
cfgtree Documentation

Release

Gaetan Semet

Jan 28, 2018

Contents:

1	Overview	1
1.1	Configuration Tree Description	1
1.2	Configuration Storage	1
1.3	Access to settings	2
2	Model	3
3	CfgTree API Reference	5
3.1	Model Recipes	5
3.2	Types	7
3.3	Storages	10
3.4	Command line parsers	11
4	Release Notes	13
4.1	1.0.0	13
4.2	0.1.1	14
4.3	0.1.0	14
5	Introduction	17
6	Similar opensource projects	19
	Python Module Index	21

1.1 Configuration Tree Description

Configuration hierarchy is to be described in a `cfgtree.EnvironmentConfig` inherited instance, inside the member `.cfgtree`, using helper classes such as `StringCfg`, `IntCfg`, `UserCfg` or `PasswordCfg`. Each setting can be set by environment variable, command line parameter or by the storage file(s) itself.

Let's take an example of an item defined at the first level of the hierarchy. It is defined as a `IntCfg` with name `count`. User can set this setting by:

- environment variable `APPLICATIONNAME_COUNT` (where `APPLICATIONNAME` is an optional, developer-defined prefix added to every environment variable of the application to avoid conflicts)
- command line argument `--count`
- item `count` at the first level of a json file

Hierarchical structure is reflected in these different ways, to avoid conflicts. Now, let's imagine the 'count' setting is set in a group called 'general':

- environment variable is: `APPLICATIONNAME_GENERAL_COUNT`
- command line argument is: `--general-count`
- Json has a first level named `general`, and inside one of the items is called `count`:

```
{
  "general": {
    "count": 1
  }
}
```

1.2 Configuration Storage

The trivial storage is a simple json file. The complete settings are placed inside it, such as:

```
{
  'setting1': 'value1',
  'setting2': 'value2',
  'setting3': 'value3',
}
```

But developer may want to organize in a more hierarchical structure, splitting into different files, etc.

cfgtree allows complete customization of the file storage, developers can even develop their own.

Another typical file format for configuration is YAML, which is more human readable and allow inserting comments and so. INI files is often found as configuration format, or TOML.

But, ultimately, all file formats actually store settings in hierarchical configuration.

Current Support:

- single Json file

Future support:

- Yaml file (with inplace save keeping comments and overall organization)
- Set of Yaml files
- Configuration server

1.3 Access to settings

In your application, an xpath-like syntax allows you to reach any item of the configuration: `<key1>.<key2>.<key3>.<item>`. See the documentation for full explanation.

CHAPTER 2

Model

The core part of `cfgtree` is the definition of the model. A “model” is a Python dictionary that describes the hierarchical organization of your settings, like a data schema.

For example, if I want to organize my settings into two groups, one “general” and one “others”, I would place the descriptions in a model such as:

```
model = {
    "general" {
        # ...
    },
    "others": {
        # ...
    },
}
```

Here an example of a comdel:

```
{
    "configfile": ConfigFileCfg(
        default_filename=config,
        long_param="--config-file",
        summary="Config file"),
    "version": ConfigVersionCfg(),
    "group1": {
        "string_opt": StringCfg(
            short_param='-s',
            long_param="--string-opt",
            summary='Help msg string'),
        "int_opt": IntCfg(
            short_param='-i',
            long_param="--int-opt",
            summary='Help msg int'),
        "float_opt": FloatCfg(
            short_param='-f',
            long_param="--float-opt",
```

```

        summary='Help msg float'),
    "bool_opt": BoolCfg(
        short_param='-b',
        long_param="--bool-opt",
        summary='Help msg bool'),
    "list_opt": ListOfStringCfg(
        short_param='-l',
        long_param="--list-opt",
        summary='Help msg lst'),
    "dict_opt": {
        "key1": StringCfg(summary='Help msg string'),
        "key2": StringCfg(summary='Help msg string'),
    }
}

```

This model matches a configuration JSON file such as:

```

{
  "version": 1,
  "group1": {
    "string_opt": "a string",
    "int_opt": 123,
    "float_opt": 2.0,
    "bool_opt": true,
    "list_opt": [
      "a",
      "b",
      "c"
    ],
    "dict_opt": {
      "key1": "val1",
      "key2": "val2"
    }
  }
}

```

Or this TOML file:

```

version = 1

[group1]
string_opt = "a string"
int_opt = 123
float_opt = 2.0
bool_opt = true
list_opt = [ "a", "b", "c", ]

[group1.dict_opt]
key1 = "val1"
key2 = "val2"

```


3.1 Model Recipes

3.1.1 Anyconfig Model Recipe

This is the easiest way of using cfgtree. Anyconfig abstract the loading of the configuration file, handling a large variety of file format transparently.

