Marsh

Release 0.2.6

Adrian Sahlman

Jan 02, 2023

CONTENTS

`marsh` is an (un)marshaling library for Python 3.8+

CHAPTER

ONE

GETTING STARTED

1.1 Entry Points

1.1.1 Create

Normally when creating an entry point for a script (or larger application) one uses `argparse.ArgumentParser` to accept arguments from the command line.

In *marsh* this can be done with a single decorator over a function. The decorator inspects the function arguments, including their types and any descriptions that can be linked to them from the docstring of the function.

Listing 1: Creating an entry point

```
# app.py
import marsh


@marsh.main
def run(
    a: int,
    b: float,
    c: dict[str, bool],
) -> None:
    """Example of an entry point.

    Arguments:
        a: An integer argument.
        b: A floating point argument.
        c: A dictionary with string keys and
            bool values. If this was python 3.8
            we would instead use typing.Dict[str, bool] as
            type hint as the builtin dict did not support
            type annotations.
    """
    print(a, type(a))
    print(b, type(b))
    print(c, type(c))


if __name__ == '__main__':
    run()
```

Note: Mypy will report an error on the line `run()` as no arguments were supplied. Set the default value of arguments to *marsh.MISSING* to retain the same behavior from marsh without mypy errors.

Note: Any arguments passed to `run()` become new default values. For the code above, if we call `run(b=0)` the `b` argument becomes optional on the command line as there is now a default value. This can also be accomplished by having a default value in the function signature.

1.1.2 Run

When the decorated function is called without arguments an `argparse.ArgumentParser` is initialized and console arguments are read in an attempt to populate the arguments to our entry point function.

Calling the file on the command line using python allows us to pass arguments to the application.

Listing 2: Running the application

```
$ python app.py a=1 b=5e-1 c.key1=true c.key2=false
1 <class 'int'>
0.5 <class 'float'>
{'key1': True, 'key2': False} <class 'dict'>
```

1.1.3 Argument Validation

When giving invalid values or when required arguments are missing an error message is printed and the application exits.

Listing 3: Using an incorrect value

```
$ python app.py a=1.5 b=0 c.some_key=true
failed to unmarshal config: int: could not convert: 1.5
    path: a
```

Listing 4: Missing a required value

```
$ python app.py a=1 c.some_key=true
failed to unmarshal config: MissingValueError
    path: b
```

1.1.4 Help

Using –help we can also get a help message for the arguments. Here the output was piped to `tail` to truncate the output into displaying only the arguments of our entry point.

Listing 5: Printing a help message

```
$ python app.py --help | tail
fields:
```

(continues on next page)

(continued from previous page)

```
  a: <int>              An integer argument.

  b: <float>            A floating point argument.

  c: {<str>: <bool>, ...}
                        A dictionary with string keys and bool values. If this
                        was python 3.8 we would instead use typing.Dict[str,
                        bool] as type hint as the builtin dict did not support
                        type annotations.
```

1.2 Marshal

Marshaling values simply means taking a python object and turning it into JSON-like data.

Listing 6: Marshaling an object

```python
# marshal.py
import dataclasses
import marsh


@dataclasses.dataclass
class Config:
    a: int
    b: float


config = Config(1, 5e-1)
print(marsh.marshal(config))
```

```
$ python marshal.py
{'a': 1, 'b': 0.5}
```

1.3 Unmarshal

Unmarshaling is the opposite of marshaling. A type is instantiated using JSON-like data.

Listing 7: Unmarshaling a type

```python
# unmarshal.py
import dataclasses
import typing
import marsh


class Range(typing.NamedTuple):
    start: typing.Optional[int]
    stop: int
```

(continues on next page)

(continued from previous page)

```
@dataclasses.dataclass
class Config:
    a: int
    b: float
    c: Range


config = marsh.unmarshal(
    Config,
    {
        'a': 1,
        'b': 1.5,
        'c': {
            'start': None,
            'stop': 5,
        },
    }
)
print(config)
```

```
$ python umarshal.py
Config(a=1, b=1.5, c=Range(start=None, stop=5))
```

CHAPTER

TWO

ENTRY POINT

Please see *Getting Started* for an introduction to entry points.

