
quoter Documentation

Release 1.6.8

Jonathan Eunice

May 15, 2017

Contents

1	Introduction	3
2	Discussion	5
3	We Can Do Better	7
3.1	Construction Details	8
4	Cloning and Setting	9
5	Formatting and Encoding	11
6	Shortcuts	13
7	StyleSets	15
7.1	Visiting the Factory	15
8	HTML	17
9	XML	19
10	Named Styles	21
11	Dynamic Quoters	23
12	Markdown	25
13	Joiners	27
14	API Reference	29
15	Notes	33
16	Installation	35
16.1	Testing	35
17	Change Log	37

`quoter` provides a simple, powerful, systematic way to accomplish one of the most common low-level operations in Python programming: combining strings and data objects into other strings. It does so with remarkable intelligence, including for complex textual languages such as HTML and XML.

CHAPTER 1

Introduction

`quoter` provides a simple, powerful, systematic way do accomplish one of the most common low-level operations in Python programming: combing strings and data objects into other strings. For example:

```
from quoter import *

print single('this')      # 'this'
print double('that')     # "that"
print backticks('ls -l') # `ls -l`
print braces('curlycue') # {curlycue}
print braces('curlysue', padding=1)
                             # { curlysue }
```

Cute...but way too simple to be useful, right? Fair enough. Any of those could have been programmed with a simple utility function.

Let's try something more complicated, where the output has to be intelligently based on context. Here's a taste of quoting some HTML content:

```
print html.p("A para", ".focus")
print html.img('.large', src='file.jpg')
print html.br()
print html.comment("content ends here")
```

Yields:

```
<p class='focus'>A para</p>
<img class='large' src='file.jpg'>
<br>
<!-- content ends here -->
```

This goes well beyond “simply wrapping some text with other text.” The output format varies widely, correctly interpreting CSS Selector-based controls, using void/self-closing elements where needed, and using specialized markup such as the comment format when needed. The HTML quoter and its companion XML quoter are competitive in power and simplicity with bespoke markup-generating packages.

A similar generator for Markdown is also newly included, though it's a the “demonstration” rather than “use in production code” stage.

Finally, `quoter` provides a drop-dead simple, highly functional, `join` function:

```
mylist = list("ABCD")
print join(mylist)
print join(mylist, sep=" | ", endcaps=braces)
print join(mylist, sep=" | ", endcaps=braces.but(padding=1))
print and_join(mylist)
print and_join(mylist[:2])
print and_join(mylist[:3])
print and_join(mylist, quoter=double, lastsep=" and ")
```

Yields:

```
A, B, C, D
{A | B | C | D}
{ A | B | C | D }
A and B
A, B, and C
A, B, C, and D
"A", "B", "C" and "D"
```

Which shows a range of separators, separation styles (both Oxford and non-Oxford commas), endcaps, padding, and individual item quoting. I daresay you will not find a more flexible or configurable `join` function *anywhere* else, in any programming language, at any price.

And if you like any particular style of formatting, make it your own:

```
>>> my_join = join.but(sep=" | ", endcaps=braces.but(padding=1))
>>> print my_join(mylist)
{ A | B | C | D }
```

Now you have a convenient specialized formatter to your own specifications.

Programs stringify and quote values all the time. They wrap both native strings and the string representation of other values in all manner of surrounding text. Single quotes. Double quotes. Curly quotes. Backticks. Separating whitespace. Unicode symbols. HTML or XML markup. *Et cetera*.

There are a *lot* of ways to do this text formatting and wrapping. For example:

```
value = 'something'
print '{x}'.replace('x', value)           # {something}
print "{0}".format(value)                # 'value'
print "" + value + ""                    # 'value'
print "{0}{1}{2}".format('', value, '')  # "value"
print ''.join(['', value, ''])           # "value"
```

But for such a simple, common task as wrapping values in surrounding text, these look pretty ugly, low-level, and dense. Writing them out, it's easy to mistype a character here or there, or to forget some of the gotchas. Say you're formatting values, some of which are strings, but others are integers or other primitive types. Instant `TypeError`! Only strings can be directly concatenated with strings in Python.

The repetitive, *ad hoc* nature of textual quoting and wrapping is tiresome and error-prone. It's never more so than when constructing multi-level quoted strings, such as Unix command line arguments, SQL commands, or HTML attributes.

`quoter` provides a clean, consistent, higher-level alternative. It also provides a mechanism to pre-define your own quoting styles that can then be easily reused.

