textdata Documentation

Release 1.7.2

Jonathan Eunice

May 31, 2017

Contents

One often needs to state data in program source. Python, however, needs its lines indented *just so*. Multi-line strings therefore often have extra spaces and newline characters you didn't really want. Many developers "fix" this by using Python `list` literals, but that has its own problems: it's tedious, more verbose, and often less legible.

The `textdata` package makes it easy to have clean, nicely-whitespaced data specified in your program, but to get the data without extra whitespace cluttering things up. It's permissive of the layouts needed to make Python code look and work right, without reflecting those requirements in the resulting data.

Python string methods give easy ways to clean text up, but it's no joy reinventing that particular wheel every time you need it–especially since many of the details are nitsy, low-level, and a little tricky. `textdata` is a "just give me the text!" module that replaces *a la carte* text cleanups with simple, well-tested code that doesn't lengthen your program or require constant wheel-reinvention.

CHAPTER 1

A Few Examples

```
data = lines("""
    There was an old woman who lived in a shoe.
    She had so many children, she didn't know what to do;
    She gave them some broth without any bread;
    Then whipped them all soundly and put them to bed.
""")
```

will result in:

```
['There was an old woman who lived in a shoe.',
 "She had so many children, she didn't know what to do;",
 'She gave them some broth without any bread;',
 'Then whipped them all soundly and put them to bed.']
```

Note that the "extra" newlines and leading spaces have been taken care of and discarded.

In addition to `lines`, `text` works similarly and with the same parameters, but joins the resulting lines into a unified string.:

```
data = text("""
    There was an old woman who lived in a shoe.
    She had so many children, she didn't know what to do;
    She gave them some broth without any bread;
    Then whipped them all soundly and put them to bed.
""")
```

Yields:

```
"There was an old woman who lived in a shoe.\nShe ... to bed."
# where the ... abbreviates exactly the characters you'd expect
```

So it does the same stripping of pointless whitespace at the beginning and end, returning the data as a clean, convenient string.

Note that while `text` returns a single string, it maintains the (potentially useful) newlines. Its result is still line-oriented by default. If you want to elide the newlines, use `text(text, join=' ')` and the newline characters will be replaced with spaces.

A new `textline` call makes this even easier. It gives a single, no-breaks string by default. It's particularly useful for rendering single, very long lines.

Note: Earlier versions of this library described a routine `textlines`. It is the same as `text`, just renamed to be consistent with the rest of the module. The shorter `text` is now the preferred name, and `textlines` is deprecated.

CHAPTER 2

Lines and Textlines

Both `lines` and `text` provide provide routinely-needed cleanups:

- remove starting and ending blank lines (which are usually due to Python source formatting)
- remove blank lines internal to your text block
- remove common indentation
- strip leading/trailing spaces other than the common prefix (leading spaces removed by request, trailing by default)
- strip any comments from the end of lines
- join lines together with your choice of separator string

```
lines(source, noblanks=True, dedent=True, lstrip=False, rstrip=True,
cstrip=True, join=False)
```

Returns text as a series of cleaned-up lines.

- `source` is the text to be processed.
- `noblanks` => all blank lines are eliminated, not just starting and ending ones. (default `True`).
- `dedent` => strip a common prefix (usually whitespace) from each line (default `True`).
- `lstrip` => strip all left (leading) space from each line (default `False`). Note that `lstrip` and `dedent` are mutually exclusive ways of handling leading space.
- `rstrip` => strip all right (trailing) space from each line (default `True`)
- `cstrip` => strip comments (from `#` to the end of each line (default `True`)
- `join` => either `False` (do nothing), `True` (concatenate lines with `\n`), or a string that will be used to join the resulting lines (default `False`)

```
text(source, noblanks=True, dedent=True, lstrip=False, rstrip=True,
cstrip=True, join='\n')
```

Does the same helpful cleanups as `lines()`, but returns result as a single string, with lines separated by newlines (by default) and without a trailing newline.

Note: Text cleanups inherently convert tabs to sequences of spaces, consistent with Python's `str.expandtabs`. There is currently no option to turn this behavior off.

CHAPTER 3

Words

Often the data you need to encode is almost, but not quite, a series of words. A list of names, a list of color names–values that are mostly single words, but sometimes have an embedded spaces. `textdata` has you covered:

```
>>> words(' Billy Bobby "Mr. Smith" "Mrs. Jones"  ')
['Billy', 'Bobby', 'Mr. Smith', 'Mrs. Jones']
```

Embedded quotes (either single or double) can be used to construct "words" (or phrases) containing whitespace (including tabs and newlines).

`words` isn't a full parser, so there are some extreme cases like arbitrarily nested quotations that it can't handle. It isn't confused, however, by embedded apostrophes and other common gotchas. For example:

```
>>> words("don't be blue")
["don't", "be", "blue"]

>>> words(""" "'this'" works '"great"' """)
["'this'", 'works', '"great"']
```

`words` is a good choice for situations where you want a compact, friendly, whitespace-delimited data representation–but a few of your entries need more than just `str.split()`.

