FAQ

What features does StrictYAML remove?

Stupid feature	Example YAML	Example pyyaml/ruamel/poyo	Example StrictYAML

Implicit typing	`x: yes`
	`y: null`
		`load(yaml) == \`
		`  {"x": True, "y": None}`
			`load(yaml) == \`
			`  {"x": "yes", "y": "null"}`

Binary data	`evil: !!binary`
	` evildata`
		`load(yaml) == \`
		`{'evil': b'z\xf8\xa5u\xabZ'}`
			`raises TagTokenDisallowed`

Explicit tags	`x: !!int 5`	`load(yaml) == {"x": 5}`	`raises TagTokenDisallowed`

Node anchors and refs	`x: &id001`
	`  a: 1`
	`y: *id001`
		`load(yaml) == \`
		` {'x': {'a': 1}, 'y': {'a': 1}}`
			`raises NodeAnchorDisallowed`

Flow style	`x: 1`
	`b: {c: 3, d: 4}`
		`load(yaml) == \`
		`{'x': {'a': 1}, 'y': {'a': 1}}`
			`raises FlowStyleDisallowed`

Duplicate keys	`x: 1`
	`x: 2`
		`load(yaml) == \`
		`{'x': 2}`
			`raises DuplicateKeysDisallowed`

What is YAML?

YAML is a simple, human readable format for representing associative and hierarchical data:

```
receipt: Oz-Ware Purchase Invoice
date: 2012-08-06
customer:
  first name: Harry
  family name: Potter
address: |-
  4 Privet Drive,
  Little Whinging,
  England
items:
  - part_no: A4786
    description: Water Bucket (Filled)
    price: 1.47
    quantity:  4
    - part_no: E1628
      description: High Heeled "Ruby" Slippers
      size: 8
      price: 133.7
      quantity: 1
```

Key features:

- Things which are associated with other things - delimited by the colon (:).
- Ordered lists of things - delimited by the prepended dash (-).
- Multi-line strings - delimited by the bar (|) if there is another newline at the end of the string, or bar + dash (|-) if not.
- Indentation describing the hierarchy of data.
- Maps directly to data types common to most high level languages - lists, dicts, scalars.

This is all you really need to know.

Why should I care about YAML?

YAML is the clearest and easiest to read format for representing hierarchical data.

It is an ideal format for configuration and simple DSLs. It easily maps on to python's lists and dicts, making the data that is parsed easy to manipulate and use.

When should I use a validator and when should I not?

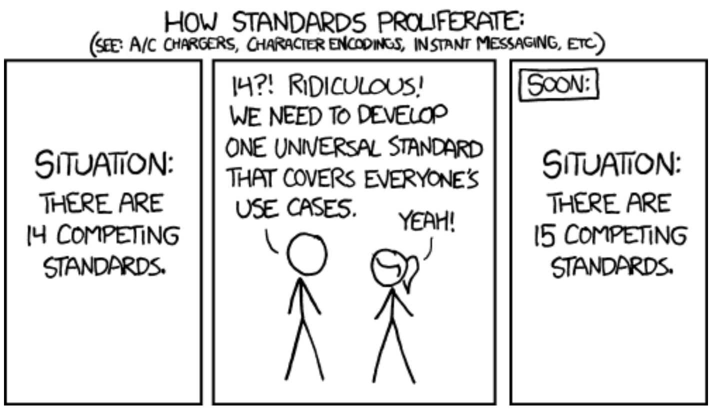
When starting out on greenfield projects it's much quicker not to create a validator. In such cases it's often more prudent to just parse the YAML and convert the strings explicitly on the fly (e.g. int(yaml['key'])).

If the YAML is also going to be largely under the control of the developer it also might not make sense to write a validator either.

If you have written software that is going to parse YAML from a source you do *not* control - especially by somebody who might make a mistake - then it probably does make sense to write a validator.

You can start off without using a validator and then add one later.

Why should I use strictyaml instead of ordinary YAML?


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

StrictYAML is *not* a new standard. It's:

- YAML without implicit typing - which was a terrible idea.
- YAML with all of the other bullshit removed.
- An optional YAML validator.

If you already have YAML, StrictYAML will usually parse it.

What is wrong with implicit typing?

