Introduction to YAML

YAML is a very simple, text/human-readable annotation format that can be used to store data introduced by Clark Evans in 2001. It is commonly used for configuration files, but could be used in many applications where data is being stored (like XML).

It supports JSON, another minimalist data serialization format where braces and brackets are used instead of indentation.

Why ?

- This is Human readable
- Really permissive
- Syntax highlighting is often available by default in major editors
- Implementations is widespread
- Allows for more sophisticated structures than a typical ini file
- Most support sections/options already. But many coders require/prefer one that allow for as many settings on a per-domain base, so they need another level of structure.

Basic principles

- Always save using the UTF-8 encoding to minimize the possibility of errors.
- Never, EVER use the TAB character anywhere in it. This doesn't work.
- Editing with a WYSIWYG text processor, such as Microsoft Word, is not recommended.
- Use a monowidth/monospaced font to view the contents. Examples you may have in your computer: Fixedsys, Lucida Console, Consolas, Monaco, DejaVu Sans Mono, Courier New.

Whenever you see the **#** character anywhere in the file that isn't first enclosed in '' (quotes) or "" (double quotes), it marks the beginning of a **comment**. This means all the text after it, up until the end of the line, is *completely ignored* and has no effect. You can use this to write notes on the file or temporarily disable one or more options.

These are all comments:

```
#debug: false
#Hello there, how are you?
debug: false # <- The line is read up to this character!
```

This is not commented:

```
name: '#changeme'
```

But this is:

```
#name: '#changeme'
```

Key-value pairs

YAML keeps data stored as a map containing keys and values associated with those keys. This map is in no particular order, so you can reorder it at will. Each pair is in the format KEY: VALUE. For example:

```
min-tag: 'minecraft'
can-tag: 'cancelled'
```

Note the 'quotes' around the value. When the value is a text string, we use the quotes to make sure any special characters aren't given special meaning, and instead are all kept as part of the value. So even though they are optional, using them is highly recommended.

Other than text, numbers and true/false, the value associated to a key can also be another map of key-value pairs. To achieve this, omit the value and instead write the key-value pairs in the following lines. However, you must prefix them with at least one more SPACE character than the key.

YAML will consider that lines prefixed with more spaces than the parent key are contained inside it; Moreover, all lines must be prefixed with the same number of spaces to belong to the same map. So this works:

```
formatting:
  fromgame:
    chat: '(%sender%) %message%'
    action: '* %sender% %message%'
```

This works too:

```
formatting:
        from-game:
                chat: '(%sender%) %message%'
```

```
                action: '* %sender% %message%'
```

But this doesn't work:

```
formatting:
from-game:
chat: '(%sender%) %message%'
action: '* %sender% %message%'
```

Alternative format

YAML supports JSON syntax to store key-value maps, useful for compressing small maps into a single line. Syntax like : {KEY: VALUE, KEY: VALUE, ...} works. The above example would become:

```
formatting: {from-game: {chat: '(%sender%) %message%', action: '* %
```

Lists

There is one other data structure from YAML we need to use - The list. Lists are used to store a collection of ordered values. The values are not associated with a key, only with a positional index obtained from the order in which they are specified (item 1, item 2, etc.).

```
mylist:
- 'item 1'
- 'item 2'
```

Like key-value pairs, list items are defined in the lines below the list key, all with the same number of spaces prefixing them (at least as many spaces as the parent key). The difference is that they begin with a dash (**-**). Here's another valid list:

```
mylist:
- 100
- 200
```

Now, remember how you could put key-value maps inside key-value maps? You can do the same with lists. You can have: * Map inside Map (as seen in the previous section) * Lists inside maps (as seen immediately above - lists are, by default, inside maps, since they need a key) * Maps inside lists

```
channels:
  - name: '#mychannel'
    password: ''
  - name: '#myprivatechannel'
    password: 'mypassword'
```

- List inside List

```
twobytwotable:
  - - 'a1'
    - 'a2'
  - - 'b1'
    - 'b2'
```

Alternative format

Guess what, there's also a one-line format for lists! It is [ITEM1, ITEM2, ITEM3, ...]. Here's the above example:

```
twobytwotable: [ ['a1', 'a2'], ['b1', 'b2'] ]
```

Multiple lines

Values can span multiple lines using `|` or `>`. Spanning multiple lines using a `|` will include the newlines. Using a `>` will ignore newlines; it's used to make what would otherwise be a very long line easier to read and edit. In either case the indentation will be ignored. Examples are::

```
include_newlines: |
  exactly as you see
  will appear these three
  lines of poetry

ignore_newlines: >
  this is really a
  single line of text
  despite appearances
```

Let's combine what we learned so far in an arbitrary YAML example. This really has nothing to do with Ansible, but will give you a feel for the format::

```
# An employee record
name: Martin D'vloper
job: Developer
skill: Elite
employed: True
foods:
  - Apple
  - Orange
  - Strawberry
  - Mango
languages:
  perl: Elite
  python: Elite
  pascal: Lame
education: |
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
```

That's all you really need to know about YAML to start writing YAML.

