Dotconf Documentation

Release 1.6 dev

Antoine Millet

July 31, 2013

CONTENTS

Dotconf is an advanced configuration parsing library which can be used by developers who are limited with standard ConfigParser library. Dotconf is also shipped with a schema validator which can be used to check the content of your application's configuration file before to start.

CHAPTER

ONE

FEATURES

- Simple configuration format with sections
- Typing:
  - Typing in the syntax (“42” is a string, 42 a number)
  - Four types availables: string, boolean, number or list
- Validation of configuration using a schema
- Configuration includes

CHAPTER

TWO

EXAMPLE

This is an example of what you can do with Dotconf:

```
from dotconf.schema import many, once
from dotconf.schema.containers import Section, Value
from dotconf.schema.types import Boolean, Integer, Float, String

# Schema definition:

class UserSection(Section):
    password = Value(String())
    _meta = {'repeat': many, 'unique': True}

class PathSection(Section):
    rate_limit = Value(Float(), default=0)
    enable_auth = Value(Boolean(), default=False)
    user = UserSection()

class VirtualHostSection(Section):
    base_path = Value(String())
    enable_ssl = Value(Boolean(), default=False)
    path = PathSection()
    _meta = {'repeat': many, 'unique': True}

class MyWebserverConfiguration(Section):
    daemon = Value(Boolean()default=False)
    pidfile = Value(String(), default=None)
    interface = Value(String(), default='127.0.0.1:80')
    interface_ssl = Value(String(), default='127.0.0.1:443')
    host = VirtualHostSection()
```

Then, to use the parser:

```
>>> conf = '''
... daemon = True
... pidfile = '/var/run/myapp.pid'
... interface = '0.0.0.0:80'
... interface_ssl = '0.0.0.0:443'
...
... host 'example.org' {
...     path '/' {
...         rate_limit = 30
...     }
... }
...
```

```
... host 'protected.example.org' {
...     enable_ssl = yes
...
...     path '/files' {
...         enable_auth = yes
...         user 'foo' {
...             password = 'bar'
...         }
...     }
... }
... '''
>>> from dotconf import Dotconf
>>> from myconfschema import MyWebserverConfiguration
>>> parsed_conf = Dotconf(conf, schema=MyWebserverConfiguration)
>>> print 'daemon:', parsed_conf.get('daemon')
True
>>> for vhost in parsed_conf.subsections('host'):
>>>     print vhost.args[0]
>>>     if vhost.get('enable_ssl'):
>>>         print '  SSL enabled'
>>>     for path in vhost.subsections('path'):
>>>         print '  ' + path.args[0]
>>>         if path.get('enable_auth'):
>>>             print '    Following users can access to this directory:'
>>>             for user in path.subsections('user'):
>>>                 print '      - ' + user.args[0]
>>>
example.org
  /
protected.example.org
  SSL enabled
  /files
    Following users can access to this directory:
      - foo
```

CHAPTER

THREE

CONTENTS

3.1 Schema validation API

3.1.1 Containers

Section container

class dotconf.schema.containers.**Section**(***kwargs*)

A section container used to store a mapping between name and other containers.

Parameters ****kwargs** – parameters used to override meta

Typically, this class has to be derived to specify accepted key for the section:

```
class MySection(Section):

    a_key = Value(Integer())
```

A section can also be defined imperatively:

```
>>> section = MySection()
>>> section.add('another_key', Value(Integer()))
```

Metadata

A section can hold some metadata, for example to define arguments schema or how many time the section can appear. This metadata can be defined in several different ways:

- In the _meta dict of a section definition (metadata are inherited from parent classes):

  ```
  class MySection(Section):
      _meta = {'unique': True}
  ```

- In the meta dict of a section object:

  ```
  >>> section = MySection()
  >>> section.meta['unique'] = True
  ```

- In keyword parameters on instance initialization:

  ```
  >>> section = MySection(unique=True)
  ```

Accepted metadata

args The `args` metadata (default `None`) store the container used to validate the section argument (value between the section name and the opening brace). You can use any type of container but a `Section`.

If the metadata is set to `None` (the default), argument for the section is forbidden.

