configclasses Documentation

Release 0.4.5

Jeff Belgum

Oct 11, 2018

Contents

1 Introduction **3**
- 1.1 A Basic Example . . . 3
- 1.2 A *Slightly* More Advanced Example . . . 3
- 1.3 Features . . . 4
- 1.4 Planned work . . . 5

2 Installation **7**
- 2.1 Installation . . . 7

3 User's Guide **9**
- 3.1 User's Guide . . . 9

4 API Documentation **15**
- 4.1 API Documentation . . . 15

5 Contribution **21**
- 5.1 Contribution . . . 21

6 License **23**
- 6.1 LICENSE . . . 23

7 Indices and tables **29**

Python Module Index **31**

Release v0.4.5. (*Installation*)

CHAPTER 1

Introduction

configclasses provides a simple yet powerful way to define and fetch configuration values for your application by extending python's dataclasses (PEP-557) with additional functionality.

Configuration values are fetched on demand from various sources, validated, and stored in a single strongly typed configuration object.

1.1 A Basic Example

```
from configclasses import configclass

# Wrap your configuration class in the `configclass` decorator
@configclass
class Configuration:
    HOST: str
    PORT: int

# Fields are populated when you construct a Configuration instance
config = Configuration()

# Access fields by name.
config.HOST == "localhost"
```

That's it!

You now have an easy to use configuration class that fetches and validates all the configuration values your application requires. It defaults to searching environment variables to populate fields. In this case, it expects environment variables to be set for `HOST` and `PORT`.

1.2 A *Slightly* More Advanced Example

The `configclass` decorator also accepts user-specified sources of configuration data.

```
from configclasses import configclass, sources
from configclasses.sources import CommandLineSource, DotEnvSource, EnvironmentSource

# Create multiple sources of configuration information, and pass them to the
# `configclass` decorator.
@configclass(sources=[DotEnvSource(path=".env"), EnvironmentSource(),␣
↪CommandLineSource()])
class Configuration:
    HOST: str = "localhost"  # Set a default value
    PORT: int

# Instantiating `Configuration` will always return the same
# singleton object. This way you can create a reference to
# it from any module you like and the configuration values
# will be consistent from instance to instance.
config = Configuration()

# Access fields by name.
config.HOST == "localhost"
```

The `Configuration` class will now search command line arguments, environment variables, and a *.env* file for `HOST` and `PORT`.

If a field name is found in multiple sources, sources are prioritized based on how they are passed to the `configclass` decorator. Sources are prioritized from left to right, giving the last source the highest priority.

1.3 Features

- Globally accessible configuration classes
- Easily pull from many sources of configuration:
  - Environment variables
  - Command line arguments
  - Dotenv files
  - Json files
  - Toml files
  - Ini files
  - Consul Key/Value store
  - Planned sources: AWS Parameter Store, Etcd, Redis
- Specify prioritization when multiple sources are used together.
- Support for strongly typed configuration values out of the box:
  - primitive types such as `int`, `float`, and `str` are supported.
  - `Enum` types can be used to specify valid values
  - `converter` functions can turn stringly typed values complex types such as dicts or your own types.

1.4 Planned work

- Deal with sources that only provide stringly typed values and values that provide other primitives
- Some sources might be case-insensitive.
- Async/Sync versions of sources
- Research and design push updates (as opposed to polling updates)
- Better error messages when config values are missing from all sources
- Audit exception types raised.
- Comprehensive docs
  - Includes docs on adding your own sources.

CHAPTER 2

Installation

2.1 Installation

`configclasses` can be installed with all the traditional python tools.

2.1.1 Pip Install configclasses

To install configclasses, simply run this command in your terminal:

```
$ pip install configclasses
```

If you don't have pip installed, this Python installation guide can guide you through the process.

2.1.2 Suggested Alternative: Pipenv

pipenv is a new tool that solves the problems of isolated virtual environment, package installation, and dependency tracking in a simple but comprehensive manner:

```
$ pip install pipenv
$ pipenv install configclasses
```

Full documentation can be found on readthedocs. Why not give it a try!

