yaml Documentation

Release dev

Anthon van der Neut

December 21, 2016

Contents

BitBucket | PyPI

Contents:

CHAPTER 1

Overview

`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)
- anchors 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 is not preserved, you can specify both indentation 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.

CHAPTER 2

Installing

`ruamel.yaml` can be installed from **PyPI_** using:

```
pip install ruamel.yaml
```

There is a 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)

2.1 Optional requirements

If you have the C yaml library and headers installed, as well as the header files for your Python executables then you can use the non-roundtrip but faster C loader and emitter.

On Debian systems you should use:

```
sudo apt-get install libyaml-dev python-dev 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 libyaml-devel python-devel
```

CHAPTER 3

Details

- 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 yaml 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.

3.1 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 `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 routine also has an additional `block_seq_indent` parameter that can be used to push the dash inwards, *within the space defined by* `indent`.

The above example with the often seen `indent=4, 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 `indent=3` resp. `block_seq_indent=2`, gives correct, but counter intuitive results.

3.2 Positioning ':' in top level mappings, prefix in ':'

If you want your toplevel mappings to look like:

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

then call `round_trip_dump()` with `top_level_colon_align=True` (and `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 `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 `top_level_colon_align=12` where `12` has to be an integer that is one more than length of the widest key.

3.3 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:

    ruamel.yaml.load(some_str, Loader=ruamel.yaml.RoundTripLoader, version=(1, 1))

or the equivalent (version can be a tuple, list or string):

    ruamel.yaml.round_trip_load(some_str, version="1.1")

this also works for `load_all`/`round_trip_load_all`.

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

3.4 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.

3.4.1 adding/replacing comments

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

```
from __future__ import print_function

import ruamel.yaml

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

data = ruamel.yaml.load(inp, ruamel.yaml.RoundTripLoader)
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)

print(ruamel.yaml.dump(data, Dumper=ruamel.yaml.RoundTripDumper), end='')
```

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).

3.5 Config file formats

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

3.6 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, eventuallly 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).

3.7 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, Loader=yaml.RoundTripLoader)

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

CHAPTER 4

Examples

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

```
from __future__ import print_function

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'

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()`:

```
import ruamel.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
"""

data = ruamel.yaml.load(inp, ruamel.yaml.RoundTripLoader)
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:

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

data = ruamel.yaml.round_trip_load(yaml_str)
data.insert(1, 'last name', 'Vandelay', comment="new key")
print(ruamel.yaml.round_trip_dump(data))
```

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.

CHAPTER 5

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

5.1 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.

5.2 PY2/PY3 reintegration

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

5.3 Fixes

- `ruamel.yaml` follows the `indent` keyword argument on scalars when dumping.

5.4 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 Python versions, PyPI as well as flake8/pep8