OmegaConf Documentation

Release 2.3.0

Omry Yadan

Dec 20, 2022

CONTENTS

CHAPTER

ONE

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 Usage

1.1.1 Installation

Just pip install:

```
pip install omegaconf
```

OmegaConf requires Python 3.6 and newer.

1.1.2 Creating

You can create OmegaConf objects from multiple sources.

Empty

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

From a dictionary

```
>>> conf = OmegaConf.create({"k" : "v", "list" : [1, {"a": "1", "b": "2", 3: "c"}]})
>>> print(OmegaConf.to_yaml(conf))
k: v
list:
- 1
- a: '1'
  b: '2'
  3: c
```

Here is an example of various supported key types:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     BLUE = 2
>>>
>>> conf = OmegaConf.create(
...     {"key": "str", 123: "int", True: "bool", 3.14: "float", Color.RED: "Color", b"123
↪": "bytes"}
... )
>>>
>>> print(conf)
{'key': 'str', 123: 'int', True: 'bool', 3.14: 'float', <Color.RED: 1>: 'Color', b'123':
↪'bytes'}
```

OmegaConf supports `str`, `int`, `bool`, `float` `bytes`, and `Enum` as dictionary key types.

From a list

```
>>> conf = OmegaConf.create([1, {"a":10, "b": {"a":10, 123: "int_key"}}])
>>> print(OmegaConf.to_yaml(conf))
- 1
- a: 10
  b:
    a: 10
    123: int_key
```

Tuples are supported as a valid option too.

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
```

From a YAML string

```
>>> s = """
... a: b
... b: c
... list:
... - item1
... - item2
... 123: 456
... """
>>> conf = OmegaConf.create(s)
>>> print(OmegaConf.to_yaml(conf))
a: b
b: c
list:
- item1
- item2
123: 456
```

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
```

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
```

From structured config

You can create OmegaConf objects from structured config classes or objects. This provides static and runtime type safety. See *Structured Configs* 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 provide 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
```

1.1.3 Access and manipulation

Input YAML file for this section:

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

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'
```

Default values

You can provide default values directly in the accessing code:

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

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
```

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}
```

1.1.4 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.

Save/Load YAML file

```
>>> conf = OmegaConf.create({"foo": 10, "bar": 20, 123: 456})
>>> 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.

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 versions of OmegaConf.

```
>>> conf = OmegaConf.create({"foo": 10, "bar": 20, 123: 456})
>>> with tempfile.TemporaryFile() as fp:
...     pickle.dump(conf, fp)
...     fp.flush()
...     assert fp.seek(0) == 0
...     loaded = pickle.load(fp)
...     assert conf == loaded
```

Note for Python3.6 users: due to limitations in pickling support, *structured configs* with complex type hints (such as *nested container types* or *containers with optional element types*) cannot be pickled using Python3.6.

1.1.5 Variable interpolation

OmegaConf supports variable interpolation. Interpolations are evaluated lazily on access.

Config node interpolation

The interpolated variable can be the path to another node in the configuration, and in that case the value will be the value of that node. This path may use either dot-notation (`foo.1`), brackets (`[foo][1]`) or a mix of both (`foo[1]`, `[foo].1`).

Interpolations are absolute by default. Relative interpolation are prefixed by one or more dots: The first dot denotes the level of the node itself and additional dots are going up the parent hierarchy. e.g. `${..foo}` points to the `foo` sibling of the parent of the current node.

NOTE: Interpolations may cause config cycles. Such cycles are forbidden and may cause undefined behavior.

Input YAML file:

```
server:
  host: localhost
  port: 80

client:
  url: http://${server.host}:${server.port}/
  server_port: ${server.port}
  # relative interpolation
  description: Client of ${.url}
```

Example:

```
>>> conf = OmegaConf.load('source/config_interpolation.yaml')
>>> def show(x):
...     print(f"type: {type(x).__name__}, value: {repr(x)}")
>>> # Primitive interpolation types are inherited from the reference
>>> show(conf.client.server_port)
type: int, value: 80
>>> # String interpolations concatenate fragments into a string
>>> show(conf.client.url)
type: str, value: 'http://localhost:80/'
>>> # Relative interpolation example
>>> show(conf.client.description)
type: str, value: 'Client of http://localhost:80/'
```

Nested interpolation

Interpolations may be nested, enabling more advanced behavior like dynamically selecting a sub-config:

```
>>> cfg = OmegaConf.create(
...     {
...         "plans": {
...             "A": "plan A",
...             "B": "plan B",
...         },
```

(continues on next page)

(continued from previous page)

```
...         "selected_plan": "A",
...         "plan": "${plans[${selected_plan}]}",
...     }
... )
>>> cfg.plan # default plan
'plan A'
>>> cfg.selected_plan = "B"
>>> cfg.plan # new plan
'plan B'
```

Interpolated nodes can be any node in the config, not just leaf nodes:

```
>>> cfg = OmegaConf.create(
...     {
...         "john": {"height": 180, "weight": 75},
...         "player": "${john}",
...     }
... )
>>> (cfg.player.height, cfg.player.weight)
(180, 75)
```

Resolvers

Add new interpolation types by registering resolvers using `OmegaConf.register_new_resolver()`. Such resolvers are called when the config node is accessed. The minimal example below shows its most basic usage, see *Resolvers* for more details.

```
>>> OmegaConf.register_new_resolver(
...     "add", lambda *numbers: sum(numbers)
... )
>>> c = OmegaConf.create({'total': '${add:1,2,3}'})
>>> c.total
6
```

Built-in resolvers

OmegaConf comes with a set of built-in custom resolvers:

- *oc.create*: Dynamically generating config nodes
- *oc.decode*: Parsing an input string using interpolation grammar
- *oc.deprecated*: Deprecate a key in your config
- *oc.env*: Accessing environment variables
- *oc.select*: Selecting an interpolation key, similar to interpolation but more flexible
- *oc.dict.{keys,value}*: Viewing the keys or the values of a dictionary as a list

1.1.6 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.

OmegaConf.merge()

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
```

OmegaConf.unsafe_merge()

OmegaConf offers a second faster function to merge config objects:

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

Unlike OmegaConf.merge(), unsafe_merge() is destroying the input configs and they should no longer be used after this call. The upside is that it's substantially faster.

1.1.7 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.

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
```

Struct flag

By default, OmegaConf dictionaries allow write access to unknown fields. If a field does not exist, writing it will create the field, and attempting to access the field before creation will raise an exception (either `ConfigKeyError` or `ConfigAttributeError`, depending on the mode of access). It's sometime useful to change this behavior. Using `OmegaConf.set_struct`, it is possible to prevent the creation of fields that do not exist:

```
>>> 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' is not in struct
    full_key: a.cc
    reference_type=Any
    object_type=dict
```

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

```
>>> from omegaconf import open_dict
>>> 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
```

1.1.8 Utility functions

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

Using `throw_on_missing`

You can control how missing values are handled by `OmegaConf.to_container()` using the `throw_on_missing` keyword argument.

