textdata Documentation

Release 2.4.0

Jonathan Eunice

Jan 23, 2019

Contents

`textdata` makes it easy to have clean, nicely-whitespaced data specified in your program (or a separate text file).

It helps manage both formatting and data type conversions, keeping extra code, gratuitious whitespaces, and data-specification syntax out of your way.

It's permissive of the human-oriented layouts needed to make code and data blocks look good, 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 "do what I mean!" 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

Lines and Text

```
>>> 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.
... """)
['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.

```
>>> 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.
... """)

"There was an old woman who lived in a shoe.\nShe ...put them 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 `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.

1.1 API Options

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 (dedent)
- expand tabs into spaces (optional)
- 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. It can be presented as a single string, or as a list of lines.
- `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`).
- `expandtabs` => expand tabs in string (if `True`, by default amount, otherwise by given amount).
- `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.

CHAPTER 2

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.

```
>>> 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" (really, 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.

```
>>> 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()`.

2.1 Explicit Separators

There is a second mode of operation for `words` in which you provide explicit separators. This is handy if, for example, you have a number of phrases with embedded spaces. This happens often when importing data from spreadsheets.

```
>>> words('First Name / Last Name / Age / Best Feature', sep='/')
['First Name', Last Name', 'Age', 'Best Feature']
```

Here you have a very terse specification of the phrases, without the need to quote in order to preserve embedded spaces.

CHAPTER 3

Paragraphs

Sometimes you want to collect "paragraphs"–contiguous runs of text lines delineated by blank lines. Markdown and RST document formats, for example, use this convention.

```
>>> 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 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 4

Attributes (Dicts)

Dictionaries are hugely useful in Python, but not always the most compact to state. In the literal form, key names must 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, 'b': 2, 'c': 'something more'}
```

(The order in which key-value pairs appear may vary depending on what Python verssion you're running. Python prior to 3.6 was almost perversly eager to randomize dictionary order; see below for some workarounds.)

Note that:

1. Quotes are not required for keys; they're assumed to be strings.
2. No separators are required between key-value pairs (though commas and semicolons may be optionally used).
3. What would "natrually" be a numercial value in Python is indeed a numerical value, not the string representation you might assume a parsing routine would render.

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' d=sweet!")
{'a': 1, 'b': 2, 'c': 'something more', d: 'sweet!'}
```

To make it easier to import from CSS, semicolons may optionally 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 interpreted as strings, not lteral values, and the Python triple quote is not supported.

You might think that this level of 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.) And supporting all these forms makes importing content directly from from JavaScript, JSON, HTML, CSS, or XML quite straightforward.

4.1 Evaluation

`attrs` tries hard to "do the right thing" with data presented to it, including parsing the string form of numbers and other data types into natural Python data types. However, that behavior is controllable. To disable the parsing of Python literal values, set `evaluate='minimal'` (alternatively, `evaluate=False`).

Evaluation behavior in general is configurable with the `evaluate` keyword parameter. `natural` is the default, attempting to convert values that "look like" `int`, `float`, `complex`, `bool`, or `None` types into their corresponding Python values.

The hard case is converting from HTML or XML, in which values are often quoted regardless of intended type, so context is the only way to know if the type should be textual or something else. Quotes are a very strong indicaton that you want a string value type back. As a result, if you use quoted HTML/XML forms, you have to specifically ask for `full` evaluation to get back numeric and other value types.

```
>>> # Note values returned as strings, even though they look like numbers
>>> # That's because they're explicitly quoted
>>> attrs('a="1" b="2" c="something more"')
{'a': '1', 'b': '2', 'c': 'something more'}

>>> # Request 'full' evaluation to get numeric values
>>> attrs('a="1" b="2" c="something more"', evaluate='full')
{'a': 1, 'b': 2, 'c': 'something more'}
```

4.2 Return Type

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 OrdredDict
>>> attrs("a=1 b=2 c='something more'", dict=OrderedDict)
OrderedDict([('a', 1), ('b', 2), ('c', 'something more')])
```

Terse, yet returns an `OrderedDict` with its keys in the expected order.