Node anchors

Two features that distinguish YAML from the capabilities of other data serialization languages are structures and data typing.

YAML structures enable storage of multiple documents within single file, usage of references for repeated nodes, and usage of arbitrary nodes as keys.

For clarity, compactness, and avoiding data entry errors, YAML provides node anchors (using `&`) and references (using `*`). References to the anchor work for all data types (see the ship-to reference in the example below).

Below is an example of a queue in an instrument sequencer in which two steps are reused repeatedly without being fully described each time.

```
# sequencer protocols for Laser eye surgery
---
- step:  &id001  # defines anchor label &id001
  instrument: Lasik 2000
  energy: 5.4
  duration: 12
  repetition: 1000
  size: 1mm

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

Types

Explicit data typing is seldom seen in the majority of YAML documents since YAML autodetects simple types. Data types can be divided into three categories: core, defined, and user-defined. Core are ones expected to exist in any parser (e.g. floats, ints, strings, lists, maps, ...). Many more advanced data types, such as binary data, are defined in the YAML specification but not supported in all implementations. Finally YAML defines a way to extend the data type definitions locally to accommodate user-defined classes, structures or primitives (e.g. quad-precision floats).

YAML autodetects the datatype of the entity. Sometimes one wants to cast the datatype explicitly. The most common situation is where a single-word string that looks like a number, boolean or tag requires disambiguation by surrounding it with quotes or using an explicit datatype tag.

```
a: 123                    # an integer
b: "123"                  # a string, disambiguated by quotes
c: 123.0                  # a float
d: !!float 123            # also a float via explicit data type pr
e: !!str 123              # a string, disambiguated by explicit ty
f: !!str Yes              # a string via explicit type
g: Yes                    # a boolean True (yaml1.1), string "Yes'
h: Yes we have No bananas  # a string, "Yes" and "No" disambiguated
```

Not every implementation of YAML has every specification-defined data type. These built-in types use a double exclamation sigil prefix (`!!`).

Particularly interesting ones not shown here are sets, ordered maps, timestamps, and hexadecimal. Here's an example of base64 encoded

binary data.

```
gif_file: !!binary |
  R0lGODlhDAAMAIQAAP//9/X17unp5WZmZgAAAOfn515eXvPz7Y6OjuDg4J+fn5
  OTk6enp56enmlpaWNjY6Ojo4SEhP/++f/++f/++f/++f/++f/++f/++f/+
  +f/++f/++f/++f/++f/++SH+Dk1hZGUgd2l0aCBHSU1QACwAAAAADAAMAAAFLC
  AgjoEwnuNAFOhpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcgggoBADs=
```

Date can be handle too:

```
datetime: 2001-12-15T02:59:43.1Z
datetime_with_spaces: 2001-12-14 21:59:43.10 -5
date: 2002-12-14
```

Multiples documents

Core to YAML is the concept of documents. A document is not just a separate file in this case. Instead, think of a document as just a chunk of YAML. You can have multiple documents in a single stream of YAML, if each one is separated by "---", like:

```
---
document: this is doc 1
---
document: this is doc 2
...
```

Using an ellipsis explicitly ends a document. The nice thing about documents is you can treat them as different entities. Let's say, "people" and "cars" are in the same file. You can use them for a bunch of entities that look alike, e.g.:

```
---
employed: true
family: [wife, toddler]
hobby: python
limbs: {arms: 2, legs: 2}
person: jesse
---
employed: true
family: [wife, toddler]
hobby: python
limbs: {arms: 2, legs: 2}
person: jesse
```

Variables

There is actually no support of variables into the last specifications of YAML. But there is some implementations to use a system handling variables. Extensions library or template system like Jinja2 or Twig can be a way to add this feature to your YAML files.

Conclusion

Now that you're a more knowledgeable person, I hope you can add new items to a list or keep your key-value pairs aligned without making a scene. You can freely create or adjust your YAML files to your taste by swapping to/from the alternative formats, changing the order of the keys in a map and adding/removing comments.

References

- https://github.com/Animosity/CraftIRC/wiki/Complete-idiot's-introduction-to-yaml
- http://jessenoller.com/blog/2009/04/13/yaml-aint-markup-language-completely-different
- http://docs.ansible.com/ansible/latest/YAMLSyntax.html
- https://en.m.wikipedia.org/wiki/YAML
- https://github.com/yaml/YAML2/wiki/Who-uses-it-and-why%3F

External links

- http://yaml.org/
- Online YAML > JSON parser
- Online JSON > YAML parser
- YAML validator