```
>>> conf = OmegaConf.create({"foo": "bar", "missing": "???"})
>>> has_missing = OmegaConf.to_container(conf, throw_on_missing=False)
>>> show(has_missing)
type: dict, value: {'foo': 'bar', 'missing': '???'}
>>> OmegaConf.to_container(conf, throw_on_missing=True)
Traceback (most recent call last):
...
omegaconf.errors.MissingMandatoryValue: Missing mandatory value: missing
    full_key: missing
    object_type=dict
```

By default, `throw_on_missing=False`. Setting `throw_on_missing=True` can be useful if you want your program to fail fast when there are missing values in the config.

Using `structured_config_mode`

You can customize the treatment of `OmegaConf.to_container()` for Structured Config nodes using the `structured_config_mode` option. The default, `structured_config_mode=SCMode.DICT`, converts Structured Config nodes to plain dict.

Using `structured_config_mode=SCMode.DICT_CONFIG` causes such nodes to remain as `DictConfig`, allowing attribute style access on the resulting node.

Using `structured_config_mode=SCMode.INSTANTIATE`, Structured Config nodes are converted to instances of the backing dataclass or attrs class. Note that when `structured_config_mode=SCMode.INSTANTIATE`, interpolations nested within a structured config node will be resolved, even if `OmegaConf.to_container` is called with the the keyword argument `resolve=False`, so that interpolations are resolved before being used to instantiate dataclass/attr class instances. Interpolations within non-structured parent nodes will be resolved (or not) as usual, according to the `resolve` keyword arg.

```
>>> from omegaconf import SCMode
>>> conf = OmegaConf.create({"structured_config": MyConfig})
>>> container = OmegaConf.to_container(conf,
...     structured_config_mode=SCMode.DICT_CONFIG)
>>> show(container)
type: dict, value: {'structured_config': {'port': 80, 'host': 'localhost'}}
>>> show(container["structured_config"])
type: DictConfig, value: {'port': 80, 'host': 'localhost'}
```

OmegaConf.to_object

The `OmegaConf.to_object` method recursively converts `DictConfig` and `ListConfig` objects into plain Python dicts and lists, with the exception that Structured Config objects are converted into instances of the backing dataclass or attr class. Interpolations in the config are always resolved by `OmegaConf.to_object`.

```
>>> container = OmegaConf.to_object(conf)
>>> show(container)
type: dict, value: {'structured_config': MyConfig(port=80, host='localhost')}
>>> show(container["structured_config"])
type: MyConfig, value: MyConfig(port=80, host='localhost')
```

Note that here, `container["structured_config"]` is actually an instance of `MyConfig`, whereas in the previous examples we had a `dict` or a `DictConfig` object that was duck-typed to look like an instance of `MyConfig`.

The call `OmegaConf.to_object(conf)` is equivalent to `OmegaConf.to_container(conf, resolve=True, throw_on_missing=True, structured_config_mode=SCMode.INSTANTIATE)`.

OmegaConf.resolve

```
def resolve(cfg: Container) -> None:
    """
    Resolves all interpolations in the given config object in-place.
    :param cfg: An OmegaConf container (DictConfig, ListConfig)
                Raises a ValueError if the input object is not an OmegaConf container.
    """
```

Normally interpolations are resolved lazily, at access time. This function eagerly resolves all interpolations in the given config object in-place. Example:

```
>>> cfg = OmegaConf.create({"a": 10, "b": "${a}"})
>>> show(cfg)
type: DictConfig, value: {'a': 10, 'b': '${a}'}
>>> assert cfg.a == cfg.b == 10 # lazily resolving interpolation
>>> OmegaConf.resolve(cfg)
>>> show(cfg)
type: DictConfig, value: {'a': 10, 'b': 10}
```

OmegaConf.select

`OmegaConf.select()` allows you to select a config node or value, using either a dot-notation or brackets to denote sub-keys.

```
>>> cfg = OmegaConf.create({
...     "foo" : {
...         "missing" : "???",
...         "bar": {
...             "zonk" : 10,
...         }
...     }
... })
>>> assert OmegaConf.select(cfg, "foo") == {
...     "missing" : "???",
...     "bar":  {
...         "zonk" : 10,
...     }
... }
>>> assert OmegaConf.select(cfg, "foo.bar") == {
...     "zonk" : 10,
... }
>>> assert OmegaConf.select(cfg, "foo.bar.zonk") == 10    # dots
>>> assert OmegaConf.select(cfg, "foo[bar][zonk]") == 10  # brackets
>>> assert OmegaConf.select(cfg, "no_such_key", default=99) == 99
>>> assert OmegaConf.select(cfg, "foo.missing") is None
>>> assert OmegaConf.select(cfg, "foo.missing", default=99) == 99
>>> OmegaConf.select(cfg,
...     "foo.missing",
...     throw_on_missing=True
... )
Traceback (most recent call last):
...
omegaconf.errors.MissingMandatoryValue: missing node selected
    full_key: foo.missing
```

OmegaConf.update

`OmegaConf.update()` allows you to update values in your config using either a dot-notation or brackets to denote sub-keys.

The merge flag controls the behavior if the input is a `dict` or a `list`. If `merge=True` true (the default), dicts and lists are merged instead of being assigned. The `force_add` flag ensures that the path is created even if it will result in insertion of new values into struct nodes.

```
>>> cfg = OmegaConf.create({"foo" : {"bar": 10}})
>>> OmegaConf.update(cfg, "foo.bar", 20)
>>> assert cfg.foo.bar == 20
>>> # Set dictionary value (using dot notation)
>>> OmegaConf.update(cfg, "foo.bar", {"zonk" : 30}, merge=False)
>>> assert cfg.foo.bar == {"zonk" : 30}
>>> # Merge dictionary value (using bracket notation)
>>> # note that merge is True by default, so you don't really need it here.
>>> OmegaConf.update(cfg, "foo[bar]", {"oompa" : 40}, merge=True)
>>> assert cfg.foo.bar == {"zonk" : 30, "oompa" : 40}
>>> # force_add ignores nodes in struct mode or Structured Configs nodes
>>> # and updates anyway, inserting keys as needed.
>>> OmegaConf.set_struct(cfg, True)
>>> OmegaConf.update(cfg, "a.b.c.d", 10, force_add=True)
>>> assert cfg.a.b.c.d == 10
```

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

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

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")
```

OmegaConf.is_interpolation

Tests if a value is an interpolation.

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

OmegaConf.{is_config, is_dict, is_list}

`OmegaConf.is_config` tests whether an object is an OmegaConf object (e.g. `DictConfig` or `ListConfig`). `OmegaConf.is_dict(cfg)` is equivalent to `isinstance(cfg, DictConfig)`, and `OmegaConf.is_list(cfg)` is equivalent to `isinstance(cfg, ListConfig)`.

```
>>> # 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)
```

OmegaConf.missing_keys

`OmegaConf.missing_keys(cfg)` returns a set of missing keys present in the input `cfg`. Each missing key is represented as a `str`, using a dotlist style. This utility function can be used after creating a config object, after merging sources and so on, to check for missing mandatory fields and aid in creating a proper error message.

```
>>> missings = OmegaConf.missing_keys({
...      "foo": {"bar": "???"},
...      "missing": "???",
...       "list": ["a", None, "???"]
... })
>>> assert missings == {'list[2]', 'foo.bar', 'missing'}
```

The function raises a *ValueError* on input not representing a config.