Two variant are provided:

- `AnyConfigModel`: that can read enviroment variables and configuration file
- `AnyConfigCliModel`: that tries to parse the command line argument

3.1.2 Base Model

`cfgtree.version()`

Returns the PEP440 version of the cfgtree package

class `cfgtree.ConfigBaseModel` (*model=None, environ_var_prefix=None, storage=None, cmd_line_parser=None*)

Main configuration class

You need to inherit from this base class and implement the following members:

- `model`: hierarchical dictionary representing the configuration tree
- `environ_var_prefix`: prefix for environment variable, to avoid conflicts
- `storage`: class to use for configuration storage
- `cmd_line_parser`: which command line argument parser to use

Usage:

```

from cfgtree import ConfigBaseModel
from cfgtree.cmdline_parsers.argparse import ArgparseCmdlineParser

class MyAppConfig(ConfigBaseModel):

    # All environment variables should start by MYAPP_ to avoid collision with
    # other
    # application's or system's environment variables
    environ_var_prefix = "MYAPP_"

    # My configuration should be read from a single JSON file
    storage = JsonConfigFile(
        # User can overwrite the configuration file name with this environment
        # variable
        environ_var="MYAPP_COMMON_CONFIG_FILE",
        # or this command line parameter
        long_param="--config-file",
        short_param="-c",
        # If not set, search for the `config.json` file in the current directory
        default_filename="config.json",
    )

    # Use `argparse` to parse the command line
    cmd_line_parser = ArgparseCmdlineParser()

    # Here is the main settings model for the application
    model = {
        # Redefine configfile with ConfigFileCfg so that it appears in --help
        "configfile": ConfigFileCfg(long_param="--config-file",
                                     short_param="-c",
                                     summary="Configuration file"),

        # can holds a version information for the storage file
        "version": VersionCfg(),

        "general": {
            "verbose": BoolCfg(short_param='-v',
                               long_param="--verbose",
                               summary='Enable verbose output logs'),
            "logfile":
                StringCfg(short_param="-l",
                          summary='Output log to file'),
        },
    }

    # As an example, the 'verbose' setting can then be configured by:
    # - environment variable "MYAPP_GENERAL_VERBOSE"
    # - command line option ``--verbose``
    # - key ``general.verbose`` in configuration file ``config.json``

    cfg = MyAppConfig()
    # Read
    cfg.get_cfg_value("group1.dict_opt.key1")
    # Write
    cfg.set_cfg_value("group1.dict_opt.key1", "newval")

```

cmd_line_parser = None

```

environ_var_prefix = None
find_configuration_values (argv=None)
    Main cfgtree entrypoint
get_cfg_value (xpath, default=None)
    Get a value from cfgtree.
json (safe=False)
    Dumps current configuration tree into a human readable json
model = None
save_configuration ()
    Save configuration to storage
set_cfg_value (xpath, value)
    Set a value in cfgtree.
storage = None

```

3.2 Types

Here are the type you can use in your model

```

class cfgtree.types._CfgBase (long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>)

```

```

action
arg_type = None
cfgfile_value
    Return value to save in config file.
cmd_line_name
default = None
environ_var
environ_var_prefix = None
get_cmd_line_params ()
ignore_in_args = False
ignore_in_cfg = False
ignore_in_envvars = False
long_param
metavar
n_args
name = None
read_environ_var ()
safe_value
    Return value as a string without compromising information.