2.1 Customization

When wrapping a function in *marsh.main()* some functionality can be configured through keyword arguments to the decorator.

Descriptions for these keyword arguments can be found in the docstring of *marsh.main()*. One of them is showcased below.

2.1.1 Config type

If a config object is to be passed around in the application it might be preferable to use that as a single argument to the entry point.

Listing 1: Using a configuration class

```
# app.py
import dataclasses
import marsh


@dataclasses.dataclass
class Config:
    a: int
    b: float


@marsh.main
def main(
    config: Config
) -> None:
    print(config)


if __name__ == '__main__':
    main()
```

However, this requires us to use the keyword `'config'` before each argument on the command line. It also creates an unnecessary nesting of arguments in the help message.

Listing 2: Help message with nested fields

```
$ python app.py --help | tail -n 6
fields:
  config: Config(...)

    config.a: <int>

    config.b: <float>
```

Listing 3: Longer path for arguments

```
$ python app.py config.a=0 config.b=1.5
Config(a=0, b=1.5)
```

Specifying a type for the `config` argument to *marsh.main()* makes the entry point use the fields/arguments of that type instead of the function being wrapped.

Listing 4: Specifying the config type

```
# app.py
...

@marsh.main(config=Config)
def main(
    config: Config
) -> None:
    print(config)
```

Listing 5: Help message without nested fields

```
$ python app.py --help | tail -n 4
fields:
  a: <int>

  b: <float>
```

Listing 6: Shorter path for arguments

```
$ python app.py a=0 b=1.5
Config(a=0, b=1.5)
```

2.2 Command Line

2.2.1 Help

The basic help message that is included looks as follows.

Listing 7: Standard help message

```
usage: prog [-h [PATH]] [-c PATH] [-o [PATH]] [--config-out [PATH]]
            [overrides [overrides ...]]

positional arguments:
  overrides             Assign new values for fields in the configuration
                        using `=` or assign the content of a config file
                        using `@`. Adding `+` in front (`+=`, `+@`) combines
                        the current value of the specified field with its
                        new assigned value. Prepending a field with `~`
                        removes it.

optional arguments:
  -h [PATH], --help [PATH]
                        Show this message and exit. Optionally display
                        config documentation for a path in the config
                        structure
  -c PATH, --config-dir PATH
                        Root directory for config files. Config paths given
                        in override arguments will be evaluated relative
                        `--config-dir`. Defaults to the working directory of
                        the caller.
  -o [PATH], --output [PATH]
                        Change the current working directory when launching
                        the application (after loading the final
                        configuration). If no path is given when using this
                        argument it defaults to "jobs/%Y%m%dT%H%M%S.%fZ"
  --config-out [PATH]   Write the current input configuration, then proceed.
                        If no path is specified, the configuration is
                        written to stdout. Relative paths are affected by
                        the --output argument.
```

Note: Setting `setup_logging=True` in the *marsh.main()* decorator adds the keyword arguments `--logging-level` and `--logging-format`.

After this part of the documentation the arguments of the decorated function (or a config class if specified) are printed. By default, only 2 levels nested fields are documented in the help message. If the types of the entry point arguments

are rich with deeply nested subfields these might not be shown.

To allow for all type documentation to be viewed `--help` supports an optional argument which is a path to a nested part of the entry point argument types.

Listing 8: Nested types

```
# app.py
import dataclasses
import marsh


@dataclasses.dataclass
class A:
    """Example of a nested documentation class."""
    some_int: int
    some_str: str
    some_bool: bool


@dataclasses.dataclass
class B:
    a: A


@dataclasses.dataclass
class C:
    b: B


@marsh.main
def main(
    a: A,
    b: B,
    c: C,
) -> None:
    ...
```

Listing 9: Help message

```
$ python app.py --help | tail -n 4
fields:
  c: C(...)

    c.b: B(...)
```

Using the optional path argument we can display the documentation for the `A` class.

Listing 10: Help message

```
$ python app.py --help c.b.a
A

Example of a nested documentation class.
```

(continues on next page)

(continued from previous page)

```
  some_int: <int>

  some_str: <str>

  some_bool: <bool>
```

2.2.2 Overrides

An override sets or alters the values passed on to the unmarshaler before being passed to the entry point function.

