config$_{resolver}$

Release 5.0.0a1

Feb 22, 2020

Contents

Full Documentation https://config-resolver.readthedocs.org/en/latest/

Repository https://github.com/exhuma/config_resolver

PyPI https://pypi.python.org/pypi/config_resolver

`config_resolver` provides a simple, yet flexible way to provide configuration to your applications. It follows the XDG Base Dir Spec (This instance is based on 0.8 of this spec) for config file locations, and adds additional ways to override config locations. The aims of this package are:

- Provide a simple API
- Follow well-known standards for config-file locations
- Be as close to pure-Python as possible
- Be framework agnostic
- Allow custom configutaion types (`.ini` and `.json` support is shipped by default)
- Allow to provide system-wide defaults but allow overriding of values for more specific environments. These are (in increasing order of specificity):
    1. System-wide configuration (potentially requiring root-access to modify)
    2. User-level configuration (for all instances running as that user)
    3. Current Working Directory configuration (for a running instance)
    4. Per-Instance configuration

CHAPTER 1

Table of Contents

1.1 Usage

1.1.1 Basics

The module provides one function to retrieve a config instance:

- *get_config()*

and one function to create a config from a text-string:

- *from_string()*

A simple usage looks like this:

```
from config_resolver imoprt get_config
result = get_config('bird_feeder', 'acmecorp')
cfg = result.config  # The config instance (its type depends on the handler)
meta = result.meta  # Metadata for the loading-process
```

This will look for config files in (in that order):

- `/etc/acmecorp/bird_feeder/app.ini`
- `/etc/xdg/acmecorp/bird_feeder/app.ini`
- `~/.config/acmecorp/bird_feeder/app.ini`
- `./.acmecorp/bird_feeder/app.ini`

If all files exist, one which is loaded later, will override the values of an earlier file. No values will be removed, this means you can put system-wide defaults in `/etc` and specialise/override from there.

Note: The above is true for the file handlers included with *config_resolver*. Since version 5.0 it is possible to provide custom file-handlers, which may behave differently. If using a custom file-handler make sure to understand how it behaves! See *Custom Handlers*.

The Freedesktop XDG standard

freedesktop.org standardises the location of configuration files in the XDG specification Since version 4.1.0, `config_resolver` reads these paths as well, and honors the defined environment variables. To ensure backwards compatibility, those paths have only been added to the resolution order. They have a higher precedence than the old locations though. So the following applies:

XDG item	overrides
`/etc/xdg/<group>/<app>`	`/etc/<group>/<app>`
`~/.config/<group>/</app>`	`~/.<group>/<app>`
`$XDG_DATA_HOME`	`$GROUP_APP_PATH`
`$XDG_CONFIG_DIRS`	`$GROUP_APP_PATH`

By default, files are parsed using the default Python `configparser.ConfigParser` (i.e. `ini` files). Custom file "handlers" may read other formats. See *Custom Handlers*.

1.1.2 Advanced Usage

The way config_resolver finds files can be controlled by an optional `lookup_options` argument to *`get_config()`*. This is a dictionary controlling how the files are searched and which files are valid. The default options are:

```
default_options = {
    'search_path': '',  # <- empty string here triggers the default search path
    'filename': 'app.ini',  # <- this depends on the file-handler
    'require_load': False,
    'version': None,
    'secure': False,
}
```

All values in the dictionary are optional. Not all values have to be supplied. Missing values will use the default value shown above.

Versioning

It is pretty much always useful to keep track of the expected "schema" of a config file. If in a later version of your application, you decide to change a configuration value's name, remove a variable, or require a new one the end-user needs to be notified.

For this use-case, you can use the lookup option `version` to allow only files of the proper version to be loaded. If the version differs in a detected file, a log message will be emitted:

```
result = get_config('group', 'app', {'version': '2.1'})
```

Config file example:

```
[meta]
version=2.1

[database]
dsn=foobar
```

If you don't specify a version number in the constructor versioning will trigger automatically on the first file encountered which has a version number. The reason this triggers is to prevent accidentally loading files further down the chain which have an incompatible version.

Only "major" and "minor" numbers are supported. If the application encounters a file with a different "major" value, it will emit a log message with severity `ERROR` and the file will be skipped. If the minor version of a file is smaller than the expected version, an error is logged as well and the file is skipped. If the minor version is equal or larger (inside the config file), then the file will be loaded.

In other words, for a file to be loaded, the major versions that the application expected (via the `get_config` call) must match the major version in the config-file **and** the expectes minor version must be **smaller** than the minor version inside the config-file.

