classyconf Documentation

Release 0.5.2

Hernan Lozano

Jun 23, 2021

Contents

Configuration is just another API of your app. It allows us to preset or modify it's behavior based on where it is installed and how it will be executed, providing more flexibility to the users of such software.

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.

Configuration management is an important aspect of the architecture of any system. But it is sometimes overlooked.

Classyconf is here to help, it's the configuration management solution for perfectionists with deadlines.

CHAPTER 1

What's classyconf

Classyconf is a framework agnostic python library created to make easy the separation of configuration and code.

It adds a declarative way to define settings for your projects contained in a class that can be extended, config objects can be passed around modules and settings are lazily loaded, plus some other goodies (aka dunder methods).

It's classy, it's pretty, it's good.

1.1 Motivation

Configuration is just another API of your 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.

1.2 Settings discoverability

Well designed applications allow different ways to be configured. For example command line args are great to explore an app from the shell, but when you already know what you want, it would be great to set some defaults in a configuration file somewhere.

But what happens if a setting is passed as command line arg but also exist in the config file?

A proper settings-discoverability chain goes as follows:

1. First command line 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 as defaults.

Each one of this sources of configuration need to be properly collected and overwritten with an explicit level of hierarchy.

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

1.3 Parsing and casting

Not only each different source of configuration needs to be parsed differently but also each setting might need to be converted from a generic type like strings to proper types like integers or db connection structs.

Also each source of configuration follows some naming conventions, CLI args look like this `--flag=true` while environment variables can be `FLAG=on`.

1.4 A settings architecture

Classyconf was born as a wrapper around prettyconf, inspired by goodconf, originally trying to follow the recomendations of 12 Factor's topic about configs, but expanded to address all the cases stated above.

The good practices that this library suggest have an agnostic approach to configure applications, no matter if they are web, CLI or GUI apps, hosted on the cloud or running in your desktop.

Classyconf aims to be the settings management solution for perfectionists with deadlines.

CHAPTER 2

Requirements

- Python 3.7+

CHAPTER 3

Installation

To install the `classyconf` library simply run:

```
pip install classyconf
```

CHAPTER 4

Getting started

ClassyConf aims to be the configuration management solution for perfectionists with deadlines.

It solves many problems, so let's get started with an incremental introduction to what it can do for you.

4.1 1. Declaring settings

The simplest ways to get started with classyconf is to use the `Configuration` class, to declare all the settings of your app.

This pythonic class allows you to encapsulate all configuration in one object by declaring your settings using the `Value` descriptor.

In this case, we will use a single debug setting, but there could be as many as you need.

```
from classyconf import Configuration, Value, as_boolean


class AppConfig(Configuration):

    DEBUG = Value(default=False, cast=as_boolean, help="Toggle debugging on/off.")
```

We are using the `as_boolean` cast for the `DEBUG` setting. The `as_boolean` cast converts values like `On|Off`, `1|0`, `yes|no`, `true|false`, `t|f` into Python boolean `True` or `False`.

See also:

Visit the *Casts* section to find out more about other casts or how to write your own.

Since we provided a boolean default, there is no need to explicitely set a cast in this case, classyconf will choose the `as_boolean` to save you some typing. Visit the *Implicit casts* section to see how to customize this.

4.2 2. Discovering settings

Now that we defined the settings we needed, we will define where to obtain them.

Loaders will help you customize how configuration discovery works. This is a list of objects that will be used to discover settings from different sources.

Loaders can be declared in the `Meta` class:

```
from classyconf import Configuration, Value, Environment


class AppConfig(Configuration):

    DEBUG = Value(default=False, help="Toggle debugging on/off.")

    class Meta:
        loaders=[Environment()]
```

In this case we are telling classyconf to only search for settings in the `os.enviroment` variables. I know, this is not very useful, and seems like an overkill. Let's override this default loaders list and introduce another loader to gather setting from a *.ini* file.

```
>>> from classyconf import Environment, IniFile
>>> config = AppConfig(loaders=[
...     Environment(),
...     IniFile("/etc/myapp/conf.ini", section="settings")
... ])
```

Now you might be asking, is this reading a file? do I have to create it? How do I access my settings?

