OmegaConf Documentation

Release 2.0.6

Omry Yadan

Mar 11, 2021

Contents

CHAPTER 1

Overview

OmegaConf is a YAML based hierarchical configuration system, with support for merging configurations from multiple sources (files, CLI argument, environment variables) providing a consistent API regardless of how the configuration was created. OmegaConf also offers runtime type safety via Structured Configs.

1.1 Installation

Just pip install:

```
pip install omegaconf
```

OmegaConf requires Python 3.6 and newer.

1.2 Creating

You can create OmegaConf objects from multiple sources.

1.2.1 Empty

```
>>> from omegaconf import OmegaConf
>>> conf = OmegaConf.create()
>>> print(OmegaConf.to_yaml(conf))
{}
```

1.2.2 From a dictionary

```
>>> conf = OmegaConf.create({"k" : "v", "list" : [1, {"a": "1", "b": "2"}]})
>>> print(OmegaConf.to_yaml(conf))
k: v
list:
- 1
- a: '1'
  b: '2'
```

1.2.3 From a list

```
>>> conf = OmegaConf.create([1, {"a":10, "b": {"a":10}}])
>>> print(OmegaConf.to_yaml(conf))
- 1
- a: 10
  b:
    a: 10
```

Tuples are supported as an valid option too.

1.2.4 From a YAML file

```
>>> conf = OmegaConf.load('source/example.yaml')
>>> # Output is identical to the YAML file
>>> print(OmegaConf.to_yaml(conf))
server:
  port: 80
log:
  file: ???
  rotation: 3600
users:
- user1
- user2
```

1.2.5 From a YAML string

```
>>> s = """
... a: b
... b: c
... list:
... - item1
... - item2
... """
>>> conf = OmegaConf.create(s)
>>> print(OmegaConf.to_yaml(conf))
a: b
b: c
list:
- item1
- item2
```

1.2.6 From a dot-list

```
>>> dot_list = ["a.aa.aaa=1", "a.aa.bbb=2", "a.bb.aaa=3", "a.bb.bbb=4"]
>>> conf = OmegaConf.from_dotlist(dot_list)
>>> print(OmegaConf.to_yaml(conf))
a:
  aa:
    aaa: 1
    bbb: 2
  bb:
    aaa: 3
    bbb: 4
```

1.2.7 From command line arguments

To parse the content of sys.arg:

```
>>> # Simulating command line arguments
>>> sys.argv = ['your-program.py', 'server.port=82', 'log.file=log2.txt']
>>> conf = OmegaConf.from_cli()
>>> print(OmegaConf.to_yaml(conf))
server:
  port: 82
log:
  file: log2.txt
```

1.2.8 From structured config

New in OmegaConf 2.0, API Considered experimental and may change.

You can create OmegaConf objects from structured config classes or objects. This provides static and runtime type safety. See *Structured config* for more details, or keep reading for a minimal example.

```
>>> from dataclasses import dataclass
>>> @dataclass
... class MyConfig:
...     port: int = 80
...     host: str = "localhost"
>>> # For strict typing purposes, prefer OmegaConf.structured() when creating␣
↪structured configs
>>> conf = OmegaConf.structured(MyConfig)
>>> print(OmegaConf.to_yaml(conf))
port: 80
host: localhost
```

You can use an object to initialize the config as well:

```
>>> conf = OmegaConf.structured(MyConfig(port=443))
>>> print(OmegaConf.to_yaml(conf))
port: 443
host: localhost
```

OmegaConf objects constructed from Structured classes offers runtime type safety:

```
>>> conf.port = 42       # Ok, type matches
>>> conf.port = "1080"   # Ok! "1080" can be converted to an int
>>> conf.port = "oops"   # "oops" cannot be converted to an int
Traceback (most recent call last):
...
omegaconf.errors.ValidationError: Value 'oops' could not be converted to Integer
```

