
yaml Documentation

Release dev

Anthon van der Neut

Jun 25, 2017

Contents

1	Overview	3
2	Installing	5
2.1	Optional requirements	5
3	Basic Usage	7
3.1	More examples	8
4	Details	9
4.1	Indentation of block sequences	9
4.2	Positioning ':' in top level mappings, prefixing ':'	10
4.3	Document version support.	10
4.4	Round trip including comments	11
4.5	Config file formats	12
4.6	Extending	12
4.7	Smartening	12
5	Examples	15
6	Departure from previous API	19
7	Loading	21
7.1	Duplicate keys	21
8	Dumping	23
9	Transparent usage of new and old API	25
10	Reason for API change	27
11	Differences with PyYAML	29
11.1	Defaulting to YAML 1.2 support	29
11.2	PY2/PY3 reintegration	29
11.3	Fixes	30
11.4	Testing	30
11.5	API	30

BitBucket | PyPI

Contents:

`ruamel.yaml` is a YAML 1.2 loader/dumper package for Python. It is a derivative of Kirill Simonov's [PyYAML 3.11](#).

`ruamel.yaml` supports [YAML 1.2](#) and has round-trip loaders and dumpers that preserves, among others:

- comments
- block style and key ordering are kept, so you can diff the round-tripped source
- flow style sequences ('a: b, c, d') (based on request and test by Anthony Sottile)
- anchor names that are hand-crafted (i.e. not of the form "idNNN")
- merges in dictionaries are preserved

This preservation is normally not broken unless you severely alter the structure of a component (delete a key in a dict, remove list entries). Reassigning values or replacing list items, etc., is fine.

For the specific 1.2 differences see [Defaulting to YAML 1.2 support](#)

Although individual indentation of lines is not preserved, you can specify both indentation (counting for this does **not** include the dash for a sequence element) and specific offset of block sequence dashes within that indentation. There is a utility function that tries to determine the correct values for `indent` and `block_seq_indent` that can then be passed to the dumper.

Although *ruamel.yaml* still allows most of the PyYAML way of doing things, adding features required a different API than the transient nature of PyYAML's `Loader` and `Dumper`. Starting with `ruamel.yaml` version 0.15.0 this new API gets introduced. Old ways that get in the way will be removed, after first generating warnings on use, then generating an error. In general a warning in version 0.N.x will become an error in 0.N+1.0

Many of the bugs filed against PyYAML, but that were never acted upon, have been fixed in `ruamel.yaml`

Make sure you have a recent version of `pip` and `setuptools` installed. The later needs environment marker support (`setuptools>=20.6.8`) and that is e.g. bundled with Python 3.4.6 but not with 3.4.4. It is probably best to do:

```
pip install -U pip setuptools wheel
```

in your environment (*virtualenv*, (Docker) container, etc) before installing `ruamel.yaml`.

`ruamel.yaml` itself should be installed from **PyPI** using:

```
pip install ruamel.yaml
```

If you want to process `jinj2/YAML` templates (which are not valid `YAML` with the default `jinj2` markers), do `pip install ruamel.yaml[jinja2]` (you might need to quote the last argument because of the `[]`)

There also is a commandline utility `yaml` available after installing:

```
pip install ruamel.yaml.cmd
```

that allows for round-trip testing/re-indenting and conversion of `YAML` files (`JSON,INI,HTML` tables)

Optional requirements

If you have the the header files for your Python executables installed then you can use the (non-roundtrip), but faster, C loader and emitter.

On Debian systems you should use:

```
sudo apt-get install python3-dev
```

you can leave out `python3-dev` if you don't use `python3`

For CentOS (7) based systems you should do:

```
sudo yum install python-devel
```


CHAPTER 3

Basic Usage

This is the new (0.15+) interface for “ruamel.yaml“, it is still in the process of being fleshed out. Please pin your dependency to ruamel.yaml<0.15 for production software.