Configuration discovery only happens when a `Value` setting is first accessed, so nothing gets evaluated until then.

The config instance can accessed as dict or object. Let's trigger a look up:

```
>>> config.DEBUG  # config["DEBUG"] also works!
False
```

Each loader is checked in the given order. In this case, we will first lookup each setting in the `os.enviroment` variables and, when not found, the declared *.ini* file (inside the `settings` section), but if this file doesn't exist or is broken, this loader is ignored.

If a setting is not found by any loader, the default value is returned, if set, or a `UnknownConfiguration` exception is thrown.

Now we all know that the industry practices have set different naming conventions for diffent configuration formats. Is it `camelCase` for *.json* files? Is it `UPPER_CASE` for the enviroment variables and `lower_case` for *.ini* files? Don't worry, classyconf has your back.

Most loaders include a `keyfmt` callable argument. This allows you to alter the name of the setting for each individual loader.

Let's customize this:

```
>>> from classyconf import EnvPrefix
>>> config = AppConfig(loaders=[
...     Environment(keyfmt=EnvPrefix("MY_APP_")),
...     IniFile("/etc/myapp/conf.ini", section="settings", keyfmt=str.lower)
... ])
```

Now if you access `config.DEBUG`, classyconf will first check for `MY_APP_DEBUG=xxx` in the `os.enviroment` but for `debug=xxx` in the `.ini` file.

See also:

The rationale for `keyfmt` is to follow the best practices for naming variables, and respecting namespaces for each source of config.

Read more at *Naming conventions and namespaces for settings*.

4.3 3. Extending settings

As you know, the same code might run in several different enviroments, like dev, staging, prod, etc.

Although `Configuration` classes can be extended to define new `Value` attributes or override them, the recomended way is to simply override the settings sources per enviroment.

```
from classyconf import EnvFile

class StagingConfig(AppConfig):
    class Meta:
        loaders = [EnvFile("staging.env")]
```

As we saw earlier, loaders can also be overridden at instantiation time.

```
from classyconf import Dict, EnvFile

test_config = AppConfig(loaders=[Dict({"DEBUG": True}), EnvFile("test.env")]
```

In the snippet above we used the *`Dict`* loader, which comes handy to ensure certain hardcoded settings always get picked up.

4.4 4. Inspecting settings

Later this object can be used to print configuration, this will evaluate every setting.

```
>>> config = AppConfig()
>>> print(config)
DEBUG=False - Toggle debugging on/off.
```

Or with `__repl__()` you get a preview of how it was instantiated.

```
>>> config = AppConfig()
>>> config
AppConf(loaders=[Environment()])
```

It can also be iterated. This gives you the field key and the `Value` instance for you to keep inspecting (this doesn't evaluate the setting).

```
>>> for setting in config:
...     print(setting)
...
('DEBUG', Value(key="DEBUG", help="Toggle debugging on/off."))
```

CHAPTER 5

Configuration Loaders

Loaders are in charge of loading configuration from various sources, like `.ini` files or *environment* variables, and expose configuration as a dict-like object. Loaders are ment to be chained, so that classyconf checks one by one for a given configuration variable.

If a loader doesn't find the configuration variable it raises a `KeyError` so that the next loader get's checked. If no loader returns any value, and no default value was set, an `UnknownConfiguration` exception is thrown.

Classyconf comes with some loaders already included in `classyconf.loaders`.

By default the library will check the environment with the *`Environment`* loader. You can change that behaviour, by customizing the loaders and the order in wich configuration discovery happens.