In addition, the config class can be used as type annotation for static type checkers or IDEs:

```
>>> def foo(conf: MyConfig):
...     print(conf.port) # passes static type checker
...     print(conf.pork) # fails static type checker
```

1.3 Access and manipulation

Input YAML file for this section:

```
server:
  port: 80
log:
  file: ???
  rotation: 3600
users:
  - user1
  - user2
```

1.3.1 Access

```
>>> # object style access of dictionary elements
>>> conf.server.port
80

>>> # dictionary style access
>>> conf['log']['rotation']
3600

>>> # items in list
>>> conf.users[0]
'user1'
```

1.3.2 Default values

You can provide default values directly in the accessing code:

```
>>> # providing default values
>>> conf.missing_key or 'a default value'
'a default value'

>>> conf.get('missing_key', 'a default value')
'a default value'
```

1.3.3 Mandatory values

Use the value ??? to indicate parameters that need to be set prior to access

```
>>> conf.log.file
Traceback (most recent call last):
...
omegaconf.MissingMandatoryValue: log.file
```

1.3.4 Manipulation

```
>>> # Changing existing keys
>>> conf.server.port = 81

>>> # Adding new keys
>>> conf.server.hostname = "localhost"

>>> # Adding a new dictionary
>>> conf.database = {'hostname': 'database01', 'port': 3306}
```

1.4 Serialization

OmegaConf objects can be saved and loaded with OmegaConf.save() and OmegaConf.load(). The created file is in YAML format. Save and load can operate on file-names, Paths and file objects.

1.4.1 Save/Load YAML file

```
>>> conf = OmegaConf.create({"foo": 10, "bar": 20})
>>> with tempfile.NamedTemporaryFile() as fp:
...     OmegaConf.save(config=conf, f=fp.name)
...     loaded = OmegaConf.load(fp.name)
...     assert conf == loaded
```

Note that this does not retain type information.

1.4.2 Save/Load pickle file

Use pickle to save and load while retaining the type information. Note that the saved file may be incompatible across different major versions of OmegaConf.

```
>>> conf = OmegaConf.create({"foo": 10, "bar": 20})
>>> with tempfile.TemporaryFile() as fp:
...     pickle.dump(conf, fp)
...     fp.flush()
...     assert fp.seek(0) == 0
...     loaded = pickle.load(fp)
...     assert conf == loaded
```

1.5 Variable interpolation

OmegaConf support variable interpolation, Interpolations are evaluated lazily on access.

1.5.1 Config node interpolation

The interpolated variable can be the dot-path to another node in the configuration, and in that case the value will be the value of that node.

Input YAML file:

```
server:
  host: localhost
  port: 80

client:
  url: http://${server.host}:${server.port}/
  server_port: ${server.port}
```

Example:

```
>>> conf = OmegaConf.load('source/config_interpolation.yaml')
>>> # Primitive interpolation types are inherited from the referenced value
>>> print(conf.client.server_port)
80
>>> print(type(conf.client.server_port).__name__)
int

>>> # Composite interpolation types are always string
>>> print(conf.client.url)
http://localhost:80/
>>> print(type(conf.client.url).__name__)
str
```

1.5.2 Environment variable interpolation

Environment variable interpolation is also supported.

Input YAML file:

```
user:
  name: ${env:USER}
  home: /home/${env:USER}
```

```
>>> conf = OmegaConf.load('source/env_interpolation.yaml')
>>> print(conf.user.name)
omry
>>> print(conf.user.home)
/home/omry
```

You can specify a default value to use in case the environment variable is not defined. The following example sets *12345* as the the default value for the *DB_PASSWORD* environment variable.

```
>>> cfg = OmegaConf.create({
...       'database': {'password': '${env:DB_PASSWORD,12345}'}
... })
>>> print(cfg.database.password)
12345
>>> OmegaConf.clear_cache(cfg) # clear resolver cache
>>> os.environ["DB_PASSWORD"] = 'secret'
>>> print(cfg.database.password)
secret
```

