BespON: Extensible config files with multiline strings, lossless round-tripping, and hex floats

Geoffrey M. Poore[‡*]

Abstract—BespON is a human-editable data format focused on expressive syntax, lossless round-tripping, and advanced features for scientific and technical tasks. Nested data structures can be represented concisely without multiple levels of either brackets or significant whitespace. The open-source Python implementation of BespON can modify data values while otherwise perfectly preserving config file layout, including comments. BespON also provides doc comments that can be preserved through arbitrary data modification. Additional features include integers (binary, octal, decimal, and hex), floats (decimal and hex, including Infinity and NaN), multiline string literals that only preserve indentation relative to delimiters, and an extensible design that can support user-defined data types.

Index Terms—configuration, data serialization, data interchange

Introduction

Many software projects need a human-editable data format, often for configuration purposes. For Python programs, INI-style files, JSON [JSON], and YAML [YAML] are popular choices. More recently, TOML [TOML] has become an alternative. While these different formats have their strengths, they also have some significant weaknesses when it comes to scientific and technical computing.

This paper introduces BespON [BespON], a new human-editable data format focused on scientific and technical features, and the `bespon` package for Python [pkg:bespon]. An overview of INI-style files, JSON, YAML, and TOML provides the motivation for BespON as well as context for its particular feature set.

Though this overview focuses on the features of each format, it also considers Python support for round-tripping—for loading data, potentially modifying it, and then saving it. While round-tripping will not lose data, it will typically lose comments and fail to preserve data ordering and formatting. Since comments and layout can be important in the context of configuration, some libraries provide special support for preserving them under round-tripping. That allows manual editing to be avoided while still minimizing the differences introduced by modifying data.

INI-style formats

Python's `configparser` module [py:configparser] supports a simple config format similar to Microsoft Windows INI files. For example:

```
[key]
subkey = value
```

Because all values are interpreted as strings, any typed values must be retrieved from the parsed data using getter functions that perform type conversion, introducing significant potential for ambiguity. Multiline string values that preserve newlines are permitted, but all indentation and all trailing whitespace (including a final newline) is stripped, so storing precise chunks of text for tasks such as templating is difficult. Another issue is that the format is not strictly specified, so that the Python 2 and Python 3 versions of the package are not fully compatible. This was a primary reason for `configparser` being rejected in PEP 518 [PEP518] as a possible format for storing Python build system requirements.

A more powerful and sophisticated INI-style format is provided by the `configobj` package [pkg:configobj]. All values are still strings as with `configparser`, but the package also provides a validator that allows the required format of a config file to be specified, along with type conversions for each data element. Multiline triple-quoted string literals are supported, though they are somewhat limited since they lack backslash-escapes and thus cannot contain triple-quoted strings or represent special characters using escape sequences. One particularly nice feature of `configobj` is round-trip support. Data can be loaded, modified, and saved, while preserving the order of values and retaining comments.

JSON

JSON [JSON] was designed as a lightweight interchange format. Its focus on a small number of common data types has enabled broad cross-language support, while its simple syntax is amenable to fast parsing. With JSON syntax, the earlier example data becomes:

```
{"key": {"subkey": "value"}}
```

Only dicts, lists, strings, numbers (floats), booleans, and null (`None`) are supported, so binary data and other unsupported types must be dealt with in an ad-hoc manner. As in the INI-style formats, dict keys can only be strings.

For configuration purposes, JSON has disadvantages. It lacks comments. Comments are not necessary in the common case of exchanging JSON data between machines, but in human-edited configuration data, they can be very useful. Similarly, JSON's brackets, braces, and quotation marks are sometimes criticized as verbose for human editing (for example, [PEP518]). For scientific

* Corresponding author: gpoore@uu.edu
‡ Union University

and technical tasks, JSON's lack of an integer type and of floating-point Infinity and NaN can be an issue. In fact, Python's standard library JSON implementation [py:json] explicitly does not comply with the JSON specification by adding extensions for integer, Infinity, and NaN support, and enabling these by default. Another drawback is that a string in JSON must be on a single line; there are no multiline string literals.

JSON's simplicity and limitations are an advantage when it comes to round-tripping data. Since there are no comments, a primary source of complexity is avoided altogether. Since there is only a single possible representation of most data if whitespace is ignored, lossless round-tripping primarily amounts to preserving indentation, line break locations, and the exact manner in which numerical values are represented.

YAML

YAML [YAML] was designed as a general serialization format. It can create a text-based representation of essentially arbitrary data structures, including some programming language-specific types. As a result, it supports integers (decimal, octal, hex), Infinity and NaN floating-point values, Base64-encoded binary data, and a variety of other data types. It also allows non-string dict keys. Its use of significant whitespace to avoid JSON's brackets and braces is reminiscent of Python's own avoidance of braces. In YAML syntax, the example data could be represented without quotation marks or braces:

```
key:
    subkey: value
```

The serialization capabilities of YAML can actually be a disadvantage by blurring the distinction between data and executable code. PyYAML [pkg:PyYAML], perhaps the most common Python YAML implementation, can execute arbitrary code during deserialization unless the special `yaml.safe_load()` function is used. For example, during YAML loading it is possible to run the default Python and include its `--help` output:

```
>>> yaml.load("""
help: !!python/object/apply:subprocess.check_output
      [['python', '--help']]
""")
```

YAML libraries in other languages can exhibit similar behavior by default; YAML deserialization was the source of a major security vulnerability in Ruby on Rails in 2013 [RoR].