We Can Do Better

Unlike native Python concatenation operators, `quoter` isn't flustered if you give it non-string data. It knows you want a string output, so it auto-stringifies non-string values:

```
assert brackets(12) == '[12]'  
assert braces(4.4) == '{4.4}'  
assert double(None) == '"None"'  
assert single(False) == "'False'"
```

The module pre-defines callable `Quoters` for a handful of the most common quoting styles:

- `braces` {example}
- `brackets` [example]
- `angles` <example>
- `parens` (example)
- `double` “example”
- `single` ‘example’
- `backticks` `example`
- `anglequote` «example»
- `curlysingle` ‘example’
- `curlydouble` “example”

But there are a *huge* number of ways you might want to wrap or quote text. Even considering just “quotation marks,” there are *well over a dozen*. There are also *numerous bracketing symbols in common use*. That’s to say nothing of the constructs seen in markup, programming, and templating languages. So `quoter` couldn’t possibly provide a default option for every possible quoting style. Instead, it provides a general-purpose mechanism for defining your own:

```
from quoter import Quoter  
  
bars = Quoter('| |')
```

```
print bars('x')           # |x|

plus = Quoter('+', '')
print plus('x')          # +x

para = Quoter('<p>', '</p>')
print para('this is a paragraph') # <p>this is a paragraph</p>
                                   # NB simple text quoting - see below
                                   # for higher-end HTML handling

variable = Quoter('${', '}')
print variable('x')      # ${x}
```

Note that `bars` is specified with just one symbol. If only one is given, the prefix and suffix are considered to be identical. If you really only want a prefix or a suffix, and not both, then instantiate the `Quoter` with two, one of which is an empty string, as in `plus` above.

In most cases, it's cleaner and more efficient to define a style, but there's nothing preventing you from an on-the-fly usage:

```
print Quoter('+[ ', ' ]+')('castle') # +[ castle ]+
```

Construction Details

The examples above generally use a flag argument style of construction. Note, however, that `Quoter` is converting these into respective `prefix` and `suffix` values. If you prefer, you can simply state the prefix and or suffix as direct kwargs:

```
vars = Quoter(prefix='${', suffix='}')
print vars('y')           # ${y}
```

And for the very common cases where quotes are paired, equal-length strings, those can be specified with the `pair` kwarg:

```
onetwo = Quoter(pair="1221")
print onetwo('this')     # 12this21
```

Cloning and Setting

Quoter parameters can be changed (set) in real time.:

```
bars = Quoter('|')
print bars('x')           # |x|
bars.set(prefix='||', suffix='||')
print bars('x')           # ||x||
bars.set(padding=1)
print bars('x')           # || x ||
```

And Quoter instances you like can be cloned, optionally with several options changed in the clone:

```
bart = bars.clone(prefix=']', suffix='[')
assert bart('x') == '] x ['
```

The method `but` is a synonym for `clone`. It is used to suggest “I like everything there, but...change this and that.”:

```
bartwide = bart.but(margin=2)
assert bartwide('x') == ' ] x [ '
```

Note that if any of the options for `bart` besides `margin` change, those changes will be reflected in `bartwide` as well. `bartwide` has decided what its own margins will be, but delegated all other choices to its parent object.

Formatting and Encoding

The Devil, as they say, is in the details. We often don't just want quote marks wrapped around values. We also want those values set apart from the rest of the text. `quoter` supports this with `padding` and `margin` settings patterned on the [CSS box model](#). In CSS, moving out from content one finds padding, a border, and then a margin. Padding can be thought of as an internal margin, and the prefix and suffix strings like the border. With that in mind:

```
print braces('this')                # '{this}'
print braces('this', padding=1)     # '{ this }'
print braces('this', margin=1)     # ' {this} '
print braces('this', padding=1, margin=1) # ' { this } '
```

If desired, the `padding` and `margin` can be given explicitly, as strings. If given as integers, they are interpreted as a number of spaces.

One can also define the `encoding` used for each call, per instance, or globally. If some of your quote symbols use Unicode characters, yet your output medium doesn't support them directly, this is an easy fix. E.g.:

```
Quoter.options.encoding = 'utf-8'
print curlydouble('something something')
```

Now `curlydouble` will output UTF-8 bytes. But in general, this is not a great idea; you should work in Unicode strings in Python, encoding or decoding only at the time of input and output, not as each piece of content is constructed.