Imagine you are parsing a DSL to represent movie scripts:

```
- Don Corleone: Do you have faith in my judgment?
- Clemenza: Yes
- Don Corleone: Do I have your loyalty?
```

Parse output of pyyaml, ruamel.yaml and Poyo:

```
>>> from ruamel.yaml import load
>>> load(the_godfather)
[{'Don Corleone': 'Do you have faith in my judgement?'},
 {'Clemenza': True},
 {'Don Corleone': 'Do I have your loyalty?'},]
```

Wait, Clemenza said what??

Parse output of StrictYAML without validators:

```
>>> from strictyaml import load, List, MapPattern, Str
>>> load(the_godfather)
[{'Don Corleone': 'Do you have faith in my judgement?'},
 {'Clemenza': 'Yes'},
 {'Don Corleone': 'Do I have your loyalty?'},]
```

Let's try the Matrix instead:

```
- Morpheus: Do you believe in fate, Neo?
- Neo: No
```

Parse output from pyyaml, ruamel.yaml and poyo:

```
>>> load(the_matrix) == [{"Morpheus": "Do you belive in fate, Neo?"}, {"Neo": False}]
```

It isn't just a problem in movie scripts:

```
python: 3.5.3
postgres: 9.3
```

```
>>> load(versions) == [{"python": "3.5.3", "postgres": 9.3}]    # oops those *both* should have been strings
```

It's also makes Christopher Null unhappy:

```
first name: Christopher
surname: Null
```

```
# Is it okay if we just call you Christopher None instead?
>>> load(name) == {"first name": "Christopher", "surname": None}
```

In the above cases, implicit typing represents a major violation of the principle of least astonishment.

What is wrong with explicit syntax typing in a readable configuration languages?

Explicit syntax typing is the process of using syntax to define types in markup. So, for instance in JSON, quotation marks are used to define name as a string and age as a number:

```
{"name": "Arthur Dent", "age": 42}
```

This helps distinguish the types for the parser, which is useful for JSON, but it also comes with two disadvantages:

- The distinction is subtle and not particularly clear to *non-programmers*, who will not necessarily understand that a directive needs to be given to the parser to avoid it being misinterpreted.

- It's not necessary if the type structure is maintained outside of the markup.
- Verbosity - two extra characters per string makes the markup longer and noisier.

In JSON when being used as a REST API, syntax typing is often an *advantage* - it is explicit to the machine reading the JSON that "string" and "age" is an integer and it can convert accordingly *in the absence of a schema*.

StrictYAML assumes all values are strings unless the schema explicitly indicates otherwise (e.g. Map(Int(), Int())).

StrictYAML does require quotation marks for strings that are implicitly converted to other types (e.g. yes or 1.5), but it does require quotation marks for strings that are syntactically confusing (e.g. "{ text in curly brackets }")

Regular YAML has explicit syntax typing to explicitly declare strings, although it's confusing as hell to know when it's required and when it is not. For example:

```
a: text               # not necessary
b: "yes"              # necessary
c: "0"                # necessary
d: "3.5"              # necessary
e: in                 # not necssary
f: out                # not necesary
g: shake it all about # not necessary
h: "on"               # necessary
```

Several other configuration language formats also have syntax typing in lieu of schemas. They are:

- TOML
- JSON5
- HJSON
- SDLang
- HOCON

INI does not.

What is wrong with binary data?

StrictYAML doesn't allow binary data to be parsed and will throw an exception if it sees it:

```
evildata: !!binary |
  R0lGODdhDQAIAIAAAAAAANn
  Z2SwAAAAADQAIAAACF4SDGQ
  ar3xxbJ9p0qa7R0YxwzaFME
  1IAADs=
```

This idiotic feature led to Ruby on Rails' spectacular security fail.

What is wrong with explicit tags?

Explicit tags are tags that have an explicit type attached that is used to determine what type to convert the data to when it is parsed.

For example, if it were to be applied to "fix" the Godfather movie script parsing issue described above, it would look like this:

```
- Don Corleone: Do you have faith in my judgment?
- Clemenza: !!str Yes
- Don Corleone: Do I have your loyalty?
```

Explicit typecasts in YAML markup are not only ugly, they confuse non-programmers. StrictYAML's philosophy is that type information should be kept strictly separated from data, so this 'feature' of YAML is switched off.

If data like this is seen in a YAML file it will raise a special exception.

What is wrong with node anchors and references?