2.1.3 Get the Source Code

configclasses is under active development on GitHub, where the code is always available.

You can clone the repository:

```
$ git clone git://github.com/jeffbelgum/configclasses.git
```

Once you have a copy of the source, you can embed it in your own Python package or install it into your site-packages easily:

```
$ python setup.py install
```

CHAPTER 3

User's Guide

Tutorials to guide you through the most common uses of the library as well as more advanced scenarios.

3.1 User's Guide

Starting with the example shown in the introduction, let's dig into configclasses a bit:

```
from configclasses import configclass

# Wrap your configuration class in the `configclass` decorator
@configclass
class Configuration:
    HOST: str
    PORT: int

# Fields are populated when you construct a Configuration instance
config = Configuration()

# Access fields by name.
config.HOST == "localhost"
```

You start by defining your own configuration class with the fields that you will need for your application. This is done exactly in the same way that it is done with dataclasses (PEP-557).

Note: If you're not familiar with dataclasses, they are a way of describing classes in python using type annotations that removes much of the boilerplate. The ideas have existed for some time in alternative forms as attrs, recordType, namedtuple, etc. I would suggest familiarizing yourself with the functionality before continuing to get the most out of this guide.

The key distinction between a dataclass and a configclass is that the fields of a dataclass are not populated from within the code itself. Instead, a configclass knows how to fetch the value for each field from sources of configuration that *live outside the code*.

When a dataclass is constructed, the `__init__` method searches for configuration variables that match the field names, and assigns the values to the matching configclass field. By default, that source of configuration is the application's environment variables.

```
$ HOST=localhost PORT=8000 python application.py
```

`Configuration` has a field named `HOST`, so it will search the environment for a variable with the same name. The value of the environment variable is assigned to the `HOST` field.

`PORT` is also found and assigned to the matching field. Notice that it is defined as an `int` type. Because of this, it is converted into an integer value before assignment. If the `PORT` environment variable cannot be converted into an `int`, an exception is raised.

3.1.1 Field Types

So far, we have discussed string and integer fields. But configclasses supports other types as well. These include bools, floats, json objects, lists, key-value pairs, and custom types. It also includes enums for when you want to limit the valid set of values for a field.

Let's see that in action:

```
from configclasses import configclass, enums

# Wrap your configuration class in the `configclass` decorator
@configclass
class Configuration:
    HOST: str
    PORT: int
    ENABLE_AUTHENTICATION: bool
    LOG_LEVEL: enums.LogLevel

# Fields are populated when you construct a Configuration instance
config = Configuration()

config.ENABLE_AUTHENTICATION == True
config.LOG_LEVEL.value == logging.DEBUG
```

```
$ HOST=localhost PORT=8000 ENABLE_AUTHENTICATION=true LOG_LEVEL=DEBUG python␣
↪application.py
```

You'll notice that the fields are converted from strings in the environment into the correct python types. Bool values should be case insensitive "true"/"false" or 1/0 respectively.

Note: Later we will look at sources that provide python primitive types instead of just string types. Those primitive types can be converted into strings using python's truthy value rules.

`LOG_LEVEL` uses a convenience enum that the library provides which maps the logging constants in the stdlib's `logging` module into an enum class.

class configclasses.enums.**LogLevel**(*Enum*)

Python logging module log level constants represented as an `enum.Enum`.

NotSet = logging.NOTSET = 0

Debug = logging.DEBUG = 10

Info = logging.INFO = 20

```
Warning  = logging.WARNING  = 30
Error    = logging.ERROR    = 40
Critical = logging.CRITICAL = 50
```

Values are considered valid for enums when they are either the case-insensitive name of an enum variant or the case-insensitive value of an enum variant. There is nothing special about the `LogLevel` enum defined in the library. You can define use any subclass of `enum.Enum` from the python stdlib, and the same rules will apply.