Loaders can be set in the `Meta` class when extending `Configuration` or passed as a param when instantiating it. The later takes precedence and overrides loaders defined in `Meta`. The order within the list of loaders matters and defines the lookup order.

```python
from classyconf import Configuration, IniFile, Environment, Value, EnvFile

class AppConfig(Configuration):

    class Meta:
        loaders = [
            Environment(),
            IniFile("/path/to/config.ini")
        ]

    DEBUG = Value(default=False)
    OTHER_CONFIG = Value(default=0)


# Checks for enviroment variables first
# Then lookup settings in the `config.ini` file
config = AppConfig()

# Override loaders and only lookup in the `.env` file
test_config = AppConfig(loaders=[EnvFile("/path/to/.env")])
```

5.1 Naming conventions and namespaces for settings

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

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

Since the environment is a global and shared dictionary, it is a good practice to also apply some prefix to each setting to avoid collisions with other known settings, like `LOCALE`, `TZ`, etc. This prefix works as a namespace for your app.

```
$ MY_APP_DEBUG=yes MY_APP_OTHER_CONFIG=10 ./app.py
```

but if you were to set this config in an `.ini` file, each setting should probably be in lower case, the namespace is implicit in the file path, i.e: `/etc/myapp/config.ini`.

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

Command line arguments have yet another conventions:

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

Classyconf let's you follow these aesthetics patterns by setting a `keyfmt` function when instantiating the loaders.

By default, the `Environment` is instantiated with `keyfmt=EnvPrefix('')` so that it looks for UPPER_CASED settings. But it can be easyly tweaked to address the prefix issue by using `keyfmt=EnvPrefix("MY_APP_")`, and look for MY_APP_UPPER_CASED to play nice with other env variables.

```
from classyconf import Configuration, IniFile, Environment, Value, EnvPrefix

class AppConfig(Configuration):

    class Meta:
        loaders = [
            Environment(keyfmt=EnvPrefix(prefix="MY_APP_")),
            IniFile("/etc/myapp/config.ini")
        ]

    DEBUG = Value(default=False)
    OTHER_CONFIG = Value(default=0)

config = AppConfig()

# looks for `MY_APP_DEBUG` in environment, then  `debug` in `settings` section of␣
↪config.ini
config.DEBUG
```

Keep reading to find out more about different loaders and their configurations.

5.2 Environment

class classyconf.loaders.**Environment**(*keyfmt=EnvPrefix("")*)

Get's configuration from the environment, by inspecting `os.environ`.

Parameters **keyfmt** (*function*) – A function to pre-format variable names.

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

Note: In the case of CLI apps, it would be recommended to set some sort of namespace so that you don't accidentally override other programs behaviour, like LOCALE or EDITOR, but instead MY_APP_LOCALE, etc. So consider using the `EnvPrefix("MY_APP_")` approach.

```
from classyconf import Configuration, Environment, Value

class AppConf(Configuration):
    debug = Value(default=False)


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

5.3 EnvFile

class `classyconf.loaders.`**`EnvFile`**(*filename='.env', keyfmt=EnvPrefix("")*)

Parameters

- **filename** (*str*) – Path to the `.env` file.
- **keyfmt** (*function*) – A function to pre-format variable names.

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 classyconf import Configuration, EnvFile, Value

class AppConf(Configuration):
    debug = Value(default=False)


config = AppConf(loaders=[EnvFile(file='.env', keyfmt=str.upper)])
config.debug  # will look for a `DEBUG` variable instead of `debug`
```

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

5.4 IniFile

class `classyconf.loaders.`**`IniFile`**(*filename, section='settings', keyfmt=<function IniFile.<lambda>>*)

Parameters

- **filename** (*str*) – Path to the `.ini/.cfg` file.
- **section** (*str*) – Section name inside the config file.
- **keyfmt** (*function*) – A function to pre-format variable names.

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.

5.5 CommandLine

class classyconf.loaders.**CommandLine**(*parser*, *get_args=<function get_args>*)

Extract configuration from an `argparse` parser.

Parameters

- **parser** (*argparse.ArgumentParser*) – An *argparse* parser instance to extract variables from.
- **get_args** (*function*) – A function to extract args from the parser.