1.1.9 Debugger integration

OmegaConf is packaged with a PyDev.Debugger extension which enables better debugging experience in PyCharm, VSCode and other PyDev.Debugger powered IDEs.

The debugger extension enables OmegaConf-aware object inspection:

- providing information about interpolations.
- properly handling missing values ("???").

The plugin comes in two flavors:

- USER: Default behavior, useful when debugging your OmegaConf objects.
- DEV: Useful when debugging OmegaConf itself, shows the exact data model of OmegaConf.

The default flavor is `USER`. You can select which flavor to use using the environment variable `OC_PYDEVD_RESOLVER`, Which takes the possible values `USER`, `DEV` and `DISABLE`.

1.2 Resolvers

1.2.1 Custom resolvers

You can add additional interpolation types by registering custom resolvers with `OmegaConf.register_new_resolver()`:

```
def register_new_resolver(
    name: str,
    resolver: Resolver,
    *,
    replace: bool = False,
    use_cache: bool = False,
) -> None
```

Attempting to register the same resolver twice will raise a `ValueError` unless using `replace=True`.

The example below creates a resolver that adds `10` to the given value.

```
>>> OmegaConf.register_new_resolver("plus_10", lambda x: 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. In a variadic argument list, whitespace is stripped from the ends of each value ("foo,bar" gives the same result as "foo, bar "). You can use literal commas and spaces anywhere by escaping (`\,` and `\ `), or simply use quotes to bypass character limitations in strings.

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

You can take advantage of nested interpolations to perform custom operations over variables:

```
>>> OmegaConf.register_new_resolver("sum", lambda x, y: x + y)
>>> c = OmegaConf.create({"a": 1,
...                       "b": 2,
...                       "a_plus_b": "${sum:${a},${b}}"})
>>> c.a_plus_b
3
```

More advanced resolver naming features include the ability to prefix a resolver name with a namespace, and to use interpolations in the name itself. The following example demonstrates both:

```
>>> OmegaConf.register_new_resolver("mylib.plus1", lambda x: x + 1)
>>> c = OmegaConf.create(
...     {
...         "func": "plus1",
...         "x": "${mylib.${func}:3}",
...     }
... )
>>> c.x
4
```

By default a custom resolver is called on every access, but it is possible to cache its output by registering it with `use_cache=True`. This may be useful either for performance reasons or to ensure the same value is always returned. Note that the cache is based on the string literals representing the resolver's inputs, not on the inputs themselves:

```
>>> import random
>>> random.seed(1234)
>>> OmegaConf.register_new_resolver(
...     "cached", random.randint, use_cache=True
... )
>>> OmegaConf.register_new_resolver("uncached", random.randint)
>>> c = OmegaConf.create(
...     {
...         "uncached": "${uncached:0,10000}",
...         "cached_1": "${cached:0,10000}",
...         "cached_2": "${cached:0, 10000}",
...         "cached_3": "${cached:0,${uncached}}",
...     }
... )
>>> # not the same since the cache is disabled by default
>>> assert c.uncached != c.uncached
>>> # same value on repeated access thanks to the cache
>>> assert c.cached_1 == c.cached_1 == 122
>>> # same input as `cached_1` => same value
>>> assert c.cached_2 == c.cached_1 == 122
>>> # same string literal "${uncached}" => same value
>>> assert c.cached_3 == c.cached_3 == 1192
```

Custom interpolations can also receive the following special parameters:

- `_parent_`: The parent node of an interpolation.
- `_root_`: The config root.

This can be achieved by adding the special parameters to the resolver signature. Note that special parameters must be defined as named keywords (after the `*`).

In the example below, we use `_parent_` to implement a sum function that defaults to `0` if the node does not exist. This is in contrast to the sum we defined earlier where accessing an invalid key, e.g. `"a_plus_z": ${sum:${a}, ${z}}`, would result in an error.

```
>>> def sum2(a, b, *, _parent_):
...     return _parent_.get(a, 0) + _parent_.get(b, 0)
>>> OmegaConf.register_new_resolver("sum2", sum2)
>>> cfg = OmegaConf.create(
...     {
...         "node": {
...             "a": 1,
...             "b": 2,
...             "a_plus_b": "${sum2:a,b}",
...             "a_plus_z": "${sum2:a,z}",
...         },
...     }
... )
>>> cfg.node.a_plus_b
3
>>> cfg.node.a_plus_z
1
```

1.2.2 Built-in resolvers

oc.env

Access to environment variables is supported using `oc.env`:

Input YAML file:

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

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

You can specify a default value to use in case the environment variable is not set. In such a case, the default value is converted to a string using `str(default)`, unless it is `null` (representing Python `None`) - in which case `None` is returned.

The following example falls back to default passwords when `DB_PASSWORD` is not defined:

```
>>> cfg = OmegaConf.create(
...     {
...         "database": {
...             "password1": "${oc.env:DB_PASSWORD,password}",
...             "password2": "${oc.env:DB_PASSWORD,12345}",
...             "password3": "${oc.env:DB_PASSWORD,null}",
...         },
...     }
... )
>>> # default is already a string
>>> show(cfg.database.password1)
type: str, value: 'password'
>>> # default is converted to a string automatically
>>> show(cfg.database.password2)
type: str, value: '12345'
>>> # unless it's None
>>> show(cfg.database.password3)
type: NoneType, value: None
```

oc.create

`oc.create` may be used for dynamic generation of config nodes (typically from Python `dict` / `list` objects or YAML strings, similar to *OmegaConf.create*).

```
>>> OmegaConf.register_new_resolver("make_dict", lambda: {"a": 10})
>>> cfg = OmegaConf.create(
...     {
...         "plain_dict": "${make_dict:}",
...         "dict_config": "${oc.create:${make_dict:}}",
```

(continues on next page)

(continued from previous page)

```
...         "dict_config_env": "${oc.create:${oc.env:YAML_ENV}}",
...     }
... )
>>> os.environ["YAML_ENV"] = "A: 10\nb: 20\nC: ${.A}"
>>> show(cfg.plain_dict)  # `make_dict` returns a Python dict
type: dict, value: {'a': 10}
>>> show(cfg.dict_config)  # `oc.create` converts it to DictConfig
type: DictConfig, value: {'a': 10}
>>> show(cfg.dict_config_env)  # YAML string to DictConfig
type: DictConfig, value: {'A': 10, 'b': 20, 'C': '${.A}'}
>>> cfg.dict_config_env.C  # interpolations work in a DictConfig
10
```

oc.deprecated

`oc.deprecated` enables you to deprecate a config node. It takes two parameters:

- `key`: An interpolation key representing the new key you are migrating to. This parameter is required.
- `message`: A message to use as the warning when the config node is being accessed. The default message is `'$OLD_KEY' is deprecated. Change your code and config to use '$NEW_KEY'`.