Shortcuts

One often sees very long function calls and expressions as text parts are being assembled. In order to reduce this problem, `quoter` defines aliases for single, double, and triple quoting, as well as backticks, and double backticks:

```
from quoter import qs, qd, qt, qb, qdb

print qs('one'), qd('two'), qt('three'), qb('and'), qdb('four')
# 'one' "two" ""three"" `and` ``four``
```

You can, of course, define your own aliases as well, and/or redefine existing styles. If, for example, you like braces but wish it added a padding space by default, it's simple to redefine:

```
sbraces = Quoter('{', '}', padding=1)
print sbraces('braces plus spaces!') # '{ braces plus spaces! }'
```

You could alternatively riff off of the existing braces:

```
sbraces = braces.but(padding=1)
```

You could still get the no-padding variation with:

```
print braces('no space braces', padding=0) # '{no space braces}'
```


As an organizational assist, quoters are available as named attributes of a pre-defined `quote` object. For those who like strict, minimalist imports, this permits `from quoter import quote` without loss of generality. For example:

```
from quoter import quote

quote.double('test')    # "test"
quote.braces('test')   # {test}
# ...and so on...
```

`quote` is a `StyleSet`—a group of related named quoters (i.e. “quoting styles”) conveniently packaged through attributes of a single object.

Visiting the Factory

Each `StyleSet` has a factory function for creating new styles; in the case of `quote` the factory is the `Quoter` class. You can use the `_define` method if you like to create new members:

```
colon = quote._define('colon', ':')
assert colon('this') == quote.colon('this') == ':this:'
```

The assignment to a standalone name `colon` here is optional; you could just always refer to `quote.colon` after the definition if you wish.

You may even call a `StyleSet` in immediate mode:

```
print quote("super")    # "'super'"
```

To define your own set of named styles:

```
cq = StyleSet(factory=Quoter,
              immediate=Quoter(':'))

cq._define("two", Quoter('::'))
```

Now:

```
print cq('this')          # 'this:'  
print cq.two('this')     # 'this::'
```

Quoting does not need to be a simple matter of string concatenation. It can involve sophisticated on-the-fly decisions based on content and context.

For example, there is an extended quoting mode designed for XML and HTML construction. Instead of prefix and suffix strings, `XMLQuoter` and `HTMLQuoter` classes build valid HTML out of tag names and “CSS selector” style specifications (similar to those used by `jQuery`). This is a considerable help in Python, which defines and/or reserves some of the attribute names most used in HTML (e.g. `class` and `id`). Using the CSS selector style neatly gets around this annoyance—and is more compact and more consistent with modern web development idioms to boot.:

```
from quoter import *

print html.p('this is great!', {'class':'emphatic'})
print html.p('this is great!', '.spastic')
print html.p('First para!', '#first')
```

Yields:

```
<p class='emphatic'>this is great!</p> <p class='spastic'>this is great!</p> <p id='first'>First para!</p>
```

Note that the order in which attributes appear is not guaranteed. They’re stored in `dict` objects, which have different orderings on different versions of Python. This generally isn’t a problem, in that ordering isn’t significant in HTML. It can, however, make string-based testing more annoying.

The following CSS selectors are understood:

CSS Spec	Result X/HTML
tag	<tag>
#ident	id="ident"
.classname	class="classname"
[key=value]	key="value"

Note that with the exception of tagnames and ids, multiple setters are possible in the same CSS spec. So `p#one.main.special[lang=en]` defines `<p id='one' class='main special' lang='en'>`.

HTML quoting also understands that some elements are “void” or “self-closing,” meaning they do not need closing tags (and in some cases, not even content). So for example:

```
>>> print html.br()
<br>

>>> print html.img('.big', src='afile')
<img class='big' src='afile'>
```

The `html` object for `HTMLQuoter` (or corresponding `xml` for `XMLQuoter`) is a convenient front-end that can be immediately used to provide simple markup language construction. (It's actually a `StyleSet` that knows how to create new styles on-the-fly.)

You can also access the underlying classes directly, and/or define your own customized quoters. Your own quoters can be called as a function would be. Or, if you give them a name, they can be called through the `html` front-end, just like the pre-defined tags. For instance:

```
para_e = html._define('para_e', 'p.emphatic')
print para_e('this is great!')
print html.para_e('this is great?', '.question')
print html.img('.large', src='somefile')
print html.br()
```

Yields:

```
<p class='emphatic'>this is great!</p>
<p class='question emphatic'>this is great?</p>
<img class='large' src='somefile'>
<br>
```

`HTMLQuoter` quotes attributes by default with single quotes. If you prefer double quotes, you may set them when the element is defined:

```
div = HTMLQuoter('div', attquote=double)
```

Note: Some output may show HTML and XML elements in a different order than described in the documentation. This is because Python `dict` data structures in which keyword arguments are stored are expressly unordered. In practice, their order is implementation dependent, and varies based on whether you're running on Python 2, Python 3, or PyPy. `quoter` always produces correct output, but the ordering may be subtly different from the order suggested by the source code. If this variance bothers you, please join me in lobbying for dictionary ordering (`OrderedDict`) to become the standard behavior for `kwargs` in future versions of Python.

