prettyconf Documentation

Release 2.2.1

Osvaldo Santana Neto

Jan 07, 2021

Contents

Contents:

CHAPTER 1

What's prettyconf

Prettyconf is a framework agnostic python library created to make easy the separation of configuration and code following the recomendations of 12 Factor's topic about configs.

1.1 Motivation

Configuration is just another API of you app, aimed for users who will install and run it, that allows them to *preset* the state of a program, without having to interact with it, only through static files or environment variables.

It is an important aspect of the architecture of any system, yet it is sometimes overlooked.

It is important to provide a clear separation of configuration and code. This is because config varies substantially across deploys and executions, code should not. The same code can be run inside a container or in a regular machine, it can be executed in production or in testing environments.

Well designed applications allow different ways to be configured. A proper settings-discoverability chain goes as follows:

1. First CLI args are checked.
2. Then Environment variables.
3. Config files in different directories, that also imply some hierarchy. For example: config files in `/etc/myapp/settings.ini` are applied system-wide, while `~/.config/myapp/settings.ini` take precedence and are user-specific.
4. Hardcoded constants.

This raises the need to consolidate configuration in a single source of truth to avoid having config management scattered all over the codebase.

CHAPTER 2

Requirements

- Python 2.7+ or 3.4+

CHAPTER 3

Installation

First you need to install `prettyconf` library:

```
pip install prettyconf
```

The `AwsParameterStore` configuration loader depends on the `boto3` package. If you need to use it, install `prettyconf` with the optional feature `aws`:

```
pip install prettyconf[aws]
```

CHAPTER 4

Usage

You can import and use `prettyconf` in your Python code:

```
from prettyconf import config

MY_CONFIG = config("PROJECT_MY_CONFIG")
```

If `PROJECT_MY_CONFIG` is not defined in an environment variable neither in a `.env` (or `*.cfg`) file, `prettyconf` will raise a `UnknownConfiguration` exception.

> **Warning:** `prettyconf` will skip configuration files inside `.zip`, `.egg` or wheel packages.

In these cases you could define a default configuration value:

```
MY_CONFIG = config("PROJECT_MY_CONFIG", default="default value")
```

You can also use the `cast` argument to convert a string value into a specific value type:

```
DEBUG = config("DEBUG", default=False, cast=config.boolean)
```

The `boolean` cast converts strings values like `On|Off`, `1|0`, `yes|no`, `true|False` into Python boolean `True` or `False`.

See also:

Find out more about other casts or how to write your own at *Casts*.

4.1 Configuration files discovery

By default library will use the directory of the file where `config()` was called as the start directory to look for configuration files. Consider the following file structure:

```
project/
  settings.ini
  app/
    settings.py
```

If you call `config()` from `project/app/settings.py` the library will start looking for configuration files at `project/app` until it finds `.env|*.ini|*.cfg` files.

See also:

This behavior is described more deeply on the `RecursiveSearch` loader. *Loaders* will help you customize how configuration discovery works. Find out more at *Customizing the configuration discovery*.

CHAPTER 5

Casts

5.1 Buitin Casts

1. `config.boolean` - converts values like `On|Off`, `1|0`, `yes|no`, `true|false`, `t|f` into booleans.
2. `config.eval` - safely evaluate strings with Python literals to Python objects (alias to Python's `ast.literal_eval`).
3. `config.list` - converts comma separated strings into lists.
4. `config.tuple` - converts comma separated strings into tuples.
5. `config.json` - unserialize a string with JSON object into Python.
6. `config.option` - get a return value based on specific options:

```
environments = {
    "production": ("spam", "eggs"),
    "local": ("spam", "eggs", "test"),
}

# Will return a tuple with ("spam", "eggs") when
# ENVIRONMENT is undefined or defined with `production`
# and a tuple with ("spam", "eggs", "test") when
# ENVIRONMENT is set with `local`.
MODULES = config("ENVIRONMENT",
                 default="production",
                 cast=Option(environment))
```

5.2 Custom casts

You can implement your own custom casting function:

```
def number_list(value):
    return [int(v) for v in value.split(";")]

NUMBERS = config("NUMBERS", default="1;2;3", cast=number_list)
```

5.3 Useful third-parties casts

Django is a popular python web framework that imposes some structure on the way its settings are configured. Here are a few 3rd party casts that help you adapt strings into that inner structures:

- dj-database-url - Parses URLs like `mysql://user:pass@server/db` into Django `DATABASES` configuration format.
- django-cache-url - Parses URLs like `memcached://server:port/prefix` into Django `CACHES` configuration format.
- dj-email-url - Parses URLs like `smtp://user@domain.com:pass@smtp.example.com:465/?ssl=True` with parameters used in Django `EMAIL_*` configurations.
- dj-admins-setting - Parses emails lists for the `ADMINS` configuration.