3.1.2 Converters

Richer data types require the introducion of a couple of new concepts. The field function and its converter argument:

```
def field(converter=None, default=MISSING, default_factory=MISSING, init=True,␣
↪repr=True, hash=None, compare=True, metadata=None)
```

If you are familiar with dataclasses, the function is identical to the same function in that module, with one key difference. That is the converter argument. A `converter` is any function that takes a single argument and knows how to convert it into the datatype of the configclass field. You are probably familiar with one such function already, `json.loads`. `json.loads` takes a string as an argument and produces a python object as long as the string contains valid json.

If we have a json config file such as logging_conf.json:

```
{
     "version": 1,
     "formatters": {
         "default": {
             "format": "%(asctime)s %(levelname)s %(name)s %(message)s"
         },
     },
     "handlers": {
         "console": {
             "class": "logging.StreamHandler",
             "formatter":"default"
         }
     },
     "root": {
         "handlers": ["console"],
         "level": "DEBUG",
     },
 }
```

We would put it to use like so:

```
import json
from configclasses import configclass, field

# Wrap your configuration class in the `configclass` decorator
@configclass
class Configuration:
    LOGGING_CONF: dict = field(converter=json.loads)

# Fields are populated when you construct a Configuration instance
config = Configuration()

type(config.LOGGING_CONF) == dict
```

configclasses also provides a handful of useful converters. `_list` takes comma seperated values and splits them into a list. It strips whitespace unless values are quoted. `"foo, bar, baz, ' quix'"` is transformed into `["foo", "bar", "baz", " quix"]`

`kv_list` takes a comma seperated list of key value pairs. `"foo=bar, baz=' quix'"` becomes `{"foo": "bar", "baz": " quix"}`

3.1.3 Sources

So far we have glossed over exactly how fields are populated from the environment. Those details are determined by `Source` classes. The default source is an `EnvironmentSource` and the constructor looks like this `EnvironmentSource(namespace=None)`. With the namespace argument, we can limit the environment variables that can populate our configclass. Suppose we have the following environment variables:

```
HOST=localhost
PORT=8000
MYAPP_HOST=0.0.0.0
MYAPP_PORT=80
```

Let's see it in action:

```
from configclasses import configclass
from configclasses.sources import EnvironmentSource
```

Field Types Defaults Sources Singleton instances Errors Enums Converters Reload Advanced patterns: - custom sources - field dependent sources - bootstrapping one configclass with values from another configclass. - Gotcha: Some sources produce python types and some always produce strings. - Make sure that your converter functions can handle that distinction

Example:

```
from configclasses import configclass, LogLevel, Environment

# Wrap your configuration class in the `configclass` decorator
# By default, it looks for matching variables in the environment.
@configclass
class Configuration:
    ENVIRONMENT: Environment    # Enum
    LOG_LEVEL: LogLevel         # Enum
    HOST: str
    PORT: int
    DB_HOST: str = "localhost"  # Default when field not found
    DB_PORT: int = 5432         # Default when field not found

# Instantiating a `Configuration` will always return the same object
config = Configuration()

# access fields by name
config.HOST == "localhost"

# `int` typed fields will be ints
config.PORT == 8080

# Fields with `Enum` types will have variants as values
config.ENVIRONMENT == Environment.Development

# Reload config values from sources
```

(continues on next page)

(continued from previous page)

```
config.reload()

# Configuration objects can now have different values
config.ENVIRONMENT == Environment.Production


# Config classes can also be configured with other `sources`
from configclasses.sources import DotEnvSource, EnvironmentSource

@configclass(sources=[DotEnvSource(), EnvironmentSource()])
class Configuration:
    HOST: str
    PORT: int
    DB_ADDRESS: str = "localhost"
    DB_PORT: int = 5432
    ENVIRONMENT: Environment
    LOG_LEVEL: LogLevel

# First, a `.env` file will be searched for values, then
# any values that are not present there will be searched
# for in the program's environment.
config = Configuration()
```

CHAPTER 4

API Documentation

Here is where you'll find comprehensive documentation for the public api.