All overrides are supplied as positional arguments.

Path

The path of an override is a set of dot-separated fields.

If the combination of input values produce

```
{'a': 0, 'b': {'c': 1}}
```

then the path `'a.b.c'` would point to the integer `1`.

For sequences such as lists and tuples the index is specified directly in the path:

```
{'a': [1, {'b': 2}]}
```

The path `'a.1'` would point to the dictionary `{'b': 2}` in the above value.

Note: An empty path is valid and points at the root of the values. This is the same as using a single dot `'.'`

Values

Values can be set, combined or removed.

Note: Some of the characters needed for specifying container objects (mappings or sequences) may be reserved by the shell. Use quotes or escape characters that are consumed by the shell.

See *marsh.parse.element()* for information on parsing rules for the values.

To set the value for a path directly on the command line use `path.to.field=value`.

To combine the value with any existing value on the same path, use `path.to.field+=value`. This will fail unless the previous value and new value both are mappings or sequences.

To remove an existing value, use `~path.to.field`.

Note: Containers such as mappings may be specified as a single value or by specifying the each nested value separately. `a_map={a:0,b:1}` is the same as `a_map.a=0 a_map.b=1`.

Configs

Config paths are absolute if they start with `/`. Otherwise they are relative to the current working directory (unless `--config-dir` has been specified, in which case the paths are relative to the specified directory).

Currently supported config types are files ending with `.json`, `.yml` and `.yaml` (can be extended to accept more types). When specifying a path to a config it is possible to omit the file extension in which cast marsh will attempt look for files with the same path and a supported extension.

Configs are specified in a similar way to values. To set the value of specific path to the content of a config, use `path.to.field@path/to/config`.

To combine the current value with the content of a config, use `path.to.field+@path/to/config`.

Note: The root path of input values is the empty string so if a config contains the entire configuration it would be set without specifying a path (`@path/to/config`).

Variable Interpolation

marsh uses the variable interpolation supplied by omegaconf. Please see its documentation for how to use or extend this functionality.

Meta Data

Any values under the path `_meta_.` are removed before the input values are being used for unmarshaling.

The meta field may be useful for storing constants and other values used for variable interpolation.

CHAPTER

THREE

NAMESPACES

Namespaces enable covariance in the type returned by unmarshaling. This means that a superclass can be specified as the type and one of its subclasses is constructed.

A namespace is a set of types, each associated with its own unique name. There are no other limitations to the inheritance hierarchy beyond that all types must be subtypes of the base class of the namespace.

Note: Namespaces are applied through the entire framework. In the examples below `marsh.unmarshal(...)()` is used but the namespace functionality is applied recursively and thus affects nested types (for example, the types of the attributes of a dataclass) as well.

A new namespace is created by supplying a name and a base class.

```
import marsh

class Model:

    def predict(
        self,
        input: List[float]
    ) -> float:
        raise NotImplementedError

namespace = marsh.namespaces.new('model', Model)
```

Types can be added to the namespace through the registration function.

```
@namespace.register(name='a')
class ModelA(Model):

    def __init__(
        self,
        layers: int
    ) -> None:
        ...

@namespace.register(name='b')
class ModelB(Model):

    def __init__(
        self,
```

(continues on next page)

(continued from previous page)

```
        clusters: int
    ) -> None:
        ...
```

Once added, the types can be selected during unmarshaling by specifying a `'name'` field in the input. The name field is stripped before the rest of the inputs are passed to the constructor of the type corresponding to the name.

Listing 1: Covariant return type

```
model = marsh.unmarshal(Model, dict(name='a', layers=3))
assert isinstance(model, ModelA)
```

If no name is supplied with the input when unmarshaling the return type is invariant relative to the input type.

