Python YAML package documentation

Release 0.16.7

Anthon van der Neut

Mar 14, 2023

CONTENTS

Github | PyPI

Contents:

CHAPTER

ONE

OVERVIEW

`ruyaml` is a YAML 1.2 loader/dumper package for Python. It is a derivative of Kirill Simonov's PyYAML 3.11.

`ruyaml` supports YAML 1.2 and has round-trip loaders and dumpers.

- 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 separate indentation levels for mappings and sequences (counting for sequences does **not** include the dash for a sequence element) and specific offset of block sequence dashes within that indentation.

Although `ruyaml` still allows most of the PyYAML way of doing things, adding features required a different API then the transient nature of PyYAML's `Loader` and `Dumper`. Starting with `ruyaml` 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 `ruyaml`

CHAPTER

TWO

INSTALLING

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 ruyaml.

ruyaml itself should be installed from PyPI using:

```
pip install ruyaml
```

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

There also is a commandline utility yaml available after installing:

```
pip install ruyaml.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 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

THREE

BASIC USAGE

You load a YAML document using:

```
from ruyaml import YAML

yaml=YAML(typ='safe')  # default, if not specfied, is 'rt' (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, otherwise the faster C libraries will be used when possible/available but these behave slightly different (and sometimes more like a YAML 1.1 loader).

Dumping works in the same way:

```
from ruyaml 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)
```

3.1 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 ruyaml 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 ruyaml import YAML

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

CHAPTER

FOUR

DUMPING PYTHON CLASSES

Only `yaml = YAML(typ='unsafe')` loads and dumps Python objects out-of-the-box. And since it loads **any** Python object, this can be unsafe.

If you have instances of some class(es) that you want to dump or load, it is easy to allow the YAML instance to do that explicitly. You can either register the class with the `YAML` instance or decorate the class.

Registering is done with `YAML.register_class()`:

```
import sys
import ruyaml


class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age


yaml = ruyaml.YAML()
yaml.register_class(User)
yaml.dump([User('Anthon', 18)], sys.stdout)
```

which gives as output:

```
- !User
  name: Anthon
  age: 18
```

The tag `!User` originates from the name of the class.

You can specify a different tag by adding the attribute `yaml_tag`, and explicitly specify dump and/or load *class-methods* which have to be called `from_yaml` resp. `from_yaml`:

```
import sys
import ruyaml


class User:
    yaml_tag = u'!user'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def to_yaml(cls, representer, node):
```

(continues on next page)

(continued from previous page)

```
        return representer.represent_scalar(cls.yaml_tag,
                                            u'{.name}-{.age}'.format(node, node))

    @classmethod
    def from_yaml(cls, constructor, node):
        return cls(*node.value.split('-'))


yaml = ruyaml.YAML()
yaml.register_class(User)
yaml.dump([User('Anthon', 18)], sys.stdout)
```

which gives as output:

```
- !user Anthon-18
```

When using the decorator, which takes the `YAML()` instance as a parameter, the `yaml = YAML()` line needs to be moved up in the file:

```
import sys
from ruyaml import YAML, yaml_object

yaml = YAML()


@yaml_object(yaml)
class User:
    yaml_tag = u'!user'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def to_yaml(cls, representer, node):
        return representer.represent_scalar(cls.yaml_tag,
                                            u'{.name}-{.age}'.format(node, node))

    @classmethod
    def from_yaml(cls, constructor, node):
        return cls(*node.value.split('-'))


yaml.dump([User('Anthon', 18)], sys.stdout)
```

The `yaml_tag`, `from_yaml` and `to_yaml` work in the same way as when using `.register_class()`.

CHAPTER

FIVE

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 `ruyaml` 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 ruyaml import YAML
yaml = YAML()
```

if not explicitly specified.

5.1 Indentation of block sequences

Although ruyaml 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(sequence=4)` (indentation is counted to the beginning of the sequence element).

PyYAML (and older versions of ruyaml) gives you non-indented scalars (when specifying default_flow_style=False):

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

You can use `mapping=4` to also have the mappings values indented. The dump also observes an additional `offset=2` setting that can be used to push the dash inwards, *within the space defined by* `sequence`.

The above example with the often seen `yaml.indent(mapping=2, sequence=4, offset=2)` indentation:

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

The defaults are as if you specified `yaml.indent(mapping=2, sequence=2, offset=0)`.

If the `offset` equals `sequence`, there is not enough room for the dash and the space that has to follow it. In that case the element itself would normally be pushed to the next line (and older versions of ruyaml 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 `sequence=3` resp. `offset=2`, might give correct, but counter intuitive results.

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

5.1.1 Inconsistently indented YAML

If your input is inconsistently indented, such indentation cannot be preserved. The first round-trip will make it consistent/normalize it. Here are some inconsistently indented YAML examples.

`b` indented 3, `c` indented 4 positions:

```
a:
   b:
       c: 1
```

Top level sequence is indented 2 without offset, the other sequence 4 (with offset 2):

```
- key:
    - foo
    - bar
```

5.2 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.

5.2.1 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 `ruyaml` 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 ruyaml==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 ruyaml never emitted sexagesimals, nor octal numbers, and emitted booleans always as true resp. false

5.2.2 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 resuting in Python dict/list). The basic for for this is:

```
from __future__ import print_function

import sys
import ruyaml

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

inp = """\
abc:
  - a     # comment 1
xyz:
  a: 1    # comment 2
  b: 2
  c: 3
```

(continues on next page)

(continued from previous page)

```
  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).

5.2.3 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.

5.2.4 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).

5.2.5 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
```

CHAPTER

SIX

EXAMPLES

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

```
import sys
from ruyaml 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 ruyaml

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

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

```

(continues on next page)

(continued from previous page)

```
ruyaml.dump(code, sys.stdout, Dumper=ruyaml.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(ruyaml.dump(code, Dumper=ruyaml.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 ruyaml 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 ruyaml import YAML

yaml_str = """\
first_name: Art
```

(continues on next page)

(continued from previous page)

```
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 ruyaml 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".

You can change this default indentation by e.g. using `yaml.indent()`:

```
import sys
from ruyaml import YAML

d = dict(a=dict(b=2),c=[3, 4])
yaml = YAML()
yaml.dump(d, sys.stdout)
print('0123456789')
yaml = YAML()
yaml.indent(mapping=4, sequence=6, offset=3)
yaml.dump(d, sys.stdout)
print('0123456789')
```

giving:

```
a:
  b: 2
c:
- 3
- 4
0123456789
a:
    b: 2
c:
   -  3
   -  4
0123456789
```

If a block sequence or block mapping is the element of a sequence, the are, by default, displayed compact notation. This means that the dash of the "parent" sequence is on the same line as the first element resp. first key/value pair of the child collection.

If you want either or both of these (sequence within sequence, mapping within sequence) to begin on the next line use `yaml.compact()`:

```
import sys
from ruyaml import YAML

d = [dict(b=2), [3, 4]]
yaml = YAML()
yaml.dump(d, sys.stdout)
print('='*15)
yaml = YAML()
yaml.compact(seq_seq=False, seq_map=False)
yaml.dump(d, sys.stdout)
```

giving:

```
- b: 2
- - 3
  - 4
===============
-
  b: 2
-
  - 3
  - 4
```

The following program uses three dumps on the same data, resulting in a stream with three documents:

```
import sys
from ruyaml 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(sequence=4, offset=2)
yaml.dump(data, sys.stdout)


def sequence_indent_four(s):
    # this will fail on direclty 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]
```

(continues on next page)

(continued from previous page)

```
        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, in the last document, was inspired by a question posted by *nowox* on StackOverflow.

6.1 Output of `dump()` as a string

The single most abused "feature" of the old API is not providing the (second) stream parameter to one of the `dump()` variants, in order to get a monolithic string representation of the stream back.

Apart from being memory inefficient and slow, quite often people using this did not realise that `print(round_trip_dump(dict(a=1, b=2)))` gets you an extra, empty, line after `b:  2`.

The real question is why this functionality, which is seldom really necessary, is available in the old API (and in PyYAML) in the first place. One explanation you get by looking at what someone would need to do to make this available if it weren't there already. Apart from subclassing the `Serializer` and providing a new `dump` method, which would ten or so lines, another **hundred** lines, essentially the whole `dumper.py` file, would need to be copied and to make use of this serializer.

The fact is that one should normally be doing `round_trip_dump(dict(a=1, b=2)), sys.stdout)` and do away with 90% of the cases for returning the string, and that all post-processing YAML, before writing to stream, can be handled by using the `transform=` parameter of dump, being able to handle most of the rest. But it is also much easier in the new API to provide that YAML output as a string if you really need to have it (or think you do):

```
import sys
from ruyaml import YAML
from io import StringIO

class MyYAML(YAML):
    def dump(self, data, stream=None, **kw):
        inefficient = False
        if stream is None:
            inefficient = True
            stream = StringIO()
        YAML.dump(self, data, stream, **kw)
        if inefficient:
            return stream.getvalue()

yaml = MyYAML()   # or typ='safe'/'unsafe' etc
```

with about one tenth of the lines needed for the old interface, you can once more do:

```
print(yaml.dump(dict(a=1, b=2)))
```

instead of:

```
yaml.dump((dict(a=1, b=2)), sys.stdout)
print()  # or sys.stdout.write('\n')
```

CHAPTER

SEVEN

DEPARTURE FROM PREVIOUS API

With version 0.15.0 `ruyaml` 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 ruyaml 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 `ruyaml`. 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
import ruyaml

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

after:

```
from pathlib import Path
from ruyaml 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 a 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).