1.5.3 Custom interpolations

You can add additional interpolation types using custom resolvers. This example creates a resolver that adds 10 the the given value.

```
>>> OmegaConf.register_resolver("plus_10", lambda x: int(x) + 10)
>>> c = OmegaConf.create({'key': '${plus_10:990}'})
>>> c.key
1000
```

Custom resolvers support variadic argument lists in the form of a comma separated list of zero or more values. Whitespaces are stripped from both ends of each value ("foo,bar" is the same as "foo, bar "). You can use literal commas and spaces anywhere by escaping (`\,` and `\ `).

```
>>> OmegaConf.register_resolver("concat", lambda x,y: x+y)
>>> c = OmegaConf.create({
...     'key1': '${concat:Hello,World}',
...     'key_trimmed': '${concat:Hello , World}',
...     'escape_whitespace': '${concat:Hello,\ World}',
... })
>>> c.key1
'HelloWorld'
>>> c.key_trimmed
'HelloWorld'
>>> c.escape_whitespace
'Hello World'
```

1.6 Merging configurations

Merging configurations enables the creation of reusable configuration files for each logical component instead of a single config file for each variation of your task.

Machine learning experiment example:

```
conf = OmegaConf.merge(base_cfg, model_cfg, optimizer_cfg, dataset_cfg)
```

Web server configuration example:

```
conf = OmegaConf.merge(server_cfg, plugin1_cfg, site1_cfg, site2_cfg)
```

The following example creates two configs from files, and one from the cli. It then combines them into a single object. Note how the port changes to 82, and how the users lists are combined.

example2.yaml file:

```
server:
  port: 80
users:
  - user1
  - user2
```

example3.yaml file:

```
log:
  file: log.txt
```

```
>>> from omegaconf import OmegaConf
>>> import sys
>>>
>>> # Simulate command line arguments
>>> sys.argv = ['program.py', 'server.port=82']
>>>
>>> base_conf = OmegaConf.load('source/example2.yaml')
>>> second_conf = OmegaConf.load('source/example3.yaml')
>>> cli_conf = OmegaConf.from_cli()
>>>
>>> # merge them all
>>> conf = OmegaConf.merge(base_conf, second_conf, cli_conf)
>>> print(OmegaConf.to_yaml(conf))
server:
  port: 82
users:
- user1
- user2
log:
  file: log.txt
```

1.7 Configuration flags

OmegaConf support several configuration flags. Configuration flags can be set on any configuration node (Sequence or Mapping). if a configuration flag is not set it inherits the value from the parent of the node. The default value inherited from the root node is always false.

1.7.1 Read-only flag

A read-only configuration cannot be modified. An attempt to modify it will result in omegaconf.ReadonlyConfigError exception

```
>>> conf = OmegaConf.create({"a": {"b": 10}})
>>> OmegaConf.set_readonly(conf, True)
>>> conf.a.b = 20
Traceback (most recent call last):
...
omegaconf.ReadonlyConfigError: a.b
```

You can temporarily remove the read only flag from a config object:

```
>>> conf = OmegaConf.create({"a": {"b": 10}})
>>> OmegaConf.set_readonly(conf, True)
>>> with read_write(conf):
...   conf.a.b = 20
>>> conf.a.b
20
```

1.7.2 Struct flag

By default, OmegaConf dictionaries allow read and write access to unknown fields. If a field does not exist, accessing it will return None and writing it will create the field. It's sometime useful to change this behavior.

```
>>> conf = OmegaConf.create({"a": {"aa": 10, "bb": 20}})
>>> OmegaConf.set_struct(conf, True)
>>> conf.a.cc = 30
Traceback (most recent call last):
...
omegaconf.errors.ConfigAttributeError: Error setting cc=30 : Key 'cc' in not in struct
    full_key: a.cc
    reference_type=Any
    object_type=dict
```