Example:

```
class MySection(Section):
    _meta = {'args': Value(String())}
```

Note: This meta can't be defined for the `__top__` section.

---

unique The `unique` metadata (default `False`) can be used to prevent multiple section to have the same arguments.

For example if you have a section with this kind of schema:

```
class MySubSection(Section):
    _meta = {'args': Value(String()), 'unique': True}

class MySection(Section):
    sub = MySubSection()
```

And a configuration file with this content:

```
sub "foo" {...}
sub "bar" {...}
sub "foo" {...}
```

The last "foo" section will throw a validation error because an another "sub" section already exists with this argument.

Note: This meta can't be defined for the `__top__` section.

---

repeat The `repeat` metadata (default to `(1, 1)`) allow to define how many time the section can appear. The first value is the minimum number of times, and the second the maximum number of time.

Values must be non-negative integers, and the first must be smaller or equal to the second. The second value can be set to None to express the infinite value.

Examples:

- `(1, 1)`: the section must appear once
- `(1, 2)`: the section must appear one or two times
- `(0, 1)`: the section is optionnal and can appear once
- `(0, None)`: the section is optionnal and can appear an infinite number of times
- `(1, None)`: the section mut appear at least once

Some shortcut are available in the `dotconf.schema.containers` module:

- `once` -> `(1, 1)`
- `many` -> `(0, None)`

Note: This meta can't be defined for the `__top__` section.

allow_unknown The `allow_unknow` metadata (default to `False`) allow the user to set value which are not defined in the section schema. Unknown data are then available in the validated section.

Value container

class dotconf.schema.containers.**Value**(*value_type*, *default=<object object at 0x7f1f767e06a0>*, ***kwargs*)

A value container used to store a scalar value of specified type.

Parameters

- **value_type** – the type of the value stored by container
- **default** – the default value of the container

List containers

class dotconf.schema.containers.**List**(*values_type*, *default=<object object at 0x7f1f767e06a0>*, ***kwargs*)

A list container used to store a list of scalar value of specified type.

Parameters

- **values_type** – type of values
- **default** – the default value of the container

Array containers

class dotconf.schema.containers.**Array**(*size*, **args*, ***kwargs*)

An array container used to store a fixed size list of scalar values of the specified type.

Parameters

- **size** – size of the array
- ****kwargs** – same arguments as List

class dotconf.schema.containers.**TypedArray**(*values_types*, *default=<object object at 0x7f1f767e06a0>*, ***kwargs*)

An array container used to store a fixed size list of scalar values with specified type for each of them.

Parameters

- **values_types** – types of each item in a list
- **default** – the default value of the container

3.1.2 Types

String based types

class dotconf.schema.types.**String**
A type representing a string in the configuration.

Example in configuration:

```
my_string = "hello, world!"
```

class dotconf.schema.types.**Regex**(*regex*, *error="value doesn't match"*)
A string based type validated against a regex.

class dotconf.schema.types.**NamedRegex**(*regex*, *error="value doesn't match"*)
A string based type like Regex but returning named groups in dict.

class dotconf.schema.types.**RegexPattern**(*flags=0*)
A re Python object.

Parameters **flags** – python re compile flag

Example in configuration:

```
match = '/[a-z]+(-[a-z]+)?\.css'
```

class dotconf.schema.types.**IPAddress**(*version=None*)
A string based type representing an ipv4 or ipv6 address.

This type require the "ipaddr" package to work and will return an `ipaddr.IPAddress` object.

Parameters **version** – type or ip address to validate, can be 4 (ipv4 addresses only), 6 (ipv6 addresses only), or None (both).

Example in configuration:

```
interface = "127.0.0.1"
```

class dotconf.schema.types.**IPNetwork**(*version=None*)
A string based type representing an ipv4 or ipv6 network.

This type require the "ipaddr" package to work and will return an `ipaddr.IPNetwork` object.

Parameters **version** – type or ip address to validate, can be 4 (ipv4 addresses only), 6 (ipv6 addresses only), or None (both).

Example in configuration:

```
allow = "10.0.0.0/8"
```

class dotconf.schema.types.**Url**
A string based type representing an URL.

This type return an urlparse.ParseResult object.

