Manyconfig

Oct 11, 2018

Contents:

CHAPTER 1

Basic usage

In most cases, building a MetaConfig is meant to be declarative: You only declare and assemble objects, without the need to code the logic.

It integrates Marshmallow, which is a plentyful of features in itself, so look into it! (for example, use marshmallow for default values and runtime transformations)

1.1 Single source

For example, if you have a single json file named *foo.json* that contains your configuration, you can retrieve it by running this snippet:

```
from manyconfig import FileMetaConfig
metaconfig = FileMetaConfig("json", "foo.json")
config = metaconfig.load()
```

Or, if your configuration resides in the environment and all its keys begins with *APPLICATION_*, you could do:

```
from manyconfig import EnvironmentMetaConfig
metaconfig = EnvironmentMetaConfig("APPLICATION_")
config = metaconfig.load()
```

1.2 Multiple sources

Now, the two examples above can easily be dealt with without using Manyconfig. But it's when there's multiple sources of configuration that it comes in handy.

Say, you use both of above configuration sources, with the environment overriding the file. It's still easy enough but think about configurations of programs like Bash, or flake8. They are far more complex but could be handled with MetaConfig blocks.

Let's come back to our example and see how it looks:

```
from manyconfig import EnvironmentMetaConfig, FileMetaConfig, ManyConfig
env_metaconfig = EnvironmentMetaConfig("APPLICATION_")
file_metaconfig = FileMetaConfig("json", "foo.json")
metaconfig = ManyConfig(file_metaconfig, env_metaconfig)
config = metaconfig.load()
```

1.3 Validating configuration

Even though it's not a dependency, Manyconfig is built with Marshmallow in mind. You can define a Schema class and use it to validate the configuration.

This is integrated in Manyconfig so no need to do it yourself. Just pass the instanciated schema to the MetaConfig block when you build it.

```
from manyconfig import FileMetaConfig
from marshmallow import Schema
from marshmallow.fields import Integer, String


class ConfigSchema(Schema):
    database = String()
    verbosity_level = Integer()


metaconfig = FileMetaConfig("json", "foo.json", schema=ConfigSchema())
config = metaconfig.load()
# yields something like {"database": "sql://foo", "verbosity_level": 42}
```

It can also be overriden at load time:

```
config = metaconfig.load(schema=SomeOtherSchema())
```

CHAPTER 2

Extend Manyconfig

2.1 Adding a file format

You can easily add support for a file format by adding a parser to Manyconfig.

Suppose you have a format with key/value pairs separated by an equal sign. This is how you would add support for your format:

```
from manyconfig import format_parsers


@format_parsers.add("key=value")
def parse(file_object):
    config = {}
    for line in file_object:
        key, value = line.split("=")
        config[key] = value
    return config


metaconfig = FileMetaConfig("key=value", "config")
```

You need to add the format parser before instanciating the FileMetaConfig.

2.2 Extend MetaConfig

To extend the *MetaConfig* class, you need to override the `_load()` method. It should return a dictionnary.

You can also override the *__init__* method to capture arguments during instanciation, but please forward keyword-arguments to *super*.

CHAPTER 3

API Reference

3.1 Config

exception manyconfig.config.**InvalidConfigException**(*message*, *errors*)
Exception raises when a configuration is invalid.

class manyconfig.config.**MetaConfig**(*silent=False*, *schema=None*)
Base class for metaconfigurations.

A schema passed at instantiation override any schema set at level class.

Parameters

- **silent** (*bool*) – Don't raise exceptions on invalid configurations
- **schema** – A marshmallow schema to validate loaded configuration

load(*schema=None*)
Load the configuration

Return dict Dict representing the configuration

schema = None
A marshmallow schema to validate loaded configuration

3.2 Environment

class manyconfig.environment.**EnvironmentMetaConfig**(*namespace*, ***kwargs*)
Pull configuration from environment

An environment variable is considered to be in a namespace if it begins by it. That is, FOO_BAR is part of the FOO_ namespace.

All environment variables of the given namespace will be collected, and the namespace removed from its beginning. It is then inserted in the configuration with its value.

Parameters **namespace** – The namespace to pull from

3.3 File

class manyconfig.file.**DecoratorDict**

class manyconfig.file.**FileMetaConfig**(*format*, *filepath*, *binary=False*, ***kwargs*)
Pull configuration from a file.

Parameters

- **filepath** – the path of the configuration file.
- **binary** (*bool*) – Open the file in binary mode.

exception manyconfig.file.**InvalidFormatError**

manyconfig.file.**parse_ini**(*file_object*)
Parse the INI in the file.

3.4 Manyconfig

class manyconfig.manyconfig.**AnyConfig**(**metaconfigs*, ***kwargs*)
Pull configuration from the first existing source

It will iter through the different MetaConfig given and yield the first non-empty valid configuration.

Parameters **metaconfigs** – A list of configurations to pull values from

class manyconfig.manyconfig.**ManyConfig**(**metaconfigs*, ***kwargs*)
Pull configuration from many others.

This class is the real plus-value of ManyConfig. It takes some configuration sources, load them and merge them in a single configuration.

Configurations are loaded in the given order, and new values for the same configuration key overrides older ones.

This allow implementation of Bash-like configurations, where multiple files are read and each one take precedence over the last one.

Parameters **metaconfigs** – A list of configurations to pull values from

manyconfig.manyconfig.**merge**(**configs*)
Merge several dicts to produce one.

If a key is present in two or more dicts, the value of the lattest occurence is took.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index