environment.py Documentation

Release 1.0.0-dev

Gittip, LLC

Mar 09, 2017

Contents

1 Installation 3

2 Rationale 5

3 Tutorial 7

4 API Reference 9

Python Module Index 11

This library provides parsing and validation of environment variables.

CHAPTER 1

Installation

environment is available on GitHub and on PyPI:

```
$ pip install environment
```

We test against Python 2.6, 2.7, 3.2, and 3.3.

environment is MIT-licensed.

CHAPTER 2

Rationale

Configuration via environment variables has become popular with the rise of twelve-factor apps, yet few Python libraries exist to help with it (despite the abundance of libraries for command line and file configuration).

When I looked around, most of the solutions I found involved using `os.environ` directly, or overloading somewhat related libraries such as `argparse` or `formencode`. The former are not robust enough with regards to typecasting and error handling. The latter are inappropriate and overengineered: the reason to prefer envvar configuration in the first place is to reduce complexity, not compound it. We need something designed specifically and solely for taking configuration from environment variables.

The one library I found is python-decouple, which does indeed rationalize typecasting of environment variables. However, it also handles file configuration, which adds unwanted complexity and (ironically) muddying of concerns. Additionally, it doesn't enable robust error messaging. The problem with error handling in `decouple` and in ad-hoc usage of `os.environ` is that if you have four environment variables wrong, you only find out about them one at a time. We want to find out about all problems with our configuration at once, so that we can solve them all at once instead of playing configuration roulette ("Will it work this time? No! How about now?").

This present library is designed to be small in scope, limited to environment variables only, and to support robust error messaging. Look into foreman and honcho for process management tools to complement this library.

CHAPTER 3

Tutorial

First let's pretend that this is our `os.environ`:

```
>>> pretend_os_environ = { 'FOO': '42'
...                       , 'BAR_BAZ': 'buz'
...                       , 'BAR_BLOO_BLOO': 'yes'
...                       , 'BAD': 'to the bone'
...                        }
```

And let's import our stuff:

```
>>> from environment import Environment, is_yesish
```

The way the *`environment`* library works is you instantiate an *`Environment`* class like so:

```
>>> env = Environment( FOO=int
...                  , BLAH=str
...                  , BAD=int
...                  , environ=pretend_os_environ
...                   )
```

Keyword arguments specify which variables to look for and how to typecast them. Since a process environment contains a lot of crap you don't care about, we only parse out variables that you explicitly specify in the keyword arguments.

The resulting object has lowercase attributes for all variables that were asked for and found:

```
>>> env.foo
42
```

There are also `missing` and `malformed` attributes for variables that weren't found or couldn't be typecast:

```
>>> env.missing
['BLAH']
>>> env.malformed
[('BAD', "ValueError: invalid literal for int() with base 10: 'to the bone'")]
```

You're expected to inspect the contents of `missing` and `malformed` and do your own error reporting. You're also expected to handle defaults yourself at a higher level—this is not a general-purpose configuration library—though the `parsed` dictionary should help with that:

```
>>> env.parsed
{'foo': 42}
```

If all of the environment variables you care about share a common prefix, you can specify this to the constructor to save yourself some clutter:

```
>>> bar = Environment( 'BAR_'
...                   , BAZ=str
...                   , BLOO_BLOO=is_yesish
...                   , environ=pretend_os_environ
...                    )
>>> bar.baz
'buz'
>>> bar.bloo_bloo
True
```

CHAPTER 4

API Reference

class environment.**Environment**(*prefix=''*, *spec=None*, *environ=None*, ***kw*)

Represent a whitelisted, parsed subset of a process environment.

Parameters

- **prefix** (*string*) – If all of the environment variables of interest to you share a common prefix, you can specify that here. We will use this prefix when pulling values out of the environment, and the attribute names you end up with won't include the prefix.
- **spec** (*mapping*) – A mapping of environment variable names to typecasters.
- **environ** (*mapping*) – By default we look at os.environ, of course, but you can override that with this. We operate on a shallow copy of this mapping (though it's effectively a deep copy in the normal case where all values are strings, since strings are immutable).
- **kw** – Keyword arguments are folded into spec.

The constructor for this class loops through the items in environ, skipping those variables not also named in spec, and parsing those that are, using the type specified. Under Python 2, we harmonize with Python 3's behavior by decoding environment variable values to unicode using the result of sys.getfilesystemencoding before typecasting. The upshot is that if you want typecasting to be a pass-through for a particular variable, you should specify the Python-version-appropriate string type: str for Python 3, unicode for Python 2. We store variables using lowercased names, so MYVAR would end up at env.myvar:

```
>>> env = Environment(MYVAR=int, environ={'MYVAR': 42})
>>> env.myvar
42
```

If a variable is mentioned in spec but is not in environ, the variable name is recorded in the missing list. If typecasting a variable raises an exception, the variable name and an error message are recorded in the malformed list:

```
>>> env = Environment(MYVAR=int, OTHER=str, environ={'MYVAR': 'blah'})
>>> env.missing
['OTHER']
```

```
>>> env.malformed
[('MYVAR', "ValueError: invalid literal for int() with base 10: 'blah'")]
```

If `prefix` is provided, then we'll add that to the variable names in `spec` when reading the environment:

```
>>> foo = Environment('FOO_', BAR=int, environ={'FOO_BAR': '42'})
>>> foo.prefix
'FOO_'
>>> foo.bar
42
```

The copy of `environ` that we act on is stored at `environ`:

```
>>> foo.environ
{'FOO_BAR': '42'}
```

All parsed variables are stored in the dictionary at `parsed`:

```
>>> foo.parsed
{'bar': 42}
```

Use the `parsed` dictionary, for example, to fold configuration from the environment together with configuration from other sources (command line, config files, defaults) in higher-order data structures. Attribute access for non-class attributes on *`Environment`* instances uses `parsed` rather than `__dict__`, which means that you can set attributes on the instance and they're reflected in `parsed`:

```
>>> foo.bar = 537
>>> foo.parsed
{'bar': 537}
```

But setting attributes doesn't modify `environ`:

```
>>> foo.environ
{'FOO_BAR': '42'}
```

static `parse`(*prefix*, *spec*, *environ*, *encoding*)

Heavy lifting, with no side-effects on `self`.

Parameters

- **`prefix`** (*`string`*) – The string to prefix to variable names when looking them up in `environ`.
- **`spec`** (*`mapping`*) – A mapping of environment variable names to typecasters.
- **`environ`** (*`mapping`*) – A mapping of environment variable names to values.
- **`encoding`** (*`string`*) – The encoding with which to decode environment variable values before typecasting them, or `None` to suppress decoding.

Returns A three-tuple, corresponding to `missing`, `malformed`, and `parsed`.

`environment`**`.is_yesish`**(*value*)

Typecast booleanish environment variables to `bool`.

Parameters **`value`** (*`string`*) – An environment variable value.

Returns `True` if `value` is `1`, `true`, or `yes` (case-insensitive); `False` otherwise.

Python Module Index

e

environment, 3

Index

E

Environment (class in environment), 9
environment (module), 1

I

is_yesish() (in module environment), 10

P

parse() (environment.Environment static method), 10