YAML has been criticized for its complexity (for example, [PEP518] and [TOML]). This is partially due to the comparatively long YAML specification and the plethora of features it defines. For instance, most characters are allowed unquoted, but in a context-dependent manner. When YAML loads "`a#comment`", it returns the string `a#comment`, but add a space before the `#`, and this becomes the string `a` followed by a line comment. Similarly, Python's `None` may be represented as `null`, `Null`, `NULL`, `~`, or as an empty value (for example, "`k:`" is identical to "`k: null`"). Some YAML issues were resolved in the transition from the version 1.1 specification (2005) to version 1.2 (2009). Among other things, the treatment of `Yes`, `No`, `On`, `Off`, and their lowercase and titlecase variants as boolean values was removed. However, since PyYAML is still based on the version 1.1 specification, the impact of version 1.2 for Python users has been minimal, at least until the `ruamel.yaml` package [pkg:ruamel.yaml] defaulted to the version 1.2 specification in 2016.

YAML does provide multiline string literals. For example:

```
key: |
    a multiline string
    in which line breaks are preserved
```

The multiline string begins on the line after the pipe `|`, and contains all text indented relative to the parent node (`key` in this case). This is a simple and efficient approach with minimal syntax for short snippets of text. It can become complex, however, if whitespace or indentation are important. Since the multiline string has no explicit ending delimiter, by default all trailing whitespace except for the final line break is stripped. This may be customized by using `|-` (remove all trailing whitespace, including the last line break) or `|+` (keep all trailing whitespace). Unfortunately, the `|+` case means that the string content depends on the relative positive of the next data element (or the end of the file, if the string is not followed by anything). Similarly, there are complications if all lines of the string contain leading whitespace or if the first line of the string is indented relative to subsequent lines. In such cases, the pipe must be followed immediately by an integer that specifies the indentation of the string relative to the parent node (`key` in the example).

All line breaks in multiline strings are normalized to line feeds (`\n`). Because backslash-escapes are not allowed in multiline strings, there is no way to wrap long lines, to specify other line break characters explicitly, or to use code points that are prohibited as literals in YAML files (for example, most control characters).

PyYAML provides no round-tripping support. The `ruamel.yaml` package does provide round-trip features. It can maintain comments, key ordering, and most styling so long as dict keys and list values are not deleted. While it supports modifying dict and list values, it does not provide built-in support for renaming dict keys.

TOML

TOML [TOML] is a more recent INI-inspired format. In TOML, the example data could be represented as:

```
[key]
subkey = "value"
```

TOML supports dicts (only with string keys), lists (only with all elements of the same type), strings, floats, integers, and booleans, plus date and time data. There are multiline string literals, both raw (delimited by `'''`) and with backslash-escapes (delimited by `"""`). Though these are very similar to Python multiline strings, they do have the difference that a line feed (`\n`) *immediately* following the opening delimiter is stripped, while it is retained otherwise, even if only preceded by a space.

String keys may be unquoted if they match the pattern for an ASCII identifier, and sections support what might be called "key paths." This allows nested data to be represented in a very compact manner without either brackets and braces or significant indentation. For example:

```
[key.subkey]
subsubkey = "value"
```

would be equivalent to the JSON

```
{"key": {"subkey": {"subsubkey": "value"}}}
```

TOML aims to be obvious, minimal, and more formally standardized than typical INI-style formats. In many ways it succeeds. It is used by Rust's Cargo package manager [Cargo] and in May 2016 was accepted as the future format for storing Python build system dependencies in PEP 518 [PEP518].

For scientific and technical tasks, TOML has some drawbacks. While there are integers, only decimal integers are supported. Decimal floats are supported, but with the notable exception of Infinity and NaN. Unlike YAML, multiline strings cannot be indented for clarity, because any indentation becomes part of the literal string content. There is no built-in support for any form of encoded binary data, and no extension mechanism for unsupported data types. These limitations may make sense in a format whose expanded acronym contains "obvious" and "minimal," but they do make TOML less appropriate for some projects.

In addition to these issues, some current features have the potential to be confusing. Inline dicts of the form

```
{"key" = "value"}
```

are supported, but they are not permitted to break over multiple lines. Meanwhile, inline lists *are* permitted to span multiple lines. When unquoted `true` appears as a dict key, it is a string, because only strings are allowed as keys. However, when it appears as a value, it is boolean true. Thus, `true = true` is a mapping of a string to a boolean.

Two of the more popular TOML implementations for Python are the `toml` package [pkg:toml] and the `pytoml` package [pkg:pytoml], which is being used in PEP 518. Currently, neither provides any round-trip support.