You can temporarily remove the struct flag from a config object:

```
>>> conf = OmegaConf.create({"a": {"aa": 10, "bb": 20}})
>>> OmegaConf.set_struct(conf, True)
>>> with open_dict(conf):
...   conf.a.cc = 30
>>> conf.a.cc
30
```

1.8 Utility functions

1.8.1 OmegaConf.is_missing

Tests if a value is missing ('???').

```
>>> cfg = OmegaConf.create({
...         "foo" : 10,
...         "bar": "???"
...     })
>>> assert not OmegaConf.is_missing(cfg, "foo")
>>> assert OmegaConf.is_missing(cfg, "bar")
```

1.8.2 OmegaConf.is_interpolation

Tests if a value is an interpolation.

```
>>> cfg = OmegaConf.create({
...         "foo" : 10,
...         "bar": "${foo}"
```

(continues on next page)

(continued from previous page)

```
...     })
>>> assert not OmegaConf.is_interpolation(cfg, "foo")
>>> assert OmegaConf.is_interpolation(cfg, "bar")
```

1.8.3 OmegaConf.is_none

Tests if a value is None.

```
>>> cfg = OmegaConf.create({
...         "foo" : 10,
...         "bar": None,
...     })
>>> assert not OmegaConf.is_none(cfg, "foo")
>>> assert OmegaConf.is_none(cfg, "bar")
>>> # missing keys are interpreted as None
>>> assert OmegaConf.is_none(cfg, "no_such_key")
```

1.8.4 OmegaConf.{is_config, is_dict, is_list}

Tests if an object is an OmegaConf object, or if it's representing a list or a dict.

```
>>> # dict:
>>> d = OmegaConf.create({"foo": "bar"})
>>> assert OmegaConf.is_config(d)
>>> assert OmegaConf.is_dict(d)
>>> assert not OmegaConf.is_list(d)
>>> # list:
>>> l = OmegaConf.create([1,2,3])
>>> assert OmegaConf.is_config(l)
>>> assert OmegaConf.is_list(l)
>>> assert not OmegaConf.is_dict(l)
```

1.8.5 OmegaConf.to_container

OmegaConf config objects looks very similar to python dict and list, but in fact are not. Use OmegaConf.to_container(cfg : Container, resolve : bool) to convert to a primitive container. If resolve is set to True, interpolations will be resolved during conversion.

```
>>> conf = OmegaConf.create({"foo": "bar", "foo2": "${foo}"})
>>> assert type(conf) == DictConfig
>>> primitive = OmegaConf.to_container(conf)
>>> assert type(primitive) == dict
>>> print(primitive)
{'foo': 'bar', 'foo2': '${foo}'}
>>> resolved = OmegaConf.to_container(conf, resolve=True)
>>> print(resolved)
{'foo': 'bar', 'foo2': 'bar'}
```

1.8.6 OmegaConf.select

OmegaConf.select() allow you to select a config node or value using a dot-notation key.

```
>>> cfg = OmegaConf.create({
...     "foo" : {
...         "bar": {
...             "zonk" : 10,
...             "missing" : "???"
...         }
...     }
... })
>>> assert OmegaConf.select(cfg, "foo") == {
...     "bar":  {
...         "zonk" : 10,
...         "missing" : "???"
...     }
... }
>>> assert OmegaConf.select(cfg, "foo.bar") == {
...     "zonk" : 10,
...     "missing" : "???"
... }
>>> assert OmegaConf.select(cfg, "foo.bar.zonk") == 10
>>> assert OmegaConf.select(cfg, "foo.bar.missing") is None
>>> OmegaConf.select(cfg,
...     "foo.bar.missing",
...     throw_on_missing=True
... )
Traceback (most recent call last):
...
omegaconf.errors.MissingMandatoryValue: missing node selected
    full_key: foo.bar.missing
```

1.8.7 OmegaConf.update

OmegaConf.update() allow you to update values in your config using a dot-notation key.