Requiring files (bail out if no config is found)

Since version 3.3.0, you have a bit more control about how files are loaded. The *`get_config()`* function takes the lookup_options value `require_load`. If this is set to `True`, an `OSError` is raised if no config file was loaded. Alternatively, and, purely a matter of presonal preference, you can leave this on it's default `False` value and inspect the `loaded_files` attribute on the `meta` attribute of the returned result. If it's empty, nothing has been loaded.

1.1.3 Overriding internal defaults

Both the search path and the basename of the file (`app.ini`) can be overridden by the application developer via the API and by the end-user via environment variables.

By the application developer

Apart from the "group name" and "application name", the *`get_config()`* function accepts `search_path` and `filename` as values in `lookup_options`. `search_path` controls to what folders are searched for config files, `filename` controls the basename of the config file. `filename` is especially useful if you want to separate different concepts into different files:

```
app_cfg = get_config('acmecorp', 'bird_feeder').config
db_cfg = get_config('acmecorp', 'bird_feeder', {'filename': 'db.ini'})
```

By the end-user

The end-user has access to two environment variables:

- `<GROUP_NAME>_<APP_NAME>_PATH` overrides the default search path.
- `XDG_CONFIG_HOME` overrides the path considered as "home" locations for config files (default = `~/.config`)
- `XDG_CONFIG_DIRS` overrides additional path elements as recommended by the freedesktop.org XDG basedir spec. Paths are separated by `:` and are sorted with descending precedence (leftmost is the most important one).
- `<GROUP_NAME>_<APP_NAME>_FILENAME` overrides the default basename of the config file (default = `app.ini`).

1.1.4 Logging

All operations are logged using the default `logging` package with a logger with the name `config_resolver`. All operational logs (opening/reading file) are logged with the `INFO` level. The log messages include the absolute names of the loaded files. If a file is not loadable, a `WARNING` message is emitted. It also contains a couple of `DEBUG` messages. If you want to see those messages on-screen you could do the following:

```
import logging
from config_resolver import Config
logging.basicConfig(level=logging.DEBUG)
conf = get_config('mycompany', 'myapplication').config
```

If you want to use the `INFO` level in your application, but silence only the config_resolver logs, add the following to your code:

```
logging.getLogger('config_resolver').setLevel(logging.WARNING)
```

As of version 4.2.0, all log messages are prefixed with the group and application name. This helps identifying log messages if multiple packages in your application use `config_resolver`. The prefix filter can be accessed via the "meta" member `prefix_filter` if you want to change or remove it:

```
from config_resolver import Config
conf = get_config('mycompany', 'myapplication')
print(conf.meta.prefix_filter)
```

More detailed information about logging is out of the scope of this document. Consider reading the logging tutorial of the official Python docs.

1.1.5 Environment Variables

The resolver can also be manipulated using environment variables to allow different values for different running instances. The variable names are all upper-case and are prefixed with both group- and application-name.

<group_name>_<app_name>_PATH The search path for config files. You can specify multiple paths by separating it by the system's path separator default (`:` on Linux).

If the path is prefixed with +, then the path elements are *appended* to the default search path.

<group_name>_<app_name>_FILENAME The file name of the config file. Note that this should *not* be given with leading path elements. It should simply be a file basename (f.ex.: `my_config.ini`)

XDG_CONFIG_HOME and XDG_CONFIG_DIRS See the XDG specification

Debugging

Calling *`get_config()`* will not raise an error (except if explicitly asked to do so). Instead it will always return a valid, (but possibly empty) instance. So errors can be hard to see sometimes.

The idea behind this, is to encourage you to have sensible default values, so that the application can run, even without configuration.

Your first stop should be to configure logging and look at the emitted messages.

In order to determine whether any config file was loaded, you can look into the `loaded_files` "meta" variable. It contains a list of all the loaded files, in the order of loading. If that list is empty, no config has been found. Also remember that the order is important. Later elements will override values from earlier elements (depending of the used `handler`).

Additionally, another "meta" variable named `active_path` represents the search path after processing of environment variables and runtime parameters. This may also be useful to display information to the end-user.