Introducing BespON

"BespON" is short for *Bespoken*, or custom-made, *Object Notation*. It originally grew out of a need for a config format with a `key=value` syntax that also offers excellent multiline string support. I am the creator of PythonTeX [PythonTeX], which allows executable code in Python and several other programming languages to be embedded within LaTeX documents. Future PythonTeX-related software will need a LaTeX-style `key=value` syntax for configuration. Because PythonTeX involves a significant amount of templating with Python code, a config format with multiline strings with obvious indentation would also be very useful. Later, BespON was influenced by some of my other software projects and by my work as a physics professor. This resulted in a focus on features related to scientific and technical computing.

- Integers, with binary, octal, and hexadecimal integers in addition to decimal integers.
- Full floating-point support including Infinity and NaN, and support for hexedecimal floating-point numbers.
- Multiline strings designed with templating and similar tasks in mind.
- A binary data type.
- Support for lossless round-tripping including comment preservation, at least when data is only modified.
- An extensible design that can allow for user-defined data types.

The `bespon` package for Python [pkg:bespon] was first released in April 2017, after over a year of development. It is used in all examples below. Like Python's `json` module [py:json], `bespon` provides `load()` and `loads()` functions for loading data from file-like objects or strings, and `dump()` and `dumps()` functions for dumping data to file-like objects or strings. `bespon` is compatible with Python 2.7 and 3.3+.

None and booleans

Python's `None` and boolean values are represented in BespON as `none`, `true`, and `false`. As in JSON and TOML, all keywords are lowercase. For example:

```
>>> import bespon
>>> bespon.loads("[none, true, false]")
[None, True, False]
```

Numbers

Integers

BespON supports binary, octal, decimal, and hexadecimal integers. Non-decimal integers use `0b`, `0o`, and `0x` base prefixes. Underscores are allowed between adjacent digits and after a base prefix, as in numbers in Python 3.6+ [PEP515]. For example:

```
>>> bespon.loads("[0b_1, 0o_7, 1_0, 0x_f]")
[1, 7, 10, 15]
```

Floats

Decimal and hexadecimal floating point numbers are supported, with underscores as in integers. Decimal numbers use `e` or `E` for the exponent, while hex use `p` or `P`, just as in Python float literals [py:stdtypes]. Infinity and NaN are represented as `inf` and `nan`.

```
>>> bespon.loads("[inf, nan, 2.3_4e1, 0x5_6.a_fp-8]")
[inf, nan, 23.4, 0.3386077880859375]
```

The support for hexadecimal floating-point numbers is particularly important in scientific and technical computing. Dumping and then loading a floating-point value in decimal form will typically involve small rounding errors [py:stdtypes]. The hex representation of a float allows the value to be represented exactly, since both the in-memory and serialized representation use base 2. This allows BespON files to be used in fully reproducible floating-point calculations. When the `bespon` package dumps data, the `hex_floats` keyword argument may be used to specify that all floats be saved in hex form.

Strings

BespON provides both inline strings, which do not preserve literal line breaks, and multiline strings, which do.

Raw and escaped versions of both are provided. Raw strings preserve all content exactly. Escaped strings allow code points to be represented with backslash-escapes. BespON supports Python-style `\xhh`, `\uhhhh`, and `\Uhhhhhhhh` escapes using hex digits `h`, as well as standard shorthand escapes like `\r` and `\n`. It also supports escapes of the form `\u{h...h}` containing 1 to 6 hex digits, as used in Rust [rs:tokens] and some other languages.

In addition, single-word identifier-style strings are allowed unquoted.

Inline strings

Raw inline strings are delimited by a single backtick `` ` ``, double backticks ``` `` ```, triple backticks ```` ``` ````, or a longer sequence that is a multiple of three. This syntax is inspired by [Markdown]; the case of single backticks is similar to Go's raw strings [Go]. A raw inline string may contain any sequence of backticks that is either longer or shorter than its delimiters. If the first non-space character in a raw string is a backtick, then the first space is stripped; similarly, if the last non-space character is a backtick, then the last space is stripped. This allows, for example, the sequence ` ``` ` to represent the literal triple backticks ```, with no leading or trailing spaces.

The overall result is a raw string syntax that can enclose essentially arbitrary content while only requiring string modification (adding a leading or trailing space) in one edge case. Other common raw string syntaxes avoid any string modification, but either cannot enclose arbitrary content or require multiple different delimiting characters. For example, Python does not allow r"\". It does allow r"""\""", but this is not a complete string representing the backslash; rather, it is the start of a raw string that will contain the literal sequence \""" and requires """ as a closing delimiter [py:lexical]. Meanwhile, Rust represents the literal backslash as r#"\"# in raw string syntax, while literal \# would require r##"\#"## [rs:tokens].

Escaped inline strings are delimited by single quotation characters, either a single quote ' or double quote ". These end at the first unescaped delimiting character. Escaped inline strings may also be delimited by triple quotation mark sequences ''' or """, or longer sequences that are a multiple of three. In these cases, any shorter or longer sequence of the delimiting character is allowed unescaped. This is similar to the raw string case, but with backslash-escapes.

