options Documentation

Release 1.4.6

Jonathan Eunice

May 15, 2017

Contents

`options` sotres option and configuration data in a clean, high-function way. Changes can "overlay" defaults or earlier settings.

For most code, `options` is flexibility overkill. Not everyone wants to be a world-class gymnast, yogi, or contortionist. For most functions and classes, Python's regular arguments, `*args`, `**kwargs`, and inheritance patterns are elegant and sufficient. `options` is for the top 1% that need:

- extremely functional classes, functions, and methods,
- with many different features and options,
- the settings for which might be adjusted or overriden at any time,
- yet that need "reasonable" or "intelligent" defaults, and
- that yearn for a simple, unobtrusive API.

In those cases, Python's built-in, inheritance-based model stops being the simple approach. Non-trivial argument-management code and complexity begins to pervade. This is where `options`'s layered, delegation-based approach begins to shine. Almost regardless of how varied the options it wrangles, or how much flexibility is required, code complexity remains flat.

Python has very flexible arguments for functions and methods, and good connection of values from classes to sub-classes to methods. It doesn't, however, connect those very well to configuration files, module defaults, method parameters, and other uses. `options`, in contrast, seamlessly connects all of these varied layers and cases.

For more backstory, see this StackOverflow.com discussion of how to combat "configuration sprawl". For examples of `options` in use, see say, quoter, and show.

CHAPTER 1

Usage

In a typical use of `options`, your highly-functional class defines default values. Subclasses can add, remove, or override options. Instances use class defaults, but they can be overridden when each instance is created. For any option an instance doesn't override, the class default "shines through."

So far, this isn't very different from a typical use of Python's standard instance and class variables. The next step is where `options` gets interesting.

Individual method calls can similarly override instance and class defaults. The options stated in each method call obtain only for the duration of the method's execution. If the call doesn't set a value, the instance value applies. If the instance didn't set a value, the class default applies (and so on, to its superclasses, if any).

One step further, Python's `with` statement can be used to set option values for just a specific duration. As soon as the `with` block exists, the option values automagically fall back to what they were before the block. (In general, if any option is unset, its value falls back to what it was in the next higher layer.)

To recap: Python handles class, subclass, and instance settings. `options` handles these as well, but also adds method and transient settings. By default when Python overrides a setting, it's destructive; the value cannot be "unset" without additional code. When a program using `options` overrides a setting, it does so non-destructively, layering the new settings atop the previous ones. When attributes are unset, they immediately fall back to their prior value (at whatever higher level it was last set).

CHAPTER 2

An Example

Because the capability of `options` is designed for high-end, edge-case situations, it's hard to demonstrate its virtues with simple code. But we'll give it a shot.

```
from options import Options, attrs

class Shape(object):

    options = Options(
        name   = None,
        color  = 'white',
        height = 10,
        width  = 10,
    )

    def __init__(self, **kwargs):
        self.options = Shape.options.push(kwargs)

    def draw(self, **kwargs):
        opts = self.options.push(kwargs)
        print attrs(opts)

one = Shape(name='one')
one.draw()
one.draw(color='red')
one.draw(color='green', width=22)
```

yielding:

```
color='white', width=10, name='one', height=10
color='red', width=10, name='one', height=10
color='green', width=22, name='one', height=10
```

So far we could do this with instance variables and standard arguments. It might look a bit like this:

```
class ClassicShape(object):

    def __init__(self, name=None, color='white', height=10, width=10):
        self.name   = name
        self.color  = color
        self.height = height
        self.width  = width
```

but when we got to the `draw` method, things would be quite a bit messier.:

```
def draw(self, **kwargs):
    name   = kwargs.get('name',   self.name)
    color  = kwargs.get('color',  self.color)
    height = kwargs.get('height', self.height)
    width  = kwargs.get('width',  self.width)
    print "color={0!r}, width={1}, name={2!r}, height={3}".format(color, width, name,␣
↪height)
```

One problem here is that we broke apart the values provided to `__init__()` into separate instance variables, now we need to re-assemble them into something unified. And we need to explicitly choose between the `**kwargs` and the instance variables. It gets repetitive, and is not pretty. Another classic alternative, using native keyword arguments, is no better:

```
def draw2(self, name=None, color=None, height=None, width=None):
    name   = name   or self.name
    color  = color  or self.color
    height = height or self.height
    width  = width  or self.width
    print "color={0!r}, width={1}, name={2!r}, height={3}".format(color, width, name,␣
↪height)
```

If we add just a few more instance variables, we've arrived at the Mr. Creosote of class design. For every instance variable that might be overridden in a method call, that method needs one line of code to decide whether the override is, in fact, in effect. And that line will appear in every method call needing to support such overrides. Suddenly, dealing with parameters starts to be a full-time job and responsibility of every method. That's neither elegant nor scalable. Pretty soon we're in "just one more wafer-thin mint..." territory.