The default loader (`typ='rt'`) is a direct derivative of the safe loader, without the methods to construct arbitrary Python objects that make the `unsafe` loader unsafe, but with the changes needed for round-trip preservation of comments, etc.. For trusted Python classes a constructor can of course be added to the round-trip or safe-loader, but this has to be done explicitly (`add_constructor`).

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).

If a parse or dump fails, and throws and exception, the state of the `YAML()` instance is not guaranteed to be able to handle further processing. You should, at that point to recreate the YAML instance before proceeding.

7.1 Loading

7.1.1 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 = ruyaml.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.

When a duplicate key is found it and its value are discarded, as should be done according to the YAML 1.1 specification.

7.2 Dumping a multi-documents YAML stream

The "normal" `dump_all` expected as first element a list of documents, or something else the internals of the method can iterate over. To read and write a multi-document you would either make a `list`:

```
yaml = YAML()
data = list(yaml.load_all(in_path))
# do something on data[0], data[1], etc.
yaml.dump_all(data, out_path)
```

or create some function/object that would yield the `data` values.

What you now can do is create `YAML()` as an context manager. This works for output (dumping) only, requires you to specify the output (file, buffer, `Path`) at creation time, and doesn't support `transform` (yet).

```
with YAML(output=sys.stdout) as yaml:
    yaml.explicit_start = True
    for data in yaml.load_all(Path(multi_document_filename)):
        # do something on data
        yaml.dump(data)
```

Within the context manager, you cannot use the `dump()` with a second (stream) argument, nor can you use `dump_all()`. The `dump()` within the context of the `YAML()` automatically creates multi-document if called more than once.

To combine multiple YAML documents from multiple files:

```
list_of_filenames = ['x.yaml', 'y.yaml', ]
with YAML(output=sys.stdout) as yaml:
    yaml.explicit_start = True
    for path in list_of_filename:
        with open(path) as fp:
            yaml.dump(yaml.load(fp))
```

The output will be a valid, uniformly indented YAML file. Doing `cat {x,y}.yaml` might result in a single document if there is not document start marker at the beginning of `y.yaml`

7.3 Dumping

7.3.1 Controls

On your `YAML()` instance you can set attributes e.g with:

```
yaml = YAML(typ='safe', pure=True)
yaml.allow_unicode = False
```

available attributes include:

`unicode_supplementary`
Defaults to `True` if Python's Unicode size is larger than 2 bytes. Set to `False` to enforce output of the form `\U0001f601` (ignored if `allow_unicode` is `False`)

7.4 Transparent usage of new and old API

If you have multiple packages depending on `ruyaml`, 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 `import ruyaml` somewhere at the top of your file and some `istream` and `ostream` apropriately opened for reading resp. writing.

Loading and dumping using the `SafeLoader`:

```
yml = ruyaml.YAML(typ='safe', pure=True)  # 'safe' load and dump
data = yml.load(istream)
yml.dump(data, ostream)
```

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

```
yml = ruyaml.YAML(typ='safe')
data = yml.load(istream)
ymlo = ruyaml.YAML()  # or yaml.YAML(typ='rt')
ymlo.width = 1000
ymlo.explicit_start = True
ymlo.dump(data, ostream)
```

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

```
# in myyaml.py
class MyYAML(yaml.YAML):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        yaml.YAML.__init__(self)
        self.preserve_quotes = True
        self.indent(mapping=4, sequence=4, offset=2)
# in your code
from myyaml import MyYAML

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

yml = MyYAML()
# no need for with statement when using pathlib.Path instances
data = yml.load(inf)
yml.dump(data, outf)
```

CHAPTER

EIGHT

REASON FOR API CHANGE

`ruyaml` inherited the way of doing things from `PyYAML`. In particular when calling the function `load()` or `dump()` 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 `ruyaml>=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. Representers, etc., are added to a reusable instance and different YAML instances can co-exists.

This change eases development and helps prevent regressions.