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

```
import argparse
from classyconf import Configuration, Value, NOT_SET, CommandLine


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

class AppConf(Configuration):
    DEBUG = Value(default=False)

config = AppConf(loaders=[CommandLine(parser=parser)])
print(config.DEBUG)
```

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 classyconf, you must set one. But that would break the discoverability chain that classyconf encourages. So by setting this special default value, you will allow classyconf to keep the lookup going.

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

5.6 Dict

class classyconf.loaders.**Dict**(*values_mapping*)

Parameters values_mapping (*dict*) – A dictionary of hardcoded settings.

This loader is great when you want to pin certain settings without having to change/override other loaders, files or defaults. It really comes handy when you are extending a `Configuration` class.

```
from classyconf import Configuration, Value, IniFile, Dict

class AppConfig(Configuration):
    class Meta:
```

(continues on next page)

(continued from previous page)

```
        loaders = [IniFile("/opt/myapp/config.ini"), IniFile("/etc/myapp/config.ini")]

    NUMBER = Value(default=1)
    DEBUG = Value(default=False)
    LABEL = Value(default="foo")
    OTHER  = Value(default="bar")


class TestConfig(AppConfig):
    class Meta:
        loders = [Dict({"DEBUG": True, "NUMBER": 0})]
```

5.7 RecursiveSearch

class classyconf.loaders.**RecursiveSearch**(*starting_path=None, filetypes=((’.env’, <class ’classyconf.loaders.EnvFile’>), ((’*.ini’, ’*.cfg’), <class ’classyconf.loaders.IniFile’>)), root_path=’/’*)

Parameters

- **starting_path** (*str*) – The path to begin looking for configuration files.
- **filetypes** (*tuple*) – tuple of tuples with configuration loaders, order matters. Defaults to `(('*.env', EnvFile), (('*.ini', *.cfg',), IniFile)`
- **root_path** (*str*) – Configuration lookup will stop at the given path. Defaults to the current user directory

This loader tries to find `.env` or `*.ini|*.cfg` files and load them with the *EnvFile* and *IniFile* loaders respectively.

It will start looking at the `starting_path` directory for configuration files and walking up the filesystem tree until it finds any or reaches the `root_path`.

> **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 classyconf import config
```

(continues on next page)

(continued from previous page)

```
from classyconf.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.

Suppose 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 classyconf import Configuration
from classyconf.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`, otherwise it raises an `InvalidPath` exception:

```
from classyconf.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")
```

5.8 Writing your own loader

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

class classyconf.loaders.**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, classyconf knows that it has to keep looking down the loaders chain for a specific config.

```
import yaml
from classyconf.loaders import AbstractConfigurationLoader


class YamlFile(AbstractConfigurationLoader):
    def __init__(self, filename, keyfmt=str.lower):
```

(continues on next page)

(continued from previous page)

```
        self.filename = filename
        self.config = None
        self.keyfmt = keyfmt

    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 self.keyfmt(item) in self.config

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

        return self.config[self.keyfmt(item)]

    def reset(self):
        self.config = None
```

Then configure classyconf to use it.

```
from classyconf import Configuration

class AppConf(Configuration):
    class Meta:
        loaders = [YamlFile('/path/to/config.yml')]
```

CHAPTER 6

Casts

Loaders gather configuration from different sources, but that configuration usually is digested as strings and it might not be the correct type you need in your programs.

That’s why you can specify cast functions for each individual setting.

```
from classyconf import Configuration, Value, Environment, as_boolean
from decimal import Decimal


class Config(Configuration)
    class Meta:
      loaders = [Environment()]

    BASE_PRICE = Value(default=Decimal(10), cast=Decimal, help="Base product price.")
    DEBUG = Value(default=False, cast=as_boolean, help="Enables debug mode.")
```

6.1 Buitin Casts

In `classyconf.casts` you can find some common cast functions that ship by default. If the cast fails it will rise an `InvalidConfiguration` exception.