But with `options`, it's easy. No matter how many configuration variables there are to be managed, each method needs just one line of code to manage them:

```
opts = self.options.push(kwargs)
```

Changing things works simply and logically:

```
Shape.options.set(color='blue')
one.draw()
one.options.set(color='red')
one.draw(height=100)
one.draw(height=44, color='yellow')
```

yields:

```
color='blue', width=10, name='one', height=10
color='red', width=10, name='one', height=100
color='yellow', width=10, name='one', height=44
```

In one line, we reset the default color for all `Shape` objects. That's visible in the next call to `one.draw()`. Then we

set the instance color of `one` (all other `Shape` instances remain blue). Finally, we call one with a temporary override of the color.

A common pattern makes this even easier:

```
class Shape(OptionsClass):
    ...
```

The `OptionsClass` base class will provide a `set()` method so that you don't need `Shape.options.set()`. `Shape.set()` does the same thing, resulting in an even simpler API. The `set()` method is a "combomethod" that will take either a class or an instance and "do the right thing." `OptionsClass` also provides a `settings()` method to easily handle transient `with` contexts (more on this in a minute), and a `__repr__()` method so that it prints nicely.

The more options and settings a class has, the more unwieldy the class and instance variable approach becomes, and the more desirable the delegation alternative. Inheritance is a great software pattern for many situations–but it's poor pattern for complex option and configuration handling.

For richly-featured APIs, `options`'s delegation pattern is simpler. As the number of options grows, delegation imposes almost no additional coding, complexity, or failure modes. Options are consolidated in one place, providing neat attribute-style access and keeping everything tidy. We can add new options or methods with confidence:

```
def is_tall(self, **kwargs):
    opts = self.options.push(kwargs)
    return opts.height > 100
```

Under the covers, `options` uses a variation on the `ChainMap` data structure (a multi-layer dictionary) to provide option stacking. Every option set is stacked on top of previously set option sets, with lower-level values shining through if they're not set at higher levels. This stacking or overlay model resembles how local and global variables are managed in many programming languages.

This makes advanced use cases, such as temporary value changes, easy:

```
with one.settings(height=200, color='purple'):
    one.draw()
    if is_tall(one):
        ...         # it is, but only within the ``with`` context

if is_tall(one):    # nope, not here!
    ...
```

Note: You will still need to do some housekeeping in your class's `__init__()` method, including creating a new options layer. If you don't wish to inherit from `OptionsClass`, you can manually add `set()` and `settings()` methods to your own class. See the `OptionsClass` source code for details.

As one final feature, consider "magical" parameters. Add the following code to your class description:

```
options.magic(
    height = lambda v, cur: cur.height + int(v) if isinstance(v, str) else v,
    width  = lambda v, cur: cur.width  + int(v) if isinstance(v, str) else v
)
```

Now, in addition to absolute `height` and `width` parameters specified with `int` (integer/numeric) values, your module auto-magically supports relative parameters for `height` and `width` given as string parameters.:

```
one.draw(width='+200')
```

yields:

```
color='blue', width=210, name='one', height=10
```

Neat, huh?

For more backstory, see this StackOverflow.com discussion of how to combat "configuration sprawl". For examples of `options` in use, see say and show.

CHAPTER 3

Design Considerations

`options` is not intended to replace every class's or method's parameter passing mechanisms–just the most highly-optioned ones that multiplex a package's functionality to a range of use cases. These are generally the highest-level, most outward-facing classes, objects, and APIs. They will generally have at least four configuration variables (e.g. kwargs used to create, configure, and define each instance).

In general, classes will define a set of methods that are "outwards facing"–methods called by external code when consuming the class's functionality. Those methods should generally expose their options through `**kwargs`, creating a local variable (say `opts`) that represents the sum of all options in use–the full stack of call, instance, and class options, including any present magical interpretations.

Internal class methods–the sort that are not generally called by external code, and that by Python convention are often prefixed by an underscore (_)–these generally do not need `**kwargs`. They should accept their options as a single variable (say `opts` again) that the externally-facing methods will provide.