You load a YAML document using:

```
from ruamel.yaml import YAML

yaml=YAML(typ='safe')    # default if not specified is round-trip

yaml.load(doc)
```

in this `doc` can be a file pointer (i.e. an object that has the `.read()` method, a string or a `pathlib.Path()`). `typ='safe'` accomplishes the same as what `safe_load()` did before: loading of a document without resolving unknown tags. Provide `pure=True` to enforce using the pure Python implementation (faster C libraries will be used when possible/available)

Dumping works in the same way:

```
from ruamel.yaml import YAML

yaml=YAML()
yaml.default_flow_style = False
yaml.dump({'a': [1, 2]}, s)
```

in this `s` can be a file pointer (i.e. an object that has the `.write()` method, or a `pathlib.Path()`). If you want to display your output, just stream to `sys.stdout`.

If you need to transform a string representation of the output provide a function that takes a string as input and returns one:

def tr(s): return `s.replace('n', '<n')` # such output is not valid YAML!

```
yaml.dump(data, sys.stdout, transform=tr)
```

More examples

Using the C based SafeLoader (at this time is inherited from libyaml/PyYAML and e.g. loads 0o52 as well as 052 load as integer 42):

```
from ruamel.yaml import YAML

yaml=YAML(typ="safe")
yaml.load("""a:\n b: 2\n c: 3\n""")
```

Using the Python based SafeLoader (YAML 1.2 support, 052 loads as 52):

```
from ruamel.yaml import YAML

yaml=YAML(typ="safe", pure=True)
yaml.load("""a:\n b: 2\n c: 3\n""")
```

- support for simple lists as mapping keys by transforming these to tuples
- `!!omap` generates `ordereddict` (C) on Python 2, `collections.OrderedDict` on Python 3, and `!!omap` is generated for these types.
- Tests whether the C yml library is installed as well as the header files. That library doesn't generate `CommentTokens`, so it cannot be used to do round trip editing on comments. It can be used to speed up normal processing (so you don't need to install `ruamel.yaml` and `PyYaml`). See the section *Optional requirements*.
- Basic support for multiline strings with preserved newlines and chomping (`'|'`, `'|+'`, `'|-'`). As this subclasses the string type the information is lost on reassignment. (This might be changed in the future so that the preservation/folding/chomping is part of the parent container, like comments).
- anchors names that are hand-crafted (not of the form `'idNNN'`) are preserved
- `merges` in dictionaries are preserved
- adding/replacing comments on block-style sequences and mappings with smart column positioning
- collection objects (when read in via `RoundTripParser`) have an `lc` property that contains line and column info `lc.line` and `lc.col`. Individual positions for mappings and sequences can also be retrieved (`lc.key('a')`, `lc.value('a')` resp. `lc.item(3)`)
- preservation of whitelines after block scalars. Contributed by Sam Thursfield.

In the following examples it is assumed you have done something like::

```
from ruamel.yaml import YAML
yaml = YAML()
```

if not explicitly specified.

Indentation of block sequences

Although `ruamel.yaml` doesn't preserve individual indentations of block sequence items, it does properly dump:

```
x:
- b: 1
- 2
```

back to:

```
x:
- b: 1
- 2
```

if you specify `yaml.indent = 4`.

PyYAML (and older versions of ruamel.yaml) gives you non-indented scalars (when specifying `default_flow_style=False`):

```
x:
- b: 1
- 2
```

The dump also observes an additional `block_seq_indent` setting that can be used to push the dash inwards, *within the space defined by indent*.

The above example with the often seen `yaml.indent = 4; yaml.block_seq_indent = 2` indentation:

```
x:
  - b: 1
  - 2
```

If the `block_seq_indent` equals `indent`, there is not enough room for the dash and the space that has to follow. In that case the element itself would normally be pushed to the next line (and older versions of ruamel.yaml did so). But this is prevented from happening. However the `indent` level is what is used for calculating the cumulative indent for deeper levels and specifying `yaml.indent = 3` resp. `yaml.block_seq_indent = 2`, might give correct, but counter intuitive results.