6.1.1 Boolean

Converts values like `On|Off`, `1|0`, `yes|no`, `y|n`, `true|false`, `t|f` into booleans.

These options can be also extended by passing an extra True/False mapping.

```
from classyconf import Boolean

boolean = Boolean({"sim": True, "não": False})
assert boolean("sim")
```

(continues on next page)

(continued from previous page)

```
assert boolean("yes")
assert not boolean("não")
assert not boolean("no")
```

6.1.2 List

Converts comma separated strings into lists by default.

This cast can accept other separators.

```
from classyconf import List

as_list = List(delimiter=";")
assert as_list("1; 2;3; ' 4; ';") == ['1', '2', '3', "' 4; '"]
```

6.1.3 Tuple

Same as `List`, but converts comma separated strings into tuples.

```
from classyconf import Tuple

as_tuple = Tuple()
assert as_tuple("a, b, c") == ['a', 'b', 'c']
```

6.1.4 Option

Gets a return value based on specific options:

```
from classyconf import Option

choices = {
    'option1': "asd",
    'option2': "def",
}
option = Option(choices)

assert option("option1") == "asd"
assert option("option2") == "def"
```

6.1.5 Evaluate

Safely evaluate strings with Python literals to Python objects (alias to Python's `ast.literal_eval`).

```
from classyconf import evaluate


assert evaluate("None") is None
```

6.1.6 Identity

It is the no-op type of cast, returns anything it receives as is.

```
from classyconf import Identity


as_is = Identity()

assert as_is("None") is "None"
```

6.2 Shortcuts for standard casts

`classyconf` ships with cast instances already configured for convenience.

```
from classyconf import as_list, as_tuple, as_boolean, as_option, as_is, evaluate
```

They are pretty much self explanatory, but `as_is` is an instance of `Identity` cast.

6.3 Custom casts

You can implement your own custom casting function by passing any callable:

```
from classyconf import Configuration, Environment


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


class Config(Configuration)
    class Meta:
      loaders = [Environment()]

    NUMBERS = Value(default="1;2;3", cast=number_list, help="Semicolon separated␣
↪numbers.")
```

6.4 Implicit casts

`classyconf` tries to provide some sensible default casts based on the default's value type.

1. If the user provides a cast function, we use that one, no questions asked.
2. If the user sets a default that is an `int`, `str`, `boolean`, `float`, etc, and doesn't set a cast function, we can set a default one: `int()`, `str()`, `as_boolean()` and `float()` respectively.
3. If the user doesn't set a default value we use the Identity cast (`as_is()`).
4. If the user sets a non callable value as cast, we raise a `TypeError` exception.

So following the first example:

```
from classyconf import Configuration, Value, Environment
from decimal import Decimal


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


class Config(Configuration)
    class Meta:
      loaders = [Environment()]

    NUMBERS = Value("NUMBERS", default="1;2;3", cast=number_list)  # cast is number_
↪list
    BASE_PRICE = Value(default=Decimal(10), help="Base product price.")  # cast is␣
↪Decimal
    DEBUG = Value(default=False, help="Enables debug mode.")  # cast is as_boolean
```

CHAPTER 7

Advanced

7.1 Caching

Everytime you access a Value, classyconf peek on the loaders one by one until a loader returns a setting. If a setting is not found by any loader, the default value is returned, if set, or a `UnknownConfiguration` exception is thrown.

If the loaders chain is long or you are accessing the settings too often, there is an optimization you can use, which is the cache property:

```python
from classyconf import Configuration, Value, Environment, EnvFile, IniFile


class AppConfig(Configuration):

    DEBUG = Value(default=False, help="Toggle debugging on/off.")

    class Meta:
        loaders=[Environment(), EnvFile(".env"), IniFile("config.ini")]
        cache = True
```

This property can also be set at runtime:

```
>>> config = AppConfig(cache=True)
>>>
```

It will make the lookup to have a `O(1)` performance the second time it is accesed.