For example, if `options` didn't provide the nice formatting function `attrs`, we might have designed our own:

```
def _attrs(self, opts):
    nicekeys = [ k for k in opts.keys() if not k.startswith('_') ]
    return ', '.join([ "{}={}".format(k, repr(opts[k])) for k in nicekeys ])

def draw(self, **kwargs):
    opts = self.options.push(kwargs)
    print self._attrs(opts)
```

`draw()`, being the outward-facing API, accepts general arguments and manages their stacking (by `push``ing ``kwargs` onto the instance options). When the internal `_attrs()` method is called, it is handed a pre-digested `opts` package of options.

A nice side-effect of making this distinction: Whenever you see a method with `**kwargs`, you know it's outward-facing. When you see a method with just `opts`, you know it's internal.

Objects defined with `options` make excellent "callables." Define the `__call__` method, and you have a very nice analog of function calls.

`options` has broad utility, but it's not for every class or module. It best suits high-level front-end APIs that multiplex lots of potential functionality, and wish/need to do it in a clean/simple way. Classes for which the set of instance

variables is small, or functions/methods for which the set of known/possible parameters is limited–these work just fine with classic Python calling conventions. For those, `options` is overkill. "Horses for courses."

CHAPTER 4

Setting and Unsetting

Using `options`, objects often become "entry points" that represent both a set of capabilities and a set of configurations for how that functionality will be used. As a result, you may want to be able to set the object's values directly, without referencing their underlying `options`. It's convenient to add a `set()` method, then use it, as follows:

```
def set(self, **kwargs):
    self.options.set(**kwargs)

one.set(width='*10', color='orange')
one.draw()
```

yields:

```
color='orange', width=100, name='one', height=10
```

`one.set()` is now the equivalent of `one.options.set()`. Or continue using the `options` attribute explicitly, if you prefer that.

Values can also be unset.:

```
from options import Unset

one.set(color=Unset)
one.draw()
```

yields:

```
color='blue', width=100, name='one', height=10
```

Because `'blue'` was the color to which `Shape` had be most recently set. With the color of the instance unset, the color of the class shines through.

Note: While options are ideally accessed with an attribute notion, the preferred of setting options is through method calls: `set()` if accessing directly, or `push()` if stacking values as part of a method call. These perform the in-

terpretation and unsetting magic; straight assignment does not. In the future, `options` may provide an equivalent `__setattr__()` method to allow assignment–but not yet.

CHAPTER 5

Leftovers

`options` expects you to define all feasible and legitimate options at the class level, and to give them reasonable defaults.

None of the initial settings ever have magic applied. Much of the expected interpretation "magic" will be relative settings, and relative settings require a baseline value. The top level is expected and demanded to provide a reasonable baseline.

Any options set "further down" such as when an instance is created or a method called should set keys that were already-defined at the class level.

However, there are cases where "extra" `**kwargs` values may be provided and make sense. Your object might be a very high level entry point, for example, representing very large buckets of functionality, with many options. Some of those options are relevant to the current instance, while others are intended as pass-throughs for lower-level modules/objects. This may seem a doubly rarefied case–and it is, relatively speaking. But it does happen–and when you need multi-level processing, it's really, really super amazingly handy to have it.

`options` supports this in its core `push()` method by taking the values that are known to be part of the class's options, and deleting those from `kwargs`. Any values left over in the `kwargs dict` are either errors, or intended for other recipients.

As yet, there is no automatic check for leftovers.

CHAPTER 6

Magic Parameters

Python's `*args` variable-number of arguments and `**kwargs` keyword arguments are sometimes called "magic" arguments. `options` takes this up a notch, allowing setters much like Python's `property` function or `@property` decorator. This allows arguments to be interpreted on the fly. This is useful, for instance, to provide relative rather than just absolute values. As an example, say that we added this code after `Shape.options` was defined:

```
options.magic(
    height = lambda v, cur: cur.height + int(v) if isinstance(v, str) else v,
    width  = lambda v, cur: cur.width  + int(v) if isinstance(v, str) else v
)
```