CHAPTER 6

Advanced Usage

Most of the time you can use the `prettyconf.config` function to get your settings and use the `prettyconf`'s standard behaviour. But some times you need to change this behaviour.

To make this changes possible you can always create your own `Configuration()` instance and change it's default behaviour:

```
from prettyconf import Configuration

config = Configuration()
```

Warning: `prettyconf` will skip configuration files inside `.zip`, `.egg` or wheel packages.

6.1 Customizing the configuration discovery

By default the library will use the envrionment and the directory of the file where `config()` was called as the start directory to look for a `.env` configuration file. Consider the following file structure:

```
project/
  app/
    .env
    config.ini
    settings.py
```

If you call `config()` from `project/app/settings.py` the library will inspect the envrionment and then look for configuration files at `project/app`.

You can change that behaviour, by customizing configuration loaders to look at a different `path` when instantiating your `Configuration()`:

```
# Code example in project/app/settings.py
import os

from prettyconf import Configuration
from prettyconf.loaders import Environment, EnvFile

project_path = os.path.realpath(os.path.join(os.path.dirname(__file__), '..'))
env_file = f"{project_path}/.env"
config = Configuration(loaders=[Environment(), EnvFile(filename=env_file)])
```

The example above will start looking for configuration in the environment and then in a `.env` file at `project/` instead of `project/app`.

Because `config` is nothing but an already instantiated `Configuration` object, you can also alter this `loaders` attribute in `prettyconf.config` before use it:

```
# Code example in project/app/settings.py
import os

from prettyconf import config
from prettyconf.loaders import Environment, EnvFile

project_path = os.path.realpath(os.path.join(os.path.dirname(__file__), '..'))
env_file = f"{project_path}/.env"
config.loaders = [Environment(), EnvFile(filename=env_file)]
```

Read more about how loaders can be configured in the *loaders section*.

6.2 Naming conventions for variables

There happen to be some formating conventions for configuration paramenters based on where they are set. For example, it is common to name environment variables in uppercase:

```
$ DEBUG=yes OTHER_CONFIG=10 ./app.py
```

but if you were to set this config in an `.ini` file, it should probably be in lower case:

```
[settings]
debug=yes
other_config=10
```

command line argments have yet another conventions:

```
$ ./app.py --debug=yes --another-config=10
```

Prettyconf let's you follow these aesthetics patterns by setting a `var_format` function when instantiating the *loaders*.

By default, the `Environment` is instantiated with `var_format=str.upper` so that lookups play nice with the env variables.

```
from prettyconf import Configuration
from prettyconf.loaders import Environment

config = Configuration(loaders=[Environment(var_format=str.upper)])
debug = config('debug', default=False, cast=config.boolean)  # lookups for␣
↪DEBUG=[yes|no]
```

6.3 Writing your own loader

If you need a custom loader, you should just extend the `AbstractConfigurationLoader`.

For example, say you want to write a Yaml loader. It is important to note that by raising a `KeyError` exception from the loader, prettyconf knows that it has to keep looking down the loaders chain for a specific config.

```
import yaml
from prettyconf.loaders import AbstractConfigurationLoader

class YamlFile(AbstractConfigurationLoader):
    def __init__(self, filename):
        self.filename = filename
        self.config = None

    def _parse(self):
        if self.config is not None:
            return
        with open(self.filename, 'r') as f:
            self.config = yaml.load(f)

    def __contains__(self, item):
        try:
            self._parse()
        except:
            return False

        return item in self.config

    def __getitem__(self, item):
        try:
            self._parse()
        except:
            # KeyError tells prettyconf to keep looking elsewhere!
            raise KeyError("{!r}".format(item))

        return self.config[item]
```

Then configure prettyconf to use it.

```
from prettyconf import config
config.loaders = [YamlFile('config.yml')]
```

CHAPTER 7

Configuration Loaders

Loaders are in charge of loading configuration from various sources, like `.ini` files or *environment* variables. Loaders are ment to chained, so that prettyconf checks one by one for a given configuration variable.

Prettyconf comes with some loaders already included in `prettyconf.loaders`.

See also:

Some loaders include a `var_format` callable argument, see *Naming conventions for variables* to read more about it's purpose.

7.1 Environment

The `Environment` loader gets configuration from `os.environ`. Since it is a common pattern to write env variables in caps, the loader accepts a `var_format` function to pre-format the variable name before the lookup occurs. By default it is `str.upper()`.

```
from prettyconf import config
from prettyconf.loaders import Environment


config.loaders = [Environment(var_format=str.upper)]
config('debug')  # will look for a `DEBUG` variable
```

7.2 EnvFile

The `EnvFile` loader gets configuration from `.env` file. If the file doesn't exist, this loader will be skipped without raising any errors.

```
# .env file
DEBUG=1
```

```
from prettyconf import config
from prettyconf.loaders import EnvFile


config.loaders = [EnvFile(file='.env', required=True, var_format=str.upper)]
config('debug')  # will look for a `DEBUG` variable
```

Note: You might want to use dump-env, a utility to create `.env` files.