Inline strings may be wrapped over multiple lines, in a manner similar to YAML. This allows BespON data containing long, single-line strings to be embedded within a LaTeX, Markdown, or other document without requiring either lines longer than 80 characters or the use of multiline strings with newline escapes. When an inline string is wrapped over multiple line, each line break is replaced with a space unless it is preceded by a code point with the Unicode White_Space property [UAX44], in which case it is stripped. For example:

```
>>> bespon.loads("""
'inline value
 that wraps'
""")
'inline value that wraps'
```

When an inline string is wrapped, the second line and all subsequent lines must have the same indentation.

Multiline strings

Multiline strings also come in raw and escaped forms. Syntax is influenced by heredocs in shells and languages like Ruby [rb:literals]. The content of a multiline string begins on the line *after* the opening delimiter, and ends on the line *before* the closing delimiter. All line breaks are preserved as literal line feeds (\n); even if BespON data is loaded from a file using Windows line endings \r\n, newlines are always normalized to \n. The opening delimiter consists of a pipe | followed immediately by a sequence of single quotes ', double quotes ", or backticks ` whose length is a multiple of three. Any longer or shorter sequence of quote/backtick characters is allowed to appear literally within the string without escaping. The quote/backtick determines whether backslash-escapes are enabled, following the rules for inline strings. The closing delimiter is the same as the opening delimiter with a slash / appended to the end. This enables opening and closing delimiters to be distinguished easily even in the absence of syntax highlighting, which is convenient when working with long multiline strings.

In a multiline string, total indentation is not preserved. Rather, indentation is only kept relative to the delimiters. For example:

```
>>> bespon.loads("""
  |'''
   first line
    second line
  |'''/
""")
' first line\n  second line\n'
```

This allows the overall multiline string to be indented for clarity, without the indentation becoming part of the literal string content.

Unquoted strings

BespON also allows unquoted strings. By default, only ASCII identifier-style strings are allowed. These must match the regular expression:

```
_*[A-Za-z][0-9A-Z_a-z]*
```

There is the additional restriction that no unquoted string may match a keyword (none, true, false, inf, nan) or related reserved word when lowercased. This prevents an unintentional miscapitalization like FALSE from becoming a string and then yielding true in a boolean test.

Unquoted strings that match a Unicode identifier pattern essentially the same as that in Python 3.0+ [PEP3131] may optionally be enabled. These are not used by default because they introduce potential usability and security issues. For instance, boolean false is represented as false. When unquoted Unicode identifier-style strings are enabled, the final e could be replaced with the lookalike code point \u0435, CYRILLIC SMALL LETTER IE. This would represent a string rather than a boolean, and any boolean tests would return true since the string is not empty.

Lists

Lists are supported using an indentation-based syntax similar to YAML as well as a bracket-delimited inline syntax like JSON or TOML.

In an indentation-style list, each list element begins with an asterisk * followed by the element content. For example:

```
>>> bespon.loads("""
* first
* second
* third
""")
['first', 'second', 'third']
```

Any indentation before or after the asterisk may use spaces or tabs, although spaces are preferred. In determining indentation levels and comparing indentation levels, a tab is never treated as identical to some number of spaces. An object that is indented relative to its parent object must share its parent object's indentation exactly. This guarantees that in the event that tabs and spaces are mixed, relative indentation will always be preserved.

In an inline list, the list is delimited by square brackets [], and list elements are separated by commas. A comma is permitted after the last list element (dangling comma), unlike JSON:

```
>>> bespon.loads("[first, second, third,]")
['first', 'second', 'third']
```

An inline list may span multiple lines, as long as everything it contains and the closing bracket are indented at least as much as the line on which the list begins. When inline lists are nested, the required indentation for all of the lists is simply that of the outermost list.

Dicts

Dicts also come in an indentation-based form similar to YAML as well as a brace-delimited inline syntax like JSON or TOML.

In an indentation-style list, keys and values are separated by an equals sign, as in INI-style formats and TOML. For example:

```
>>> bespon.loads("""
key =
    subkey = value
""")
{'key': {'subkey': 'value'}}
```

The rules for indentation are the same as for lists. A dict value that is a string or collection may span multiple lines, but it must always have at least as much indentation as its key if it starts on the same line as the key, or more indentation if it starts on a line after the key. This may be demonstrated with a multiline string:

```
>>> bespon.loads("""
key = |```
   first line
    second line
  |```/
""")
{'key': ' first line\n  second line\n'}
```

Because the multiline string starts on the same line as `key`, the opening and closing delimiters are not required to have the same indentation, and the indentation of the string content is relative to the closing deliter.

In an inline dict, the dict is delimited by curly braces `{}`, and key-value pairs are separated by commas:

```
>>> bespon.loads("""
{key = {subkey = value}}
""")
{'key': {'subkey': 'value'}}
```

As with inline lists, a dangling comma is permitted, as is spanning multiple lines so long as all content is indented at least as much as the line on which the dict begins. When inline dicts are nested, the required indentation for all of the dicts is simply that of the outermost dict.