Now, in addition to absolute `height` and `width` parameters which are provided by specifying `int` (integer/numeric) values, your module auto-magically supports relative parameters for `height` and `width`.:

```
one.draw(width='+200')
```

yields:

```
color='blue', width=210, name='one', height=10
```

This can be as fancy as you like, defining an entire domain-specific expression language. But even small functions can give you a great bump in expressive power. For example, add this and you get full relative arithmetic capability (`+`, `-`, `*`, and `/`):

```
def relmath(value, currently):
    if isinstance(value, str):
        if value.startswith('*'):
            return currently * int(value[1:])
        elif value.startswith('/'):
            return currently / int(value[1:])
        else:
            return currently + int(value)
    else:
        return value
```

```
...

options.magic(
    height = lambda v, cur: relmath(v, cur.height),
    width  = lambda v, cur: relmath(v, cur.width)
)
```

Then:

```
one.draw(width='*4', height='/2')
```

yields:

```
color='blue', width=40, name='one', height=5
```

Magically interpreted parameters are the sort of thing that one doesn't need very often or for every parameter–but when they're useful, they're *enormously* useful and highly leveraged, leading to much simpler, much higher function APIs.

We call them 'magical' here because of the "auto-magical" interpretation, but they are really just analogs of Python object properties. The magic function is basically a "setter" for a dictionary element.

CHAPTER 7

The Magic APIs

The callables (usually functions, lambda expressions, static methods, or methods) called to preform magical interpretation can be called with 1, 2, or 3 parameters. `options` inquires as to how many parameters the callable accepts. If it accepts only 1, it will be the value passed in. Cleanups like "convert to upper case" can be done, but no relative interpretation. If it accepts 2 arguments, it will be called with the value and the current option mapping, in that order. (NB this order reverses the way you may think logical. Caution advised.) If the callable requires 3 parameters, it will be `None`, value, current mapping. This supports method calls, though has the defect of not really passing in the current instance.

A decorator form, `magical()` is also supported. It must be given the name of the key exactly:

```
@options.magical('name')
def capitalize_name(self, v, cur):
    return ' '.join(w.capitalize() for w in v.split())
```

The net is that you can provide just about any kind of callable. But the meta-programming of the magic interpretation API could use a little work.

CHAPTER 8

Subclassing

Subclass options may differ from superclass options. Usually they will share many options, but some may be added, and others removed. To modify the set of available options, the subclass defines its options with the `add()` method to the superclass options. This creates a layered effect, just like `push()` for an instance. The difference is, `push()` does not allow new options (keys) to be defined; `add()` does. It is also possible to assign the special null object `Prohibited`, which will disallow instances of the subclass from setting those values.:

```
options = Superclass.options.add(
    func   = None,
    prefix = Prohibited,  # was available in superclass, but not here
    suffix = Prohibited,  # ditto
)
```

Because some of the "additions" can be prohibitions (i.e. removing particular options from being set or used), this is "adding to" the superclass's options in the sense of "adding a layer onto" rather than strict "adding options."

An alternative is to copy (or restate) the superclass's options. That suits "unlinked" cases–where the subclass is highly independent, and where changes to the superclass's options should not effect the subclass's options. With `add()`, they remain linked in the same way as instances and classes are.

CHAPTER 9

Transients and Internal Options

Some options do not make sense as permanent values–they are needed only as transient settings in the context of individual method calls. The special null value `Transient` can be assigned as an option value to signal this.

Other options are useful, but only internal to your class. They are not meant to be exposed as part of the external API. In this case, they can be signified by prefixing with an underscore, such as `_cached_value`. This is consistent with Python naming practice.

CHAPTER 10

Flat Arguments

Sometimes it's more elegant to provide some arguments as flat, sequential values rather than by keyword. In this case, use the `addflat()` method:

```
def __init__(self, *args, **kwargs):
    self.options = Quoter.options.push(kwargs)
    self.options.addflat(args, ['prefix', 'suffix'])
```

to consume optional `prefix` and `suffix` flat arguments. This makes the following equivalent:

```
q1 = Quoter('[', ']')
q2 = Quoter(prefix='[', suffix=']')
```

An explicit `addflat()` method is provided not as much for Zen of Python reasons ("Explicit is better than implicit."), but because flat arguments are commonly combined with abbreviation/shorthand conventions, which may require some logic to implement. For example, if only a `prefix` is given as a flat argument, you may want to use the same value to implicitly set the `suffix`. To this end, addflat returns the set of keys that it consumed:

```
if args:
    used = self.options.addflat(args, ['prefix', 'suffix'])
    if 'suffix' not in used:
        self.options.suffix = self.options.prefix
```

CHAPTER 11

Choosing Option Names

You can choose pretty much any option name that is a legitimate Python keyword argument. The exceptions: Names that are already defined by methods of `Options` or `OptionsChain`. To wit: `add`, `addflat`, `clear`, `copy`, `fromkeys`, `get`, `items`, `iteritems`, `iterkeys`, `itervalues`, `keys`, `magic`, `magical`, `new_child`, `parents`, `pop`, `popitem`, `push`, `read`, `set`, `setdefault`, `update`, `values`, and `write` are off-limits.