4.1 API Documentation

The API documentation covers the full public api for the library including examples for more complicated features. The configclass decorator is described followed by pluggable sources of configuration values, and finally convenience enums and data type conversion functionality.

4.1.1 Configclass

`configclasses.`**`configclass`**(*source: Source=None*, *sources: List[Source]=None*)

Turn a class into a configclass with the default EnvironmentSource used.

For example, configuring the host and port for a web application might look like this:

```
>>> from configclasses import configclass
>>> @configclass
... class Configuration:
...     HOST: str
...     PORT: int
```

Turn a class into a configclass using the user provided source or sources list.

Parameters

- **`source`** – single `Source` used to fetch values.
- **`sources`** – list of `Source` used to fetch values, prioritized from first to last.

Raises **`ValueError`** – The user must pass *either* the source *or* a list of sources. It is an error to provide both.

Configuring the host and port for a web application using both command line arguments and environment variables as sources:

```
>>> from configclasses import configclass, sources
>>> env_source = EnvironmentSource()
>>> cli_source = CommandLineSource()
>>> @configclass(sources=[cli_source, env_source])
... class Configuration:
...     HOST: str
...     PORT: int
```

Because the `cli_source` comes *after* the `env_source` in the list of `sources`, it will be prioritized when fetching values that are found in both sources.

Decorate your configuration classes with the *configclass* decorator to turn them into Configuration Classes.

The returned configclass will have a `.reload()` method present, that can be used to reload values from configuration sources on demand. This reload affects *all* instances of the configclass you are reloading.

`configclasses.`**`field`**(*converter=None*, *default=MISSING*, *default_factory=MISSING*, *init=True*, *repr=True*, *hash=None*, *compare=True*, *metadata=None*)

This function can be used if the field differs from the default functionality. It is the same as the field function in the dataclasses module except that it includes a `converter` argument that can be used to convert from a primitive type to a more complex type such as a dict or custom class.

Parameters

- **`converter`** – is a function that takes a single argument and constructs a return value that is the same as the conficlass field's type annotation.
- **`converter`** – is a function that takes a single argument and returns True or False depending on whether that argument is considerd a valid value.
- **`default`** – is the default value of the field.
- **`default_factory`** – is a 0-argument function called to initialize a field's value.
- **`init`** – if True, the field will be a parameter to the class's __init__() function.
- **`repr`** – if True, the field will be included in the object's repr().
- **`hash`** – if True, the field will be included in the object's hash().
- **`compare`** – if True, the field will be used in comparison functions.
- **`metadata`** – if specified, must be a mapping which is stored but not otherwise examined by dataclass.

Raises **`ValueError`** – It is an error to specify both default and default_factory.

4.1.2 Sources

Source classes know how to fetch configuration values from all kinds of different sources of configuration values. A number of Source classes are provided by the library, and users can implement their own sources.

TODO: link to documentation on implementing custom sources.

Builtin sources:

`class` `configclasses.sources.`**`EnvironmentSource`**(*namespace=None*, *environ=os.environ*)

Get configuration values from case insensitive environment variables.

Parameters

- **namespace** – An optional string prefix to match on with environment variables.
- **environ** – A different source of environment variables can be passed if you don't want to use os.environ.

If `namespace` is provided, only environment variable names that start with the namespace value will be considered. The namespace is also stripped off the variable name before it is stored.

reload()
Fetch and parse values from the environment dict and store them.

class configclasses.sources.**DotEnvSource**(*path='.env'*, *filehandle=None*, *namespace=None*)
Get configuration values from a *.env* (dotenv) formatted file.

Parameters

- **path** – path to read from.
- **filehandle** – open file handle to read from.
- **namespace** – string prefix for values this sources will fetch from.

