# cfgtree Documentation

*Release 1.2.0.dev5*

**Gaetan Semet**

**Jan 29, 2018**

# Contents:

CHAPTER 1

# Overview

## 1.1 Configuration Tree Description

Configuration hierarchy is to be described in a `cfgtree.ConfigBaseModel` inherited instance, inside the member `.model`, using helper classes such as `StringCfg`, `IntCfg`, `IPCfg` 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:

```
{
    "group1": {
        "string_opt": "a string",
        "int_opt": 123,
        "float_opt": 2.0,
        "bool_opt": true
    }
}
```

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

Current Support:

- single Json file
- Anyconfig supported file type (yaml, toml, json,...)

Future support:

- Multiple file example
- Configuration server

## 1.3 Access to settings

In your application, an xpath-like syntax allows you to reach any item of the configuration, using a xpath-like syntax `<key1>.<key2>.<key3>.<item>`, for example:

```
cfg = AnyConfigModel(model={
    "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'),
})
```

Setting values can be done by:

```
cfg.set_cfg_value("group1.float_opt", 2.0)
```

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 JSON 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": {
        # ...
    },
}
```

Various setting types are provided, covering most of the data types to be stored in configuration a file.

For example, `StringCfg` descrive a string value, `IntCfg` any integer, `BoolCfg` a boolean and so on.

Each type has the same base arguments, such as:

- `summary`: human readable short description
- `description`: human readable long and exhauxtive description
- `short_param`: which short command line argument to expose (ex: `-c`)
- `long_param`: which long command line argument to expose (ex: `--config-file`)

Here an example of a complex model:

```
{
    "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"
```

CHAPTER 3

# CfgTree API Reference

## 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*, *autosave=False*)
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")
```

**autosave = False**

**cmd_line_parser = None**

**disable_autosave**()

**enable_autosave**()

**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: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <object object>*)

**action**

**arg_type = None**
argument type

**cfgfile_value**
Return value to save in config file.

**cmd_line_name**

**default = None**
Default value

**environ_var**

**environ_var_prefix = None**
prefix to use for environemn

**get_cmd_line_params**()

**ignore_in_args = False**

**ignore_in_cfg = False**

**ignore_in_envvars = False**

**long_param**

**metavar**

**n_args**

**name = None**
name of the item

**read_environ_var**()

**safe_value**
Return value as a string without compromizing information.

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

**value**

**xpath = None**
xpath to reach this element

**class** cfgtree.types.**BoolCfg**(*long_param: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <object object>*)
Boolean value

Handle automatic integer convertion 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: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <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: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <object object>*)
Directory name

Example:

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

**class** cfgtree.types.**FloatCfg**(*long_param: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <object object>*)
Float or double value

Example:

```
1,23
```

**class** cfgtree.types.**HardcodedCfg**(*long_param: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <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: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <object object>*)

IPv4 or IPv6 value

Example:

```
"192.168.0.1"
```

**class** cfgtree.types.**IntCfg**(*long_param: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <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 mode value between several string value

Example:

```
"a_value"
```

**class** cfgtree.types.**PasswordCfg**(*long_param: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <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: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <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: str = None*, *description: str = None*, *short_param: str = None*, *summary: str = None*, *required: bool = False*, *default: typing.Any = <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*, *default_filename=None*)

Settings are stored in a single JSON file

Example:

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

Usage:

```
class MyAppConfig(ConfigBaseModel):

    ...
```

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

    ...
```

**default_filename = None**
Default filename for the JSON configuration file

Example:

```
myconfig.json
```

**environ_var = None**
Environment variable to set the configuration file name

Example:

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

**find_default_filename**(*model*)

**get_config_file**()

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

**long_param = None**
Short parameter to specify the configure file name

Example:

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

**short_param = None**
Short parameter to specify the configure file name

Example:

```
-g myconfig.json
```

### 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*)

CHAPTER 4

# Release Notes

## 4.1 1.1.0

### 4.1.1 Release Summary

This release add support for AnyConfig

### 4.1.2 New Features

- New model: `cfgtree.models.anyconfig.AnyConfigModel`, that automatically loads configuration file from a large variety of file formats: ini, json, pickle, properties, shellvars, xml, yaml
- No more need to specify both `ConfigFileCfg` and `default_filename` in storage class. Now, when trying to load the configuration file, the models search for a `ConfigFileCfg` type argument and inject its `default_filename` value.

## 4.2 1.0.0

### 4.2.1 Release Summary

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

### 4.2.2 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`

### 4.2.3 Deprecations

- Type `UserCfg` has been deprecated. Use `StringCfg` instead.

## 4.3 0.1.1

### 4.3.1 Release Summary

New version increase the user documentation.

### 4.3.2 New Features

- User documentation at readthedocs
- API documentation

## 4.4 0.1.0

### 4.4.1 Release Summary

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

### 4.4.2 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

This package provides an easy yet comprehensive way of describing, storing, parsing, modifying user configuration for a modern application.

It requires the following acknolegdments:

- Application settings are stored in a hierarchical structure, they can be organized into group of settings, sub-groups, and they entierely depends on the application itself.

  This structure is called in cfgtree a "bare configuration", or "configuration tree", and is described by a "model".

- User settings may come from different inputs:
  - environment variables ("12-factors" approach). Example: `MYAPP_VERBOSE`.
  - command line argument. Example: `--verbose`
  - configuration storage such as file (json, yaml, ini) or configuration server. Example:

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

This allows you to define once your settings structure, and let the user of your application define the settings throught 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

# Python Module Index

## C

# Index

## Symbols



## A



## B



## C



## D



## E



## F



## G



## H



## I



## J



## L

## M



## N



## P



## R



## S



## V



## X