XMLQuoter with its `xml` front-end is a similar quoter with markup intelligence. It offers one additional attribute beyond HTMLQuoter: `ns` for namespaces. Thus:

```
item = xml._define("item inv_item", tag='item', ns='inv')
print item('an item')
print xml.item('another')
print xml.inv_item('yet another')
print xml.thing('something')
print xml.special('else entirely', '#unique')
```

yields:

```
<inv:item>an item</inv:item>
<inv:item>another</inv:item>
<inv:item>yet another</inv:item>
<thing>something</thing>
<special id='unique'>else entirely</special>
```

Note: `item` was given two names. Multiple aliases are supported. While the `item` object carries its namespace specification through its different invocations, the calls to non-`item` quoters have no persistent namespace. Finally, that the CSS specification language heavily used in HTML is present and available for XML, though its use may be less common.

In general, `xml.tagname` auto-generates quoters just like `html.tagname` does on first use. There are also pre-defined utility methods such as `html.comment()` and `xml.comment()` for commenting purposes.

CHAPTER 10

Named Styles

Quoting via the functional API or the attribute-accessed front-ends (`quote`, `lambdaq`, `html`, and `xml`) is probably the easiest way to go. But there's one more way. If you provide the name of a defined style via the `style` attribute, that's the style you get. So while `quote('something')` gives you single quotes by default (`'something'`), if you invoke it as `quote('something', style='double')`, you get double quoting as though you had used `quote.double(...)`, `double(...)`, or `qd(...)`. This even works through named front-ends; `quote.braces('something', style='double')` still gets you `"something"`. If you don't want to be confused by such double-bucky forms, don't use them. The best use-case for named styles is probably when you don't know how something will be quoted (or what tag it will use, in the HTML or XML case), but that decision is made dynamically. Then `style=desired_style` makes good sense.

Style names are stored in the class of the quoter. So all `Quoter` instances share the same named styles, as do `HTMLQuoter`, `XMLQuoter`, and `LambdaQuoter`.

Dynamic Quoters

`XMLQuoter` and `HTMLQuoter` show that it's straightforward to define `Quoters` that don't just concatenate text, but that examine it and provide dynamic rewriting on the fly.

`LambdaQuoter` is a further generalization of this idea. It allows generic formatting to be done by a user-provided function. For example, in finance, one often wants to present numbers with a special formatting:

```
from quoter import *

f = lambda v: ('(', abs(v), ')') if v < 0 else ('', v, '')
financial = LambdaQuoter(f)
print financial(-3)           # (3)
print financial(45)          # 45

password = LambdaQuoter(lambda v: ('', 'x' * len(v), ''))
print password('secret!')    # xxxxxxxx

wf = lambda v: ('**', v, '**') if v < 0 else ('', v, '')
warning = lambdaq._define("warning", wf)
print warning(12)            # 12
print warning(-99)          # **-99**
```

The trick is instantiating `LambdaQuoter` with a callable (e.g. lambda expression or even a full function) that accepts one value and returns a tuple of three values: the quote prefix, the value (possibly rewritten), and the suffix. The rewriting mechanism can be entirely general, doing truncation, column padding, content obscuring, hashing, or...just anything.

`LambdaQuoter` named instances are accessed through the `lambdaq` front-end (because `lambda` is a reserved word). Given the code above, `lambdaq.warning` is active, for example.

`LambdaQuoter` shows how general a formatting function can be made into a `Quoter`. That has the virtue of providing a consistent mechanism for tactical output transformation with built-in margin and padding support. It's also able to encapsulate complex quoting / representation decisions that would otherwise muck up "business logic," making representation code much more unit-testable. But, one might argue that such full transformations are "a bridge too far" for a quoting module. So use this dynamic component, or not, as you see fit.

An experimental Markdown formatter has been added. It is quite simple at present, supporting both span:

Function	Markdown Span
<code>md.i</code>	<i>*italics*</i>
<code>md.b</code>	**bold**
<code>md.a</code>	anchor, aka link

and some block functions:

Function	Markdown Block
<code>md.h</code>	heading
<code>md.h1</code>	heading level 1
<code>md.h2</code>	heading level 2
...	...
<code>md.h6</code>	heading level 6
<code>md.p</code>	paragraph
<code>md.hr</code>	horizontal rule
<code>md.doc</code>	document

All functions are accessed through the `md` style set.

List, image, blockquote, and code-block formatting are next steps. At this demonstration stage, the goal is to stretch the `quoter` use-case and prove/harden its extension mechanisms, which it is already doing. A much more extensive block-oriented quoting mechanism is in the works to flesh out Markdown construction. Stay tuned for more extensive functions and documentation.