Examples

A simple config instance (with logging):

```
import logging
from config_resolver import get_config

logging.basicConfig(level=logging.DEBUG)
cfg = get_config("bird_feeder", "acmecorp").config
print(cfg.get('section', 'var'))
```

An instance which will not load unsecured files:

```
import logging
from config_resolver import get_config

logging.basicConfig(level=logging.DEBUG)
cfg = get_config("bird_feeder", "acmecorp", {"secure": True}).config
print(cfg.get('section', 'var'))
```

Loading a versioned config file:

```
import logging
from config_resolver import get_config

logging.basicConfig(level=logging.DEBUG)
cfg = get_config("bird_feeder", "acmecorp", {"version": "1.0"}).config
print(cfg.get('section', 'var'))
```

Inspect the "meta" variables:

```
from config_resolver import get_config

cfg = get_config("bird_feeder", "acmecorp")
print(cfg.meta)
```

1.2 Changelog

1.2.1 Release 5.0.0

Warning: Major API changes! Read the full documentation before upgrading!

- Python 2 support is now dropped!
- Add the possibility to supply a custom file "handler" (f.ex. YAML or other custom parsers).
- Add *config_resolver.handler.json* as optional file-handler.
- Refactored from a simple module to a full-fledged Python package

- Retrieving a config instance no longer returns a subclass of the `configparser.ConfigParser` class. Instead, it will return whatever the supplied handler creates.
- External API changed to a functional API. You no longer call the `Config` constructor, but instead use the `get_config()` function. See the API docs for the changes in function signature.
- Retrieval meta-data is returned along-side the retrieved config. This separation allows a custom handler to return any type without impacting the internal logic of `config_resolver`.
- Dropped the deprectaed lookup in `~/.group-name/app-name` in favor of the XDG standard `~/.config/group-name/app-name`.
- Fully type-hinted

Upgrading from 4.x

- Replace `Config` with `get_config`
- The result from the call to `get_config` now returns a named-tuple with two objects: The config instance (`.config`) and additional metadata (`.meta`).
- The following attributes moved to the meta-data object:
  - `active_path`
  - `prefix_filter`
  - `loaded_files`
- Return types for INI files is now a standard library instance of `configparser.ConfigParser`. This means that the `default` keyword argument to `get` has been replaced with `fallback`.

1.2.2 Release 4.2.0

Features added

- GROUP and APP names are now included in the log messages.

1.2.3 Release 4.1.0

Features added

- XDG Basedir support

  `config_resolver` will now search in the folders/names defined in the *XDG specification*.

1.2.4 Release 4.0.0

Features added

- Config versioning support.

  The config files can now have a section `meta` with the key `version`. The version is specified in dotted-notation with a major and minor number (f.ex.: `version=2.1`). Configuration instances take an optional `version` argument as well. If specified, config_resolver expects the `meta.version` to be there. It will raise

a `config_resolver.NoVersionError` otherwise. Increments in the major number signify an incompatible change. If the application expectes a different major number than stored in the config file, it will raise a `config_resolver.IncompatibleVersion` exception. Differences in minor numbers are only logged.

Improvments

- The `mandatory` argument **has been dropped**! It is now implicitly assumed it the `.get` method does not specify a default value. Even though "explicit is better than implicit", this better reflects the behaviour of the core `ConfigParser` and is more intuitive.
- Legacy support of old environment variable names **has been dropped**!
- Python 3 support.
- When searching for a file on the current working directory, look for `./.group/app/app.ini` instead of simply `./app.ini`. This solves a conflict when two modules use config_resolver in the same application.
- Better logging.

1.2.5 Release 3.3.0

Features added

- New (optional) argument: `require_load`. If set to `True` creating a config instance will raise an error if no appropriate config file is found.
- New class: `SecuredConfig`: This class will refuse to load config files which are readable by other users than the owner.

Improvments

- Documentation updated/extended.
- Code cleanup.

1.2.6 Release 3.2.2

Improvments

- Unit tests added

1.2.7 Release 3.2.1

Fixes/Improvments

- The "group" name has been prefixed to the names of the environment variables. So, instead of APP_PATH, you can now use GROUP_APP_PATH instead. Not using the GROUP prefix will still work but emit a DeprecationWarning.

1.2.8 Release 3.2

Features added

- The call to `get` can now take an optional default value. More details can be found in the docstring.

1.2.9 Release 3.1

Features added

- It is now possible to extend the search path by prefixing the `<APP_NAME>_PATH` variable value with a `+`
- Changelog added

1.3 Custom Handlers

When requesting a config-instance using *`get_config()`* it is possible to specify a custom *file-handler* using the `handler` keyword arg. For example:

```
from config_resolver import get_config
from config_resolver.handlers.json import JsonHandler

result = get_config('foo', 'bar', handler=JsonHandler)
```

Each handler has full control over the data type which is returned by *`get_config()`*. `get_config` always returns a named-tuple with two arguments:

- `config`: This contains the object returned by the `handler`.
- `meta`: This is a named-tuple which is generated by config_resolver and not modifyable by a `handler`. See *The Meta Object*.

