python-configuration

Tiago Requeijo

Aug 05, 2023

CONTENTS

This library is intended as a helper mechanism to load configuration files hierarchically. Current format types are:

- Python files
- Dictionaries
- Environment variables
- Filesystem paths
- JSON files
- INI files
- dotenv type files

and optionally

- YAML files
- TOML files
- Azure Key Vault credentials
- AWS Secrets Manager credentials
- GCP Secret Manager credentials

CHAPTER

ONE

INSTALLING

To install the library:

```
$ pip install python-configuration
```

To include the optional TOML and/or YAML loaders, install the optional dependencies *toml* and *yaml*. For example,

```
$ pip install python-configuration[toml,yaml]
```

CHAPTER

TWO

GETTING STARTED

This library converts the config types above into dictionaries with dotted-based keys. That is, given a config `cfg` from the structure

```
{
    'a': {
        'b': 'value'
    }
}
```

we are able to refer to the parameter above as any of

```
cfg['a.b']
cfg['a']['b']
cfg['a'].b
cfg.a.b
```

and extract specific data types such as dictionaries:

```
cfg['a'].as_dict == {'b': 'value'}
```

This is particularly useful in order to isolate group parameters. For example, with the JSON configuration

```
 {
  "database.host": "something",
  "database.port": 12345,
  "database.driver": "name",
  "app.debug": true,
  "app.environment": "development",
  "app.secrets": "super secret",
  "logging": {
    "service": "service",
    "token": "token",
    "tags": "tags"
  }
}
```

one can retrieve the dictionaries as

```
cfg.database.as_dict()
cfg.app.as_dict()
cfg.logging.as_dict()
```

or simply as

```
dict(cfg.database)
dict(cfg.app)
dict(cfg.logging)
```

CHAPTER

THREE

CONFIGURATION

There are two general types of objects in this library. The first one is the `Configuration`, which represents a single config source. The second is a `ConfigurationSet` that allows for multiple `Configuration` objects to be specified.

3.1 Single Config

3.1.1 Python Files

To load a configuration from a Python module, the `config_from_python()` can be used. The first parameter must be a Python module and can be specified as an absolute path to the Python file or as an importable module.

Optional parameters are the `prefix` and `separator`. The following call

```
config_from_python('foo.bar', prefix='CONFIG', separator='__')
```

will read every variable in the `foo.bar` module that starts with `CONFIG__` and replace every occurrence of `__` with a `.`. For example,

```
# foo.bar
CONFIG__AA__BB_C = 1
CONFIG__AA__BB__D = 2
CONF__AA__BB__D = 3
```

would result in the configuration

```
{
    'aa.bb_c': 1,
    'aa.bb.d': 2,
}
```

Note that the single underscore in `BB_C` is not replaced and the last line is not prefixed by `CONFIG`.

3.1.2 Dictionaries

Dictionaries are loaded with `config_from_dict()` and are converted internally to a flattened `dict`.

```
{
    'a': {
        'b': 'value'
    }
}
```

becomes

```
{
    'a.b': 'value'
}
```

3.1.3 Environment Variables

Environment variables starting with `prefix` can be read with `config_from_env()`:

```
config_from_env(prefix, separator='_')
```

3.1.4 Filesystem Paths

Folders with files named as `xxx.yyy.zzz` can be loaded with the `config_from_path()` function. This format is useful to load mounted Kubernetes ConfigMaps. or Secrets.

3.1.5 JSON, INI, .env, YAML, TOML

JSON, INI, YAML, TOML files are loaded respectively with `config_from_json()`, `config_from_ini()`, `config_from_dotenv()`, `config_from_yaml()`, and `config_from_toml()`. The parameter `read_from_file` controls whether a string should be interpreted as a filename.

3.1.6 Caveats

In order for `Configuration` objects to act as `dict` and allow the syntax `dict(cfg)`, the `keys()` method is implemented as the typical `dict` keys. If `keys` is an element in the configuration `cfg` then the `dict(cfg)` call will fail. In that case, it's necessary to use the `cfg.as_dict()` method to retrieve the `dict` representation for the `Configuration` object.

The same applies to the methods `values()` and `items()`.