Joiner is a type of Quoter that combines sequences. The simplest invocation `join(mylist)` is identical to `' , ' .join(mylist)`. But of course it doesn't stop there. The `sep` parameter determines what string is placed between each list item. But the separator need not be uniform. For the common (and linguistically important) case where there are two items in list, the `twosep` parameter provides an alternate value. The final separator can be defined via the `lastsep` parameter, permitting proper [Oxford commas](#), or if you prefer, a non-Oxford heathen style. The standard `prefix`, `suffix`, `margin` and `padding` parameters are available. Finally, individual sequence items can be formatted (`quoter`) and the entire “core” of joined material can be wrapped by an `endcap` quoter.

Some examples:

```
mylist = list("ABCD")
print join(mylist)
print join(mylist, sep=" | ", endcaps=braces)
print join(mylist, sep=" | ", endcaps=braces.but(padding=1))
print and_join(mylist)
print and_join(mylist[:2])
print and_join(mylist[:3])
print and_join(mylist, quoter=double, lastsep=" and ")
```

Yields:

```
A, B, C, D
{A | B | C | D}
{ A | B | C | D }
A and B
A, B, and C
A, B, C, and D
"A", "B", "C" and "D"
```

It's a bit of a historical accident that both the `prefix/suffix` pair and `endcap` are available, as they accomplish the same goal. If an `endcap` quoter is used, note that any desired padding (spaces inside the endcaps) must be provided by the endcapper, as it operates earlier than, and in conflict with, the application of normal padding. E.g.:

```
print join(mylist, sep=" | ", endcaps=braces.but(padding=1))
print join(mylist, sep=" | ", prefix="{", suffix="}", padding=1)
```

Do the same thing. But mixing and matching the two styles may not give you what you wanted.

Various defined `Joiner` objects may be of use:: `and_join`, `or_join`, `joinlines`, and `concat`.

A start on a more complete, method-by-method reference:

class `quoter.Quoter` (*args, **kwargs)

A quote style. Instantiate it with the style information. Call it with a value to quote the value.

`__call__` (*args, **kwargs)

Quote the value, according to the current options.

`__init__` (*args, **kwargs)

Create a quoting style.

but (**kwargs)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

clone (**kwargs)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

options = `Options(suffix=None, sep=' ', encoding=None, padding=0, prefix=None, pair=Transient, margin=0)`

set (*args, **kwargs)

Change the receiver's settings to those defined in the `kwargs`. An update-like function. This uplevels calls that would look like `Class.options.set(...)` to the simpler `Class.set(...)`. Works on either class or instance receivers. Requires that one uses the instance variable `options` to store persistent configuration data.

settings (**kwargs)

Open a context manager for a *with* statement. Temporarily change settings for the duration of the *with*.

class `quoter.LambdaQuoter` (*args, **kwargs)

A Quoter that uses code to decide what quotes to use, based on the value.

`__call__` (value, **kwargs)

Quote the value, based on the instance's function.

`__init__` (*args, **kwargs)

Create a quoting style.

but (***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

clone (***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

options = **Options**(`suffix=Prohibited`, `sep=''`, `encoding=None`, `padding=0`, `prefix=Prohibited`, `func=None`, `pair=Prohibited`)

set (**args*, ***kwargs*)

Change the receiver's settings to those defined in the `kwargs`. An update-like function. This uplevels calls that would look like `Class.options.set(...)` to the simpler `Class.set(...)`. Works on either class or instance receivers. Requires that one uses the instance variable `options` to store persistent configuration data.

settings (***kwargs*)

Open a context manager for a *with* statement. Temporarily change settings for the duration of the *with*.

class `quoter.XMLQuoter` (**args*, ***kwargs*)

A more sophisticated quoter for XML elements that manages tags, namespaces, and the idea that some elements may not have contents.

`__call__` (**args*, ***kwargs*)

Quote a value in X/HTML style, with optional attributes.

`__init__` (**args*, ***kwargs*)

Create an XMLQuoter

but (***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

clone (***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

options = **Options**(`atts={}`, `suffix=Prohibited`, `sep=''`, `void=False`, `encoding=None`, `attquote=Quoter(suffix='\"'`, `sep=''`, `e`)

set (**args*, ***kwargs*)

Change the receiver's settings to those defined in the `kwargs`. An update-like function. This uplevels calls that would look like `Class.options.set(...)` to the simpler `Class.set(...)`. Works on either class or instance receivers. Requires that one uses the instance variable `options` to store persistent configuration data.

settings (***kwargs*)

Open a context manager for a *with* statement. Temporarily change settings for the duration of the *with*.

class `quoter.HTMLQuoter` (**args*, ***kwargs*)

A more sophisticated quoter that supports attributes and void elements for HTML.