The merge flag controls the behavior if the input is a dict or a list. If it's true, those are merged instead of being assigned.

```
>>> cfg = OmegaConf.create({"foo" : {"bar": 10}})
>>> OmegaConf.update(cfg, "foo.bar", 20, merge=True) # merge has no effect because␣
↪the value is a primitive
>>> assert cfg.foo.bar == 20
>>> OmegaConf.update(cfg, "foo.bar", {"zonk" : 30}, merge=False) # set
>>> assert cfg.foo.bar == {"zonk" : 30}
>>> OmegaConf.update(cfg, "foo.bar", {"oompa" : 40}, merge=True) # merge
>>> assert cfg.foo.bar == {"zonk" : 30, "oompa" : 40}
```

1.8.8 OmegaConf.masked_copy

Creates a copy of a DictConfig that contains only specific keys.

```
>>> conf = OmegaConf.create({"a": {"b": 10}, "c":20})
>>> print(OmegaConf.to_yaml(conf))
a:
  b: 10
c: 20
```

(continues on next page)

(continued from previous page)

```
>>> c = OmegaConf.masked_copy(conf, ["a"])
>>> print(OmegaConf.to_yaml(c))
a:
  b: 10
```

1.9 Structured config

Structured configs are used to create OmegaConf configuration object with runtime type safety. In addition, they can be used with tools like mypy or your IDE for static type checking.

Two types of structures classes that are supported: dataclasses and attr classes.

- dataclasses are standard as of Python 3.7 or newer and are available in Python 3.6 via the *dataclasses* pip package.
- attrs Offset slightly cleaner syntax in some cases but depends on the attrs pip package.

This documentation will use dataclasses, but you can use the annotation *@attr.s(auto_attribs=True)* from attrs instead of *@dataclass*.

Basic usage involves passing in a structured config class or instance to OmegaConf.structured(), which will return an OmegaConf config that matches the values and types specified in the input. OmegaConf will validate modifications to the created config object at runtime against the schema specified in the input class.

1.9.1 Simple types

Simple types include

- int : numeric integers
- float : numeric floating point values
- bool : boolean values (True, False, On, Off etc)
- str : Any string
- Enums : User defined enums

The following class defines fields with all simple types:

```
>>> class Height(Enum):
...     SHORT = 0
...     TALL = 1

>>> @dataclass
... class SimpleTypes:
...     num: int = 10
...     pi: float = 3.1415
...     is_awesome: bool = True
...     height: Height = Height.SHORT
...     description: str = "text"
```

You can create a config based on the SimpleTypes class itself or instances of it. Those would be equivalent by default, but the Object variant allows you to set the values of specific fields during construction.

```
>>> conf1 = OmegaConf.structured(SimpleTypes)
>>> conf2 = OmegaConf.structured(SimpleTypes())
>>> # The two configs are identical in this case
>>> assert conf1 == conf2
>>> # But the second form allow for easy customization of the values:
>>> conf3 = OmegaConf.structured(
...   SimpleTypes(num=20,
...   height=Height.TALL))
>>> print(OmegaConf.to_yaml(conf3))
num: 20
pi: 3.1415
is_awesome: true
height: TALL
description: text
```

The resulting object is a regular OmegaConf DictConfig, except it will utilize the type information in the input class/object and will validate the data at runtime. The resulting object and will also rejects attempts to access or set fields that are not already defined (similarly to configs with their to *Struct flag* set, but not recursive).

```
>>> conf = OmegaConf.structured(SimpleTypes)
>>> with raises(AttributeError):
...     conf.does_not_exist
```

You can create a config with specified fields that can also accept arbitrary values by extending Dict:

```
>>> @dataclass
... class DictWithFields(Dict[str, Any]):
...     num : int = 10
>>>
>>> conf = OmegaConf.structured(DictWithFields)
>>> assert conf.num == 10
>>>
>>> conf.foo = "bar"
>>> assert conf.foo == "bar"
```