7.3 IniFile

The `IniFile` loader gets configuration from `.ini` or `.cfg` files. If the file doesn't exist, this loader will be skipped without raising any errors.

7.4 CommandLine

This loader lets you extract configuration variables from parsed CLI arguments. By default it works with argparse parsers.

```
from prettyconf import Configuration, NOT_SET
from prettyconf.loaders import CommandLine

import argparse


parser = argparse.ArgumentParser(description='Does something useful.')
parser.add_argument('--debug', '-d', dest='debug', default=NOT_SET, help='set debug␣
↪mode')

config = Configuration(loaders=[CommandLine(parser=parser)])
print(config('debug', default=False, cast=config.boolean))
```

Something to notice here is the `NOT_SET` value. CLI parsers often force you to put a default value so that they don't fail. In that case, to play nice with prettyconf, you must set one. But that would break the discoverability chain that prettyconf encourages. So by setting this special default value, you will allow prettyconf to keep the lookup going.

The `get_args` function converts the argparse parser's values to a dict that ignores `NOT_SET` values.

7.5 RecursiveSearch

This loader tries to find `.env` or `*.ini|*.cfg` files and load them with the `EnvFile` and `IniFile` loaders respectively. It will start at the `starting_path` directory to look for configuration files.

Warning: It is important to note that this loader uses the glob module internally to discover `.env` and `*.ini|*.cfg` files. This could be problematic if the project includes many files that are unrelated, like a `pytest.ini` file along side with a `settings.ini`. An unexpected file could be found and be considered as the configuration to use.

Consider the following file structure:

```
project/
  settings.ini
  app/
    settings.py
```

When instantiating your `RecursiveSearch`, if you pass `/absolute/path/to/project/app/` as `starting_path` the loader will start looking for configuration files at `project/app`.

```
# Code example in project/app/settings.py
import os

from prettyconf import config
from prettyconf.loaders import RecursiveSearch

app_path = os.path.dirname(__file__)
config.loaders = [RecursiveSearch(starting_path=app_path)]
```

By default, the loader will try to look for configuration files until it finds valid configuration files **or** it reaches `root_path`. The `root_path` is set to the root directory `/` initialy.

Consider the following file structure:

```
/projects/
  any_settings.ini
  project/
    app/
      settings.py
```

You can change this behaviour by setting any parent directory of the `starting_path` as the `root_path` when instantiating `RecursiveSearch`:

```
# Code example in project/app/settings.py
import os

from prettyconf import Configuration
from prettyconf.loaders import RecursiveSearch

app_path = os.path.dirname(__file__)
project_path = os.path.realpath(os.path.join(app_path, '..'))
rs = RecursiveSearch(starting_path=app_path, root_path=project_path)
config = Configuration(loaders=[rs])
```

The example above will start looking for files at `project/app/` and will stop looking for configuration files at `project/`, actually never looking at `any_settings.ini` and no configuration being loaded at all.

The `root_path` must be a parent directory of `starting_path`:

```
# Code example in project/app/settings.py
from prettyconf.loaders import RecursiveSearch

# /baz is not parent of /foo/bar, so this raises an InvalidPath exception here
rs = RecursiveSearch(starting_path="/foo/bar", root_path="/baz")
```

7.6 AwsParameterStore

The `AwsParameterStore` loader gets configuration from the AWS Parameter Store, part of AWS Systems Manager. The loader will be skipped if the parameter store is unreachable (connectivity, unavailability, access permissions). The loader respects parameter hierarchies, performing non-recursive discoveries. The loader accepts AWS access secrets and region when instantiated, otherwise, it will use system-wide defaults (if available). The AWS parameter store supports three parameter types: `String`, `StringList` and `SecureString`. All types are read as strings, however, decryption of `SecureStrings` is not handled by the loader.

```
from prettyconf import config
from prettyconf.loaders import AwsParameterStore


config.loaders = [AwsParameterStore(path='/api')]
config('debug')  # will look for a parameter named "/api/debug" in the store
```

CHAPTER 8

FAQ

8.1 Why not use environment variables directly?

There is a common pattern to read configurations in environment variable that look similar to the code below:

```
if os.environ.get("DEBUG", False):
    print(True)
else:
    print(False)
```

But this code have some issues:

1. If *envvar* `DEBUG=False` this code will print `True` because `os.environ.get("DEBUG", False)` will return an string *'False'* instead of a boolean *False*. And a non-empty string has a `True` boolean value.
2. We can't (dis|en)able debug with *envvars* `DEBUG=yes|no`, `DEBUG=1|0`, `DEBUG=True|False`.
3. If we want to use this configuration during development we need to define this *envvar* all the time. We can't define this setting in a configuration file that will be used if *DEBUG envvar* is not defined.