`attrs` also exports `Dict`, an attribute-accessible `dict` subclass. (Note, in future versions this will been replaced with items.Item, an inherently ordered, attribute-accessible dictionary.

```
>>> attrs("a=1 b=2 c='something more'", dict=Dict)
Dict(a=1, b=2, c='something more')

>>> d = attrs("a=1 b=2 c='something more'", dict=Dict)
>>> d.a
```

(continues on next page)

(continued from previous page)

```
1
>>> d.a = 12
>>> d
Dict(a=12, b=2, c='something more')
```

CHAPTER 5

Tables

Much data comes in tabular format. The `table()` and `records()` functions help you extract it in convenient ways...either as a list of lists, or as a list of dictionaries.

```
>>> text = """
...      name  age  strengths
...      ----  ---  ---------------
...      Joe   12   woodworking
...      Jill  12   slingshot
...      Meg   13   snark, snapchat
... """

>>> table(text)
[['name', 'age', 'strengths'],
 ['Joe', 12, 'woodworking'],
 ['Jill', 12, 'slingshot'],
 ['Meg', 13, 'snark, snapchat']]

>>> records(text)
[{'name': 'Joe', 'age': 12, 'strengths': 'woodworking'},
 {'name': 'Jill', 'age': 12, 'strengths': 'slingshot'},
 {'name': 'Meg', 'age': 13, 'strengths': 'snark, snapchat'}]
```

The `table()` function returns a list of lists, while the `records()` function uses the table header as keys and returns a list of dictionaries.

`table()` and `records()` work even if you have a lot of extra fluff:

```
>>> fancy = """
... +------+-----+-----------------+
... | name | age | strengths       |
... +------+-----+-----------------+
... | Joe  |  12 | woodworking     |
... | Jill |  12 | slingshot       |
... | Meg  |  13 | snark, snapchat |
```

(continues on next page)

(continued from previous page)

```
... +------+-----+-----------------+
... """
>>> assert table(text) == table(fancy)
>>> assert records(text) == records(fancy)
```

The parsing algorithm is heuristic, but it's a good heuristic. It works well with tables formatted in a wide variety of conventional ways including Markdown, RST, ANSI/Unicode line drawing characters, plain text columns and borders, ... See the table tests for *dozens* of samples of formats that work.

What constitutes table columns are contiguous bits of text, without intervening whitespace. Typographic "rivers" of whitespace define column breaks. For this reason, it's recommended that every table column have a separator line, consisting of −, =, or Unicode box drawing characters, to control column width.

```
>>> ma_text = """
...     id  art          source
...     133 Kempo Karate Japan
...     201 Judo         Japan
...     217 BJJ          Brazil via Japan
...     322 Wushu        China
... """

>>> table(ma_text)
[['id', 'art', '', 'source'],
 [133, 'Kempo', 'Karate', 'Japan'],
 [201, 'Judo', '', 'Japan'],
 [217, 'BJJ', '', 'Brazil via Japan'],
 [322, 'Wushu', '', 'China']]
```

Not so good! There is that unfortunate extra assumed column with no name and only the word `'Karate'`. That's because there is a river of space right before the word, and no unambiguous clues that should not be a real column. (We don't assume or insist that all tables will have titles for each column.) To fix, just add a clear definition of where the columns should go:

```
>>> ma_text2 = """
...     id  art          source
...     --  ------------ ----------------
...     133 Kempo Karate Japan
...     201 Judo         Japan
...     217 BJJ          Brazil via Japan
...     322 Wushu        China
... """

>>> table(ma_text2)
[['id', 'art', 'source'],
 [133, 'Kempo Karate', 'Japan'],
 [201, 'Judo', 'Japan'],
 [217, 'BJJ', 'Brazil via Japan'],
 [322, 'Wushu', 'China']]
```

If there are # characters in your table data, best to call the routines with the keyword argument `cstrip=False` so that they will not be erroneously interpreted as comments.

5.1 Headers

The header or column titles for a table can be provided in the table itself, or via the `header` keyword arg. If a string is provided, it will be split using the `words` function. If a list, that list will be exclusively used. In general, it's just as good to provide the headers in the provided text. Note, a header given explicitly is prepended to the data rows; if both explicit and embedded headers are provided, both will appear in the resulting table.

5.2 Records and Keys

Records depends on there being a header row available.

Many tables use natural language headers, such as `First Name` and `Item Price`. When retrieving records (dicts), this is not impossible, but it's often also not entirely convenient–especially for attribute-accessible dictionary keys. So `records()` provides a `keyclean` feature that passes each key through a cleanup function. By default whitespace at the start and end of the key are removed, multiple interior whitespace characters are collapsed and replaced with underscore characters (_).

You can provide your own custom keyclean function if you like, or `None` if you like your keys as-is.

CHAPTER 6

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 (or other text that includes the hashmark / number sign character), set `cstrip=False` (though that is probably more useful with the `lines` and `text` APIs than for `words`).