1.9.2 Static type checker support

Python type annotation can be used by static type checkers like Mypy/Pyre or by IDEs like PyCharm.

```
>>> conf: SimpleTypes = OmegaConf.structured(SimpleTypes)
>>> # Passes static type checking
>>> conf.description = "text"
>>> # Fails static type checking (but will also raise a Validation error)
>>> with raises(ValidationError):
...     conf.num = "foo"
```

This is duck-typing, the actual object type of *conf* is *DictConfig*. You can access the underlying type using *OmegaConf.get_type()*:

```
>>> type(conf).__name__
'DictConfig'

>>> OmegaConf.get_type(conf).__name__
'SimpleTypes'
```

1.9.3 Runtime type validation and conversion

OmegaConf supports merging configs together, as well as overriding from the command line. This means some mistakes can not be identified by static type checkers, and runtime validation is required.

```
>>> # This is okay, the string "100" can be converted to an int
>>> # Note that static type checkers will not like it and you should
>>> # avoid such explicit mistyped assignments.
>>> conf.num = "100"
>>> assert conf.num == 100

>>> with raises(ValidationError):
...     # This will fail at runtime because num is an int
...     # and foo cannot be converted to an int
...     # Note that the static type checker can't help here.
...     conf.merge_with_dotlist(["num=foo"])
```

Runtime validation and conversion works for all supported types, including Enums:

```
>>> conf.height = Height.TALL
>>> assert conf.height == Height.TALL

>>> # The name of Height.TALL is TALL
>>> conf.height = "TALL"
>>> assert conf.height == Height.TALL

>>> # The ordinal of Height.TALL is 1
>>> conf.height = 1
>>> assert conf.height == Height.TALL
```

1.9.4 Nesting structured configs

Structured configs can be nested.

```
>>> @dataclass
... class User:
...     # A simple user class with two missing fields
...     name: str = MISSING
...     height: Height = MISSING
>>>
>>> @dataclass
... class DuperUser(User):
...     duper: bool = True
...
>>> # Group class contains two instances of User.
>>> @dataclass
... class Group:
...     name: str = MISSING
...     # data classes can be nested
...     admin: User = User()
...
...     # You can also specify different defaults for nested classes
...     manager: User = User(name="manager", height=Height.TALL)

>>> conf : Group = OmegaConf.structured(Group)
>>> print(OmegaConf.to_yaml(conf))
```

(continues on next page)

(continued from previous page)

```
name: ???
admin:
  name: ???
  height: ???
manager:
  name: manager
  height: TALL
```

OmegaConf will validate that assignment of nested objects is of the correct type:

```
>>> with raises(ValidationError):
...     conf.manager = 10
```

You can assign subclasses:

```
>>> conf.manager = DuperUser()
>>> assert conf.manager.duper == True
```

1.10 Containers

Python container types are fully supported in Structured configs.

1.10.1 Lists

Lists can hold any type supported by OmegaConf (int, float. bool, str, enum and Structured configs). Lists can be created from Lists and Tuples.

```
>>> from typing import List, Tuple
>>> @dataclass
... class User:
...     name: str = MISSING

>>> @dataclass
... class Lists:
...     # Typed list can hold Any, int, float, bool, str and Enums as well
...     # as arbitrary Structured configs
...     ints: List[int] = field(default_factory=lambda: [10, 20, 30])
...     bools: Tuple[bool, bool] = field(default_factory=lambda: (True, False))
...     users: List[User] = field(default_factory=lambda: [User(name="omry")])
```

OmegaConf verifies at runtime that your Lists contains only values of the correct type.

```
>>> conf : Lists = OmegaConf.structured(Lists)

>>> # Okay, 10 is an int
>>> conf.ints.append(10)
>>> # Okay, "20" can be converted to an int
>>> conf.ints.append("20")

>>> conf.bools.append(True)
>>> conf.users.append(User(name="Joe"))
>>> # Not okay, 10 cannot be converted to a User
```

(continues on next page)