An example of a snippet of YAML that uses node anchors and references is described on the wikipedia page:

```yaml
# sequencer protocols for Laser eye surgery
---
- step:  &id001                  # defines anchor label &id001
    instrument:      Lasik 2000
    pulseEnergy:     5.4
    pulseDuration:   12
    repetition:      1000
    spotSize:        1mm

- step: &id002
    instrument:      Lasik 2000
    pulseEnergy:     5.0
    pulseDuration:   10
    repetition:      500
    spotSize:        2mm
- step: *id001                   # refers to the first step (with anchor &id001)
- step: *id002                   # refers to the second step
- step:
    <<: *id001
    spotSize: 2mm                # redefines just this key, refers rest from &id001
- step: *id002
```

While the intent of the feature is obvious (it lets you deduplicate code), the effect is to make the markup more or less unreadable to non-programmers.

The example above could be refactored to be clearly as follows:

```yaml
# sequencer protocols for Laser eye surgery
---
- step:
    instrument:      Lasik 2000
    pulseEnergy:     5.4
    pulseDuration:   12
    repetition:      1000
    spotSize:        1mm
- step:
    instrument:      Lasik 2000
    pulseEnergy:     5.0
    pulseDuration:   10
    repetition:      500
    spotSize:        2mm
- step:
    instrument:      Lasik 2000
    pulseEnergy:     5.4
    pulseDuration:   12
    repetition:      1000
    spotSize:        1mm
- step:
    instrument:      Lasik 2000
    pulseEnergy:     5.0
    pulseDuration:   10
    repetition:      500
    spotSize:        2mm
- step:
    instrument:      Lasik 2000
    pulseEnergy:     5.4
    pulseDuration:   12
    repetition:      1000
    spotSize:        2mm
- step:
    instrument:      Lasik 2000
    pulseEnergy:     5.0
    pulseDuration:   10
    repetition:      500
    spotSize:        2mm
```

While much more repetitive, the intent of the above is *so much* clearer and easier for non-programmers to work with, that it more than compensates for the increased repetition.

While it makes little sense to refactor the above snippet to deduplicate repetititve data it may make sense to refactor the *structure* as it grows larger (and more repetitive). However, there are a number of ways this could be done without using YAML's nodes and anchors (e.g. splitting the file into two files - step definitions and step sequences), depending on the nature and quantity of the repetitiveness.

What is wrong with flow style?

Flow style is a feature of YAML that uses curly brackets - {, } and looks a little like embedded JSON.

```
a: 1
b: {c: 3, d: 4}
```

This use of JSONesque { and } is also ugly and hampers readability - *especially* when { and } are used for other purposes (e.g. templating) and the human reader/writer of YAML has to give themselves a headache figuring out what *kind* of curly bracket it is.

The *first* question in the FAQ of pyyaml actually subtly indicates that this feature wasn't a good idea - see "why does my YAML look wrong?".

To take a real life example, use of flow style in this saltstack YAML definition blurs the distinction between flow style and jinja2, confusing the reader.

What is wrong with duplicate keys?

Duplicate keys are allowed in regular YAML - as parsed by pyyaml, ruamel.yaml and poyo:

```
x: cow
y: dog
x: bull
```

Not only is it unclear whether x should be "cow" or "bull" (the parser will decide 'bull', but did you know that?), if there are 200 lines between x: cow and x: bull, a user might very likely change the *first* x and erroneously believe that the resulting value of x has been changed - when it hasn't.

In order to avoid all possible confusion, StrictYAML will simply refuse to parse this and will *only* accept associative arrays where all of the keys are unique. It will throw a DuplicateKeysDisallowed exception.

Why not use INI files for configuration or DSLs?

INI is a very old and quite readable configuration format. Unfortunately it suffers from two *major* problems:

- Different parsers will operate in subtly different ways that can lead to often obscure bugs regarding the way whitespace is used, case sensitivity, comments and escape characters.
- It doesn't let you represent hierarchical data.

Why shouldn't I just use python code for configuration?

This isn't uncommon and can often seem like a nice, simple solution although using a turing complete language for configuration will often have nasty side effects.

Why does using YAML (or indeed, any configuration language) avoid this? Because they are *less powerful* languages than python.

While this may not intrinsically seem like a good thing (more power seems better at first glance), it isn't:

> - We need less powerful languages.
> - Rule of least power (wikipedia).
> - Principle of least power by Tim Berners Lee.
> - Principle of least power by Jeff Atwood (coding horror blogger / stack overflow founder).

A good way of refactoring, in fact, is to take a large chunk of turing complete python code that *can* be transformed directly into YAML with no loss in expressiveness and and to transform it - for example, a list of translation strings, countries or other configuration information.