Listing 2: Invariant return type

```
model = marsh.unmarshal(Model, {})
assert isinstance(model, Model)
```

It is not required that the type given to *marsh.unmarshal()* is the base class of the namespace. Other classes in the inheritance hierarchy may be provided (even if they themselves are not registered to the namespace).

```
class StatefulModel(Model):
"""Part of inheritance hierarchy but not registered
to the namespace."""

    def get_state(
        self
    ) -> bytes:
        raise NotImplementedError


@namespace.register(name='c')
class ModelC(StatefulModel):
    ...

@namespace.register(name='d')
class ModelD(StatefulModel):
    ...

# here we can use name='c' or name='d'. The other models
# do not inherit `StatefulModel` and specifying their names
# would result in an error.
model = marsh.unmarshal(StatefulModel, dict(name='c'))
assert isinstance(model, ModelC)
```

CHAPTER

FOUR

EXTENDING

It is possible to increase the number of types that are supported by the framework (or customize how certain types are marshaled/unmarshaled).

4.1 Schema

All (un)marshaling is supported through the *marsh.schema.core.Schema* class (there are two separate variants of this, one for marshaling and one for unmarshaling).

The common part of the schema is their matching function which always takes a single value and returns a boolean based on if the schema class matched the value or not. If the schema matched the value it will be initialized with the same value which is accessible through `self.value`.

4.2 Registration

A new implementation of a schema must be registered if it is to be available for use by *marsh*. This is done through the function *marsh.schema.register()* which acts as a decorator for a registered schema class.

4.2.1 Priority

Registered schemas are ordered by their priority which can be set during registration. The priority order affects which schemas are matched before others.

The base priority is an integer value. Higher values correspond to higher priority. There is also a relative priority where a schema may preceed or succeed one or more other schemas. The relative priority is considered before the base priority.

4.3 Marshal

Lets consider the steps for supporting marshaling of `complex` values. We need to implement and register a new schema. This schema should inherit the base class *marsh.schema.MarshalSchema*.

```
import marsh


@marsh.schema.register
```

(continues on next page)

(continued from previous page)

```
class ComplexMarshalSchema(marsh.schema.MarshalSchema):

    @classmethod
    def match(
        cls,
        value
    ) -> bool:
        # we match an instance of `complex`, not its type
        return isinstance(value, complex)

    def marshal(
        self
    ) -> dict:
        return {
            'real': self.value.real,
            'imag': self.value.imag,
        }
```

Marshaling for the `complex` type is now supported through the framwork.

4.4 Unmarshal

For unmarshaling we perform similar steps as with marshaling but with slight differences. For starters, we need to use a different base class; *marsh.schema.UnmarshalSchema*. We also need to consider returning a default value when the input is missing. For example, if a field in a dataclass is of type `complex` and has a default value of `complex(1, 2)` then our schema class would match that field and contain the type as well as the default value.

```
import marsh


@marsh.schema.register
class ComplexUnmarshalSchema(marsh.schema.UnmarshalSchema[complex]):

    @classmethod
    def match(
        cls,
        value
    ) -> bool:
        # We match the type of `complex`, not its instance
        return value == complex

    def unmarshal(
        self,
        element: marsh.element.ElementType
    ) -> complex:
        # we first need to check if the input value is missing
        if marsh.utils.is_missing(element):
            # if there is a default value we return it instead
            if self.has_default():
                return self.get_default()
```

(continues on next page)

(continued from previous page)

```
            # otherwise we raise an error since we did not get
            # a value to unmarshal.
            raise marsh.errors.MissingValueError
        if isinstance(element, float):
            return complex(element)
        if marsh.utils.is_mapping(element):
            return complex(**element)
        if marsh.utils.is_sequence(element):
            return complex(*element)
        raise marsh.errors.UnmarshalError(element)
```

In the above example we allow sequence and mapping inputs without checking their actual values. We could instead take advantage of *marsh*'s ability to unmarshal typing constructs as a way to validate our input.

```
from typing import (
    Tuple,
    TypedDict,
    Union,
)
import marsh


class Kwargs(TypedDict, total=False):
    real: float
    imag: float


Args = Union[Tuple[float], Tuple[float, float]]


InputType = Union[float, Args, Kwargs]


@marsh.schema.register
class ComplexUnmarshalSchema(marsh.schema.UnmarshalSchema[complex]):

    @classmethod
    def match(
        cls,
        value
    ) -> bool:
        return value == complex

    def unmarshal(
        self,
        element: marsh.element.ElementType
    ) -> complex:
        if marsh.utils.is_missing(element):
            if self.has_default():
                return self.get_default()
            raise marsh.errors.MissingValueError
        arg = marsh.unmarshal(InputType, element)
```