```
>>> conf = OmegaConf.create({
...   "rusty_key": "${oc.deprecated:shiny_key}",
...   "custom_msg": "${oc.deprecated:shiny_key, 'Use $NEW_KEY'}",
...   "shiny_key": 10
... })
>>> # Accessing rusty_key will issue a deprecation warning
>>> # and return the new value automatically
>>> warning = "'rusty_key' is deprecated. Change your" \
...           " code and config to use 'shiny_key'"
>>> with pytest.warns(UserWarning, match=warning):
...   assert conf.rusty_key == 10
>>> with pytest.warns(UserWarning, match="Use shiny_key"):
...   assert conf.custom_msg == 10
```

oc.decode

With `oc.decode`, strings can be converted into their corresponding data types using the *"element" parser rule of the OmegaConf grammar*. This grammar recognizes typical data types like `bool`, `int`, `float`, `bytes`, `dict` and `list`, e.g. `"true"`, `"1"`, `"1e-3"`, `b"123"`, `"{a:  b}"`, `"[a, b, c]"`.

Note that:

- In most cases input strings provided to `oc.decode` should be quoted, since only a subset of the characters is allowed in unquoted strings.
- `None` (written as `null` in the grammar) is the only valid non-string input to `oc.decode` (returning `None` in that case).

This resolver can be useful for instance to parse environment variables:

```
>>> cfg = OmegaConf.create(
...     {
...         "database": {
...             "port": '${oc.decode:${oc.env:DB_PORT}}',
...             "nodes": '${oc.decode:${oc.env:DB_NODES}}',
...             "timeout": '${oc.decode:${oc.env:DB_TIMEOUT,null}}',
...         }
...     }
... )
>>> os.environ["DB_PORT"] = "3308"
>>> show(cfg.database.port)  # converted to int
type: int, value: 3308
>>> os.environ["DB_NODES"] = "[host1, host2, host3]"
>>> show(cfg.database.nodes)  # converted to a Python list
type: list, value: ['host1', 'host2', 'host3']
>>> show(cfg.database.timeout)  # keeping `None` as is
type: NoneType, value: None
>>> os.environ["DB_TIMEOUT"] = "${.port}"
>>> show(cfg.database.timeout)  # resolving interpolation
type: int, value: 3308
```

oc.select

`oc.select` enables selection similar to that performed with node interpolation, but is a bit more flexible. Using `oc.select`, you can provide a default value to use in case the primary interpolation key is not found. The following example uses "/tmp" as the default value for the node output:

```
>>> cfg = OmegaConf.create({
...  "a": "Saving output to ${oc.select:output,/tmp}"
... })
>>> print(cfg.a)
Saving output to /tmp
>>> cfg.output = "/etc/config"
>>> print(cfg.a)
Saving output to /etc/config
```

`oc.select` can also be used to select keys that are otherwise illegal interpolation keys. The following example has a key with a colon. Such a key looks like a custom resolver and therefore cannot be accessed using a regular interpolation:

```
>>> cfg = OmegaConf.create({
...     # yes, there is a : in this key
...     "a:b": 10,
...     "bad": "${a:b}",
...     "good": "${oc.select:'a:b'}",
... })
>>> print(cfg.bad)
Traceback (most recent call last):
...
UnsupportedInterpolationType: Unsupported interpolation type a
>>> print(cfg.good)
10
```

Another scenario where `oc.select` can be useful is if you want to select a missing value.

```
>>> cfg = OmegaConf.create({
...         "missing": "???",
...         "interpolation": "${missing}",
...         "select": "${oc.select:missing}",
...         "with_default": "${oc.select:missing,default value}",
...     }
... )
...
>>> print(cfg.interpolation)
Traceback (most recent call last):
...
InterpolationToMissingValueError: MissingMandatoryValue while ...
>>> print(cfg.select)
None
>>> print(cfg.with_default)
default value
```

oc.dict.{keys,value}

Some config options that are stored as a `DictConfig` may sometimes be easier to manipulate as lists, when we care only about the keys or the associated values.

The resolvers `oc.dict.keys` and `oc.dict.values` simplify such operations by offering an alternative view of a `DictConfig`'s keys or values as a list, with behavior analogous to the `dict.keys` and `dict.values` methods in plain Python dictionaries. These resolvers take as input a string that is the path to another config node (using the same syntax as interpolations), and they return a `ListConfig` that contains keys or values of the node whose path was given.

```
>>> cfg = OmegaConf.create(
...     {
...         "workers": {
...             "node3": "10.0.0.2",
...             "node7": "10.0.0.9",
...         },
...         "nodes": "${oc.dict.keys: workers}",
...         "ips": "${oc.dict.values: workers}",
...     }
... )
>>> # Keys are copied from the DictConfig:
>>> show(cfg.nodes)
type: ListConfig, value: ['node3', 'node7']
>>> # Values are dynamically fetched through interpolations:
>>> show(cfg.ips)
type: ListConfig, value: ['${workers.node3}', '${workers.node7}']
>>> assert cfg.ips == ["10.0.0.2", "10.0.0.9"]
```

1.2.3 Clearing/removing resolvers

clear_resolvers

Use `OmegaConf.clear_resolvers()` to remove all resolvers except the built-in resolvers (like `oc.env` etc).

```
def clear_resolvers() -> None
```

In the following example, first we register a new custom resolver `str.lower`, and then clear all custom resolvers.

```
>>> # register a new resolver: str.lower
>>> OmegaConf.register_new_resolver(
...     name='str.lower',
...     resolver=lambda x: str(x).lower(),
... )
>>> # check if resolver exists (after adding, before removal)
>>> OmegaConf.has_resolver("str.lower")
True
>>> # clear all custom-resolvers
>>> OmegaConf.clear_resolvers()
>>> # check if resolver exists (after removal)
>>> OmegaConf.has_resolver("str.lower")
False
>>> # default resolvers are not affected
>>> OmegaConf.has_resolver("oc.env")
True
```

clear_resolver

Use `OmegaConf.clear_resolver()` to remove a single resolver (including built-in resolvers).

```
def clear_resolver(name: str) -> bool
```

`OmegaConf.clear_resolver()` returns True if the resolver was found and removed, and False otherwise.

Here is an example.