```

set_value (*value*)
 Setter method used in *set_node_by_xpath*.

value

xpath = None

class `cfgtree.types.BoolCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Boolean value

Handle automatic integer conversion Example:

```
True
```

class `cfgtree.types.ConfigFileCfg` (**args, default_filename=None, **kwargs*)

Configuration file to load rest of configuration

Use to tell to your storage where the rest of the configuration should be used

Example:

```
"/path/to/my/config.json"
```

class `cfgtree.types.ConfigVersionCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Version of the configuration storage.

It does not present an environment variable nor a command line argument

Example:

```
"1.2.3"
```

class `cfgtree.types.DirNameCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Directory name

Example:

```
"/path/to/existing/folder"
```

class `cfgtree.types.FloatCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Float or double value

Example:

```
1, 23
```

class `cfgtree.types.HardcodedCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Placeholder only used to store application value.

It does not present an environment variable nor a command line argument.

class `cfgtree.types.IPCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

IPv4 or IPv6 value

Example:

```
"192.168.0.1"
```

class `cfgtree.types.IntCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Integer value

Example:

```
123
```

class `cfgtree.types.ListOfStringCfg` (**args, **kwargs*)
Comma separated list of string (1 argument).

Example:

```
"a,b,c,d"
```

cfgfile_value

Return value to save in config file.

class `cfgtree.types.MultiChoiceCfg` (**args, choices=None, **kwargs*)
Let user choose one or more value between several string value

Example:

```
"a_value"
```

class `cfgtree.types.PasswordCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Password value

This can be used to handle value while limiting its exposition

safe_value

Hide password in logs.

class `cfgtree.types.PortCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

Port value, with range from 1 to 65535

Example:

```
49670
```

class `cfgtree.types.SingleChoiceCfg` (**args, choices=None, **kwargs*)
Let user choose one value between several string value

Example:

```
"a_value"
```

class `cfgtree.types.StringCfg` (*long_param=None, description=None, short_param=None, summary=None, required=False, default=<object object>*)

String value

Example:

```
"a value"
```

3.3 Storages

Cfgtree does not make any assumption of the way settings are stored, appart from the fact they are all organized in a hierarchicla structure.

Some common storage format are provided out of the box by cfgtree, but developers can easily implement their own configuration file format.

3.3.1 AnyConfig handled configuration file

anyconfig is a library abstracting the load of a variety of configuration file format (ini, json, yaml,...)

3.3.2 Single Json file

```
class cfgtree.storages.json.JsonFileConfigStorage(environ_var=None,
                                                    long_param=None,
                                                    short_param=None,          de-
                                                    fault_filename=None)
```

Settings are stored in a single JSON file

Example:

```
{
  'version': 1,
  'general': {
    'verbose': true ,
    'logfile': 'logfile.log'
  }
}
```

Usage:

```
class MyAppConfig(ConfigBaseModel) :

    environ_var_prefix = "MYAPP_"

    storage = JsonFileConfigStorage(
        environ_var="MYAPP_COMMON_CONFIG_FILE",
        long_param="--config",
        short_param="-c",
        default_filename="config.json",
    )

    cmd_line_parser = # ...

    model = {
        # repeat in model so the arguments in described in --help
        "configfile": ConfigFileCfg(long_param="--config-file", summary="Config_
↪file"),
        # ...
    }
```

default_filename = None

environ_var = None

find_default_filename (*model*)

json_configstorage_default_filename = None

Default filename for the configuration file

Example:

```
myconfig.json
```

json_configstorage_envIRON_var = None

Environment variable to set the configuration file name

Example:

```
DOPPLERR_COMMON_CONFIG_FILE="myconfig.json"
```

json_configstorage_long_param = None

Short parameter to specify the configure file name

Example:

```
--config-file myconfig.json
```

json_configstorage_short_param = None

Short parameter to specify the configure file name

Example:

```
-g myconfig.json
```

load_bare_config (*config_file_path*: *pathlib.Path*)

long_param = None

short_param = None

3.3.3 Base class

class `cfgtree.storages.ConfigBaseStorage`

find_storage (*model*, *argv*=None)

Find the storage location and load the bare configuration

get_bare_config_dict ()

Returns the bare configuration tree

save_bare_config_dict (*bare_cfg*)

Return the bare configuration into a dict

3.4 Command line parsers

class `cfgtree.cmdline_parsers.CmdlineParsersBase`

parse_cmd_line (*model*, *argv*=None)

3.4.1 Base class

```
class cfgtree.cmdline_parsers.CmdlineParsersBase  
  
    parse_cmd_line (model, argv=None)
```


4.1 1.0.0

4.1.1 Release Summary

This is the first 1.0 release. It includes a major rework of the cfgtree internal code and API.