Dicts support `none`, `true`, `false`, integers, and strings as keys. Floats are not supported as keys by default, since this could produce unexpected results due to rounding.

Key paths and sections

The indentation-based syntax for dicts involves increasing levels of indentation, while the inline syntax involves accumulating layers of braces. BespON provides a key-path syntax that allows this to be avoided in some cases. A nested dict can be created with a series of unquoted, period-separated keys. For example:

```
>>> bespon.loads("""
key.subkey.subsubkey = value
""")
{'key': {'subkey': {'subsubkey': 'value'}}}
```

Key path are scoped, so that once the indentation or brace level of the top of the key path is closed, no dicts created by the key path can be modified. Consider a nested dict three levels deep, with the lowest level accessed via key paths:

```
>>> bespon.loads("""
key =
    subkey.a = value1
    subkey.b = value2
""")
{'key': {'subkey': {'a': 'value1', 'b': 'value2'}}}
```

Key paths starting with `subkey` can be used multiple times at the indentation level where `subkey` is first used. Using `subkey.c` at this level would be valid. However, returning to the indentation level of `key` and attempting to use `key.subkey.c` would result in a scope error. Scoping ensures that all data defined via key paths with common nodes remains relatively localized.

Key paths can also be used in sections similar to INI-style formats and TOML. A section consists of a pipe followed immediately by three equals signs (or a longer series that is a multiple of three), followed by a key path. Everything until the next section definition will be placed under the section key path. For example:

```
>>> bespon.loads("""
|=== key.subkey
subsubkey = value
""")
{'key': {'subkey': {'subsubkey': 'value'}}}
```

This allows both indentation and layers of braces to be avoided, while not requiring the constant repetition of the complete path to the data that is being defined (`key.subkey` in this case).

Instead of ending a section by starting a new section, it is also possible to return to the top level of the data structure using an end delimiter of the form `|===/` (with the same number of equals signs as the opening section delimiter).

Tags

All of the data types discussed so far are implicitly typed; there is no explicit type declaration. BespON provides a tag syntax that allows for explicit typing and some other features. This may be illustrated with the `bytes` type, which can be applied to strings to create byte strings (Python `bytes`):

```
>>> bespon.loads("""
(bytes)> "A string in binary"
""")
b'A string in binary'
```

Similarly, there is a `base16` type and a `base64` type:

```
>>> bespon.loads("""
(base16)> "01 89 ab cd ef"
""")
b'\x01\x89\xab\xcd\xef'
>>> bespon.loads("""
(base64)> "U29tZSBCYXNlNjQgdGV4dA=="
""")
b'Some Base64 text'
```

When applied to strings, tags also support keyword arguments `indent` and `newline`. `indent` is used to specify a combination of spaces and tabs by which all lines in a string should be indented to produce the final string. `newline` takes any code point sequence considered a newline in the Unicode standard [UnicodeNL], or the empty string, and replaces all literal line breaks with the specified sequence. This simplifies the use of literal newlines other than the default line feed (`\n`). When `newline` is applied to a byte string, only newline sequences in the ASCII range are permitted.

```
>>> bespon.loads(r"""
(bytes, indent=' ', newline='\r\n')>
|'''
A string in binary
with a break
|'''/
""")
b' A string in binary\r\n with a break\r\n'
```

Aliases and inheritance

For configuration purposes, it would be convenient to have some form of inheritance, so that settings do not need to be duplicated in multiple dicts. The tag `label` keyword argument allows lists, list elements, dicts, and dict values to be labeled. Then they can be referenced later using aliases, which consist of a dollar sign `$` followed by the label name. Aliases form the basis for inheritance.

Dicts support two keywords for inheritance. `init` is used to specify one or more dicts with which to initialize a new dict. The keys supplied by these dicts must not be overwritten by the keys put into the new dict directly. Meanwhile, `default` is used to specify one or more dicts whose keys are added to the new dict after `init` and after values that are added directly. `default` keys are only added if they do not exist; they are fallback values.

```
>>> d = bespon.loads("""
initial =
    (dict, label=init)>
    first = a
default =
    (dict, label=def)>
    last = z
    k = default_v
settings =
    (dict, init=$init, default=$def)>
    k = v
""")
>>> d['settings']
{'first': 'a', 'k': 'v', 'last': 'z'}
```

If there multiple values for `init` or `default`, these could be provided in an inline list of aliases:

```
[$alias1, $alias2, ...]
```

In similar manner, `init` can be used to specify initial elements in a list, and `extend` to add elements at the end. Other features that make use of aliases are under development.

Immutability, confusability, and other considerations

BespON and the `bespon` package contain several features designed to enhance usability and prevent confusion.

Nested collections more than 100 levels deep are prohibited by default. In such cases, the `bespon` package raises a nesting depth error. This reduces the potential for runaway parsing.