This has a number of advantages.

Less powerful languages are easier to maintain and can be given to non-programmers to maintain.

For example, a YAML translations configuration file like this could easily be edited by a non programmer:

```yaml
Hello:
  French: Bonjour
  German: Guten tag
Goodbye:
  French: Au revoir
  German: Auf wiedersehen
```

Whereas this python is more likely to cause problems, especially in a large file:

```python
TRANS = {
    "Hello": {
        "French": "Bonjour",
        "German": "Guten tag",
    },
    "Hello": {
        "French": "Bonjour",
        "German": "Auf wiedersehen",
    },
}
```

It also makes it easier to have the markup generated by another program or a templating language. While you technicall *can* do this with turing complete code, it will often lead to a debugging nightmare - just ask C++ programmers!

Why not use XML for configuration or DSLs?

XML suffers from overcomplication much like vanilla YAML does - although to an ever greater degree, thanks to the committee driven design. Doctypes and namespaces are horrendous additions to the language, for instance. XML is not only not really human readable (beyond a very basic subset of the language), it's often barely *programmer* readable despite being less expressive than most turing complete languages. It's a flagrant violation of the rule of least power.

The language was, in fact, *so* overcomplicated that it ended up increasing the attack surface of the parser itself to the point that it led to parsers with security vulnerabilities.

Unlike JSON and YAML, XML's structure also does not map well on to the default data types used by most languages, often requiring a *third* language to act as a go between - e.g. either XQuery or XPath.

XML's decline in favor of JSON as a default API format is largely due to these complications and the lack of any real benefit drawn from them. The associated technologies (e.g. XSLT) also suffered from design by committee.

Using it as a configuration language will all but ensure that you need to write extra boilerplate code to manage its quirks.

Why not use JSON for configuration or simple DSLs?

JSON is an *ideal* format for REST APIs and other APIs that send data over a wire and it probably always will be because:

- It's a simple spec.
- It has all the basic types which map on to all programming languages - number, string, list, mapping, boolean *and no more*.
- Its syntax contains a built in level of error detection - cut a JSON request in half and it is no longer still valid, eliminating an entire class of obscure and problematic bugs.
- If pretty-printed correctly, it's more or less readable - for the purposes of debugging, anyway.

However, while it is emintently suitable for REST APIs it is less suitable for configuration since:

- The same syntax which gives it decent error detection (commas, curly brackets) makes it tricky for humans to edit.
- It's not especially readable.
- It doesn't allow comments.

Why not use TOML?

TOML is a redesigned configuration language that's essentially an extended version of INI which allows the expression of both hierarchical and typed data.

TOML's main criticism of YAML is spot on:

```
TOML aims for simplicity, a goal which is not apparent in the YAML specification.
```

StrictYAML's cut down version of the YAML specification however - with implicit typing, node anchors/references and flow style cut out, ends up being simpler than TOML.

The main complication in TOML is its inconsistency in how it handles tables and arrays. For example:

```
# not clear that this is an array
[[tables]]
foo = "foo"
```

Similarly, all arrays have the type array. So even though arrays are homogenous in TOML, you can oddly do:

```
array = [["foo"], [1]]

# but not
array = ["foo", 1]
```

TOML's use of special characters for delimiters instead of whitespace like YAML makes the resulting output noiser and harder for humans to parse. Here's an example from the TOML site:

```
[[fruit]]
name = "apple"

[fruit.physical]
color = "red"
shape = "round"
```

Equivalent YAML:

```
fruit:
  name: apple
  physical:
    color: red
    shape: round
```

It also embeds type information used by the parser into the syntax:

```
flt2 = 3.1415
string = "hello"
```

Whereas strictyaml:

```
flt2: 3.1415
string: hello
```

Will yield this:

```
load(yaml) == {"flt2": "3.1415", "string": "hello"}
```

Or this:

```
load(yaml, Map({"flt2": Float(), "string": Str()})) == {"flt": 3.1415, "string": "hello"}
```

Which not only eliminates the need for syntax typing, is more type safe.

Why not HOCON?

HOCON is another "redesigned" JSON, ironically enough, taking JSON and making it even more complicated.

Along with JSON's syntax typing - a downside of most non-YAML alternatives, HOCON makes the following mistakes in its design:

- It does not fail loudly on duplicate keys.
- It has a confusing rules for deciding on concatenations and substitutions.
- It has a mechanism for substitutions similar to YAML's node/anchor feature - which, unless used extremely sparingly, can create confusing markup that, ironically, is *not* human optimized.