It is best to always have `indent >= block_seq_indent + 2` but this is not enforced. Depending on your structure, not following this advice **might lead to invalid output**.

Positioning ‘:’ in top level mappings, prefixing ‘:’

If you want your toplevel mappings to look like:

```
library version: 1
comment       : |
              this is just a first try
```

then set `yaml.top_level_colon_align = True` (and `yaml.indent = 4`). `True` causes calculation based on the longest key, but you can also explicitly set a number.

If you want an extra space between a mapping key and the colon specify `yaml.prefix_colon = ' ':`

```
- https://myurl/abc.tar.xz : 23445
#                          ^ extra space here
- https://myurl/def.tar.xz : 944
```

If you combine `prefix_colon` with `top_level_colon_align`, the top level mapping doesn’t get the extra prefix. If you want that anyway, specify `yaml.top_level_colon_align = 12` where 12 has to be an integer that is one more than length of the widest key.

Document version support.

In YAML a document version can be explicitly set by using:

```
%YAML 1.x
```

before the document start (at the top or before a ---). For `ruamel.yaml` `x` has to be 1 or 2. If no explicit version is set `version 1.2` is assumed (which has been released in 2009).

The 1.2 version does **not** support:

- sexagesimals like `12:34:56`
- octals that start with 0 only: like `012` for number 10 (`0o12` is supported by YAML 1.2)
- Unquoted Yes and On as alternatives for True and No and Off for False.

If you cannot change your YAML files and you need them to load as 1.1 you can load with `yaml.version = (1, 1)`, or the equivalent (version can be a tuple, list or string) `yaml.version = "1.1"`

If you cannot change your code, stick with `ruamel.yaml==0.10.23` and let me know if it would help to be able to set an environment variable.

This does not affect dump as `ruamel.yaml` never emitted sexagesimals, nor octal numbers, and emitted booleans always as `true` resp. `false`

Round trip including comments

The major motivation for this fork is the round-trip capability for comments. The integration of the sources was just an initial step to make this easier.

adding/replacing comments

Starting with version 0.8, you can add/replace comments on block style collections (mappings/sequences resulting in Python dict/list). The basic for for this is:

```
from __future__ import print_function

import sys
import ruamel.yaml

yaml = ruamel.yaml.YAML() # defaults to round-trip

inp = """\
abc:
  - a      # comment 1
xyz:
  a: 1     # comment 2
  b: 2
  c: 3
  d: 4
  e: 5
  f: 6 # comment 3
"""

data = yaml.load(inp)
data['abc'].append('b')
data['abc'].yaml_add_eol_comment('comment 4', 1) # takes column of comment 1
data['xyz'].yaml_add_eol_comment('comment 5', 'c') # takes column of comment 2
data['xyz'].yaml_add_eol_comment('comment 6', 'e') # takes column of comment 3
data['xyz'].yaml_add_eol_comment('comment 7', 'd', column=20)

yaml.dump(data, sys.stdout)
```

Resulting in:

```
abc:
- a      # comment 1
- b      # comment 4
xyz:
  a: 1    # comment 2
  b: 2
  c: 3    # comment 5
  d: 4    # comment 7
  e: 5    # comment 6
  f: 6    # comment 3
```

If the comment doesn't start with '#', this will be added. The key is the element index for list, the actual key for dictionaries. As can be seen from the example, the column to choose for a comment is derived from the previous, next or preceding comment column (picking the first one found).

Config file formats

There are only a few configuration file formats that are easily readable and editable: JSON, INI/ConfigParser, YAML (XML is too cluttered to be called easily readable).

Unfortunately JSON doesn't support comments, and although there are some solutions with pre-processed filtering of comments, there are no libraries that support round trip updating of such commented files.

