphs in folded style; a single blank line would be parsed as a single newline.

The header has some other features, but I've never seen them used. It consists of up to three parts, with no intervening whitespace.

1. The character indicating which block style to use.
2. Optionally, the indentation level of the indented block, relative to its parent. You only need this if the first line of the block starts with a space, because the space would be interpreted as indentation.
3. Optionally, a "chomping" indicator. The default behavior is to include the final newline as part of the string, but ignore any subsequent empty lines. You can use − here to ignore the final newline as well, or use + to preserve all trailing whitespace verbatim.

You can put a comment on the same line as the header, but a comment on the next line would be interpreted as part of the indented block. You can also put a tag or an anchor before the header, as with any other node.

Sequences

Sequences are ordered collections, with type `!!seq`. They're pretty simple.

Flow style is a comma-delimited list in square brackets, just like JSON: `[one, two, 3]`. A trailing comma is allowed, and whitespace is generally ignored. The contents must also be written in flow style.

Block style is written like a bulleted list:

```
- one
- two
- 3
- a plain scalar that's
  wrapped across multiple lines
```

Indentation determines where each element ends, and where the entire sequence ends.

Other blocks may be nested without intervening newlines:

```
- - one one
  - one two
- - two one
  - two two
```

Mappings

Mappings are unordered, er, mappings, with type `!!map`. The keys must be unique, but may be of any type. Also, they're unordered.

Did I mention that mappings are **unordered**? The order of the keys in the document is irrelevant and arbitrary. If you need order, you need a sequence.

Flow style looks unsurprisingly like JSON: `{x: 1, y: 2}`. Again, a trailing comma is allowed, and whitespace doesn't matter.

As a special case, inside a sequence, you can write a single-pair mapping without the braces. So `[a: b, c: d, e: f]` is a sequence containing three mappings. This is allowed in block sequences too, and is used for the ordered mapping type `!!omap`.

Block style is actually a little funny. The canonical form is a little surprising:

```
? x
: 1
? y
: 2
```

`?` introduces a key, and `:` introduces a value. You very rarely see this form, because the `?` is optional as long as the key and colon are all on one line (to avoid ambiguity) and the key is no more than 1024 characters long (to avoid needing infinite lookahead).

So that's more commonly written like this:

```
x: 1
y: 2
```

The explicit `?` syntax is more useful for complex keys. For example, it's the only way to use block styles in the key:

```
? >
    If a train leaves Denver at 5:00 PM traveling at 90 MPH, and another
    train leaves New York City at 10:00 PM traveling at 80 MPH, by how many
    minutes are you going to miss your connection?
: Depends whether we're on Daylight Saving Time or not.
```

Other than the syntactic restrictions, an implicit key isn't special in any way and can also be of any type:

```
true: false
null: null
up: down
[0, 1]: [1, 0]
```

It's fairly uncommon to see anything but strings as keys, though, since languages often don't support it. Python can't have lists and dicts as dict keys; Perl 5 and JavaScript only support string keys; and so on.

Unlike sequences, you may **not** nest another block inside a block mapping on the same line. This is invalid:

```
one: two: buckle my shoe
```

But this is fine:

```
- one: 1
  two: 2
- three: 3
  four: 4
```

You can also nest a block sequence without indenting:

```
foods:
- burger
- fries
drinks:
- soda
- iced tea
```

One slight syntactic wrinkle: in either style, the colon must be followed by whitespace. `foo:bar` is a single string, remember. (For JSON's sake, the whitespace can be omitted if the colon immediately follows a flow sequence, a flow mapping, or a quoted string.)

Merge keys

These are written << and have type `!!merge`. A merge key should have another mapping (or sequence of mappings) as its value. Each mapping is merged into the containing mapping, with any existing keys left alone. The actual << key is never shown to the application.

This is generally used in conjunction with anchors to share default values:

```
defaults: &DEFAULTS
    use-tls: true
    verify-host: true
host1:
    <<: *DEFAULTS
    hostname: example.com
host2:
    <<: *DEFAULTS
    hostname: example2.com
host3:
    <<: *DEFAULTS
    hostname: example3.com
    # we have a really, really good reason for doing this, really
    verify-host: false
```

More on tags

`!!str` is actually an illusion.