(continued from previous page)

```
>>> with raises(ValidationError):
...     conf.users.append(10)
```

1.10.2 Dictionaries

Dictionaries are supported as well. Keys must be strings or enums, and values can be any of any type supported by OmegaConf (Any, int, float, bool, str and Enums as well as arbitrary Structured configs)

1.11 Misc

OmegaConf supports field modifiers such as MISSING and Optional.

```
>>> from typing import Optional
>>> from omegaconf import MISSING

>>> @dataclass
... class Modifiers:
...     num: int = 10
...     optional_num: Optional[int] = 10
...     another_num: int = MISSING

>>> conf : Modifiers = OmegaConf.structured(Modifiers)
```

1.11.1 Mandatory missing values

Fields assigned the constant MISSING do not have a value and the value must be set prior to accessing the field Otherwise a MissingMandatoryValue exception is raised.

```
>>> with raises(MissingMandatoryValue):
...     x = conf.another_num
>>> conf.another_num = 20
>>> assert conf.another_num == 20
```

1.11.2 Optional fields

```
>>> with raises(ValidationError):
...     # regular fields cannot be assigned None
...     conf.num = None

>>> conf.optional_num = None
>>> assert conf.optional_num is None
```

1.11.3 Interpolations

Variable interpolation works normally with Structured configs but static type checkers may object to you assigning a string to an other types. To work around it, use SI and II described below.

```
>>> from omegaconf import SI, II
>>> @dataclass
... class Interpolation:
...     val: int = 100
...     # This will work, but static type checkers will complain
...     a: int = "${val}"
...     # This is identical to the above, but static type checkers
...     # will not complain
...     b: int = SI("${val}")
...     # This is a syntactic sugar, the input string is
...     # wrapped with ${} automatically.
...     c: int = II("val")

>>> conf : Interpolation = OmegaConf.structured(Interpolation)
>>> assert conf.a == 100
>>> assert conf.b == 100
>>> assert conf.c == 100
```

Type validation is performed on assignment, but not on values returned by interpolation, e.g:

```
>>> from omegaconf import SI
>>> @dataclass
... class Interpolation:
...     int_key: int = II("str_key")
...     str_key: str = "string"

>>> cfg = OmegaConf.structured(Interpolation)
>>> assert cfg.int_key == "string"
```

1.11.4 Frozen

Frozen dataclasses and attr classes are supported via OmegaConf *Read-only flag*, which turns the entire config node and all if it's child nodes read-only.

```
>>> @dataclass(frozen=True)
... class FrozenClass:
...     x: int = 10
...     list: List = field(default_factory=lambda: [1, 2, 3])

>>> conf = OmegaConf.structured(FrozenClass)
>>> with raises(ReadonlyConfigError):
...     conf.x = 20
```

Read-only flag is recursive:

```
>>> with raises(ReadonlyConfigError):
...     conf.list[0] = 20
```

1.12 Merging with other configs

Once an OmegaConf object is created, it can be merged with others regardless of its source. OmegaConf configs created from Structured configs contains type information that is enforced at runtime. This can be used to validate config files based on a schema specified in a structured config class

example.yaml file:

```
server:
  port: 80
log:
  file: ???
  rotation: 3600
users:
  - user1
  - user2
```

A Schema for the above config can be defined like this.

```
>>> @dataclass
... class Server:
...     port : int = MISSING

>>> @dataclass
... class Log:
...     file : str = MISSING
...     rotation: int = MISSING

>>> @dataclass
... class MyConfig:
...     server : Server = Server()
...     log : Log = Log()
...     users : List[int] = field(default_factory=list)
```

I intentionally made an error in the type of the users list (List[int] should be List[str]). This will cause a validation error when merging the config from the file with that from the scheme.

```
>>> schema = OmegaConf.structured(MyConfig)
>>> conf = OmegaConf.load("source/example.yaml")
>>> with raises(ValidationError):
...     OmegaConf.merge(schema, conf)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search