If you try to define options with verboten names, a `BadOptionName` exception will be raised. This will save you grief down the road; getting back a wrong thing at runtime is vastly harder to debug than fielding an early exception.

CHAPTER 12

Special Values

Some special values ("sentinels" values) are defined:

`Prohibited` This option cannot be used (set or accessed). Useful primarily in subclasses, to "turn off" options that apply in the superclass, but not the subclass.

`Transient` Option is not set initially or on a per-instance basis, but may be invoked on a call-by-call basis.

`Reserved` Not used, but explicitly marked as reserved for future use.

`Unset` If an option is set to `Unset`, the current value is removed, letting values from higher up the option chain shine through.

CHAPTER 13

Loading From Configuration Files

`options` values can be easily writen to, or read from, configuration files. E.g. reading from JSON and YAML with a low-level approach:

```
import json
o = Options()
jdata = json.load(open('config.json'))
o.update(jdata)
```

Or for YAML:

```
import yaml
o = Options()
ydata = yaml.load(open('config.yml').read())
o.update(ydata)
```

At a higher level, `Options` objects contain a `write` method that will directly write the object to a JSON file, and a `read` class method that will construct an `Options` object from a JSON file.

CHAPTER 14

Related Work

A huge amount of work, both in Python and beyond, has gone into the effective management of configuration information.

- Program defaults. Values pre-established by developers, often as `ALL_UPPERCASE_IDENTIFIERS` or as keyword default to functions.
- Configuration file format parsers/formatters. Huge amounts of the INI, JSON, XML, and YAML specifications and toolchains, for example, are configuration-related. There are many. anyconfig is perhaps of interest for its flexibility. You could probably lump into this group binary data marshaling schemes such as `pickle`.
- Command-line argument parsers. These are all about taking configuration information from the command line. argh is one I particularly like for its simple, declarative nature. (aaargh is similar.)
- System and environment introspection. The best known of these would be `sys.argv` and `os.environ` to get command line arguments and the values of operating system environment variables (especially when running on Unixy platforms). But any code that asks "Where am I running?" or "What is my IP address?" or otherwise inspects its current execution environment and configures itself accordingly is doing a form of configuration discovery.
- Attribute-accessible dictionary objects. It is incredibly easy to create simple versions of this idea in Python–and rather tricky to create robust, full-featured versions. Caveat emptor. stuf and treedict are cream-of-the-crop implementations of this idea. I have not tried confetti or Yaco, but they look like interesting variations on the same theme.
- The portion of Web frameworks concerned with getting and setting cookies, URL query and hash attributes, form variables, and/or HTML5 local storage. Not that these are particularly secure, scalable, or robust sources...but they're important configuration information nonetheless.
- While slightly afield, database interface modules are often used for querying configuration information from SQL or NoSQL databases.
- Some object metaprogramming systems. That's a mouthful, right? Well some modules implement metaclasses that change the basic behavior of objects. value for example provides very common-sense treatment of object instantiation with out all the Javaesque `self.x = x; self.y = y; self.z = z` repetition. `options` similarly redesigns how parameters should be passed and object values stored.

- Combomatics. Many configuration-related modules combine two or more of these approaches. E.g. yconf combines YAML config file parsing with `argparse` command line parsing. In the future, `options` will also follow this path. There's no need to take programmer time and attention for several different low-level configuration-related tasks.
- Dependency injection frameworks are all about providing configuration information from outside code modules. They tend to be rather abstract and have a high "activation energy," but the more complex and composed-of-many-different-components your system is, the more valuable the "DI pattern" becomes.
- Other things. conflib, uses dictionary updates to stack default, global, and local settings; it also provides a measure of validation.

This diversity, while occasionally frustrating, makes some sense. Configuration data, after all, is just "state," and "managing state" is pretty much what all computing is about. Pretty much every program has to do it. That so many use so many different, home-grown ways is why there's such a good opportunity.

Flask's documentation is a real-world example of how "spread everywhere" this can be, with some data coming from the program, some from files, some from environment variables, some from Web-JSON, etc.