4.1.2 Upgrade Notes

- rename `EnvironmentConfig` to `ConfigBaseModel`
- add type: `FloatCfg`
- rename type `FileVersionCfg` to `ConfigVersionCfg`
- remove short parameter for type definition:
 - `l`` to ```long_param`
 - `s`` to ```short_param`
 - `h`` to ```summary`
- Please note the API is not compatible with previous version. You need to manually update your application.
- `EnvironmentConfig` is renamed `ConfigBaseModel`.
`config_storage` is renamed `storage`.
`cfgtree` is renamed `model`.

From:

```
from cfgtree.cfgtree import EnvironmentConfig

class MyAppConfig(EnvironmentConfig):
```

```
config_storage = ...

cfgtree = ...

...
```

To:

```
from cfgtree import ConfigBaseModel

class MyAppConfig(ConfigBaseModel):

    storage = ...

    model = ...

...
```

- `DopplerrJsonConfigFile` has been moved to `cfgtree.storages.json` and its fields has been renamed.
- Type short argument `h=`, `l=`, `s=` has been renamed to more meaningful name.
 - `l=: long_param`
 - `s=: short_param`
 - `h=: summary`
 - `r=: required`

4.2 0.1.1

4.2.1 Release Summary

New version increase the user documentation.

4.2.2 New Features

- User documentation at `readthedocs`
- API documentation

4.3 0.1.0

4.3.1 Release Summary

First release of `cfgtree`. It only support basic features, that was needed for the main project I was using it internally.

4.3.2 New Features

- Only simple feature are supported on this version, single json configuration file, `argparse` command line parser and a bunch of setting types.

- Note the API may change on the next version.
- **Current support:**
 - File storage: - json (JsonFileConfigStorage)
 - Command line parser: - argparse
 - Settings types: - BoolCfg - ConfigFileCfg - DirNameCfg - HardcodedCfg - IntCfg - ListOfStringCfg - MultiChoiceCfg - PasswordCfg - SingleChoiceCfg - StringCfg - UserCfg

CHAPTER 5

Introduction

`cfgtree` provides an easy yet comprehensive way of describing, storing and parsing a user configuration for you Python application.

It requires the following acknowledgments:

- Application settings actually represents a hierarchical structure, they can be organized into group of settings, subgroups, and they entirely depends on the application itself.

This structure is called in Cfgtree a “bare configuration”, or “configuration tree”.

- User settings may come from different inputs:
 - environment variables (12 factors approach). Example: `MYAPP_VERBOSE=1`.
 - command line argument. Example: `--verbose`
 - configuration storage such as file (JSON, YAML, INI) or even configuration server. Example:

```
{  
  "verbose": true  
}
```

This allows you to define once your settings structure, and let the user of your application define the settings through different ways. For instance, your application can read some settings through command line arguments, which is very useful for containerization of your application. It is indeed recommended by [Heroku's 12 Factor Good Practices](#).

Describing your configuration through a model also allows to have a configuration validator without having to maintain both a file schema (ex: JSON Schema) and the parsing logic code.

CHAPTER 6

Similar opensource projects

- Openstack's [Olso.config](#)

C

`cfgtree`, [5](#)

`cfgtree.cmdline_parsers`, [11](#)

`cfgtree.types`, [8](#)

Symbols

`_CfgBase` (class in `cfgtree.types`), 7

A

`action` (`cfgtree.types._CfgBase` attribute), 7
`arg_type` (`cfgtree.types._CfgBase` attribute), 7