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 2 source files, care is warranted.

Best advice is: It's time to upgrade already! Python 3 is lovely and ever-improving. Python 2 is now showing its age.

If you do need to continue supporting Python 2, either make sure your literal strings are marked with a "u" prefix: `u""`. To turn Unicode literal processing on by default.

You can explicitly mark strings as unicode in Python 3.3 and following, though it's only necessary if you're maintaing backwards portability, since Python 3 strings are by default Unicode strings.

It can also be helpful (amd in Python 2, often strictly necessary) to declare your source encoding by putting a specially-formatted PEP 263 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 data embedded into source code, but there's no reason text coming from a file, a generator, or other sources can't enjoy the module's text cleanups and lightweight parsing.

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

CHAPTER 9

API

textdata.**lines**(*source*, *noblanks=True*, *dedent=True*, *lstrip=False*, *rstrip=True*, *expandtabs=False*, *cstrip=True*, *join=False*)

Grab lines from a string. Discard initial and final lines if blank.

Parameters

- **source** (*str|lines*) – Text (or list of text lines) to be processed
- **dedent** (*bool*) – a common prefix should be stripped from each line (default *True*)
- **noblanks** (*bool*) – allow no blank lines at all (default *True*)
- **lstrip** (*bool*) – all left space be stripped from each line (default *False*); dedent and lstrip are mutually exclusive
- **rstrip** (*bool*) – all right space be stripped from each line (default *True*)
- **expandtabs** (*Union[bool,int]*) – should all tabs be expanded? if int, by how much?
- **cstrip** (*bool*) – strips comment strings from # to end of each line (like Python itself)
- **join** (*bool|str*) – if False, no effect; otherwise a string used to join the lines

Returns a list of strings

Return type list

textdata.**text**(*source*, ***kwargs*)

Like `lines()`, but returns result as unified text. Useful primarily because of the nice cleanups `lines()` does.

Parameters

- **source** (*str|lines*) – Text (or list of text lines) to be processed
- **join** (*str*) – String to join lines with. Typically newline for line-oriented text but change to " " for a single continous line.

Returns the cleaned string

Return type str

textdata.**textline**(*source*, *cstrip=True*)

Like `text()`, but returns result as unified string that is not line-oriented. Really a special case of `text()`

Parameters

- **source** (*str|list*) –
- **cstrip** (*bool*) – Should comments be stripped? (default: `True`)

Returns the cleaned string

Return type str

textdata.**words**(*source*, *cstrip=True*, *sep=None*)

Returns a sequence of words, like qw() in Perl. Similar to s.split(), except that it respects quoted spans for the occasional word (really, phrase) with spaces included.) If the `sep` argument is provided, words are split on that boundary (rather like `str.split()`). Either the standard space and possibly-quoted word behavior should be used, or the explicit separator. They don't cooperate well.

Like `lines`, removes comment strings by default.

Parameters

- **source** (*str|list*) – Text (or list of text lines) to gather words from
- **cstrip** (*bool*) – Should comments be stripped? (default: `True`)
- **sep** (*Optional[str]*) – Optional explicit separator.

Returns list of words/phrases

Return type list

textdata.**paras**(*source*, *keep_blanks=False*, *join=False*, *cstrip=True*)

Given a string or list of text lines, return a list of lists where each sub list is a paragraph (list of non-blank lines). If the source is a string, use `lines` to split into lines. Optionally can also keep the runs of blanks, and/or join the lines in each paragraph with a desired separator (likely a newline if you want to preserve multi-line structure in the resulting string, or " " if you don't). Like `words`, `lines`, and `textlines`, will also strip comments by default.

Parameters

- **source** (*str|list*) – Text (or list of text lines) from which paras are to be gathered
- **keep_blanks** – Should internal blank lines be retained (default: `False`)
- **join** (*bool|str*) – Should paras be joined into a string? (default: `False`).
- **cstrip** (*bool*) – Should comments be stripped? (default: `True`)

Returns list of strings (each a paragraph)

Return type list

textdata.**attrs**(*source*, *evaluate='natural'*, *dict=<type 'dict'>*, *cstrip=True*)

Parse attribute strings into a dict (or other mapping type). By default evaluates literals as natural to Python, e.g. turning what looks like numbers into into real `int` and `float` instances, not just strings). Quoted values are always treated as strings, never evaluated.