INI files support comments, and the excellent `ConfigObj` library by Foord and Larosa even supports round trip editing with comment preservation, nesting of sections and limited lists (within a value). Retrieval of particular value format is explicit (and extensible).

YAML has basic mapping and sequence structures as well as support for ordered mappings and sets. It supports scalars various types including dates and datetimes (missing in JSON). YAML has comments, but these are normally thrown away.

Block structured YAML is a clean and very human readable format. By extending the Python YAML parser to support round trip preservation of comments, it makes YAML a very good choice for configuration files that are human readable and editable while at the same time interpretable and modifiable by a program.

Extending

There are normally six files involved when extending the roundtrip capabilities: the reader, parser, composer and constructor to go from YAML to Python and the resolver, representer, serializer and emitter to go the other way.

Extending involves keeping extra data around for the next process step, eventually resulting in a different Python object (subclass or alternative), that should behave like the original, but on the way from Python to YAML generates the original (or at least something much closer).

Smartening

When you use round-tripping, then the complex data you get are already subclasses of the built-in types. So you can patch in extra methods or override existing ones. Some methods are already included and you can do:

```
yaml_str = """\
a:
- b:
  c: 42
- d:
  f: 196
e:
```



```
g: 3.14
"""

data = yaml.load(yaml_str)

assert data.mlget(['a', 1, 'd', 'f'], list_ok=True) == 196
```


Basic round trip of parsing YAML to Python objects, modifying and generating YAML:

```
import sys
from ruamel.yaml import YAML

inp = """\
# example
name:
  # details
  family: Smith    # very common
  given: Alice    # one of the siblings
"""

yaml = YAML()
code = yaml.load(inp)
code['name']['given'] = 'Bob'

yaml.dump(code, sys.stdout)
```

Resulting in

```
# example
name:
  # details
  family: Smith    # very common
  given: Bob      # one of the siblings
```

with the old API:

```
from __future__ import print_function

import sys
import ruamel.yaml

inp = """\
# example
name:
  # details
  family: Smith    # very common
```

```

    given: Alice    # one of the siblings
"""

code = ruamel.yaml.load(inp, ruamel.yaml.RoundTripLoader)
code['name']['given'] = 'Bob'

ruamel.yaml.dump(code, sys.stdout, Dumper=ruamel.yaml.RoundTripDumper)

# the last statement can be done less efficient in time and memory with
# leaving out the end='' would cause a double newline at the end
# print(ruamel.yaml.dump(code, Dumper=ruamel.yaml.RoundTripDumper), end='')

```

Resulting in

```

# example
name:
  # details
  family: Smith    # very common
  given: Bob       # one of the siblings

```

YAML handcrafted anchors and references as well as key merging are preserved. The merged keys can transparently be accessed using [] and .get():

```

from ruamel.yaml import YAML

inp = """\
- &CENTER {x: 1, y: 2}
- &LEFT {x: 0, y: 2}
- &BIG {r: 10}
- &SMALL {r: 1}
# All the following maps are equal:
# Explicit keys
- x: 1
  y: 2
  r: 10
  label: center/big
# Merge one map
- <<: *CENTER
  r: 10
  label: center/big
# Merge multiple maps
- <<: [*CENTER, *BIG]
  label: center/big
# Override
- <<: [*BIG, *LEFT, *SMALL]
  x: 1
  label: center/big
"""

yaml = YAML()
data = yaml.load(inp)
assert data[7]['y'] == 2

```

The CommentedMap, which is the dict like construct one gets when round-trip loading, supports insertion of a key into a particular position, while optionally adding a comment:

```

import sys
from ruamel.yaml import YAML

```

```

yaml_str = """\
first_name: Art
occupation: Architect # This is an occupation comment
about: Art Vandelay is a fictional character that George invents...
"""

yaml = YAML()
data = yaml.load(yaml_str)
data.insert(1, 'last name', 'Vandelay', comment="new key")
yaml.dump(data, sys.stdout)

```

gives:

```

first_name: Art
last name: Vandelay # new key
occupation: Architect # This is an occupation comment
about: Art Vandelay is a fictional character that George invents...