CHAPTER 15

Notes

- Automated multi-version testing managed with pytest, pytest-cov, coverage, and tox. Packaging linting with pyroma.
- Version 1.4.4 updates testing for early 2017 Python versions. 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 PyPy 5.6.0 (based on 2.7.12) and PyPy3 5.5.0 (based on 3.3.5).
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions. If you're using `options` in your own code, drop me a line!

CHAPTER 16

Installation

To install or upgrade to the latest version:

```
pip install -U options
```

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

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

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

Testing

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

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

CHAPTER 17

Change Log

1.4.6 (May 15, 2017)

Updated mechanism for method-specific option setting. Still work in progress, but code now much cleaner.

1.4.5 (January 31, 2017)

Retfined testing matrix, esp for coverage.

1.4.4 (January 23, 2017)

Updates testing. Newly qualified under 2.7.13 and 3.6, as well as most recent builds of pypy and pypy3. Drops Python 3.2 support; should still work, but no longer in testing matrix.

1.4.2 (September 15, 2015)

Updated testing with PyPy 2.6.1 (based on 2.7.10).

1.4.1 (September 14, 2015)

Updated testing matrix with 3.5.0 final.

1.4.0 (August 31, 2015)

Major reorganization of implementation. The two-level `Options` and `OptionsChain` strategy replaced with single-level `Options` based directly on `ChainMap` (or more precisely, on an attribute-accessible subclass of it). It's now turtles all the way down.

Systematic enough change that by traditional versioning standards this would be a 2.0 release. But following Semantic Versioning, while the class structure changes, the effective API seen by using modules does not change, so 1.4.0 is enough.

Correctness of this systematic roto-tilling confirmed by test suite. Testing now extended to 100% line coverage (and 99% branch coverage). Some edge case issues were discovered and corrected. Thank you to coverage testing for ferreting those out.

No longer depends on stuf. Coupled with a new supporting chainmap polyfill, decisively returns compatibility for Python 2.6 in a way that doesn't depend on the release schedule or priorities of external modules.

1.3.2 (August 26, 2015)

Reorganized documentation structure.

1.3.0 (August 25, 2015)

Added test branch metrics to coverage evaluation. Line coverage now 93%; branch coverage 92%.

Integrated reading/writing of options data to JSON files now operational and tested.

1.2.5 (August 17, 2015)

Inaugurated automated test coverage analysis. Extended a few tests and cleaned up some code as a result. Published with coverage at 88%.

1.2.2 (August 11, 2015)

Simplified setup.

1.2.1 (August 4, 2015)

Added wheel distribution format. Updated test matrix.

Moved from BSD to Apache Software License.

Moved status to production/stable from beta.

1.2.0 (July 22, 2015)

Doc and config tweaks.

Python 2.6 support wavering, primarily because of failure of `stuf` 0.9.16 to build there. 0.9.14 works fine. But either `stuf` support will have to improve (I've submitted a pull request that fixes the problem), or we'll have to swap `stuf` out, or we''ll have to decomit py26.

1.1.7 (December 16, 2014)

Added snazzy badges to PyPI readme

1.1.5 (December 16, 2014)

Changed dependencies to utilize new `nulltype` package (unbundling it). Ensured tested on all lastest Python versions.

1.1.1 (October 29, 2013)

Added `OptionsClass` base class. If client classes inherit from this, they automatically get ' set()'' and `settings()` methods.

1.0.7 (October 25, 2103)

Mainly doc tweaks.

1.0.4 (October 24, 2013)

When bad option names are defined ("bad" here meaning "conflicts with names already chosen for pre-existing methods"), a `BadOptionName` exception will be raised.

Tweaked docs, adding comparison chart.

1.0.3 (September 23, 2013)

Switched to local version of `chainstuf` until bug with generator values in `stuf.chainstuf` can be tracked down and corrected. This was blocking a downstream feature-release of `say`.

1.0.2 (September 19, 2013)

Improved `setdefault` and `update` methods, and added tests, primarily in effort to work around bug that appears in `stuf`, `orderedstuf`, or `chainstuf` when a mapping value is a generator.

Documentation improved.

1.0.1 (September 14, 2013)

Moved main documentation to Sphinx format in ./docs, and hosted the long-form documentation on readthedocs.org. README.rst now an abridged version/teaser for the module.

1.0.0 (September 10, 2013)

Cleaned up source for better PEP8 conformance

Bumped version number to 1.0 as part of move to semantic versioning, or at least enough of it so as to not screw up Python installation procedures (which don't seem to understand 0.401 is a lesser version that 0.5, because 401 > 5).