BespON requires that dict keys be unique; keys are never overwritten. Similarly, there is no way to set and then modify list elements. In contrast, the JSON specification only specifies that keys "SHOULD be unique" [JSON]. Python's JSON module [py:json] allows duplicate keys, with later keys overwriting earlier ones. Although YAML [YAML] specifies that keys are unique, in practice PyYaml [pkg:PyYAML] allows duplicate keys, with later keys overwriting earlier ones. TOML [TOML] also specifies unique keys, and this is enforced by the `toml` [pkg:toml] and `pytoml` [pkg:pytoml] packages.

When the last line of an inline or unquoted string contains one or more Unicode code points with `Bidi_Class` R or AL (right-to-left languages) [UAX9], by default no other data objects or comments are allowed on the line on which the string ends. This prevents a right-to-left code point from interacting with following code points to produce ambiguous visual layout as a result of the Unicode bidirectional algorithm [UAX9] that is implemented in much text editing software. Consider an indentation-based dict mapping Hebrew letters to integers (valid BespON):

```
"א" =
    1
"ב" =
    2
```

There is no ambiguity in that case. Now consider the same data, but represented with an inline dict (still valid BespON):

```
{"\u05D0" = 1, "\u05D1" = 2}
```

There is still no ambiguity, but the meaning is less clear due to the Unicode escapes. If the literal letters are substituted, this is the rendering in most text editors (now invalid BespON):

```
{"2 = "ב" ,1 = "א}
```

Because the quotation marks, integers, comma, and equals signs have no strong left-to-right directionality, everything after the first quotation mark until the final curly brace is visually laid out from right to left. When the data is loaded, though, it will produce the correct mapping, since loading depends on the logical order of the code points rather than their visual rendering. By default, BespON prevents the potential for confusion as a result of this logical-visual mismatch, by prohibiting data objects or comments from immediately following an inline or unquoted string with one or more right-to-left code points in its last line. For the same reason, code points with the property `Bidi_Control` [UAX9] are prohibited from appearing literally in BespON data; they can only be produced via backslash-escapes.

Round-tripping

BespON has been designed with round-tripping in mind. Currently, the `bespon` package supports replacing keys and values in data. For example:

```
>>> ast = bespon.loads_roundtrip_ast("""
key.subkey.first = 123   # Comment
key.subkey.second = 0b1101
key.subkey.third = `literal \string`
""")
>>> ast.replace_key(['key', 'subkey'], 'sk')
>>> ast.replace_val(['key', 'sk', 'second'], 7)
>>> ast.replace_val(['key', 'sk', 'third'],
                    '\\another \\literal')
>>> ast.replace_key(['key', 'sk', 'third'], 'fourth')
>>> print(ast.dumps())

key.sk.first = 123   # Comment
key.sk.second = 0b111
key.sk.fourth = `\another \literal`
```

This illustrates several features of the round-trip capabilities.

- Comments, layout, and key ordering are preserved exactly.
- Key renaming works even with key paths, when a given key name appears in multiple locations.
- When a number is modified, the new value is expressed in the same base as the old value by default.
- When a quoted string is modified, the new value is quoted in the same style as the old value (at least when practical).

- As soon as a key is modified, the new key must be used for further modifications. The old key is invalid.

In the future, the bespon package will add additional round-trip capabilities beyond replacing keys and values. One of the challenges in round-tripping data is dealing with comments. BespON supports standard line comments of the form #comment. While these can survive round-tripping when data is added or deleted, dealing with them in those cases is difficult, because line comments are not uniquely associated with individual data objects. To provide an alternative, BespON defines a doc comment that is uniquely associated with individual data objects. Each data object may have at most a single doc comment. The syntax is inspired by string and section syntax, involving three hash symbols (or a multiple of three). Both inline and multiline doc comments are defined, and must come immediately before the data with which they are associated (or immediately before its tag, for tagged data):

```
key1 = ### inline doc comment for value 1 ###
       value1
key2 = |###
       multiline doc comment

       for value2
       |###/
       value2
```

Because doc comments are uniquely associated with individual data elements, they will make possible essentially arbitrary manipulation of data while retaining all relevant comments.

Performance

Since the beginning, performance has been a concern for BespON. The bespon package is pure Python. YAML's history suggested that this could be a significant obstacle to performance. PyYAML [pkg:PyYAML] can be much slower than Python's json module [py:json] for loading equivalent data, in part because the JSON module is implemented in C while the default PyYAML is pure Python. PyYAML can be distributed with LibYAML [LibYAML], a C implementation of YAML 1.1, which provides a significant performance improvement.

So far, bespon performance is promising. The package uses __slots__ and avoids global variables extensively, but otherwise optimizations are purely algorithmic. In spite of this, under CPython it can be only about 50% slower than PyYAML with LibYAML. Under PyPy [PyPy], the alternative Python implementation with a just-in-time (JIT) compiler, bespon can be within an order of magnitude of json's CPython speed.