CHAPTER 4

Comments

If you need to embed more than a few lines of immediate data in your program, you may want some comments to explain what's going on. By default, `textdata` strip out Python-like comments (from `#` to end of line). So:

```
exclude = words("""
    __pycache__ *.pyc *.pyo     # compilation artifacts
    .hg* .git*                  # repository artifacts
    .coverage                   # code tool artifacts
    .DS_Store                   # platform artifacts
""")
```

Yields:

```
['__pycache__', '*.pyc', '*.pyo', '.hg*', '.git*',
 '.coverage', '.DS_Store']
```

You could of course write it out as:

```
exclude = [
    '__pycache__', '*.pyc', '*.pyo',    # compilation artifacts
    '.hg*', '.git*',                    # repository artifacts
    '.coverage',                        # code tool artifacts
    '.DS_Store'                         # platform artifacts
]
```

But you'd need more nitsy punctuation, and it's less compact.

If however you want to capture comments, set `cstrip=False` (though that is probably more useful with the `lines` and `text` APIs than for `words`).

CHAPTER 5

Paragraphs

Sometimes you want to collect "paragraphs"–contiguous runs of text lines that are delineated by blank lines. Markdown and RST document formats, for example, use this convention. `textdata` has a `paras` routine to extract such paragraphs:

```
>>> rhyme = """
    Hey diddle diddle,

    The cat and the fiddle,
    The cow jumped over the moon.
    The little dog laughed,
    To see such sport,

    And the dish ran away with the spoon.
"""
>>> paras(rhyme)
[['Hey diddle diddle,'],
 ['The cat and the fiddle,',
  'The cow jumped over the moon.',
  'The little dog laughed,',
  'To see such sport,'],
 ['And the dish ran away with the spoon.']]
```

Or if you'd like paras, but each paragraph in a single string:

```
>>> paras(rhyme, join="\n")
['Hey diddle diddle,',
 'The cat and the fiddle,\nThe cow jumped over the moon.\nThe little dog laughed,\nTo␣
↪see such sport,',
 'And the dish ran away with the spoon.']
```

Setting `join` to a space will of course concatenate the lines of each paragraph with a space. This can be useful for converting from line-oriented paragraphs into each-paragraph as a (potentially very long) single line, a format useful for cut-and-pasting into many editors and text entry boxes on the Web or for email systems.

On the off chance you want to preserve the exact intra-paragraph spacing, setting `keep_blanks=True` will accomplish that.

CHAPTER 6

Attributes (Dicts)

Dictionaries are hugely useful in Python, but not always the most compact to state. In the literal form, key names have to be quoted (unlike JavaScript), and there are very specific key-value separation rules (using `:` in the literal form, and = in the constructor form.

`textdata` contains a more concise constructor, `attrs`:

```
>>> attrs("a=1 b=2 c='something more'")
{'a': 1, 'c': 'something more', 'b': 2}
```

Note that quotes are not required for keys, there are no required separators between key-value pairs, and that the values for numercial values are rendered from string representation into actual Python `int` (or `float`, `complex`, etc.) types. Pretty slick, huh?

Even better, colons may also be used as key-value separators, and quotes are only required if the value includes spaces.:

```
>>> attrs("a:1 b:2 c:'something more'")
{'a': 1, 'b': 2, 'c': 'something more'}
```

This makes specifying dictionary contents easier and less verbose, and makes it easier to import from JavaScript, HTML, or XML. To make it easier to import from CSS, semicolons may be used to separate key-value pairs.:

```
>>> attrs("a:1; b: green")
{'a': 1, 'b': 'green'}
```

Finally, for familiarity with Python literal forms, keys may be quoted, and key-value pairs may be separated by commas.:

```
>>> attrs(" 'a':1, 'the color': green")
{'a': 1, 'the color': 'green'}
```

About the only option that isn't available is that keys are always strings, not lteral values, and the Python triple quote is not supported.

You might think that this level of generality and flexibility would make parsing unreliable, but it doesn't seem to be so. The `attrs` parser and its support code are significantly tested. (And it's derived from a JavaScript codebase which is

itself significantly tested.)

Literals and Return Type

`attrs` tries hard to "do the right thing" with data presented to it, iincluding parsing the string form of numbers and other data types into those data types. However, that behavior is controllable. To disable the parsing of Python literal values, set `literal=False`.

It's also a sad fact of Python life that, until version 3.6 (late 2016!), there was no clean way to present a literal `dict` that would preserve the order of keys in the same order as the source code. As a result, Python developers have often needed the much less graceful `collections.OrderedDict`, which, while effective, lacked a clean literal form. `attrs` can help.:

```
from collections import OrderedDict

attrs('a=1 b=2 c=3', astype=OrderedDict)
```

Which is terse, yet returns an `OrderedDict` with its keys in the expected order.

CHAPTER 7

Unicode and Encodings

`textdata` doesn't have any unique friction with Unicode characters and encodings. That said, any time you use Unicode characters in Python source files, care is warranted–especially in Python 2!