`__call__` (**args*, ***kwargs*)

Quote a value in X/HTML style, with optional attributes.

`__init__` (**args*, ***kwargs*)

but (***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to `self.__class__.options`). `kwargs` become the cloned instance's overlay options.

clone (**kwargs)

Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

options = Options(atts={}, suffix=Prohibited, sep=',', void=False, encoding=None, attquote=Quoter(suffix='\"', sep=',', e

set (*args, **kwargs)

Change the receiver's settings to those defined in the kwargs. An update-like function. This uplevels calls that would look like Class.options.set(...) to the simpler Class.set(...). Works on either class or instance receivers. Requires that one uses the instance variable options to store persistent configuration data.

settings (**kwargs)

Open a context manager for a *with* statement. Temporarily change settings for the duration of the with.

class quoter.MDQuoter (*args, **kwargs)

A more sophisticated quoter for Markdown elements.

__call__ (*args, **kwargs)

Quote the value, according to the current options.

__init__ (*args, **kwargs)

Create an MDQuoter

a (text, href, **kwargs)

but (**kwargs)

Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

clone (**kwargs)

Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

doc (seq, **kwargs)

h (text, level=1, close=False, setext=False, **kwargs)

Headers at various levels. Either atx style (hashmark prefix) by default, or Setext (underlining) style optionally.

h1 (text, **kwargs)

h2 (text, **kwargs)

h3 (text, **kwargs)

h4 (text, **kwargs)

h5 (text, **kwargs)

h6 (text, **kwargs)

hr (**kwargs)

options = Options(suffix=None, sep=',', encoding=None, misc=Prohibited, padding=0, prefix=None, pair=Transient, ma

p (*args, **kwargs)

set (*args, **kwargs)

Change the receiver's settings to those defined in the kwargs. An update-like function. This uplevels calls that would look like Class.options.set(...) to the simpler Class.set(...). Works on either class or instance receivers. Requires that one uses the instance variable options to store persistent configuration data.

settings (***kwargs*)

Open a context manager for a *with* statement. Temporarily change settings for the duration of the with.

`quoter.quote` Default “StyleSet” for “Quoter” objects

Container for named styles.

`quoter.lambdaq` Default “StyleSet” for “LambdaQuoter” objects

Container for named styles.

`quoter.xml` Default “StyleSet” for “XMLQuoter” objects

Container for named styles.

`quoter.html` Default “StyleSet” for “HTMLQuoter” objects

Container for named styles.

`quoter.md` Default “StyleSet” for “Markdown” objects

Container for named styles.

- `quoter` provides simple transformations that could be alternatively implemented as a series of small functions. The problem is that such “little functions” tend to be constantly re-implemented, in different ways, and spread through many programs. That need to constantly re-implement such common tasks has led me to re-think how software should construct text on a grander scale. `quoter` is one facet of a project to systematize higher-level formatting operations. See `say` and `show` for other parts of the larger effort.
- `quoter` is a test case for, and leading user of, `options`, a module that supports flexible option handling. In some ways it is `options` most extensive test case, in terms of subclassing and dealing with named styles.
- In the future, additional quoting styles might appear. There is already (limited, experimental) support for Markdown, and other languages such as RST are straightforward. It’s not hard to subclass `Quoter` for new languages. Some of the things learned in the `say` project about text block management (indentation, wrapping, and such) are highly applicable to the quoting mission.
- You might look at some of the modules for ANSI-coloring text such as `ansicolors` as being special cases of the `quoter` idea. While `quoter` doesn’t provide this specific kind of wrapping, it’s in-line with the mission.
- 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 builds of PyPy and PyPy3.
- The author, Jonathan Eunice or [@jeunice on Twitter](#) welcomes your comments and suggestions.

To install or upgrade to the latest version:

```
pip install -U quoter
```

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

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

(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
```

The provided `tox.ini` and `toxcov.ini` config files do not define a preferred package index / repository. If you want to use them with a specific (presumably local) index, the `-i` option will come in very handy:

```
tox -i INDEX_URL
```


1.6.7 (February 20, 2017)

Updated testing matrix. Now certified under Python 3.6 and latest versions of PyPy and PyPy3.

1.6.6 (September 16, 2015)

Updated testing for Python 3.5.0 final and PyPy 2.6.1 (based on CPython 2.7.10).

1.6.5 (September 8, 2015)

Minor fix to Markdown module. Turns out HTML stops at h6, not h7. Oops!

1.6.4 (September 7, 2015)

Quoters can now take a variable number of arguments. A `sep` kwarg determines how multiple pieces are combined. Defaults to empty string, so rank concatenation. Full rationalization re `Joiner` awaits another day.

Minor testing cleanups.

1.6.3 (September 7, 2015)

Handful of minor cleanups.

All previously skipped tests are now re-enabled and passing.

Argument handling of `Quoter` creation is now hardened. If a prefix or suffix is explicitly set but not the other, the unset one defaults to an empty string, not to the other one that is set.

The `pair` keyword for compact setting of balanced quote prefix/suffix pairs is returned to service (in earlier incarnations, this was the `chars` kwarg).

Extended documentation. Added API Reference.

1.6.0 (September 7, 2015)

Major redesign of the named styles facility. Refactored style management out of `Quoter` and its subclasses and into separate `StyleSet` class. Much cleaner and more reliable.

The `style=` kwarg to `Quoter` classes, however, goes away. It was “a bridge too far,” complicating everything and not adding real value. Its function is now found in `StyleSet._define()`.

While this is largely an “infrastructure” rather than functional release, there were some minor functional improvements: Markdown quoting extended with `hr` and `h1` through `h7` header methods. XML quoting now has `cdata` and `pdata` methods. Both of these are part of an incremental march toward functional completeness in their respective quoting realms.

Withdrew branch coverage testing metric. While a good idea in theory, current tools do not provide sufficient actionable intelligence to directly identify source of branch misses.

1.5.6 (August 22, 2015)

Extended Markdown feature with header function `h`

1.5.5 (August 24, 2015)

Start of a Markdown capability. It is at a experimental, proof of concept, or demonstration stage rather than production-use stage at present—much as HTML and XML quoting functions were a few releases back. The initial goal is to stretch the `quoter` use- case and prove/harden its extension mechanisms. Even at its inaugural stage, it is doing just that.

1.5.3 (August 23, 2015)

Reorg of XML and HTML code into own module. Parallel reorg of tests. All tests passing.

1.5.2 (August 23, 2015)

Starts automatic collection of test branch coverage. Begins with 99% coverage.

1.5.0 (August 20, 2015)

Enables cloning and specialization of all `Quoter` subclasses, including `xml` and `html` objects docs previously warned were non-functional. Refactors majority of `Joiner` functionality as a subclass of `Quoter`, with all the rights and privileges thereunto appertaining.

Extends tests.

1.4.4 (August 20, 2015)

Added “naked” invocations of primary quoter front-ends, such as `html("joe", "b.one")`. This allows quoters to take on primary information (such as the tag name) during each call. This was long planned for the API, but had been sidelined due to now- quashed complexity.

1.4.3 (August 20, 2015)

Added `but` synonym for `clone` to have simpler, clearer invocations.

1.4.2 (August 20, 2015)

Docs were getting long. Truncated `README.rst` and converted majority to Sphinx docs to be hosted on `readthedocs`.

1.4.1 (August 20, 2015)

Documentation fix.

1.4.0 (August 20, 2015)

Major rewrite of complex argument handling for `XMLQuoter` (and `HTMLQuoter` subclass). Should be much more robust and extensible. Refactored some functions into `util.py`

Adds direct attribute setting `[key=value]` to the CSS selector specification language.

Improved testing. Added new tests of sub-cases of parsing the arguments. Updated strategy to have quicker basic tests with `sudo tox`, but easy full coverage analysis with `sudo tox -c toxcov.ini`. Advances test line coverage to 100%.

1.3.6 (August 17, 2015)

Updated testing strategy with integrated multi-version coverage testing. Achieved 98%.

1.3.5 (August 14, 2015)

Fixed RST problem in docs

1.3.4 (August 14, 2015)

Added Travis CI badge and note about 2.6 release to docs

1.3.3 (August 11, 2015)

Some code reorganization and updating of common string routines.

1.3.2 (August 11, 2015)

Initiated external CI with Travis. Dropped Python 2.6 from official support given issues getting it running on Travis (though it actually does work).

1.3.0 (August 11, 2015)