```
>>> OmegaConf.has_resolver("oc.env")
True
>>> # This will remove the default resolver: oc.env
>>> OmegaConf.clear_resolver("oc.env")
True
>>> OmegaConf.has_resolver("oc.env")
False
```

1.3 Structured Configs

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 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. At runtine, OmegaConf will validate modifications to the created config object against the schema specified in the input class.

Currently, type hints supported in OmegaConf's structured configs include:

- primitive types (`int`, `float`, `bool`, `str`, `bytes`, `Path`) and enum types (user-defined subclasses of `enum.Enum`). See the *Simple types* section below.
- unions of primitive/enum types, e.g. `Union[float, bool, MyEnum]`. See *Unions* below.
- structured config fields (i.e. MyConfig.x can have type hint MySubConfig). See the *Nesting structured configs* section below.
- dict and list types: `typing.Dict[K, V]` or `typing.List[V]`, where K is primitive or enum, and where V is any of the above (including nested dicts or lists, e.g. `Dict[str, List[int]]`). See the *Lists* and *Dictionaries* sections below.
- optional types (any of the above can be wrapped in a `typing.Optional[...]` annotation). See *Other special features* below.

1.3.1 Simple types

Simple types include

- `int`: numeric integers
- `float`: numeric floating point values
- `bool`: boolean values (True, False, On, Off etc)
- `str`: any string
- `bytes`: an immutable sequence of numbers in [0, 255]
- `pathlib.Path`: filesystem paths as represented by python's standard library `pathlib`
- `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"
...     data: bytes = b"bin_data"
...     path: pathlib.Path = pathlib.Path("hello.txt")
```

You can create a config based on the SimpleTypes class itself or an instance 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
data: !!binary |
  YmluX2RhdGE=
path: !!python/object/apply:pathlib.PosixPath
- hello.txt
```

The resulting object is a regular OmegaConf `DictConfig`, except that 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
```

1.3.2 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'
```

1.3.3 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
```

(continues on next page)

(continued from previous page)

```
>>> # This works too
>>> conf.height = "Height.TALL"
>>> assert conf.height == Height.TALL

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

1.3.4 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 = field(default_factory=User)
...
...     # You can also specify different defaults for nested classes
...     manager: User = field(default_factory=lambda: User(name="manager", height=Height.
↪TALL))

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

1.3.5 Lists

Structured Config fields annotated with `typing.List` or `typing.Tuple` can hold any type supported by OmegaConf (`int`, `float`. `bool`, `str`, `bytes`, `pathlib.Path`, `Enum` or Structured configs).

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

>>> @dataclass
... class ListsExample:
...     # Typed list can hold Any, int, float, bool, str,
...     # bytes, pathlib.Path 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. In the example below, the OmegaConf object `conf` (which is actually an instance of `DictConfig`) is duck-typed as `ListExample`.

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

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

1.3.6 Dictionaries

Dictionaries are supported via annotation of structured config fields with `typing.Dict`. Keys must be typed as one of `str`, `int`, `Enum`, `float`, `bytes`, or `bool`. Values can be any of the types supported by OmegaConf (`Any`, `int`, `float`, `bool`, `bytes`, `pathlib.Path`, `str` and `Enum` as well as arbitrary Structured configs)

```
>>> from dataclasses import dataclass, field
>>> from typing import Dict
>>> @dataclass
... class DictExample:
...     ints: Dict[str, int] = field(default_factory=lambda: {"a": 10, "b": 20, "c": 30})
...     bools: Dict[str, bool] = field(default_factory=lambda: {"Uno": True, "Zoro":␣
```

(continues on next page)

(continued from previous page)

```
↪False})
...     users: Dict[str, User] = field(default_factory=lambda: {"omry": User(name="omry
↪")})
```

Like with Lists, the types of values contained in Dicts are verified at runtime.

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

>>> # Okay, correct type is assigned
>>> conf.ints["d"] = 10
>>> conf.bools["Dos"] = True
>>> conf.users["James"] = User(name="Bond")

>>> # Not okay, 10 cannot be assigned to a User
>>> with raises(ValidationError):
...     conf.users["Joe"] = 10
```

1.3.7 Nested dict and list annotations

Dict and List annotations can be nested flexibly:

```
>>> @dataclass
... class NestedContainers:
...     dict_of_dict: Dict[str, Dict[str, int]]
...     list_of_list: List[List[int]] = field(default_factory=lambda: [[123]])
...     dict_of_list: Dict[str, List[int]] = MISSING
...     list_of_dict: List[Dict[str, int]] = MISSING
...
...
>>> cfg = OmegaConf.structured(NestedContainers(dict_of_dict={"foo": {"bar": 123}}))
>>> print(OmegaConf.to_yaml(cfg))
dict_of_dict:
  foo:
    bar: 123
list_of_list:
- - 123
dict_of_list: ???
list_of_dict: ???

>>> with raises(ValidationError):
...     cfg.list_of_dict = [["whoops"]]  # not a list of dicts
```

1.3.8 Unions

You can use typing.Union to annotate unions of *Simple types*.

```
>>> from typing import Union
>>>
>>> @dataclass
... class HasUnion:
...     u: Union[float, bool] = 10.1
...
>>> cfg = OmegaConf.structured(HasUnion)
>>> assert cfg.u == 10.1
>>> cfg.u = True  # ok
>>> cfg.u = b"binary"  # bytes not compatible with union
Traceback (most recent call last):
...
omegaconf.errors.ValidationError: Cannot assign 'b'binary'' of type 'bytes' to␣
↪Union[float, bool]
    full_key: u
    object_type=HasUnion
>>> OmegaConf.structured(HasUnion("abc"))  # str not compatible
Traceback (most recent call last):
...
omegaconf.errors.ValidationError: Cannot assign 'abc' of type 'str' to Union[float, bool]
    full_key: u
    object_type=None
```

If any argument of a `Union` type hint is `Optional`, the *whole* union is considered optional. For example, OmegaConf treats all four of the following type hints as equivalent:

- `Optional[Union[int, str]]`
- `Union[Optional[int], str]`
- `Union[int, str, None]`
- `Union[int, str, type(None)]`

Ordinarily, assignment to a structured config field results in coercion of the assigned value to the field's type. For example, assigning an integer to a field typed as `str` results in the integer being coverted to a string:

```
>>> @dataclass
... class HasStr:
...     s: str
...
>>> cfg = OmegaConf.structured(HasStr)
>>> cfg.s = 10.1
>>> assert cfg.s == "10.1"  # The assigned value has been converted to a string
```

When dealing with `Union` types, however, conversion is disabled so as to avoid ambiguity. Values assigned to a union-typed field of a structured config must precisely match one of the types in the `Union` annotation:

```
>>> @dataclass
... class StrOrInt:
...     u: Union[str, float]
...
```

(continues on next page)

(continued from previous page)

```
>>> cfg = OmegaConf.structured(StrOrInt)
>>> cfg.u = 10.1
>>> assert cfg.u == 10.1  # The assigned value remains a `float`.
>>> cfg.u = "10.1"
>>> assert cfg.u == "10.1"  # The assigned value remains a `str`.
>>> cfg.u = 123  # Conversion from `int` to `float` does not occur.
Traceback (most recent call last):
...
omegaconf.errors.ValidationError: Value '123' of type 'int' is incompatible with type␣
↪hint 'Union[str, float]'
    full_key: u
    object_type=StrOrInt
```

1.3.9 Other special features

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
...     optional_dict: Optional[Dict[str, int]] = None
...     list_optional: List[Optional[int]] = field(default_factory=lambda: [10, MISSING,␣
↪None])

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

Note for Python3.6 users: *pickling* structured configs with complex type annotations, such as dict-of-list or list-of-optional, is not supported.