Figure 1 shows an example of performance in loading data. This was generated with the BespON Python benchmarking code [bespon:benchmark]. A sample BespON data set was assembled using the template below (whitespace reformatted to fit column width), substituting the template field {num} for integers in range(1000) and then concatenating the results.

```
key{num} =
  first_subkey{num} =
    "Some text that goes on for a while {num}"
  second_subkey{num} =
    "Some more text that also goes on and on {num}"
  third_subkey{num} =
    * "first list item {num}"
    * "second list item {num}"
    * "third list item {num}"
```

Analogous data sets were generated for JSON, YAML, and TOML, using the closest available syntax. Python's json module and the PyYAML, toml, pytoml, and bespon packages were then used to load their corresponding data from strings 10 times. Load times were measured with Python's timeit module [py:timeit], and the minimum time for each package was recorded and plotted in the figure.

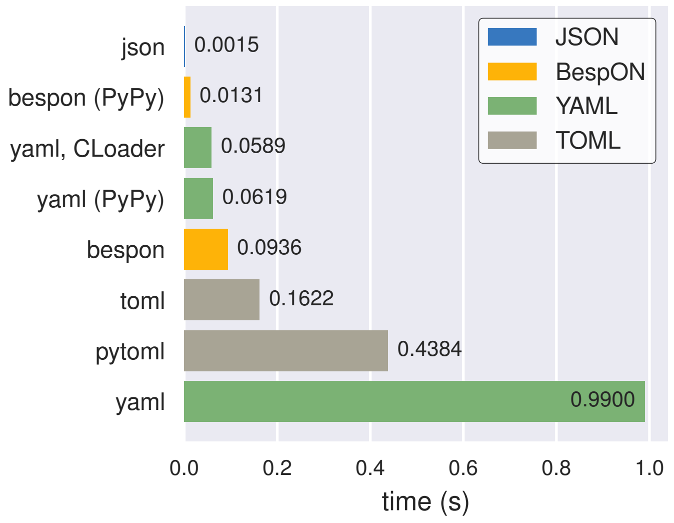

Fig. 1: Performance of Python's json module and the PyYAML, toml, pytoml, and bespon packages in loading sample data. All tests were performed under Ubuntu 16.04. All tests used Anaconda Python 3.6.1 (64-bit) except those designated with "PyPy," which used PyPy3.5 5.7.1 (64-bit). PyYAML was tested with its C library implementation (CLoader) when available.

An extended example

All examples shown so far have been short snippets loaded from Python strings using bespon.loads(). Any of those examples could instead have been saved in a text file, say data.bespon, and loaded as

```
with open('data.bespon', encoding='utf8') as f:
    data = bespon.load(f)
```

A longer example of a BespON file that could be loaded in this manner is shown below. It illustrates most BespON features.

```
# Line comments can be round-tripped if data
# elements are only modified, not added or removed.

### This doc comment can always be round-tripped.###
# Only one doc comment is allowed per data element.
# The doc comment above belongs to the key below.
"key (\x5C escapes)" = 'value (\u{5C} escapes)'

`key (no \ escapes)` = ``value (no `\` escapes)``

# Unquoted ASCII identifier-style strings.
unquoted_key = unquoted_value

# Trailing commas are fine.
inline_dict = {key1 = value1, key2 = value2,}

# Decimal, hex, octal, and binary integers.
inline_list_of_ints = [1, 0x12, 0o755, 0b1010]

list_of_floats =
    * 1.2e3
    * -inf     # Infinity and NaN are supported.
    * 0x4.3p2  # Hex floats to avoid rounding.
```

```
wrapped_string = """String with no whitespace
    lines, with line breaks converted to spaces,
    and "quotes" allowed by delimiters."""

multiline_raw_string = |```
      Linebreaks are kept (as '\n') and leading
      indentation is preserved relative to
      delimiters (which are on lines by themselves).
    |```/

multiline_escaped_string = |"""
    The same idea as the raw multiline string,
    but with backslash-escapes.
    |"""/

typed_string = (bytes)> "byte string"

# Key-path style; same as "key1 = {key2 = true}"
key1.key2 = true

# Same as "section = {subsection = {key = value}}"
|=== section.subsection
key = value
|===/  # Back to root level.  Can be omitted
       # if sections never return to root.
```

Conclusion

BespON and the `bespon` package remain under development.

The `bespon` package is largely complete as far as loading and dumping data are concerned. The standard, default data types discussed above are fully supported, and it is already possible to enable a limited selection of optional types.

The primary focus of future `bespon` development will be on improving round-tripping capabilities. Eventually, it will also be possible to enable optional user-defined data types with the tag syntax.

BespON as a configuration format will primarily be refined in the future through the creation of a more formal specification. The Python implementation is written in such a way that a significant portion of the grammar already exists in the form of Python template strings, from which it is converted into functions and regular expressions. A more formal specification will bring the possibility of implementations in additional languages.

Working with BespON will also be improved through additional revision of the programming language-agnostic test suite [bespon:test] and the syntax highlighting extension for Microsoft Visual Studio Code [bespon:vscode]. The language-agnostic test suite is a set of BespON data files containing hundreds of snippets of BespON that is designed to test implementations for conformance. It is used for testing the Python implementation before each release. The VS Code syntax highlighting extension provides a TextMate grammar [TextMate] for BespON, so it can provide a basis for BespON support in other text editors in the future.