```

Please note that the comment is aligned with that of its neighbour (if available).

The above was inspired by a [question](#) posted by *demux* on StackOverflow.

By default *ruamel.yaml* indents with two positions in block style, for both mappings and sequences. For sequences the indent is counted to the beginning of the scalar, with the dash taking the first position of the indented “space”.

The following program with three dumps:

```

import sys
from ruamel.yaml import YAML

data = {1: {1: [{1: 1, 2: 2}, {1: 1, 2: 2}], 2: 2}, 2: 42}

yaml = YAML()
yaml.explicit_start = True
yaml.dump(data, sys.stdout)
yaml.indent = 4
yaml.block_seq_indent = 2
yaml.dump(data, sys.stdout)

def sequence_indent_four(s):
    # this will fail on directly nested lists: {1; [[2, 3], 4]}
    levels = []
    ret_val = ''
    for line in s.splitlines(True):
        ls = line.lstrip()
        indent = len(line) - len(ls)
        if ls.startswith('- '):
            if not levels or indent > levels[-1]:
                levels.append(indent)
            elif levels:
                if indent < levels[-1]:
                    levels = levels[:-1]
            # same -> do nothing
        else:
            if levels:
                if indent <= levels[-1]:
                    while levels and indent <= levels[-1]:
                        levels = levels[:-1]
        ret_val += ' ' * len(levels) + line
    return ret_val

```

```
yaml = YAML()
yaml.explicit_start = True
yaml.dump(data, sys.stdout, transform=sequence_indent_four)
```

gives as output:

```
---
1:
  1:
    - 1: 1
      2: 2
    - 1: 1
      2: 2
  2: 2
2: 42
---
1:
  1:
    - 1: 1
      2: 2
    - 1: 1
      2: 2
  2: 2
2: 42
---
1:
  1:
    - 1: 1
      2: 2
    - 1: 1
      2: 2
  2: 2
2: 42
```

The transform example was inspired by a [question](#) posted by *nowox* on StackOverflow.

Departure from previous API

With version 0.15.0 *ruamel.yaml* starts to depart from the previous (PyYAML) way of loading and dumping. During a transition period the original *load()* and *dump()* in its various formats will still be supported, but this is not guaranteed to be so with the transition to 1.0.

At the latest with 1.0, but possible earlier transition error and warning messages will be issued, so any packages depending on *ruamel.yaml* should pin the version with which they are testing.

Up to 0.15.0, the loaders (*load()*, *safe_load()*, *round_trip_load()*, *load_all()*, etc.) took, apart from the input stream, a *version* argument to allow downgrading to YAML 1.1, sometimes needed for documents without directive. When round-tripping, there was an option to preserve quotes.

Up to 0.15.0, the dumpers (*dump()*, *safe_dump*, *round_trip_dump()*, *dump_all()*, etc.) had a plethora of arguments, some inherited from PyYAML, some added in *ruamel.yaml*. The only required argument is the *data* to be dumped. If the stream argument is not provided to the dumper, then a string representation is build up in memory and returned to the caller.

Starting with 0.15.0 *load()* and *dump()* are methods on a *YAML* instance and only take the stream, resp. the data and stream argument. All other parameters are set on the instance of *YAML* before calling *load()* or *dump()*

Before 0.15.0:

```
from pathlib import Path
from ruamel import yaml

data = yaml.safe_load("abc: 1")
out = path('/tmp/out.yaml')
with out.open('w') as fp:
    yaml.safe_dump(data, fp, default_flow_style=False)
```

after:

```
from ruamel.yaml import YAML

yaml = YAML(typ='safe')
yaml.default_flow_style = False
data = yaml.load("abc: 1")
out = path('/tmp/out.yaml')
yaml.dump(data, out)
```

