quoter Documentation

Release 1.6.5

Jonathan Eunice

September 08, 2015

Contents

`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. It does so with remarkable intelligence, including for complex textual langauges 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 "demonsration" 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.

CHAPTER 2

Discussion

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.

CHAPTER 3

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 ]+
```

3.1 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
```

CHAPTER 4

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.

CHAPTER 5

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.

CHAPTER 6

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}'
```

CHAPTER 7

StyleSets

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 is a `StyleSet`–a group of related named quoters (i.e. "quoting styles") conveniently packaged through attributes of a single object.

7.1 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 assignement 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::'
```

CHAPTER 8

HTML

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:

this is great! this is great! First para!

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 that 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.

CHAPTER 9

XML

`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 nave 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 predefined 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`.

CHAPTER 11

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!')       # xxxxxxx

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.

CHAPTER 12

Markdown

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

Function	Markdown Span
`md.i`	*italics*
`md.b`	**bold**
`md.a`	anchor, aka link

and some block functions:

Function	Markdown Block
`md.h`	heading
`md.h1`	heading level 1
`md.h2`	heading level 2
...	...
`md.h6`	heading level 6
`md.p`	paragraph
`md.hr`	horizontal rule
`md.doc`	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.

CHAPTER 13

Joiners

`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`.

CHAPTER 14

API Reference

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=Prohibite

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.

`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.

CHAPTER 15

Notes

- `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, 3.2, 3.3, 3.4, and 3.5 pre-release (3.5.0b3) as well as PyPy 2.6.0 (based on 2.7.9) and PyPy3 2.4.0 (based on 3.2.5).
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions.

CHAPTER 16

Installation

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.)

16.1 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 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
```

Symbols

B

C

H

L

M

O

Q

S

X