Tag names are actually URIs, using UTF-8 percent-encoding. YAML suggests using the `tag:` scheme and your domain name to help keep tags globally unique; for example, the string tag is really `tag:yaml.org,2002:str`. (Domain names can change hands over time, hence the inclusion of a year.)

That's quite a mouthful, and wouldn't be recognized as a tag anyway, because tags have to start with `!`. So tags are written in shorthand with a prefix, like `!foo!bar`. The `!foo!` is a *named tag handle* that expands to a given prefix, kind of like XML namespacing. Named tag handles must be defined by a `%TAG` directive before the document:

```
%TAG !foo! tag:example.com,2015:app/
```

A tag of `!foo!bar` would then resolve to `tag:example.com,2015:app/bar`.

I've never seen `%TAG` used in practice. Instead, everyone uses the two special tag handles.

- The *primary tag handle* is `!`, which by default expands to `!`. So `!bar` just resolves to `!bar`, a *local tag*, specific to the document and not expected to be unique.
- The *secondary tag handle* is `!!`, which by default expands to `tag:yaml.org,2002:`, the prefix YAML uses for its own built-in types. So `!!bar` resolves to `tag:yaml.org,2002:bar`, and the tag for a string would more commonly be written as `!!str`. Defining new tags that use `!!` is impolite.

Both special handles can be reassigned with `%TAG`, just like any other handle. An important (and confusing) point here is that the **resolved** name determines whether or not a tag is local; how it's written is irrelevant. You're free to do this:

```
%TAG !foo! !foo-types/
```

Now `!foo!bar` is shorthand for `!foo-types/bar`, which is a local tag. You can also do the reverse:

```
%TAG ! tag:example.com,2015:legacy-types/
```

Which would make `!bar` a global tag! This is deliberate, as a quick way to convert an entire document from local tags to global tags.

You can reassign `!!`, too. But let's not.

Tags can also be written *verbatim* as `!<foo>`, in which case `foo` is taken to be the resolved final name of the tag, ignoring `%TAG` and any other resolution mechanism. This is the only way to write a global tag without using `%TAG`, since tags must start with a `!`.

Every node has a tag, whether it's given one explicitly or not. Nodes without explicit tags are given one of two special *non-specific* tags: `!` for quoted and folded scalars; or `?` for sequences, mappings, and plain scalars.

The `?` tag tells the application to do *tag resolution*. Technically, this means the application can do any kind of arbitrary inspection to figure out the type of the node. In practice, it just means that scalars are inspected to see whether they're booleans, integers, floats, whatever else, or just strings.

The `!` tag forces a node to be interpreted as a basic built-in type, based on its kind: `!!str`, `!!seq`, or `!!map`. You can explicitly give the `!` tag to a node if you want, for example writing `! true` or `! 133` to force parsing as strings. Or you could use quotes. Just saying.

CHAPTER 3

API reference

class `camel.`**`Camel`**(*registries=()*)

`dump`(*data*)

`load`(*data*)

`load_all`(*data*)

`load_first`(*data*)

`lock_version`(*cls*, *version*)

`make_dumper`(*stream*)

`make_loader`(*stream*)

class `camel.`**`CamelDumper`**(**args*, ***kwargs*)

Subclass of yaml's *SafeDumper* that scopes representers to the instance, rather than to the particular class, because damn.

`add_multi_representer`(*data_type*, *representer*)

`add_representer`(*data_type*, *representer*)

`represent_binary`(*data*)

class `camel.`**`CamelLoader`**(**args*, ***kwargs*)

Subclass of yaml's *SafeLoader* that scopes constructors to the instance, rather than to the particular class, because damn.

`add_constructor`(*data_type*, *constructor*)

`add_implicit_resolver`(*tag*, *regexp*, *first*)

`add_multi_constructor`(*data_type*, *constructor*)

`add_path_resolver`(**args*, ***kwargs*)

class `camel.`**`CamelRegistry`**(*tag_prefix=u'!'*)

`dumper`(*cls*, *tag*, *version*)

`freeze`()

`frozen` = **False**

`inject_dumpers`(*dumper*, *version_locks=None*)

`inject_loaders`(*loader*)

`loader`(*tag*, *version*)

`run_constructor`(*constructor*, *version*, **yaml_args*)

`run_representer`(*representer*, *tag*, *dumper*, *data*)

exception `camel.`**`DuplicateVersion`**

Python Module Index

Index