SysEnv Documentation

Release 0.1.0

Ben Lopatin

August 20, 2013

CONTENTS

Contents:

CHAPTER

ONE

SYSENV: THE ENVIRONMENT VARIABLE HELPER

Simple handling of system environment variables for application deployment.

SysEnv lets you configure your Python application using environment variables, casting types as necessary, enabling smooth 12factor deployments.

It's a replacement for an inline function, with some inspiration and code from Honcho. The interface is directly inspired by Django-environ but unlike Django-environ SysEnv is not Django specific and does not replace functionality provided by existing applications.

- Free software: BSD license
- Documentation: http://sysenv.rtfd.org.

1.1 Installation

pip install sysenv

1.2 Using SysEnv

To load from your local environment, use the *load* function to return an *EnvDict* instance.:

```
>>> from sysenv import load
>>> env = load()
```

By default it loads from the system environment, but you can also include values from a *.env* or similar file.:

```
>>> from sysenv import load
>>> env = load('.env')
```

The *EnvDict* instance can be accessed like a normal Python dictionary.:

```
>>> env['DEBUG']
'True'
>>> env.get('DEBUG', False)
'True'
```

Of course your environment variables are strings, so you'll want to cast the return value to a boolean.:

```
>>> env.get('DEBUG', False, cast=bool)
True
```

Alternatively you can define a schema when you initialize your *EnvDict*.:

```
>>> env = EnvDict(schema={'DEBUG': bool, 'CACHE_COUNT': int})
>>> env = load(schema={'DEBUG': bool, 'CACHE_COUNT': int})
```

The schema should be provided as a dictionary of key names with the cast identifier given as the value in the schema.

1.2.1 Using with Django

To use SysEnv in your Django project add it to the top of your settings.py file and pull in setting values from the *env* variable (or whatever you call it).:

```
from sysenv import load
env = load('.env')
DEBUG = env.get('DEBUG', False, cast=bool)
```

See the docs for additional usage instructions.

CHAPTER

TWO

INSTALLATION

At the command line:

```
$ pip install sysenv
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv sysenv
$ pip install sysenv
```

CHAPTER

THREE

USAGE

To load from your local environment, use the *load* function to return an *EnvDict* instance.:

```
>>> from sysenv import load
>>> env = load()
```

By default it loads from the system environment, but you can also include values from a *.env* or similar file.:

```
>>> from sysenv import load
>>> env = load('.env')
```

Note: The load function returns a new *EnvDict* instance and also updates the system environment with values in the provided file so that later calles to *os.environ* will pick up on these new variables or values.

The *EnvDict* instance can be accessed like a normal Python dictionary.:

```
>>> env['DEBUG']
'True'
>>> env.get('DEBUG', False)
'True'
```

Of course your environment variables are strings, so you'll want to cast the return value to a boolean.:

```
>>> env.get('DEBUG', False, cast=bool)
True
```

3.1 Type schemas

Alternatively you can define a schema when you initialize your *EnvDict*. For example, to ensure that the value for *DEBUG* is rendered internally as a boolean value and *CACHE_COUNT* is rendred as an integer:

```
>>> env = EnvDict(schema={'DEBUG': bool, 'CACHE_COUNT': int})
>>> env.get('DEBUG')
True
>>> env['DEBUG']
True
>>> env = load(schema={'DEBUG': bool, 'CACHE_COUNT': int})
>>> env.get('CACHE_COUNT')
3600
>>> env['CACHE_COUNT']
3600
```

The schema should be provided as a dictionary of key names with the cast identifier given as the value in the schema.

3.2 Type casting

By default an EnvDict instance can cast to these types:

- *str* (default)
- *bool*
- *int*
- *float*
- *Decimal*
- *list*
- *dict*

Each type can be cast using a built-in dictionary that uses the type (by type object or name) as the key and the type constructor or other passable function as the value.

All built-in types are cast using the type as the casting dictionary key, e.g.:

```
env.get('MY_VAR', cast=list)
```

Non-built in types should be referenced by lowercased string name, e.g.:

```
env.get('MY_VAR', cast='decimal')
```

3.2.1 bool conversion

Booleans are cast by comparing the lowercased input against a tuple of "true" values:

- true
- on
- yes
- 1

True is returned if the filtered input is in the tuple, otherwise *False*.

3.2.2 list converstion

The list conversion produces a list of strings based on the input, e.g. it will turn:

```
MYVAR=1,2,3,4
```

Into:

```
>>> env.get('MYVAR', cast=list)
['1', '2', '3', '4']
```

3.2.3 dictionary conversion

The dictionary conversion produces a dictionary of string keys and values based on the input, e.g. it will turn:

```
MYVAR=a=1,b=2,c=3,d=4
```

Into:

```
>>> env.get('MYVAR', cast=dict)
{'a': '1', 'b': '2', 'c': '3', 'd': '4'}
```

3.2.4 Extending type casting

You can replace or add type casting methods. Simply pass a dictionary using the *casts* keyword when creating an *EnvDict* instance.:

```
>>> MY_TRUE_VALUES = ('true', 'for sure')
>>> env = EnvDict({'DEBUG': 'For SURE'}, casts={bool: lambda x: x.lower in MY_TRUE_VALUES})
>>> env.get('DEBUG', cast=bool)
True
```

The same dictionary can be based when loading the environment.:

```
>>> MY_TRUE_VALUES = ('true', 'for sure')
>>> env = load(casts={bool: lambda x: x.lower in MY_TRUE_VALUES})
>>> env.get('DEBUG', cast=bool)
True
```

3.3 Using with Django

To use SysEnv in your Django project add it to the top of your settings.py file and pull in setting values from the *env* variable (or whatever you call it).:

```
from sysenv import load
env = load('.env')
DEBUG = env.get('DEBUG', False, cast=bool)
```

Note that SysEnv does not offer functionality for configuring database or working with file paths. There are already libraries that do those things.

You can configure your database(s) using DJ-Database-URL.:

```
DATABASES = {'default': dj_database_url.config(default='postgres://localhost')}
```

Got search? You can use DJ-Search-URL.:

```
HAYSTACK_CONNECTIONS = {"default": dj_search_url.conf("elasticsearch://..")}
```

If you want to make working with file paths simpler you should take a look at path.py.

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/bennylope/sysenv/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

SysEnv could always use more documentation, whether as part of the official SysEnv docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/bennylope/sysenv/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *sysenv* for local development.

1. Fork the *sysenv* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/sysenv.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv sysenv
   $ cd sysenv/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 sysenv tests
$ python setup.py test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/bennylope/sysenv/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_sysenv
```

CHAPTER

FIVE

CREDITS

5.1 Development Lead

- Ben Lopatin <ben@wellfire.co>

5.2 Contributors

None yet. Why not be the first?

CHAPTER

SIX

HISTORY

6.1 0.1.0 (2013-08-11)

- First release on PyPI.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*