A handler must be subclassed from *`config_resolver.handler.base.Handler`* which allows us to provide good type-hinting.

See the existing handlers in *`config_resolver.handler`* for some practical examples.

1.4 The Meta Object

The return value of *`get_config()`* returns a named-tuple which not only contains the parsed config instance, but also some additional meta-data.

Before version 5.0 this information was melded into the returned config instance.

The reason this was split this way in version 5.0, is because with this version, the return type is defined by *the handlers*. Now, handlers may have return-types which cannot easily get additional values grafted onto them (at least not explicitly). To keep it *clear and understandable*, the values are now *explicitly* returned separately! This give the handler total freedom of which data-type they work with, and still retain useful meta-data for the end-user.

The meta-object is accessible via the second return value from *`get_config()`*:

```
_, meta = get_config('foo', 'bar')
```

Or via the `meta` attribute on the returned named-tuple:

```
result = get_config('foo', 'bar')
meta = result.meta
```

At the time of this writing, the meta-object contains the following attributes:

active_path A list of path names were used to look for files (in order of the lookup)

loaded_files A list of filenames which have been loaded (in order of loading)

config_id The internal ID used to identify the application for which the config was requested. This corresponds to the first and second argument to `get_config`.

prefix_filter A reference to the logging-filter which was added to prefix log-lines with the config ID. This exists so a user can easily get a handle on this in case it needs to be removed from the filters.

1.5 config_resolver

1.5.1 config_resolver package

Subpackages

config_resolver.handler package

Submodules

config_resolver.handler.base module

This module contains helpers for type hinting

class config_resolver.handler.base.**Handler**
Bases: typing.Generic

A generic config file handler. Concrete classes should be created in order to support new file formats.

DEFAULT_FILENAME = 'unknown'
The filename that is used when the user did not specify a filename when retrieving the config instance

static empty() → TConfig
Create an empty configuration instance.

static from_filename(*filename: str*) → TConfig
Create a configuration instance from a file-name.

static from_string(*data: str*) → TConfig
Create a configuration instance from a text-string

static get_version(*config: TConfig*) → Optional[packaging.version.Version]
Retrieve the parsed version number from a given config instance.

static update_from_file(*config: TConfig*, *filename: str*) → None
Updates an existing config instance from a given filename.

The config instance in *data* will be modified in-place!

config_resolver.handler.ini module

Handler for INI files

class config_resolver.handler.ini.**IniHandler**
Bases: *config_resolver.handler.base.Handler*

A config-resolver handler capable of reading ".ini" files.

DEFAULT_FILENAME = 'app.ini'

static empty() → configparser.ConfigParser
Create an empty configuration instance.

static from_filename(*filename: str*) → configparser.ConfigParser
Create a configuration instance from a file-name.

static from_string(*data: str*) → configparser.ConfigParser
Create a configuration instance from a text-string

static get_version(*config: configparser.ConfigParser*) → Optional[packaging.version.Version]
Retrieve the parsed version number from a given config instance.

static update_from_file(*config: configparser.ConfigParser*, *filename: str*) → None
Updates an existing config instance from a given filename.

The config instance in *data* will be modified in-place!

config_resolver.handler.json module

Handler for JSON files

class config_resolver.handler.json.**JsonHandler**
Bases: *config_resolver.handler.base.Handler*

A config-resolver handler capable of reading ".json" files.

DEFAULT_FILENAME = 'app.json'

static empty() → Dict[str, Any]
Create an empty configuration instance.

static from_filename(*filename: str*) → Dict[str, Any]
Create a configuration instance from a file-name.

static from_string(*data: str*) → Dict[str, Any]
Create a configuration instance from a text-string

static get_version(*config: Dict[str, Any]*) → Optional[packaging.version.Version]
Retrieve the parsed version number from a given config instance.

static update_from_file(*config: Dict[str, Any]*, *filename: str*) → None
Updates an existing config instance from a given filename.

The config instance in *data* will be modified in-place!

Module contents

Container package for "handlers". See *Custom Handlers*.