CHAPTER

FOUR

CONFIGURATION SETS

Configuration sets are used to hierarchically load configurations and merge settings. Sets can be loaded by constructing a `ConfigurationSet` object directly or using the simplified `config()` function.

To construct a `ConfigurationSet`, pass in as many of the simple `Configuration` objects as needed:

```
cfg = ConfigurationSet(
    config_from_env(prefix=PREFIX),
    config_from_json(path, read_from_file=True),
    config_from_dict(DICT),
)
```

The example above will read first from Environment variables prefixed with `PREFIX`, and fallback first to the JSON file at `path`, and finally use the dictionary `DICT`.

The `config()` function simplifies loading sets by assuming some defaults. The example above can also be obtained by

```
cfg = config(
    ('env', PREFIX),
    ('json', path, True),
    ('dict', DICT),
)
```

or, even simpler if `path` points to a file with a `.json` suffix:

```
cfg = config('env', path, DICT, prefix=PREFIX)
```

The `config()` function automatically detects the following:

- extension `.py` for python modules
- dot-separated python identifiers as a python module (e.g. `foo.bar`)
- extension `.json` for JSON files
- extension `.yaml` for YAML files
- extension `.toml` for TOML files
- extension `.ini` for INI files
- extension `.env` for dotenv type files
- filesystem folders as Filesystem Paths
- the strings `env` or `environment` for Environment Variables

4.1 Merging Values

`ConfigurationSet` instances are constructed by inspecting each configuration source, taking into account nested dictionaries, and merging at the most granular level. For example, the instance obtained from `cfg = config(d1, d2)` for the dictionaries below

```
d1 = {'sub': {'a': 1, 'b': 4}}
d2 = {'sub': {'b': 2, 'c': 3}}
```

is such that `cfg['sub']` equals

```
{'a': 1, 'b': 4, 'c': 3}
```

Note that the nested dictionaries of `'sub'` in each of `d1` and `d2` do not overwrite each other, but are merged into a single dictionary with keys from both `d1` and `d2`, giving priority to the values of `d1` over those from `d2`.

4.1.1 Caveats

As long as the data types are consistent across all the configurations that are part of a `ConfigurationSet`, the behavior should be straightforward. When different configuration objects are specified with competing data types, the first configuration to define the elements sets its datatype. For example, if in the example above `element` is interpreted as a `dict` from environment variables, but the JSON file specifies it as anything else besides a mapping, then the JSON value will be dropped automatically.

CHAPTER

FIVE

OTHER FEATURES

5.1 String Interpolation

When setting the `interpolate` parameter in any `Configuration` instance, the library will perform a string interpolation step using the str.format syntax. In particular, this allows to format configuration values automatically:

CHAPTER

SIX

EXTRAS

The `contrib` package contains extra implementations of the `Configuration` class used for special cases. Currently the following are implemented:

- `AzureKeyVaultConfiguration` in `azure`, which takes Azure Key Vault credentials into a `Configuration`-compatible instance. To install the needed dependencies execute

```
pip install python-configuration[azure]
```

- `AWSSecretsManagerConfiguration` in `aws`, which takes AWS Secrets Manager credentials into a `Configuration`-compatible instance. To install the needed dependencies execute

```
pip install python-configuration[aws]
```

- `GCPSecretManagerConfiguration` in `gcp`, which takes GCP Secret Manager credentials into a `Configuration`-compatible instance. To install the needed dependencies execute

```
pip install python-configuration[gcp]
```

CHAPTER

SEVEN

DEVELOPING

To develop this library, download the source code and install a local version.

CHAPTER

EIGHT

FEATURES

- Load multiple configuration types
- Hierarchical configuration
- Ability to override with environment variables
- Merge parameters from different configuration types

CHAPTER

NINE

CONTRIBUTING

If you'd like to contribute, please fork the repository and use a feature branch. Pull requests are welcome.

CHAPTER

TEN

LINKS

- Repository: https://github.com/tr11/python-configuration
- Issue tracker: https://github.com/tr11/python-configuration/issues

CHAPTER

ELEVEN

LICENSING

The code in this project is licensed under MIT license.

CHAPTER

TWELVE

TABLE OF CONTENTS

12.1 API

12.1.1 Extra Modules

12.2 Glossary

TOML
: TOML files

YAML
: YAML files

INDEX