CHAPTER

NINE

DIFFERENCES WITH PYYAML

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

`ruyaml` 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

9.1 Defaulting to YAML 1.2 support

PyYAML supports the YAML 1.1 standard, `ruyaml` 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
```

, `ruyaml` will load the document as version 1.2.

9.2 Python Compatibility

`ruyaml` requires Python 3.7 or later.

9.3 Fixes

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

9.4 Testing

`ruyaml` 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

9.5 API

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

CHAPTER

TEN

CONTRIBUTING

All contributions to ruyaml are welcome. Please post an issue or, if possible, a pull request (PR) on github.

Please don't use issues to post support questions.

TODO:: The maintainers of ruyaml don't have an official support channel yet.

10.1 Documentation

The documentation for ruyaml is written in the ReStructured Text format and follows the Sphinx Document Generator's conventions.

10.2 Code

Code changes are welcome as well, but anything beyond a minor change should be tested (tox/pytest), checked for typing conformance (mypy) and pass pep8 conformance (flake8).

In my experience it is best to use two virtualenv environments, one with the latest Python from the 2.7 series, the other with 3.5 or 3.6. In the site-packages directory of each virtualenv make a soft link to the ruyaml directory of your (cloned and checked out) copy of the repository. Do not under any circumstances run pip install -e . it will not work (at least not until these commands are fixed to support packages with namespaces).

You can install tox, pytest, mypy and flake8 in the Python3 virtualenv, or in a virtualenv of their own. If all of these commands pass without warning/error, you can create your pull-request.

10.2.1 Flake

The Flake8 configuration is part of setup.cfg:

```
[flake8]
show-source = True
max-line-length = 95
ignore = F405
```

The suppress of F405 is necessary to allow from xxx import *.

Please make sure your checked out source passes flake8 without test (it should). Then make your changes pass without any warnings/errors.

10.2.2 Tox/pytest

Whether you add something or fix some bug with your code changes, first add one or more tests that fail in the unmodified source when running `tox`. Once that is in place add your code, which should have as a result that your added test(s) no longer fail, and neither should any other existing tests.

10.2.3 Typing/mypy

You should run `mypy` from `ruyaml`'s source directory:

```
mypy --strict --follow-imports silent lib/ruyaml/*.py
```

This command should give no errors or warnings.

10.3 Vulnerabilities

If you find a vulnerability in `ruyaml` (e.g. that would show the `safe` and `rt` loader are not safe due to a bug in the software)), please contact the maintainers directly via email.

After the vulnerability is removed, and affected parties notified to allow them to update versions, the vulnerability will be published, and your role in finding/resolving this properly attributed.

CHAPTER

ELEVEN

UPSTREM MERGE

The process to merge `ruamel.yaml`'s Mercurial repository to ours is non-trivial due to non-unique Merगurial-to-git imports and squash merges.

11.1 Preparation

We create a git import of the Upstream repository. Then we add a pseudo-merge node to it which represents our version of the code at the point where the last merge happened. The commit we want is most likely named "Upstream 0.xx.yy".

So, first we get a git copy of an HG clone of the `ruamel.yaml` repository:

```
# install Mercurial (depends on your distribution)

cd /your/src
mkdir -p ruyaml/git
cd ruyaml/git; git init
cd ../
hg clone http://hg.code.sf.net/p/ruamel-yaml/code hg
```

Next we prepare our repository for merging. We need a `hg-fast-export` script:

```
cd ..
git clone git@github.com:frej/fast-export.git
```

We use that script to setup our git copy:

```
cd ../git
../fast-export/hg-fast-export.sh -r ../hg --ignore-unnamed-heads
```

Now let's create a third repository for the actual work:

```
cd ../
git clone git@github.com:pycontribs/ruyaml.git repo
cd repo
git remote add ../git ruamel
git fetch ruamel
```

Create a branch for merging:

```
git checkout -b merge main
```

This concludes setting things up.

11.2 Incremental merge

First, let's pull the remote changes (if any):

```
cd /your/src/ruyaml/hg
hg pull
cd ../git
../fast-export/hg-fast-export.sh -r ../hg --ignore-unnamed-heads
cd ../repo
git fetch --all
git checkout merge
```

Next, we need a pseudo-merge that declares "we have merged all of Upstream up to *THAT* into *THIS*", where *THIS* is the latest Merge commit in our repository (typically named "Upstream 0.xx.yy") and *THAT* is the corresponding commit in the Ruamel tree (it should be tagged 0.xx.yy):

```
git log --date-order --all --oneline
git reset --hard THIS
git merge -s ours THAT
```

Now we'll "merge" the current Upstream sources:

```
git merge --squash ruamel/main
```

This will create a heap of conflicts, but no commit yet.

Note: The reason we do a squash-merge here is that otherwise git will un-helpfully upload the complete history of `ruamel.yaml` to GitHub. It's already there, of course, but due to the diverging git hashes that doesn't help.

The next step, obviously, is to fix the conflicts. (There will be a bunch.) If git complains about a deleted `__init__.py`, the solution is to `git rm -f __init__.py`.

Then, commit your changes:

```
git commit -a -m "Merge Upstream 0.xx.yz"
git push -f origin merge
```

Now check github. If everything is OK, congratulations, otherwise fix and push (no need to repeat the `-f`).