In addition, its attempt at using "less pedantic" syntax creates a system of rules which makes the behavior of the parser much less obvious and edge cases more frequent.

Why not use HJSON?

HJSON is an attempt at fixing the aforementioned lack of readability of JSON.

It has the following criticisms of YAML:

- JSON is easier to explain (compare the JSON and YAML specs).
- JSON is not bloated (it does not have anchors, substitutions or concatenation).

As with TOML's criticism, these are spot on. However, strictyaml fixes this by *cutting out those parts of the spec*, leaving something that is actually simpler than HJSON.

It has another criticism:

- JSON does not suffer from significant whitespace.

This is not a valid criticism.

Whitespace and indentation is meaningful to people parsing any kind of code and markup (why else would code which *doesn't* have meaningful whitespace use indentation as well?) so it *should* be meaningful to computers parsing.

There is an initial 'usability hump' for first time users of languages which have significant whitespace *that were previously not used to significant whitespace* but this isn't especially hard to overcome - especially if you have a propery configured decent editor which is explicit about the use of whitespace.

Python users often report this being a problem, but after using the language for a while usually come to prefer it since it keeps the code shorter and makes its intent clearer.

Why not use JSON5?

JSON5 is also a proposed extension to JSON to make it more readable.

Its main criticism of YAML is:

```
There are other formats that are human-friendlier, like YAML, but changing from JSON to a completely different format
```

This is, I belive, mistaken. It is better if a language is not subtly different if you are going to use it as such. Subtle differences invite mistakes brought on by confusion.

JSON5 looks like a hybrid of YAML and JSON:

```
{
    foo: 'bar',
    while: true,
}
```

It has weaknesses similar to TOML:

- The noisiness of the delimiters that supplant significant whitespace make it less readable and editable.
- The use of syntax typing is neither necessary, nor an aid to stricter typing if you have a schema.

Why not use SDLang?

SDLang or "simple declarative language" is a proposed configuration language with an XML-like structure inspired by C.

Example:

```
// This is a node with a single string value
title "Hello, World"

// Multiple values are supported, too
bookmarks 12 15 188 1234

// Nodes can have attributes
author "Peter Parker" email="peter@example.org" active=true

// Nodes can be arbitrarily nested
contents {
    section "First section" {
        paragraph "This is the first paragraph"
        paragraph "This is the second paragraph"
    }
}

// Anonymous nodes are supported
"This text is the value of an anonymous node!"

// This makes things like matrix definitions very convenient
matrix {
    1 0 0
    0 1 0
    0 0 1
}
```

Advantages:

- Relatively more straightforward than other serialization languages.

Disadvantages:

- Syntax typing - leading to noisy syntax and a
- The distinction between properties and values is not entirely clear.
- Instead of having one obvious way to describe property:value mappings
- Niche

Should I use kwalify?

Kwalify is a schema validation language that is written *in* YAML.

It is a descriptive schema language suitable for validating simple YAML.

Kwalify compiles to the strictyaml equivalent but is able to do less. You cannot, for example:

- Plug generated lists that come from outside of the spec (e.g. a list of country code from pycountry).
- Validate parts of the schema which can be either one thing *or* another - e.g. a list *or* a single string.
- Plug sub-validators of a document into larger validators.

If your schema is very simple and small, there is no point to using kwalify.

If your schema needs to be shared with a 3rd party - especially a third party using another language, it may be helpful to use it.

If your schema validation requirements are more complicated - e.g. like what is described above - it's best *not* to use it.

Why not use pykwalify to validate YAML instead?

See the question above for the correct times to use kwalify to validate your code and crucially, when not to.

Why is StrictYAML built upon ruamel.yaml?

ruamel.yaml is probably the best spec-adhering YAML parser for python.

Unlike pyyaml it does not require the C yaml library to be installed, and it is capable of loading, editing and saving YAML while preserving comments, which pyyaml does not.

What if I still disagree with everything you wrote here?

If I haven't covered all aspects of this tedious, ongoing debate about what is the best configuration language, I consider that a bug.

If your favorite configuration language / tool isn't mentioned and critiqued and it isn't obviously worse then also please raise a ticket to ask me to compare it.

Please feel free to ensure all tickets come accompanied with a creative insult. I wouldn't want to spoil the long tradition of flame wars about configuration languages.