References

[BespON] G. Poore. "BespON – Bespoken Object Notation," https://bespon.org/.

[bespon:benchmark] G. Poore. "Benchmark BespON in Python," https://github.com/bespon/bespon_python_benchmark.

[bespon:test] G. Poore. "Language-agnostic tests for BespON," https://github.com/bespon/bespon_tests.

[bespon:vscode] G. Poore. "BespON syntax highlighting for VS Code," https://github.com/bespon/bespon_vscode.

[Cargo] "CARGO: packages for Rust," https://crates.io/.

[Go] "The Go Programming Language Specification," November 18, 2016, https://golang.org/ref/spec.

[JSON] T. Bray. "The JavaScript Object Notation (JSON) Data Interchange Format," https://tools.ietf.org/html/rfc7159.

[LibYAML] "LibYAML," http://pyyaml.org/wiki/LibYAML.

[Markdown] J. Gruber. "Markdown: Syntax," https://daringfireball.net/projects/markdown/syntax.

[PEP515] G. Brandl, S. Storchaka. "PEP 515 -- Underscores in Numeric Literals," https://www.python.org/dev/peps/pep-0515/.

[PEP518] B. Cannon, N. Smith, D. Stufft. "PEP 518 -- Specifying Minimum Build System Requirements for Python Projects," https://www.python.org/dev/peps/pep-0518/.

[PEP3131] M. von Löwis. "PEP 3131 -- Supporting Non-ASCII Identifiers," https://www.python.org/dev/peps/pep-3131/.

[pkg:bespon] G. Poore, "bespon package for Python," https://github.com/gpoore/bespon_py.

[pkg:configobj] M. Foord, N. Larosa, R. Dennis, E. Courtwright. "Welcome to configobj's documentation!" http://configobj.readthedocs.io/en/latest/index.html.

[pkg:pytoml] "pytoml," https://github.com/avakar/pytoml.

[pkg:PyYAML] "PyYAML Documentation," http://pyyaml.org/wiki/PyYAMLDocumentation.

[pkg:ruamel.yaml] A. van der Neut. "ruamel.yaml," http://yaml.readthedocs.io/en/latest/index.html.

[pkg:toml] "TOML: Python module which parses and emits TOML," https://github.com/uiri/toml.

[PythonTeX] G. Poore. "PythonTeX: reproducible documents with LaTeX, Python, and more," *Computational Science & Discovery* 8 (2015) 014010, http://stacks.iop.org/1749-4699/8/i=1/a=014010.

[py:configparser] Python Software Foundation. "configparser — Configuration file parser", Apr 09, 2017, https://docs.python.org/3.6/library/configparser.html.

[py:json] Python Software Foundation. "json — JSON encoder and decoder," May 27, 2017, https://docs.python.org/3/library/json.html.

[py:lexical] Python Software Foundation. "Lexical analysis," Mar 26, 2017, https://docs.python.org/3/reference/lexical_analysis.html.

[py:stdtypes] Python Software Foundation. "Built-in Types," May 16, 2017, https://docs.python.org/3/library/stdtypes.html.

[py:timeit] Python Software Foundation. "`timeit` — Measure execution time of small code snippets," Mar 26, 2017, https://docs.python.org/3/library/timeit.html.

[PyPy] "Welcome to PyPy," http://pypy.org/.

[rb:literals] "Literals," https://ruby-doc.org/core-2.4.1/doc/syntax/literals_rdoc.html.

[RoR] A. Patterson. "Multiple vulnerabilities in parameter parsing in Action Pack (CVE-2013-0156)," https://groups.google.com/forum/#!topic/rubyonrails-security/61bkgvnSGTQ/discussion.

[rs:tokens] The Rust Project Developers. "Tokens," https://doc.rust-lang.org/reference/tokens.html.

[TextMate] MacroMates Ltd. "Language Grammars," https://manual.macromates.com/en/language_grammars.

[TOML] T. Preston-Werner. "TOML: Tom's Obvious, Minimal Language, v0.4.0," https://github.com/toml-lang/toml/.

[UAX9] M. Davis, A. Lanin, and A. Glass. "Unicode Standard Annex #9: UNICODE BIDIRECTIONAL ALGORITHM," http://unicode.org/reports/tr9/.

[UAX44] Unicode, Inc., ed. M. Davis, L. Iancu, and K. Whistler. "Unicode Standard Annex #44: UNICODE CHARACTER DATABASE," http://unicode.org/reports/tr44/.

[UnicodeNL] The Unicode Consortium. *The Unicode Standard, Version 9.0.0*, chapter 5.8, "Newline Guidelines," http://www.unicode.org/versions/Unicode9.0.0/.

[YAML] O. Ben-Kiki, C. Evans, I. döt Net. "YAML Ain't Markup Language (YAML) Version 1.2, 3rd Edition, Patched at 2009-10-01," http://www.yaml.org/spec/1.2/spec.html.