Submodules

config_resolver.core module

Core functionality of *config_resolver*

class config_resolver.core.**ConfigID**(*group*, *app*)
Bases: tuple

app
Alias for field number 1

group
Alias for field number 0

class config_resolver.core.**FileReadability**(*is_readable*, *filename*, *reason*, *version*)
Bases: tuple

filename
Alias for field number 1

is_readable
Alias for field number 0

reason
Alias for field number 2

version
Alias for field number 3

class config_resolver.core.**LookupMetadata**(*active_path*, *loaded_files*, *config_id*, *prefix_filter*)
Bases: tuple

active_path
Alias for field number 0

config_id
Alias for field number 2

loaded_files
Alias for field number 1

prefix_filter
Alias for field number 3

class config_resolver.core.**LookupResult**(*config*, *meta*)
Bases: tuple

config
Alias for field number 0

meta
Alias for field number 1

config_resolver.core.**effective_filename**(*config_id: config_resolver.core.ConfigID*, *config_filename: str*) → str
Returns the filename which is effectively used by the application. If overridden by an environment variable, it will return that filename.

config_id is used to determine the name of the variable. If that does not return a value, *config_filename* will be returned instead.

config_resolver.core.**effective_path**(*config_id: config_resolver.core.ConfigID*, *search_path: str = ''*) → List[str]

Returns a list of paths to search for config files in order of increasing precedence: the last item in the list will override values of earlier items.

The value in *config_id* determines the sub-folder structure.

If *search_path* is specified, that value should have the OS specific path-separator (`:` or `;`) and will completely override the default search order. If it is left empty, the search order is dictated by the XDG standard.

As a "last-resort" override, the value of the environment variable `<GROUP_NAME>_<APP_NAME>_PATH` will be inspected. If this value is set, it will be used instead of *anything* found previously (XDG paths, `search_path` value) unless the value is prefixed with a + sign. In that case it will be *appended* to the end of the list.

Examples:

```
>>> # Search the default XDG paths (and the CWD)
>>> effective_path(config_id)

>>> # Search only in "/etc/myapp"
>>> effective_path(config_id, search_path="/etc/myapp")

>>> # Search only in "/etc/myapp" and "/etc/fallback"
>>> effective_path(config_id, search_path="/etc/myapp:/etc/fallback")

>>> # Add "/etc/myapp" to the paths defined by XDG
>>> assert os.environ["FOO_BAR_PATH"] == "+/etc/myapp"
>>> effective_path(ConfigId("foo", "bar"))
```

config_resolver.core.**env_name**(*config_id: config_resolver.core.ConfigID*) → str

Return the name of the environment variable which contains the file-name to load.

config_resolver.core.**find_files**(*config_id: config_resolver.core.ConfigID*, *search_path: Optional[List[str]] = None*, *filename: str = ''*) → Generator[str, None, None]

Looks for files in default locations. Returns an iterator of filenames.

Parameters

- **config_id** – A "ConfigID" object used to identify the config folder.
- **search_path** – A list of paths to search for files.
- **filename** – The name of the file we search for.

config_resolver.core.**from_string**(*data: str*, *handler: Optional[config_resolver.handler.base.Handler[typing.Any][Any]] = None*) → config_resolver.core.LookupResult

Load a config from the string value in *data*. *handler* can be used to specify a custom parser/handler.

config_resolver.core.**get_config**(*app_name: str*, *group_name: str = ''*, *lookup_options: Optional[Dict[str, Any]] = None*, *handler: Optional[Type[config_resolver.handler.base.Handler[typing.Any][Any]]] = None*) → config_resolver.core.LookupResult

Factory function to retrieve new config instances.

app_name is the only required argument for config lookups. If nothing else is specified, this will trigger a lookup in default XDG locations for a config file in a subfolder with that name.

group_name is an optional subfolder which is *prefixed* to the subfolder based on the *app_name*. This can be used to group related configurations together.

To summarise the two above paragraphs the relative path (relative to the search locations) will be:

- `<app_name>/<filename>` if only *app_name* is given
- `<group_name>/<app_name>/<filename>` if both *app_name* and *group_name* are given

lookup_options contains arguments which allow more fine-grained control of the lookup process. See below for details.

The *handler* may be a class which is responsible for loading the config file. *config_resolver* uses a ".ini" file handler by default and comes bundled with a JSON handler as well. They can be found in the **:py:module:'config_resolver.handler'** package.

Note: The type of the returned config-object depends on the handler. Each handler has its own config type!