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
```

Optional fields

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

>>> conf.optional_num = None
>>> assert conf.optional_num is None
>>> assert conf.list_optional[2] is None
```

Interpolations

Variable interpolation works normally with Structured configs, but static type checkers may object to you assigning a string to another type. To work around this, use the special functions `omegaconf.SI` and `omegaconf.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 = "${val}"
...     # This is equivalent to the above, but static type checkers
...     # will not complain
...     b: int = SI("${val}")
...     # This is 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
```

Interpolated values are validated, and converted when possible, to the annotated type when the interpolation is accessed, e.g:

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

>>> cfg = OmegaConf.structured(Interpolation)
>>> cfg.int_key  # fails due to type mismatch
Traceback (most recent call last):
  ...
omegaconf.errors.InterpolationValidationError: Value 'string' could not be converted to␣
↪Integer
    full_key: int_key
    object_type=Interpolation
```

(continues on next page)

(continued from previous page)

```
>>> cfg.str_key = "1234"  # string value
>>> assert cfg.int_key == 1234  # automatically convert str to int
```

Note however that this validation step is currently skipped for container node interpolations:

```
>>> @dataclass
... class NotValidated:
...     some_int: int = 0
...     some_dict: Dict[str, str] = II("some_int")

>>> cfg = OmegaConf.structured(NotValidated)
>>> assert cfg.some_dict == 0  # type mismatch, but no error
```

Frozen classes

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

```
>>> from dataclasses import dataclass, field
>>> from typing import List
>>> @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
```

The read-only flag is recursive:

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

1.3.10 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 = field(default_factory=Server)
...     log: Log = field(default_factory=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)
```

1.3.11 Using Metadata to Ignore Fields

OmegaConf inspects the metadata of dataclasss / attr class fields, ignoring any fields where `metadata["omegaconf_ignore"]` is `True`. When defining a dataclass or attr class, fields can be given metadata by passing the `metadata` keyword argument to the `dataclasses.field` function or the `attrs.field` function:

```
>>> @dataclass
... class HasIgnoreMetadata:
...     normal_field: int = 1
...     field_ignored: int = field(default=2, metadata={"omegaconf_ignore": True})
...     field_not_ignored: int = field(default=3, metadata={"omegaconf_ignore": False})
...
>>> cfg = OmegaConf.create(HasIgnoreMetadata)
>>> cfg
{'normal_field': 1, 'field_not_ignored': 3}
```

In the above example, `field_ignored` is ignored by OmegaConf.

1.4 The OmegaConf grammar

OmegaConf uses an ANTLR-based grammar to parse string expressions, where the lexer rules rules define the tokens used by the parser rules. Currently this grammar's main usage is in the parsing of *interpolations*, detailed below.

1.4.1 Interpolation strings

An interpolation string is any string containing the `${` character sequence (denoting the start of an interpolation), and is parsed using the `text` rule of the grammar:

```
text: (interpolation |
       ANY_STR | ESC | ESC_INTER | TOP_ESC | QUOTED_ESC)+;
```

Such a string can either be a single interpolation, or the concatenation of multiple fragments that can either be interpolations or regular strings (with a special handling of escaped characters, see *Escaping in interpolation strings* below). These are all examples of interpolation strings:

- `${foo.bar}`
- `https://${host}:${port}`
- `Hello ${name}`
- `${a}${oc.env:B}${c}`

1.4.2 Interpolation types

An `interpolation` as found in the rule above can either be a *Config node interpolation* (e.g., `${host}`) or a call to a *resolver* (e.g., `${oc.env:B}`). This is reflected in the following parser rules:

```
interpolation: interpolationNode | interpolationResolver;

interpolationNode:
    INTER_OPEN  // ${
    DOT*
    (configKey | BRACKET_OPEN configKey BRACKET_CLOSE)
    (DOT configKey | BRACKET_OPEN configKey BRACKET_CLOSE)*
    INTER_CLOSE;  // }

interpolationResolver:
    INTER_OPEN  // ${
```

(continues on next page)

(continued from previous page)

```
    resolverName COLON sequence?
    BRACE_CLOSE;  // }
```

The following are all valid examples of config node interpolations according to the `interpolationNode` rule (note in particular that it supports both dot and bracket notations to access child nodes):

- `${host}`
- `${.sibling}`
- `${..uncle.cousin}`
- `${some_list[3]}`
- `${some_deep_dict[key1][subkey2].subsubkey3}`

Here are also examples of resolver calls from the `interpolationResolver` rule:

- `${oc.env:B}`
- `${my_resolver_without_args:}`
- `${oc.select:  missing, default}`

Resolver arguments must be provided in a comma-separated list as per the following `sequence` parser rule:

```
sequence: (element (COMMA element?)*) | (COMMA element?)+;
```

Note that this rule currently supports empty arguments to preserve backward compatibility with OmegaConf 2.0, but this has been deprecated (see #572).

1.4.3 Element types

As seen in the `sequence` rule above, each resolver argument is parsed by an `element` rule, which currently supports four main types of arguments:

```
element:
    quotedValue
    | listContainer
    | dictContainer
    | primitive
;
```

A `quotedValue` is a quoted string that may contain basically anything in-between either double or single quotes (including interpolations, which will be resolved at evaluation time). For instance:

- `"Hello World!"`
- `'Hello ${name}!'`
- `"I ${can:  ${nest}, ${interpolations}, 'and quotes'}"`

The `quotedValue` parser rule is formally defined as:

```
quotedValue:
    (QUOTE_OPEN_SINGLE | QUOTE_OPEN_DOUBLE)
    text?
    MATCHING_QUOTE_CLOSE;
```

`listContainer` and `dictContainer` are respectively lists and dictionaries, using a familiar syntax:

- List examples: `[]`, `[1, 2, 3]`, `[${a}, ${oc.env:B}, c]`
- Dict examples: `{}`, `{a: 1, b: 2}`, `{a: ${a}, b: ${oc.env:B}}`

Their corresponding parser rules are:

```
listContainer: BRACKET_OPEN sequence? BRACKET_CLOSE;
dictContainer: BRACE_OPEN
               (dictKeyValuePair (COMMA dictKeyValuePair)*)?
               BRACE_CLOSE;
```

Regarding dictionaries, note that although values can be any `element`, keys are more restricted, and in particular quoted strings and interpolations are currently *not* allowed as dictionary keys (see the definition of `dictKey` in the grammar).

Finally, a `primitive` is everything else that is allowed, including in particular (see the full grammar for details):

- Unquoted strings (that support only a subset of characters, contrary to quoted ones): `foo`, `foo_bar`, `hello world 123`
- Integer numbers: `123`, `-5`, `+1_000_000`
- Floating point numbers (with special case-independent keywords for infinity and NaN): `0.1`, `1e-3`, `inf`, `-INF`, `nan`
- Other special keywords (also case-independent): `null`, `true`, `false`, `NULL`, `True`, `fAlSe`. **IMPORTANT**: `None` is *not* a special keyword and will be parsed as an unquoted string, you must use the `null` keyword instead (as in YAML).
- Interpolations (thus allowing for nested interpolations)

1.4.4 Escaped characters

Some characters need to be escaped, with varying escaping requirements depending on the situation. In general, however, you can use the following rule of thumb: *you only need to escape characters that otherwise have a special meaning in the current context*.

Escaping in interpolation strings

In order to define fields whose value is an interpolation-like string, interpolations can be escaped with `\${`. For instance:

```
>>> c = OmegaConf.create({"path": r"\${dir}", "dir": "tmp"})
>>> print(c.path)  # does *not* interpolate into the `dir` node
${dir}
```

If you actually want to follow a `\` with a resolved interpolation, this backslash needs to be escaped into `\\` to differentiate it from an escaped interpolation:

```
>>> c = OmegaConf.create({"path": r"C:\\${dir}", "dir": "tmp"})
>>> print(c.path)  # *does* interpolate into the `dir` node
C:\tmp
```

Note that we use Python raw strings here to make code more readable – otherwise all `\` characters would need be duplicated due to how Python handles escaping in regular string literals.

Finally, since the `\` character has no special meaning unless followed by `${`, it does *not* need to be escaped anywhere else:

```
>>> c = OmegaConf.create({"path": r"C:\foo_${dir}", "dir": "tmp"})
>>> print(c.path)  # a single \ is preserved...
C:\foo_tmp
>>> c = OmegaConf.create({"path": r"C:\\foo_${dir}", "dir": "tmp"})
>>> print(c.path)  # ... and multiple \\ too (no escape sequence)
C:\\foo_tmp
```

Escaping in unquoted strings

Unquoted strings can be found in a number of contexts, including dictionary keys/values, list elements, etc. As a result, the escape sequences are used for some special characters (`\\`, `\[`, `\]`, `\{`, `\}`, `\(`, `\)`, `\:`, `\=`, `\,`), for instance:

- `C\:\\$\{dir\}` resolves to the string `"C:\${dir}"`
- `\[a\, b\, c\]` resolves to the string `"[a, b, c]"`

In addition, leading and trailing whitespaces must be escaped in unquoted strings if we do not want them to be stripped (while inner whitespaces are always preserved):

```
>>> c = OmegaConf.create({"esc": r"${oc.decode: \ hi u \  }"})
>>> c.esc  # one leading whitespace and two trailing ones
' hi u  '
>>> # Tabs are handled similarly (NB: r-strings can't be used below)
>>> c = OmegaConf.create({"esc": "${oc.decode:\t\\\thi u\t\\\t\t}"})
>>> c.esc  # one leading tab and two trailing ones
'\thi u\t\t'
```

Escaping in unquoted strings can lead to hard-to-read expressions, and it is recommended to switch to quoted strings instead of relying heavily on the above escape sequences.

Escaping in quoted strings

As can be seen from the definition of the `quotedValue` parser rule above, quoted strings are just `text` fragments surrounded by quotes, and are thus very similar to *Interpolation strings*. As a result, the `\${` escape sequence can also be used to escape interpolations in quoted strings (as described in *Escaping in interpolation strings*):

- `"\${dir}"` resolves to the string `"${dir}"`
- `"C:\\${dir}"` resolves to the string `"C:\<value of dir>"`

However, one key difference with interpolation strings is that quotes of the same type as the enclosing quotes must be escaped, unless they are within a nested interpolation. For instance:

- `'\'Hi you\', I said'` resolves to the string `"'Hi you', I said"`
- `"'Hi ${concat:  'y', "o", u}', I said"` also resolves to the string `"'Hi you', I said"` if `concat` is a *custom resolver* concatenating its inputs. The main point to pay attention to in this example is that the quoted strings `'y'` and `"o"` found within the resolver interpolation `${concat:  ...}` do *not* need to be escaped, regardless of existing quotes outside of this interpolation.

1.5 How-To Guides

1.5.1 How to Perform Arithmetic Using `eval` as a Resolver

Sometimes it is necessary to perform arithmetic based on settings from your app's config. You can register Python's builtins.eval function as a *resolver* to perform simple computations.

First, register the `builtins.eval` function as a new resolver:

```
>>> from omegaconf import OmegaConf
>>> OmegaConf.register_new_resolver("eval", eval)
```

Now, define a config and perform some arithmetic using the `eval` resolver:

yaml

python

```
>>> yaml_data = """
... ten_squared: ${eval:'10 ** 2'}
... """
>>> cfg = OmegaConf.create(yaml_data)
>>> assert cfg.ten_squared == 100
```

```
>>> cfg = OmegaConf.create({
...     "ten_squared": "${eval:'10 ** 2'}",
... })
>>> assert cfg.ten_squared == 100
```

You can use *nested interpolation* to perform computation that involves other values from your config:

yaml

python

```
>>> yaml_data = """
... side_1: 5
... side_2: 6
... rectangle_area: ${eval:'${side_1} * ${side_2}'}
... """
>>> cfg = OmegaConf.create(yaml_data)
>>> assert cfg.rectangle_area == 30
```

```
>>> cfg = OmegaConf.create({
...     "side_1": 5,
...     "side_2": 6,
...     "rectangle_area": "${eval:'${side_1} * ${side_2}'}",
... })
>>> assert cfg.rectangle_area == 30
```

To pass string data to `eval`, you'll need to use a nested pair of quotes:

yaml

python

```
>>> yaml_data = """
... cow_say: moo
... three_cows: ${eval:'3 * "${cow_say}"'}
... """
>>> cfg = OmegaConf.create(yaml_data)
>>> assert cfg.three_cows == "moomoomoo"
```