Example in configuration:

```
proxy = "http://proxy:3128"
```

class dotconf.schema.types.**IPSocketAddress**(*default_addr='127.0.0.1'*, *default_port=None*, *version=None*)
A string based type representing an (ip address, port) couple.

This type return an IPSocketAddress.Address object.

Example in configuration:

```
interface = "0.0.0.0:80"
```

class dotconf.schema.types.**Eval**(*locals=None*, *globals=None*)
A string base type evaluating string as Python expression.

Example in configuration:

```
sum = 'sum(range(3, 10))'
```

Warning: This type can be dangerous since any Python expression can be typed by the user, like __import__("sys").exit(). Use it at your own risk.

Number based types

class dotconf.schema.types.**Integer**(*min=None*, *max=None*)
A type representing an integer in the configuration.

Parameters

- **min** – define the minimum acceptable value value of the integer
- **max** – define the maximum acceptable value value of the integer

Example in configuration:

```
my_integer = 42
my_integer = 42.0  # Will also match this type
```

class dotconf.schema.types.**Float**
A type representing a float in the configuration.

Example in configuration:

```
my_float = 42.2
my_float = 42  # All values matched by the Integer type also match
               # for the Float type
```

Boolean based types

class dotconf.schema.types.**Boolean**
A type representing a boolean value in the configuration.

Example in configuration:

```
my_boolean = yes
```

3.1.3 Argparse integration

Dotconf provide an optionnal integration with the standard `argparse` module. This compatibility brings you a way to override some configuration values using a command line argument.

Define the schema

Each compatible `Container` can take five optionnals arguments:

- `argparse_names`: the list of argument names or flags, the usage of this argument enable feature for the container.
- `argparse_metavar`: the metavar value of the argument (read the argparse documentation for more informations).
- `argparse_help`: the help to display.
- `argparse_names_invert`: only for a flag value, create an argument to invert the boolean value (eg: for a `--daemon` argument, you can create a `--foreground` value which will force to disable the daemonization).
- `argparse_help_invert`: help message for the invert argument.

Example:

```
debug = Value(Boolean, argparse_names=['-d', '--debug'],
              argparse_help='enable the debug mode')
```

Populate the argument parser and use it

Once your schema is defined, you must call the `populate_argparse()` method providing the argument parser to populate:

```
parser = argparse.ArgumentParser()
schema = MySchema()
schema.populate_argparse(parser)
args = parser.parse_args()
config = Dotconf(conf, schema=schema)
my_config = config.parse()
```

Full featured example

```
from pprint import pprint
import argparse

from dotconf import Dotconf
from dotconf.schema import ValidationError
from dotconf.parser import ParsingError
from dotconf.schema.containers import Section, Value, List
from dotconf.schema.types import Boolean, String


config_test = '''
debug = no
paths = '/bin', '/usr/bin'
'''

class MySchema(Section):

    debug = Value(Boolean(), argparse_names=['-d', '--debug'],
                  argparse_help='enable the debug mode',
                  argparse_names_invert=['-q', '--quiet'],
                  argparse_help_invert='disable the debug mode')
    paths = List(String(), argparse_names=['--paths'],
```

```
                  argparse_help='list of paths to inspect')


if __name__ == '__main__':
    # 1. Create the argument parser:
    argument_parser = argparse.ArgumentParser()

    # 2. Create the schema:
    schema = MySchema()

    # 3. Populate the argument parser using schema:
    schema.populate_argparse(argument_parser)

    # 4. Parse command line arguments
    args = argument_parser.parse_args()

    # 5. Create the configuration parser:
    config = Dotconf(config_test, schema=schema)

    # 6. Parse the configuration and show it:
    try:
        pconfig = config.parse()
    except (ValidationError, ParsingError) as err:
        if err.position is not None:
            print str(err.position)
        print err
    else:
        pprint(pconfig.to_dict())
```

And an execution example:

```
$ ./argparse_example.py --help
usage: ./argparse_example.py [-h] [-d] [--paths [PATHS [PATHS ...]]]

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           enable the debug mode
  -q, --quiet           disable the debug mode
  --paths [PATHS [PATHS ...]]
                        list of paths to inspect
$ ./argparse_example.py
{'debug': False, 'paths': ['/bin', '/usr/bin']}
$ ./argparse_example.py -d
{'debug': True, 'paths': ['/bin', '/usr/bin']}
$ ./argparse_example.py --paths
{'debug': False, 'paths': []}
$ ./argparse_example.py --paths /sbin /usr/sbin
{'debug': False, 'paths': ['/sbin', '/usr/sbin']}
```

3.2 Tips 'n Tricks

3.2.1 Validate plugins configuration

If your application use a plugin system, and you want to validate a configuration which can be different for each plugin, this trick is for you.