Raises **ValueError** – It is an error if both `path` and `filehandle` are defined *or* neither `path` nor `filehandle` are defined.

reload()
Fetch and parse values from the file source and store them.

If a `path` was provided to the source, the path will be reopened and read. If a `filehandle` was provided and the handle supports seeking, it will seek to the position the handle was at when passed to the source. If it does not support seeking, it will attempt to read from the current position.

It is up to the user to ensure that filehandles will act correctly given the above rules

class configclasses.sources.**CommandLineSource**(*argparse=None*, *argv=sys.argv*)
Get configuration values from command line arguments. Adds command line arguments for each field in the associated configclass.

Parameters

- **argparse** – Optionally pass in a preexisting *argparse.ArgumentParser* instance to add to an existing set of command line arguments rather than only using auto-generated command line arguments.
- **argv** – Optionally pass a custom argv list. Most useful for testing.

reload()
Child classes that have a sensible reload strategy should override this method

class configclasses.sources.**JsonSource**(*path=None*, *filehandle=None*, *namespace=None*)
Get configuration values from a json encoded file or filehandle.

Parameters

- **path** – path to read from.
- **filehandle** – open file handle to read from.
- **namespace** – list of keys or indices used to access a nested configuration object.

Raises **ValueError** – It is an error if both `path` and `filehandle` are defined *or* neither `path` nor `filehandle` are defined.

Namespacing for json sources is best described by example:

```
>>> json_value = """ {
...     "nested": {
...         "configuration": {
...             "FOO": "foo_value",
...             "BAR": "bar_value",
...         }
...     }
... }"""
>>> namespace = ["nested", "configuration"]
```

A `JsonSource` that reads a file with the contents of `json_value` with the `namespace` defined above would only consider the keys "FOO" and "BAR" as configuration values in scope.

reload()
Fetch and parse values from the file source and store them.

If a `path` was provided to the source, the path will be reopened and read. If a `filehandle` was provided and the handle supports seeking, it will seek to the position the handle was at when passed to the source. If it does not support seeking, it will attempt to read from the current position.

It is up to the user to ensure that filehandles will act correctly given the above rules

class configclasses.sources.**TomlSource**(*path=None*, *filehandle=None*, *namespace=None*)
Get configuration values from a *.toml* file.

Parameters

- **path** – path to read from.
- **filehandle** – open file handle to read from.
- **namespace** – optional list of nested section to search for configuration fields

Raises **ValueError** – It is an error if both `path` and `filehandle` are defined *or* neither `path` nor `filehandle` are defined.

reload()
Fetch and parse values from the file source and store them.

If a `path` was provided to the source, the path will be reopened and read. If a `filehandle` was provided and the handle supports seeking, it will seek to the position the handle was at when passed to the source. If it does not support seeking, it will attempt to read from the current position.

It is up to the user to ensure that filehandles will act correctly given the above rules

class configclasses.sources.**IniSource**(*path=None*, *filehandle=None*, *namespace=None*)
Get configuration values from a *.ini* file.

Parameters

- **path** – path to read from.
- **filehandle** – open file handle to read from.
- **namespace** – optional section to search for configuration fields

Raises **ValueError** – It is an error if both `path` and `filehandle` are defined *or* neither `path` nor `filehandle` are defined.

Note: Python ini parsing is case insensitive.

reload()
Fetch and parse values from the file source and store them.

If a `path` was provided to the source, the path will be reopened and read. If a `filehandle` was provided and the handle supports seeking, it will seek to the position the handle was at when passed to the source. If it does not support seeking, it will attempt to read from the current position.

It is up to the user to ensure that filehandles will act correctly given the above rules

class configclasses.sources.**ConsulSource**(*root*, *namespace=""*, *http=requests*)

Get configuration values from a remote consul key value store.