8.2 Is prettyconf tied to Django or Flask?

No, prettyconf was designed to be framework agnostic, be it for the web or cli applications.

8.3 What is the difference between prettyconf and python-decouple?

There is no subtantial difference between both libraries. `prettyconf` is highly inspired in `python-decouple` and provides almost the same API.

The implementation of `prettyconf` is more extensible and flexible to make behaviour configurations easier.

You can use any of them. Both are good libraries and provides a similar set of features.

8.4 Why you created a library similar to python-decouple instead of use it?

I made some contributions for python-decouple previously, but I needed to change its behaviour as described above and this change is backward incompatible, so, it could break software that relies on the old behaviour. Besides that it's hard to make this change on `python-decouple` due to the way it's implemented.

See the lookup order of configurations below

Lookup Order	prettyconf	python-decouple (<3.0)	python-decouple (>=3.0)
1	ENVVAR	.env	ENVVAR
2	.env	settings.ini	.env
3	*.cfg or *.ini	ENVVAR	settings.ini

8.5 How does prettyconf compare to python-dotenv?

python-dotenv reads the key, value pair from .env file and adds them to environment variable. It is good for some tools that simply proxy the env to some other process, like docker-compose or pipenv.

On the other hand, prettyconf does not populate the `os.environ` dictionary, because it is designed to discover configuration from diferent sources, the environment being just one of them.

CHAPTER 9

Changelog

All notable changes to this project will be documented in this file.

This project adheres to Semantic Versioning.

9.1 2.2.1

- Fix JSON cast for already parsed default configuration

9.2 2.2.0

- Add new cast *config.json* for JSON configurations

9.3 2.1.0

- Add (optional) support for AWSParameterStore. To enable it install `prettyconf[aws]` (thanks @ronaldotd)
- Replace nosetest -> pytest
- Implemented new .env parser with multiline support (thanks @jaysonsantos)
- Update and improve `casts` session documentation (thanks @hernantz)

9.4 2.0.1

- Hopeful quick and dirty fix of the discovery system (do not install version 2.0.0)

9.5 2.0.0

- Refactor strategy to find configuration files (thanks to @hernantz)
- Lots of improvements on documentation
- Dropped support for py2
- Dropped tox support

9.6 1.2.3

- Fix a blocker issue with `config.eval` cast and add a test to prevent regressions
- Add more informations about python-decouple vs. prettyconf at FAQ (Fixes #16 again)

9.7 1.2.2

- Remove testfixtures requirements (it's broken with pypy)

9.8 1.2.1

- New cast type: config.eval (uses `ast.literal_eval` to cast python settings)
- 3rd-party suggestion: dj-email-url parser in documentation

9.9 1.2.0

- New cast type: config.tuple (converts a comma-separated string in tuple)

9.10 1.1.2

- Ignore errors in `make clean`

9.11 1.1.1

- Fix a brown paper bug in the last release
- Force test running in `make release` target
- Add "pragma: no cover" in abstract methods

9.12 1.1.0

- Skip discovering files inside not-directory paths (eg. `.egg`, `.zip` or wheel packages)

9.13 1.0.1

- Fix a issue that breaks .ini/.cfg loader with "broken" files

9.14 1.0.0

- First stable release! hooray!
- Make configuration load lazy to make possible change root_path and starting_path in prettyconf.config
- Default root_path is "/" instead of $HOME (backward incompatible change)
- Add missing requirements in requirements.txt and make tox use it.
- Small PEP-8 and code formatting fixes

9.15 0.4.1

- Add MacOSX travis builds.

9.16 0.4.0

- Add `root_path` to stop looking indefinitely for configuration files until the OS root path
- Add advanced usage docs
- Include a simple (but working) tox configuration for py27 + py34 to the project

9.17 0.3.3

- Start a structure to make a better documentation with sphinx and publish it at Read the Docs

9.18 0.3.2

- Stop directories from being traversed up when valid configurations were found.
- Validates invalid unicode data on INI files (and skip them)
- Better Python3 support with use of ConfigParser.read_file
- Code cleanup
- More test cases for ConfigurationDiscovery added

9.19 0.3.1

- Fix a bad behaviour that make impossible to define a None default

9.20 0.3.0

- Make config.{cast} shortcuts easier to use. This change breaks backward compatibility.

9.21 0.2.2

- Fix an issue with .env parser that breaks with unquoted URL values
- Fix an issue with magic _get_path used by config discovery (thanks @bertonha)

9.22 0.2.0

- Add basic documentation

9.23 0.1.1

- Fix a small issue in README.txt formatting

9.24 0.1

- First version

CHAPTER 10

Indices and tables

- genindex
- modindex
- search