For example, loading JSON files can be achieved using:

```
>>> from config_resolver.handler.json import JsonHandler
>>> get_config("myapp", handler=JsonHandler)
```

lookup_options is a dictionary with the following optional keys:

filename (default='') This can be used to override the default filename of the selected handler. If left empty, the handler will be responsible for the filename.

search_path (default='[]') A list of folders that should be searched for config files. The order here is relevant. The folders will be searched in order, and each file which is found will be loaded by the *handler*. Note that the search path should not include *group_name* or *app_name* as they will be appended automatically.

require_load (default='False') A boolean value which determines what happens if *no* file was loaded. If this is set to `True` the call to `get_config` will raise an exception if no file was found. Otherwise it will log a warning.

version (default='None') This can be a string in the form `<major>.<minor>`. If specified, the lookup process will request a version number from the *handler* for each file found. The version in the file will be compared with this value. If the minor-number differs, the file will be loaded, but a warning will be logged. If the major number differs, the file will be skipped and an error will be logged. If the value is left unset, no version checking will be performed. If this is left unspecified and a config file is encountered with a version number, a sanity check is performed on subsequent config-files to ensure that no mismatching major versions are loaded in the lookup-chain.

How the version has to be stored in the config file depends on the handler.

secure (default='False') If set to `True`, files which are world-readable will be ignored. This forces you to have secure file-access rights because the file will be skipped if the rights are too open.

`config_resolver.core.`**`get_xdg_dirs`**(*config_id: config_resolver.core.ConfigID*) → List[str]
Returns a list of paths specified by the XDG_CONFIG_DIRS environment variable or the appropriate default. See *The Freedesktop XDG standard* for details.

The list is sorted by precedence, with the most important item coming *last* (required by the existing config_resolver logic).

The value in *config_id* is used to determine the sub-folder structure.

`config_resolver.core.`**`get_xdg_home`**(*config_id: config_resolver.core.ConfigID*) → str
Returns the value specified in the XDG_CONFIG_HOME environment variable or the appropriate default. See *The Freedesktop XDG standard* for details.

config_resolver.core.**is_readable**(*config_id: config_resolver.core.ConfigID*, *filename: str*, *version: Optional[packaging.version.Version] = None*, *secure: bool = False*, *handler: Optional[Type[config_resolver.handler.base.Handler[typing.Any][Any]]] = None*) → config_resolver.core.FileReadability

Check if `filename` can be read. Will return boolean which is True if the file can be read, False otherwise.

Parameters

- **filename** – The exact filename which should be checked.
- **version** – The expected version, that should be found in the file.
- **secure** – Whether we should avoid loading insecure files or not.
- **handler** – The handler to be used to open and parse the file.

config_resolver.core.**prefixed_logger**

Returns a log instance and prefix filter for a given group- & app-name pair.

It applies a filter to the logger which prefixes the log messages with group- and application-name from the config.

The call to this function is cached to ensure we only have one instance in memory.

config_resolver.exc module

Exceptions for the config_resolver package

exception config_resolver.exc.**NoVersionError**

Bases: Exception

This exception is raised if the application expects a version number to be present in the config file but does not find one.

config_resolver.util module

Helpers and utilities for the config_resolver package.

This module contains stuff which is not directly impacting the business logic of the config_resolver package.

class config_resolver.util.**PrefixFilter**(*prefix: str*, *separator: str = ' '*)

Bases: logging.Filter

A logging filter which prefixes each message with a given text.

Parameters

- **prefix** – The log prefix.
- **separator** – A string to put between the prefix and the original log message.

filter(*record: logging.LogRecord*) → bool

Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

Module contents

The `config_resolver` package provides an easy way to create an instance of a config object.

The main interface of this package is *`config_resolver.core.get_config()`* (also provided via `config_resolver.get_config`).

This function takes a fair amount of options to control how config files are loaded. The easiest example is:

```
>>> from config_resolver import get_config
>>> config, metadata = get_config("myapp")
```

This call will scan through a number of folders and load/update the config with every matching file in that chain. Some customisation of that load process is made available via the *`get_config()`* arguments.

The call retuns a config instance, and some meta-data related to the loading process. See *`get_config()`* for details.

`config_resolver` comes with support for `.json` and `.ini` files out of the box. It is possible to create your own handlers for other file types by subclassing `config_resolver.handler.Handler` and passing it to *`get_config()`*

1.6 Glossary

file-handler A file-handler is a module or class offering a minimal set of functions to load files as config files. They can optionally be supplied to *`get_config()`*. By default, handlers for INI and JSON files are supplied. Look at *Custom Handlers* for details on how to create a new one.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

H

I

J

L

M

N

P

R

U

V