Parameters

- **root** – The address of the consul api to use. Don't forget to include the scheme (http or https)!
- **namespace** – The consul kv namespace from which to fetch fields.
- **http** – http library used to make get requests. Defaults to using requests.

reload()

Child classes that have a sensible reload strategy should override this method

4.1.3 Enums

Common configuration enums provided for user convenience. However, any subclass of python's `enum.Enum` will work as expected.

class configclasses.enums.**LogLevel**(*Enum*)

Python logging module log level constants represented as an `enum.Enum`.

NotSet = logging.NOTSET

Debug = logging.DEBUG

Info = logging.INFO

Warning = logging.WARNING

Error = logging.ERROR

Critical = logging.CRITICAL

class configclasses.enums.**Environment**(*Enum*)

Common environment names.

Development = 0

Test = 1

Staging = 2

Production = 3

4.1.4 Conversions

Conversion functions that can be specified as the `converter` in a configclass field.

configclasses.conversions.**csv_list**(*value: str*) → list

csv_lists are comma separated values. Whitespace around a value is stripped unless text is quoted. Empty values are skipped.

An example usage:

```
>>> csv_list("a,b,c")
["a", "b", "c"]
```

Typically it is used in specifying a configclass:

```
>>> @configclass
... class Configuration:
...     LIST: list = field(converter=csv_list)
```

Then a string of values will be converted into a list of strings in the `Configuration` class.

`configclasses.conversions.`**`csv_pairs`**(*value: str*) → dict

Kv lists are comma separated pairs of values where a pair is defined as `"key=value"`. Whitespace around a key or value is stripped unless text is quoted. Empty pairs are skipped.

Raises **`ValueError`** – on a malformed key value pair.

An example usage:

```
>>> csv_pairs("a=1,b=2")
{"a": "1", "b": "2"}
```

Typically it is used in specifying a configclass:

```
>>> @configclass
... class Configuration:
...     PAIRS: dict = field(converter=csv_pairs)
```

Then a string of key=value pairs will be converted into a dictionary in the `Configuration` class.

CHAPTER 5

Contribution

Contributors are the best!

5.1 Contribution

Feature requests, issues, and Pull Requests welcome.

If you want to add any new functionality, please file an issue to discuss it beforehand. That way we can all avoid code that conflicts with the goals and design philosophy of the project.

CHAPTER 6

License

Licensor solely permits licensee to license under either of the following two options

- MIT license
- Apache License, Version 2.0

6.1 LICENSE

Licensor solely permits licensee to license under *either* of the following two options:

- MIT license
- Apache License, Version 2.0

6.1.1 Contribution

Unless you explicitly state otherwise, any contribution intentionally submitted for inclusion in the work by you shall be dual licensed as above, without any additional terms or conditions.

6.1.2 MIT

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

OR

6.1.3 Apache, Version 2.0

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner.

For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide ad-

ditional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "{}" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright {yyyy} {name of copyright owner}

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

configclasses, 15
configclasses.enums, 19
configclasses.sources, 16

Index

C

CommandLineSource (class in configclasses.sources), 17
configclass() (in module configclasses), 15
configclasses (module), 15
configclasses.enums (module), 19
configclasses.sources (module), 16
ConsulSource (class in configclasses.sources), 19
csv_list() (in module configclasses.conversions), 19
csv_pairs() (in module configclasses.conversions), 20

D

DotEnvSource (class in configclasses.sources), 17

E

Environment (class in configclasses.enums), 19
EnvironmentSource (class in configclasses.sources), 16

F

field() (in module configclasses), 16

I

IniSource (class in configclasses.sources), 18

J

JsonSource (class in configclasses.sources), 17

L

LogLevel (class in configclasses.enums), 19

R

reload() (configclasses.sources.CommandLineSource method), 17
reload() (configclasses.sources.ConsulSource method), 19
reload() (configclasses.sources.DotEnvSource method), 17
reload() (configclasses.sources.EnvironmentSource method), 17
reload() (configclasses.sources.IniSource method), 18
reload() (configclasses.sources.JsonSource method), 18
reload() (configclasses.sources.TomlSource method), 18

T

TomlSource (class in configclasses.sources), 18