If you previously used an keyword argument *explicit_start=True* you now do `yaml.explicit_start = True` before calling `dump()`. The `Loader` and `Dumper` keyword arguments are not supported that way. You can provide the *typ* keyword to *rt* (default), *safe*, *unsafe* or *base* (for round-trip load/dump, *safe_load/dump*, *load/dump* resp. using the `BaseLoader` / `BaseDumper`. More fine-control is possible by setting the attributes *.Parser*, *.Constructor*, *.Emitter*, etc., to the class of the type to create for that stage (typically a subclass of an existing class implementing that).

All data is dumped (not just for round-trip-mode) with *.allow_unicode = True*

You can of course have multiple YAML instances active at the same time, with different load and/or dump behaviour.

Initially only the typical operations are supported, but in principle all functionality of the old interface will be available via `YAML` instances (if you are using something that isn't let me know).

Duplicate keys

In JSON mapping keys should be unique, in YAML they must be unique. PyYAML never enforced this although the YAML 1.1 specification already required this.

In the new API (starting 0.15.1) duplicate keys in mappings are no longer allowed by default. To allow duplicate keys in mappings:

```
yaml = ruamel.yaml.YAML()
yaml.allow_duplicate_keys = True
yaml.load(stream)
```

In the old API this is a warning starting with 0.15.2 and an error in 0.16.0.

CHAPTER 8

Dumping

Transparent usage of new and old API

If you have multiple packages depending on `ruamel.yaml`, or install your utility together with other packages not under your control, then fixing your `install_requires` might not be so easy.

Depending on your usage you might be able to “version” your usage to be compatible with both the old and the new. The following are some examples all assuming `from ruamel import yaml` somewhere at the top of your file and some `istream` and `ostream` appropriately opened for reading resp. writing.

Loading and dumping using the `SafeLoader`:

```
if yaml.version_info < (0, 15):
    data = yaml.safe_load(istream)
    yaml.safe_dump(data, ostream)
else:
    yml = yaml.YAML(typ='safe', pure=True) # 'safe' load and dump
    data = yml.load(istream)
    yml.dump(ostream)
```

Loading with the `CSafeLoader`, dumping with `RoundTripLoader`. You need two `YAML` instances, but each of them can be re-used

```
if yaml.version_info < (0, 15):
    data = yaml.load(istream, Loader=yaml.CSafeLoader)
    yaml.round_trip_dump(data, ostream, width=1000, explicit_start=True)
else:
    yml = yaml.YAML(typ='safe')
    data = yml.load(istream)
    ymlo = yaml.YAML() # or yaml.YAML(typ='rt')
    ymlo.width = 1000
    ymlo.explicit_start = True
    ymlo.dump(ostream)
```

Loading and dumping from `pathlib.Path` instances using the round-trip-loader:

```
# in myyaml.py
if yaml.version_info < (0, 15):
    class MyYAML(yaml.YAML):
        def __init__(self):
            yaml.YAML.__init__(self)
            self.preserve_quotes = True
            self.indent = 4
```

```
        self.block_seq_indent = 2

try:
    from myyaml import MyYAML
except ImportError:
    if yaml.version_info >= (0, 15):
        raise

# some pathlib.Path
from pathlib import Path
inf = Path('/tmp/in.yaml')
outf = Path('/tmp/out.yaml')

if yaml.version_info < (0, 15):
    with inf.open() as ifp:
        data = yaml.round_trip_load(ifp, preserve_quotes=True)
    with outf.open('w') as ofp:
        yaml.round_trip_dump(data, ofp, indent=4, block_seq_indent=2)
else:
    yml = MyYAML()
    # no need for with statement when using pathlib.Path instances
    data = yml.load(inf)
    yml.dump(outf)
```

Reason for API change

`ruamel.yaml` inherited the way of doing things from `PyYAML`. In particular when calling the function `load()` or `dump()` a temporary instances of `Loader()` resp. `Dumper()` were created that were discarded on termination of the function.