7.2 Reloading new settings

Typically when files get parsed, their values are kept in an internal cache by each loader. If at some point you want to pickup new values, for example when using a long running daemon, call the reset method.

```python
import signal

config = AppConfig()

def signal_handler(signum, frame):
    if signum == signal.SIGHUP:  # kill -1 <pid>
        config.reset()

signal.signal(signal.SIGHUP, signal_handler)

if __name__ == '__main__':
    main(config)
```

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 When should I use configuration files?

Environment variables shouldn't hold sensitive data, there are potential security issues:

1. Accidental leaks via logging or error reporting services.
2. Child process inheritance.

Command line arguments are great for exploring the possibilities of an app, but passing lot's of arguments either in the short `-s` or long `--more-verbose` formats can be cumbersome.

Sometimes files are more convinient and documenting than command line arguments or env vars. Some file formats allow for comments and are great as templates to build upon.

If your app is a long running process, like a webserver, you can issue a `SIGHUP` signal so that it reloads it's config from files. Env vars and command line arguments cannot be easily changed from the outside after startup.

8.3 Why are executable config files a bad idea?

Executable files can be used as config sources like `.vimrc`, `Vagrantfile`, etc. This approach has some drawbacks.

First, your users now need to learn a new programming language, just to configure your application. Some apps (like the suckless bundle) go as far as requiring you to patch and compile your app to change it's configuration.

And second, your configuration is no longer hierarchical, your application cannot extract configuration from different sources by executing different files, because you cannot know in advance what is being executed. So you typically end up with one single executable file as config that takes care of everything.

On the other hand, classyconf encourages traditional formats for configuration, like enviroment variables or ini files. The best way to think of configuration is as a set of key/value dicts that need to be merged into a single config dict. No need to get fancy.

8.4 Is classyconf tied to Django or Flask?

No, classyconf was designed to be framework agnostic, can be used for web, CLI or GUI applications.

8.5 Why create a library similar to prettyconf or goodconf instead of using it?

Although *prettyconf* is great and very flexible, I don't like that the *config("debug")* call isn't lazy, so putting it into a class isn't enough:

```
from prettyconf import config

class MyConfig():
    debug = config("debug")    # this is evaluated when this module is loaded
```

I also didn't like the default *RecursiveSearch* that it provides and I also needed to implement many changes and move fast to see what would work.

I've made several contributions to *prettyconf* and even have a talk about it, but I needed to change its behaviour, break things and move fast. This is backward incompatible, so, it could break software that relies on the old behaviour.

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

Other libraries had other issues:

- Were tied to a specific web app framework.
- Didn't allow you to specify configuration sources and their hierarchy.
- Had a global configuration object, or made it really hard to override specific configuration when writing tests.
- Settings were eagerly evaluated.
- Had no facilities for auto-generating configuration documentation or inspecting it.

Classyconf is classy, it's pretty, it's good.

8.6 How does classyconf 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, classyconf does not populate the `os.environ` dictionary, because it is designed to discover configuration from diferent sources, the environment being just one of them.

Other similar projects are direnv and envdir to load environment variables from directories and files.

In case you are running your app as an systemd unit, there is a section to directly list the env vars or to suply a env file.

8.7 What are some useful third-parties casts for Django?

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 9

Changelog

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

This project adheres to Semantic Versioning.

9.1 0.5.2

- Improved `pyproject.toml` metadata.

9.2 0.5.1

- Added python 3.9 support.

9.3 0.5.0

- Migrated from `setup.py` to `pyproject.toml`.
- Refactored Makefile and added tbump directive.

9.4 0.4.0

- Changed references to the obsolete `ClassyConf` object in docs.
- Exposed `CommandLine` loader in the library import root.

9.5 0.3.0

- Added keyword only flag for `Value` and `Configuration` classes.
- Added cache option for `Configuration` class.

9.6 0.2.0

- Replaced `env_prefix` with the `EnvPrefix` class.
- Replaced coveralls with codecov.
- Replaced TravisCI with Github Actions.

9.7 0.1.0

- First version

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index