First release of integrated sequence joining. `join`, `word_join`, `and_join`, `or_join`, `joinlines`, and `items` are functional and tested, but still less mature than the rest of the codebase.

1.2.0 (August 11, 2015)

Institutes full named styles within each quoting class. Tests and docs tweaked. Embedded `nulltype` module withdrawn in favor of full `nulltype` imported by `options`.

1.1.4 (August 3, 2015)

Experimental switch to YAML format for change log. Some setup, testing, and config updates.

1.1.3 (August 3, 2015)

Cloning and changing of `Quoter` instances (though not yet `HTMLQuoter` or `XMLQuoter` instances) is now operational.

1.1.0 (August 3, 2015)

Cleans up HTML quoting, esp. re void / self-closing elements. Adds new double-backtick functions. Changed to Apache License 2.0. Updates docs and testing matrix.

1.0.3 (November 1, 2013)

HTML, XML, and lambda quoters now use class-relative styles dictionaries, as opposed to piggybacking the standard `Quoter` styles dictionary.

Improved docs and tests.

Added `lambdaq` front-end parallel to `quote`, `html`, and `xml`.

1.0.2 (October 31, 2013)

Some internal cleanups to improve code reuse among classes. Bumped from Alpha to Beta status.

1.0.1 (October 31, 2013)

A new alternate API consisting of attribute names off of a default quoting object (e.g. `quote.single` as a specialization of `quote`) has been instituted. This is mostly, but not perfectly, a superset of the previous use of a `quote()` function.

The naming infrastructure has been beefed up, with multiple names (aliases) possible for all named objects.

A new `XMLQuoter` is inserted as a superclass of `HTMLQuoter`. It has `HTMLQuoter`'s ability to parse CSS style id and class name definitions (e.g. `'#first.big.special'`), as well as namespace support (new `ns` attribute).

XML and HTML quoters for individual tags are automatically generated upon first use. E.g. `html.b('this')` creates an `HTMLQuoter(tag='b', name='b')` quoter that is cached as `html.b` for subsequent uses.

Updated versioning strategy to comply with [PEP 386](#)

Various other structural and packaging cleanups. E.g. moved into proper Python package; given introspectable version number; removed old `verno` auto-update of version number; this proper change long instituted; etc.

0.308 (October 30, 2012)

Last version before PEP 386 versioning switch. Upgrade away from these old versions if for no other reason than improving the auto- install logic.

Symbols

`__call__()` (quoter.HTMLQuoter method), 30
`__call__()` (quoter.LambdaQuoter method), 29
`__call__()` (quoter.MDQuoter method), 31
`__call__()` (quoter.Quoter method), 29
`__call__()` (quoter.XMLQuoter method), 30
`__init__()` (quoter.HTMLQuoter method), 30
`__init__()` (quoter.LambdaQuoter method), 29
`__init__()` (quoter.MDQuoter method), 31
`__init__()` (quoter.Quoter method), 29
`__init__()` (quoter.XMLQuoter method), 30

A

`a()` (quoter.MDQuoter method), 31

B

`but()` (quoter.HTMLQuoter method), 30
`but()` (quoter.LambdaQuoter method), 30
`but()` (quoter.MDQuoter method), 31
`but()` (quoter.Quoter method), 29
`but()` (quoter.XMLQuoter method), 30

C

`clone()` (quoter.HTMLQuoter method), 30
`clone()` (quoter.LambdaQuoter method), 30
`clone()` (quoter.MDQuoter method), 31
`clone()` (quoter.Quoter method), 29
`clone()` (quoter.XMLQuoter method), 30

D

`doc()` (quoter.MDQuoter method), 31

H

`h()` (quoter.MDQuoter method), 31
`h1()` (quoter.MDQuoter method), 31
`h2()` (quoter.MDQuoter method), 31
`h3()` (quoter.MDQuoter method), 31
`h4()` (quoter.MDQuoter method), 31
`h5()` (quoter.MDQuoter method), 31

`h6()` (quoter.MDQuoter method), 31
`hr()` (quoter.MDQuoter method), 31
`html` (in module quoter), 32
HTMLQuoter (class in quoter), 30

L

`lambdaq` (in module quoter), 32
LambdaQuoter (class in quoter), 29

M

`md` (in module quoter), 32
MDQuoter (class in quoter), 31

O

`options` (quoter.HTMLQuoter attribute), 31
`options` (quoter.LambdaQuoter attribute), 30
`options` (quoter.MDQuoter attribute), 31
`options` (quoter.Quoter attribute), 29
`options` (quoter.XMLQuoter attribute), 30

P

`p()` (quoter.MDQuoter method), 31

Q

`quote` (in module quoter), 32
Quoter (class in quoter), 29

S

`set()` (quoter.HTMLQuoter method), 31
`set()` (quoter.LambdaQuoter method), 30
`set()` (quoter.MDQuoter method), 31
`set()` (quoter.Quoter method), 29
`set()` (quoter.XMLQuoter method), 30
`settings()` (quoter.HTMLQuoter method), 31
`settings()` (quoter.LambdaQuoter method), 30
`settings()` (quoter.MDQuoter method), 31
`settings()` (quoter.Quoter method), 29
`settings()` (quoter.XMLQuoter method), 30

X

xml (in module quoter), 32

XMLQuoter (class in quoter), 30