Parameters

- **source** (*Union[str, List[str]]*) – Text to parse (as string or list of lines)
- **evaluate** (*Union[str, bool]*) – How to evaluate resulting values
- **dict** (*type*) – Type of mapping to return

- **cstrip** (*bool*) – Remove comments from string before interpretation?
- **astyle** – Deprecated. Use `dict` parameter instead.
- **literal** – Deprecated. Use `evaluate` parameter instead.

Returns dict (or given dict type)

class `textdata.`**Dict**(**args*, ***kwargs*)

Attribute-accessible `dict` subclass. Does whatever `dict` does, but its keys accessible via .attribute notation. Provided as a convenience. In future, will use the inherently ordered items.Item instead. It is more robust and complete, though only supporting Python 2 at the moment. But if you're on Python 3, `Items` recommended over `Dict`.

`textdata.`**table**(*source*, *header=None*, *evaluate=True*, *cstrip=True*)

Return a list of lists representing a table.

Parameters

- **source** (*Union[str, List[str]]*) – Text to parse (as string or list of lines)
- **header** (*Union[str, List, None]*) – Header for the table
- **evaluate** (*Union[str, function, None]*) – Indicates how to post-process table cells. By default, True or "natural" means as Python literals. Other options are False or 'minimal' (just string trimming), or None or 'none'. Can also provide a custom function.
- **cstrip** (*bool*) – strip comments?

Returns List of lists, where each inner list represents a row.

`textdata.`**records**(*source*, *dict=<class 'textdata.attrs.Dict'>*, *keyclean=<function keyclean>*, ***kwargs*)

Alternate table parser. Renders not a list of lists, but a list of attribute-accessible Dict (dict subclasses).

Parameters

- **source** (*Union[str, List[str]]*) – Text to parse (as string or list of lines)
- **dict** (*type*) – dictionary subtype in which to return results
- **keyclean** (*Union[Function, None]*) – function to clean table headers into more suitable dictionary keys
- ****kwargs** – All other kw args passed to textdata.table

Returns list of dictionaries, one per non-header row

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.
- Thanks to Travis CI, successfully packaged for, and tested against, all late-model versions of Python: 2.7, 3.3, 3.4, 3.5, 3.6, and 3.7, as well as recent versions of PyPy and PyPy3.
- 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
```

You make need to use a specific `pip2` or `pip3` to target a given version of Python, and on some platforms, you'll need to prefix the above command with `sudo` to authorize installation.

In environments without super-user privileges, `pip`'s `--user` option helps install only for a single user, rather than system-wide.

If your `pip` programs don't seem well configured for the version of Python you want, install directly:

```
python3.6 -m pip install -U textdata
```

11.1 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 py37        # run for a specific version only (e.g. py27, py36)
tox -c toxcov.ini  # run full coverage tests
```

CHAPTER 12

Change Log

2.4.1 (January 23, 2019)

Fixed error in `table()` parsing heuristic. Added tests. Tweaked docs.

2.4.0 (December 21, 2018)

Added explicit separator mode to `words()`.

2.3.2 (September 20, 2018)

Removed `print()` inadvertently added in last release.

2.3.0 (September 15, 2018)

Changed tab handling behavior. Previously used `str.expandtabs()` uniformly. While useful for dedent (removing common line indentation), could obscure important internal tabs. Default behavior is now to NOT expandtabs unless explicitly requested. However, leading tabs are still expanded for dedent processing.

2.2.0 (July 7, 2018)

Reorganized code. Tidied and improved comments.

Improved key cleaning for `records()`

Added `full` evaluation mode.

Strengthened table evaluations.

Improved tests and docs.

Dropped deprecated `astype` and `literal` parameters to `attrs()`.

Drops support for Python 2.6. Mainstream support ended 5 years ago. Upgrade already!

2.1.0 (July 4, 2018)

Removed debugging statement inadverntaently left in code.

Improve documentation, esp. for APIs.

Enable `attrs`, `table`, and `records` to take the same string or sequence of lines input as the outher routines.

Cleaned up exported names. `OrderedDict` no longer exported as a convenience.

2.0.2 (July 3, 2018)

Documentation tweaks.

2.0.1

Updated for new pypi.org URLs.

Plus other minor tweaks, like tightening tox targets in favor of Travis CI.

2.0.0 (October 30, 2017)

Major release.

Added `table()` and `records()` functions for ingesting tabular and record-oriented data respectively.

Regularized handling of object evaluation, comment stripping, and other attributes.

1.7.3 (October 13, 2017)

Added pyproject.toml for PEP 518 compliance.

Updated testing matrix to accomodate new PyPy3 version on Travis CI.

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.

Index