If your text includes Unicode, in Python 2 make sure to mark literal strings with a "u" prefix: `u""`. You can also do this in Python 3.3 and following. Sadly, there was a dropout of compatibility in early Python 3 releases, making it much harder to maintain a unified source base with them in the mix. (A compatibility function such as `six.u` from six can help alleviate much–though certainly not all–of the pain.)

It can also be helpful to declare your source encoding: put a specially-formatted comment as the first or second line of the source code:

```
# -*- coding: utf-8 -*-
```

This will usually endorse UTF-8, but other encodings are possible. Python 3 defaults to a UTF-8 encoding, but Python 2 sadly assumes ASCII.

Finally, if you are reading from or writing to a file on Python 2, strongly recommend you use an alternate form of `open` that supports automatic encoding (which is built-in to Python 3). E.g.:

```
from codecs import open

with open('filepath', encoding='utf-8') as f:
    data = f.read()
```

This construction works across Python 2 and 3. Just add a `mode='w'` for writing.

CHAPTER 8

Alternate Data Paths

`textdata` is primarily designed to deal with text coming from source code, but there's no reason it must be. Text coming from a file, from a generator, or other sources can enjoy the module's text cleanups and lightweight parsing.

To make this "from whatever source" ability more general, all of the `textdata` entry points (`lines`, `text`, `words`, and `paras`) can accept a sequence of lines. Most often this will be a list of lines, but it can also be an iterator, generator, or such that returns a sequence of strings.

CHAPTER 9

API

CHAPTER 10

Notes

- Those who like how `textdata` simplifies data extraction from text should also consider quoter, a module with the same philosophy about wrapping text and joining composite data into strings.
- Automated multi-version testing managed with the wonderful pytest, pytest-cov, coverage, and tox. Continuous integration testing with Travis-CI. Packaging linting with pyroma.
- Successfully packaged for, and tested against, all late-model versions of Python: 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6, as well as recent versions of PyPy and PyPy3.
- It's tempting to define a constant such as `Dedent` that might be the default for the `lstrip` parameter, instead of having separate `dedent` and `lstrip` Booleans. The more I use singleton classes in Python as designated special values, the more useful they seem.
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions.

CHAPTER 11

Installation

To install or upgrade to the latest version:

```
pip install -U textdata
```

To `easy_install` under a specific Python version (3.3 in this example):

```
python3.3 -m easy_install --upgrade textdata
```

(You may need to prefix these with `sudo` to authorize installation. In environments without super-user privileges, you may want to use `pip`'s `--user` option, to install only for a single user, rather than system-wide.)

Testing

If you wish to run the module tests locally, you'll need to install `pytest` and `tox`. For full testing, you will also need `pytest-cov` and `coverage`. Then run one of these commands:

```
tox                # normal run - speed optimized
tox -e py27        # run for a specific version only (e.g. py27, py34)
tox -c toxcov.ini  # run full coverage tests
```

CHAPTER 12

Change Log

1.7.2 (May 30, 2017)

Update compatibility strategy to make Python 3 centric. Python 2 is now the outlier. More future-proof.

Doc tweaks.

1.7.1 (January 30, 2017)

Returned test coverage to 100% of lines (introducing *attrs()* took it briefly down to 99% testing).

1.7.0 (January 30, 2017)

Added *attrs()* function for parsing *dict* instances out of text.

1.6.2 (January 23, 2017)

Updates testing. Newly qualified under 2.7.13 and 3.6, as well as most recent builds of pypy and pypy3.

1.6.1 (September 15, 2015)

Added Python 3.5.0 final and PyPy 2.6.1 to the testing matrix.

1.6.0 (September 1, 2015)

Added `textline()` routine (NB `textline` not `textlines`) as a quick "grab a single very long line" function. It actually allows multiple paragraphs to be grabbed, each as a single long line, separated by double-newlines (i.e. Markdown style).

1.5.0 (September 1, 2015)

Added `text()` as preferred synonym for `textlines()`, as that is more consistent with the rest of the naming scheme. Deprecated `textlines()`.

1.4.3 (August 26, 2015)

Reorganizes documentation using Sphinx.

1.4.2 (August 17, 2015)

Achieves 100% test coverage. Updated testing scheme to automatically evaluate and report combined coverage across multiple Python versions.

1.4.0

Allows all routines to accept a list of text lines, in addition to text as a single string.

1.3.0

Adds a paragraph constructor, `paras`.

1.2.0

Adds comment stripping. Packaging and testing also tweaked.

1.1.5

Adds the `bdist_wheel` packaging format.

1.1.3

Switches from BSD to Apache License 2.0 and integrates `tox` testing with `setup.py`.

1.1.0

Added the `words` constructor.

1.0

Misc. changes from 1.0 or prior:

Common line prefix is now computed without considering blank lines, so blank lines need not have any indentation on them just to "make things work."

The tricky case where all lines have a common prefix, but it's not entirely composed of whitespace, now properly handled. This is useful for lines that are already "quoted" such as with leading `"|"` or `">"` symbols (common in Markdown and old-school email usage styles).

`textlines()` is now somewhat superfluous, now that `lines()` has a `join` kwarg. But you may prefer it for the implicit indication that it's turning lines into text.

Python Module Index

t

Index

T