The idea is pretty simple, do a two-step validation: the first one will validate all the common configuration, and the second step validate each plugin's configuration. To avoid ValidationError in the first step, you use the `allow_unknown` feature of Sections.

Here is a full example:

```
import sys
from pprint import pprint

from dotconf import Dotconf
from dotconf.schema import ValidationError
from dotconf.parser import ParsingError
from dotconf.schema.containers import Section, Value, many
from dotconf.schema.types import Boolean, Integer, String

#
# Our Dotconf schemas:
#

class GenericPluginSchema(Section):

    # This value is common to all plugins:
    common_value = Value(Integer())

    # Enable the allow_unknown meta in order to keep unknown values
    # on the first validation:
    _meta = {'allow_unknown': True,
             'repeat': many,
             'args': Value(String())}


class MainConfigurationSchema(Section):

    global_value = Value(Integer())
    plugin = GenericPluginSchema()


#
# Our plugins:
#

class BasePlugin(object):

    name = None
    schema = None

    def __init__(self, config):
        self.config = config

    def print_config(self):
        print '%s: ' % self.__class__.__name__
        pprint(self.config.to_dict())


class BeautifulSchema(GenericPluginSchema):

    beautiful_value = Value(String())

    # The allow_unknown meta is disable to forbid bad config:
```

```
    _meta = {'allow_unknown': False}


class BeautifulPlugin(BasePlugin):

    name = 'beautiful'
    schema = BeautifulSchema()


class UglySchema(GenericPluginSchema):

    ugly_value = Value(Boolean())
    _meta = {'allow_unknown': False}


class UglyPlugin(BasePlugin):

    name = 'ugly'
    schema = UglySchema()



#
# Our test config
#

TEST_CONFIG = '''
global_value = 42

plugin 'beautiful' {
    common_value = 123
    beautiful_value = "I'm so beautiful"
}

plugin 'ugly' {
    common_value = 456
    ugly_value = no
}
'''

#
# This is where the magic happen:
#

if __name__ == '__main__':
    my_plugins = (BeautifulPlugin, UglyPlugin)
    enabled_plugins = []

    # Parse the global configuration:
    config = Dotconf(TEST_CONFIG, schema=MainConfigurationSchema())
    try:
        pconfig = config.parse()
    except (ValidationError, ParsingError) as err:
        if err.position is not None:
            print str(err.position)
        print err
        sys.exit(1)
    else:
```

```
        print 'Main configuration:'
        pprint(pconfig.to_dict())

    # Enable each used plugins:
    for plugin_conf in pconfig.subsections('plugin'):
        # Search the plugin:
        for plugin in my_plugins:
            if plugin.name == plugin_conf.args:
                break
        else:
            print 'Unknown plugin %r, exiting.' % plugin_conf.args
            sys.exit(1)
        # Check plugin configuration:
        try:
            validated_conf = plugin.schema.validate(plugin_conf)
        except ValidationError as err:
            print 'Bad plugin configuration:'
            if err.position is not None:
                print str(err.position)
            print err
            sys.exit(1)
        else:
            # Instanciate the plugin object:
            enabled_plugins.append(plugin(validated_conf))

    # Print each enabled plugin config:
    for plugin in enabled_plugins:
        print '\n' + '~' * 80 + '\n'
        plugin.print_config()
```

And the output:

```
Main configuration:
{'global_value': 42,
 'plugin': [{'beautiful_value': "I'm so beautiful", 'common_value': 123},
            {'common_value': 456, 'ugly_value': False}]}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

BeautifulPlugin:
{'beautiful_value': "I'm so beautiful", 'common_value': 123}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

UglyPlugin:
{'common_value': 456, 'ugly_value': False}
```

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*