B

`BoolCfg` (class in `cfgtree.types`), 8

C

`cfgfile_value` (`cfgtree.types._CfgBase` attribute), 7
`cfgfile_value` (`cfgtree.types.ListOfStringCfg` attribute), 9
`cfgtree` (module), 5
`cfgtree.cmdline_parsers` (module), 11
`cfgtree.types` (module), 8
`cmd_line_name` (`cfgtree.types._CfgBase` attribute), 7
`cmd_line_parser` (`cfgtree.ConfigBaseModel` attribute), 6
`CmdlineParsersBase` (class in `cfgtree.cmdline_parsers`), 11, 12
`ConfigBaseModel` (class in `cfgtree`), 5
`ConfigBaseStorage` (class in `cfgtree.storages`), 11
`ConfigFileCfg` (class in `cfgtree.types`), 8
`ConfigVersionCfg` (class in `cfgtree.types`), 8

D

`default` (`cfgtree.types._CfgBase` attribute), 7
`default_filename` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 10
`DirNameCfg` (class in `cfgtree.types`), 8

E

`environ_var` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 10
`environ_var` (`cfgtree.types._CfgBase` attribute), 7
`environ_var_prefix` (`cfgtree.ConfigBaseModel` attribute), 6
`environ_var_prefix` (`cfgtree.types._CfgBase` attribute), 7

F

`find_configuration_values()` (`cfgtree.ConfigBaseModel` method), 7
`find_default_filename()` (`cfgtree.storages.json.JsonFileConfigStorage` method), 10
`find_storage()` (`cfgtree.storages.ConfigBaseStorage` method), 11
`FloatCfg` (class in `cfgtree.types`), 8

G

`get_bare_config_dict()` (`cfgtree.storages.ConfigBaseStorage` method), 11
`get_cfg_value()` (`cfgtree.ConfigBaseModel` method), 7
`get_cmd_line_params()` (`cfgtree.types._CfgBase` method), 7

H

`HardcodedCfg` (class in `cfgtree.types`), 8

I

`ignore_in_args` (`cfgtree.types._CfgBase` attribute), 7
`ignore_in_cfg` (`cfgtree.types._CfgBase` attribute), 7
`ignore_in_envvars` (`cfgtree.types._CfgBase` attribute), 7
`IntCfg` (class in `cfgtree.types`), 9
`IPCfg` (class in `cfgtree.types`), 8

J

`json()` (`cfgtree.ConfigBaseModel` method), 7
`json_configstorage_default_filename` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 11
`json_configstorage_environ_var` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 11
`json_configstorage_long_param` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 11
`json_configstorage_short_param` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 11

JsonFileConfigStorage (class in `cfgtree.storages.json`), 10

L

ListOfStringCfg (class in `cfgtree.types`), 9

`load_bare_config()` (`cfgtree.storages.json.JsonFileConfigStorage` method), 11

`long_param` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 11

`long_param` (`cfgtree.types._CfgBase` attribute), 7

M

`metavar` (`cfgtree.types._CfgBase` attribute), 7

`model` (`cfgtree.ConfigBaseModel` attribute), 7

MultiChoiceCfg (class in `cfgtree.types`), 9

N

`n_args` (`cfgtree.types._CfgBase` attribute), 7

`name` (`cfgtree.types._CfgBase` attribute), 7

P

`parse_cmd_line()` (`cfgtree.cmdline_parsers.CmdlineParsersBase` method), 11, 12

PasswordCfg (class in `cfgtree.types`), 9

PortCfg (class in `cfgtree.types`), 9

R

`read_environ_var()` (`cfgtree.types._CfgBase` method), 7

S

`safe_value` (`cfgtree.types._CfgBase` attribute), 7

`safe_value` (`cfgtree.types.PasswordCfg` attribute), 9

`save_bare_config_dict()` (`cfgtree.storages.ConfigBaseStorage` method), 11

`save_configuration()` (`cfgtree.ConfigBaseModel` method), 7

`set_cfg_value()` (`cfgtree.ConfigBaseModel` method), 7

`set_value()` (`cfgtree.types._CfgBase` method), 7

`short_param` (`cfgtree.storages.json.JsonFileConfigStorage` attribute), 11

SingleChoiceCfg (class in `cfgtree.types`), 9

`storage` (`cfgtree.ConfigBaseModel` attribute), 7

StringCfg (class in `cfgtree.types`), 9

V

`value` (`cfgtree.types._CfgBase` attribute), 8

`version()` (in module `cfgtree`), 5

X

`xpath` (`cfgtree.types._CfgBase` attribute), 8