```
>>> cfg = OmegaConf.create({
...   "cow_say": "moo",
...   "three_cows": """${eval:'3 * "${cow_say}"'}"""
... })
>>> assert cfg.three_cows == "moomoomoo"
```

The double quotes around `"${cow_say}"` guarantee that `eval` will interpret `"moo"` as a string instead of as a variable `moo`. See *Escaping in interpolation strings* for more information.

For more complicated logic, you should consider defining a specialized resolver to encapsulate the computation, rather than relying on the general capabilities of `eval`. Follow the examples from the *Custom resolvers* docs.

1.6 API Reference

1.6.1 The `OmegaConf` API

class omegaconf.**OmegaConf**

OmegaConf primary class

classmethod clear_resolver(*name: str*) → bool

Clear(remove) any resolver only if it exists.

Returns a bool: True if resolver is removed and False if not removed.

Parameters
name – Name of the resolver.

Returns
A bool (`True` if resolver is removed, `False` if not found before removing).

static clear_resolvers() → None

Clear(remove) all OmegaConf resolvers, then re-register OmegaConf's default resolvers.

static from_dotlist(*dotlist: List[str]*) → DictConfig

Creates config from the content sys.argv or from the specified args list of not None

Parameters
dotlist – A list of dotlist-style strings, e.g. `["foo.bar=1", "baz=qux"]`.

Returns
A `DictConfig` object created from the dotlist.

static masked_copy(*conf: DictConfig*, *keys: Union[str, List[str]]*) → DictConfig

Create a masked copy of of this config that contains a subset of the keys

Parameters

- **conf** – DictConfig object
- **keys** – keys to preserve in the copy

Returns
The masked `DictConfig` object.

static merge(**configs: Union[DictConfig, ListConfig, Dict[Union[str, bytes, int, Enum, float, bool], Any], List[Any], Tuple[Any, ...], Any]*) → Union[ListConfig, DictConfig]

Merge a list of previously created configs into a single one

Parameters
configs – Input configs

Returns
the merged config object.

static missing_keys(*cfg: Any*) → Set[str]

Returns a set of missing keys in a dotlist style.

Parameters
cfg – An `OmegaConf.Container`, or a convertible object via `OmegaConf.create` (dict, list, ...).

Returns
set of strings of the missing keys.

Raises
ValueError – On input not representing a config.

static register_new_resolver(*name: str*, *resolver: Callable[[...], Any]*, ***, *replace: bool = False*, *use_cache: bool = False*) → None

Register a resolver.

Parameters

- **name** – Name of the resolver.
- **resolver** – Callable whose arguments are provided in the interpolation, e.g., with ${foo:x,0,${y.z}} these arguments are respectively "x" (str), 0 (int) and the value of `y.z`.
- **replace** – If set to `False` (default), then a `ValueError` is raised if an existing resolver has already been registered with the same name. If set to `True`, then the new resolver replaces the previous one. NOTE: The cache on existing config objects is not affected, use `OmegaConf.clear_cache(cfg)` to clear it.
- **use_cache** – Whether the resolver's outputs should be cached. The cache is based only on the string literals representing the resolver arguments, e.g., ${foo:${bar}} will always return the same value regardless of the value of `bar` if the cache is enabled for `foo`.

static resolve(*cfg: Container*) → None

Resolves all interpolations in the given config object in-place.

Parameters

cfg – An OmegaConf container (DictConfig, ListConfig) Raises a ValueError if the input object is not an OmegaConf container.

static save(*config: Any*, *f: Union[str, Path, IO[Any]]*, *resolve: bool = False*) → None

Save as configuration object to a file

Parameters

- **config** – omegaconf.Config object (DictConfig or ListConfig).
- **f** – filename or file object
- **resolve** – True to save a resolved config (defaults to False)

static select(*cfg: Container*, *key: str*, ***, *default: Any = _DEFAULT_MARKER_*, *throw_on_resolution_failure: bool = True*, *throw_on_missing: bool = False*) → Any

Parameters

- **cfg** – Config node to select from
- **key** – Key to select
- **default** – Default value to return if key is not found
- **throw_on_resolution_failure** – Raise an exception if an interpolation resolution error occurs, otherwise return None
- **throw_on_missing** – Raise an exception if an attempt to select a missing key (with the value '???') is made, otherwise return None

Returns

selected value or None if not found.

static to_container(*cfg: Any*, ***, *resolve: bool = False*, *throw_on_missing: bool = False*, *enum_to_str: bool = False*, *structured_config_mode: SCMode = SCMode.DICT*) → Union[Dict[Union[str, bytes, int, Enum, float, bool], Any], List[Any], None, str, Any]

Resursively converts an OmegaConf config to a primitive container (dict or list).

Parameters

- **cfg** – the config to convert
- **resolve** – True to resolve all values
- **throw_on_missing** – When True, raise MissingMandatoryValue if any missing values are present. When False (the default), replace missing values with the string "???" in the output container.
- **enum_to_str** – True to convert Enum keys and values to strings
- **structured_config_mode** –

  Specify how Structured Configs (DictConfigs backed by a dataclass) are handled.

  - By default (`structured_config_mode=SCMode.DICT`) structured configs are converted to plain dicts.
  - If `structured_config_mode=SCMode.DICT_CONFIG`, structured config nodes will remain as DictConfig.

- If `structured_config_mode=SCMode.INSTANTIATE`, this function will instantiate structured configs (DictConfigs backed by a dataclass), by creating an instance of the underlying dataclass.

See also OmegaConf.to_object.

Returns
A dict or a list representing this config as a primitive container.

`static to_object`(*cfg: Any*) → Union[Dict[Union[str, bytes, int, Enum, float, bool], Any], List[Any], None, str, Any]

Resursively converts an OmegaConf config to a primitive container (dict or list). Any DictConfig objects backed by dataclasses or attrs classes are instantiated as instances of those backing classes.

This is an alias for OmegaConf.to_container(..., resolve=True, throw_on_missing=True, structured_config_mode=SCMode.INSTANTIATE)

Parameters
cfg – the config to convert

Returns
A dict or a list or dataclass representing this config.

`static to_yaml`(*cfg: Any*, *, *resolve: bool = False*, *sort_keys: bool = False*) → str

returns a yaml dump of this config object.

Parameters

- **cfg** – Config object, Structured Config type or instance
- **resolve** – if True, will return a string with the interpolations resolved, otherwise interpolations are preserved
- **sort_keys** – If True, will print dict keys in sorted order. default False.

Returns
A string containing the yaml representation.

`static unsafe_merge`(**configs: Union[DictConfig, ListConfig, Dict[Union[str, bytes, int, Enum, float, bool], Any], List[Any], Tuple[Any, ...], Any]*) → Union[ListConfig, DictConfig]

Merge a list of previously created configs into a single one This is much faster than OmegaConf.merge() as the input configs are not copied. However, the input configs must not be used after this operation as will become inconsistent.

Parameters
configs – Input configs

Returns
the merged config object.

`static update`(*cfg: Container*, *key: str*, *value: Optional[Any] = None*, *, *merge: bool = True*, *force_add: bool = False*) → None

Updates a dot separated key sequence to a value

Parameters

- **cfg** – input config to update
- **key** – key to update (can be a dot separated path)
- **value** – value to set, if value if a list or a dict it will be merged or set depending on merge_config_values

- **merge** – If value is a dict or a list, True (default) to merge into the destination, False to replace the destination.
- **force_add** – insert the entire path regardless of Struct flag or Structured Config nodes.

1.6.2 Utility functions importable from the omegaconf module

omegaconf.**II**() → Any

Equivalent to ${interpolation}

Parameters
interpolation –

Returns
input ${node} with type Any

omegaconf.**SI**() → Any

Use this for String interpolation, for example "http://${host}:${port}"

Parameters
interpolation – interpolation string

Returns
input interpolation with type **Any**

omegaconf.**flag_override**(*names: Union[List[str], str]*, *values: Optional[Union[List[Optional[bool]], bool]]*) → Generator[Node, None, None]

omegaconf.**open_dict**() → Generator[Container, None, None]

omegaconf.**read_write**() → Generator[Node, None, None]

omegaconf.**MISSING**

alias of ???

1.7 Indices and tables

- genindex
- modindex
- search

INDEX