This way of doing things leads to several problems:

- it is virtually impossible to return information to the caller apart from the constructed data structure. E.g. if you would get a YAML document version number from a directive, there is no way to let the caller know apart from handing back special data structures. The same problem exists when trying to do on the fly analysis of a document for indentation width.
- these instances were composites of the various load/dump steps and if you wanted to enhance one of the steps, you needed e.g. subclass the emitter and make a new composite (dumper) as well, providing all of the parameters (i.e. copy paste)

Alternatives, like making a class that returned a `Dumper` when called and sets attributes before doing so, is cumbersome for day-to-day use.

- many routines (like `add_representer()`) have a direct global impact on all of the following calls to `dump()` and those are difficult if not impossible to turn back. This forces the need to subclass `Loaders` and `Dumpers`, a long time problem in `PyYAML` as some attributes were not *deep_copied* although a bug-report (and fix) had been available a long time.
- If you want to set an attribute, e.g. to control whether literal block style scalars are allowed to have trailing spaces on a line instead of being dumped as double quoted scalars, you have to change the `dump()` family of routines, all of the `Dumpers()` as well as the actual functionality change in `emitter.Emitter()`. The functionality change takes changing 4 (four!) lines in one file, and being able to enable that another 50+ line changes (non-contiguous) in 3 more files resulting in diff that is far over 200 lines long.
- replacing `libyaml` with something that doesn't both support `0o52` and `052` for the integer 42 (instead of 52 as per YAML 1.2) is difficult

With `ruamel.yaml>=0.15.0` the various steps “know” about the YAML instance and can pick up setting, as well as report back information via that instance. `Representer`s, etc., are added to a reusable instance and different YAML instances can co-exists.

This change eases development and helps prevent regressions.

Differences with PyYAML

If I have seen further, it is by standing on the shoulders of giants.
Isaac Newton (1676)

`ruamel.yaml` is a derivative of Kirill Simonov's [PyYAML 3.11](#) and would not exist without that excellent base to start from.

The following a summary of the major differences with PyYAML 3.11

Defaulting to YAML 1.2 support

PyYAML supports the [YAML 1.1](#) standard, `ruamel.yaml` supports [YAML 1.2](#) as released in 2009.

- YAML 1.2 dropped support for several features unquoted `Yes`, `No`, `On`, `Off`
- YAML 1.2 no longer accepts strings that start with a `0` and solely consist of number characters as octal, you need to specify such strings with `0o[0-7]+` (zero + lower-case `o` for octal + one or more octal characters).
- YAML 1.2 no longer supports [sexagesimals](#), so the string scalar `12:34:56` doesn't need quoting.
- `\` escape for JSON compatibility
- correct parsing of floating point scalars with exponentials

unless the YAML document is loaded with an explicit `version==1.1` or the document starts with:

```
% YAML 1.1
```

, `ruamel.yaml` will load the document as version 1.2.

PY2/PY3 reintegration

`ruamel.yaml` re-integrates the Python 2 and 3 sources, running on Python 2.7 (CPython, PyPy), 3.3, 3.4, 3.5 and 3.6 (support for 2.6 has been dropped mid 2016). It is more easy to extend and maintain as only a miniscule part of the code is Python version specific.

Fixes

- `ruamel.yaml` follows the `indent` keyword argument on scalars when dumping.
- `ruamel.yaml` allows `:` in plain scalars, as long as these are not followed by a space (as per the specification)

Testing

`ruamel.yaml` is tested using `tox` and `py.test`. In addition to new tests, the original PyYAML test framework is called from within `tox` runs.

Before versions are pushed to PyPI, `tox` is invoked, and has to pass, on all supported Python versions, on PyPI as well as `flake8/pep8`

API

Starting with 0.15 the API for using `ruamel.yaml` has diverged allowing easier addition of new features.