(continues on next page)

(continued from previous page)

```
        if isinstance(arg, float):
            return complex(arg)
        if marsh.utils.is_mapping(arg):
            return complex(**arg)
        else:
            return complex(*arg)
```

CHAPTER

FIVE

SUPPORTED TYPES

5.1 Marshal

Table 1: Marshal Types

Type	Description
`slice`	Marshals a `slice` into a mapping with the keys `"start"`, `"stop"` and `"step"`.
`int`, `float`, `bool`, `str`	Marshals any instance of a primitive class or subclass into its baseclass value.
`DictConfig`, `ListConfig`	Marshals an omegaconf `DictConfig` or `ListConfig`, converting all enums in the container to strings.
`None`	Simply marshals any `None` value by returning it.
`namedtuple()`, `NamedTuple`	Marshals namedtuples into a mapping where the keys correspond to the field names of the namedtuple. Supports both `namedtuple()` and `NamedTuple`.
`Logger`	Marshals a python logger into a mapping with keys `"name"` and `"level"`.
`Enum`	Marshals an enum instance into its name.
`datetime.datetime`, `datetime.time`, `datetime.date`	Marshals a datetime object into formatted string according to ISO. See `datetime.datetime.isoformat()`, .:func:*datetime.time.isoformat* and `datetime.date.isoformat()`.
`dataclass()`	Marshals a dataclass instance into a mapping with the field names as keys.
`complex`	Marshals a complex value into a dictionary with keys `"real"` and `"imag"` containing float values.
`bytes`	Marshals bytes into a URL-safe base64-encoded string.
`SequenceProtocol`	Unmarshals sequence values matched through ducktyping.Uses *`SequenceProtocol`* as reference for the ducktyped match. Marshaling outputs a JSON-style list.
`MappingProtocol`	Unmarshals mapping values matched through ducktyping.Uses *`MappingProtocol`* as reference for the ducktyped match. Marshaling outputs a JSON-style map.
`Iterable`	Fetches and marshals all items in an iterable.

5.2 Unmarshal

Table 2: Unmarshal Types

Type	Description
`Annotated`	Unwraps and delegates unmarshaling to the underlying type.
`Union`	The union type which in later versions of python may be specified using the `|` operator. The order of the types in the union matter as an attempt is made to unmarshal each type where the value of the first successful unmarshal is returned. `Optional` will always set `None` as the second type in the produced union.
`TypeVar`	Unwraps the `TypeVar`, using its bound or constraints when unmarshaling. If no bound or constraints are present the type can not be unmarshaled.
`TypedDict`	Unmarshals any subclass of `TypedDict` from a mapping input. If `total=False` then all keys are optional.
`slice`	Accepts input as strings in the python slicing format (i.e. "1:2", "::-1" e.t.c). Also accepts keys "start", "stop" and/or "step" or a sequence of up to 3 optional integer values. If a single integer is given, it is treated as "stop"
`set`	Unmarshals a sequence input into a set. Any duplicate values will be discarded.
`Protocol`	Attempts to unmarshal `runtime_checkable()` Protocols into a matching class. The available classes. that can be matched against are `float`, `int`, `bool`, `bytes`, `dict`, `list`, `str` and `complex`.
`int`, `float`, `bool`, `str`	Unmarshals any primitive class or its subclass. Special rules for `bool` is documented in *`primitive_to_bool()`*.
`None`	A missing value is valid input as well as the `None` value directly.
`namedtuple()`, `NamedTuple`	Unmarshals namedtuple types from a mapping or sequence input. Supports both `namedtuple()` and `NamedTuple`.
`tuple`	Unmarshals a sequence input into a fixed size tuple. The input sequence must contain the same number of elements as the tuple type specifies.
`Logger`	A python logging.Logger. If no arguments are given the root logger is used. Takes either a single string which is used as name for the logger or a mapping with optional keys "name" and "level". The level can be given as a string or an integer.
`Literal`	Matches the input against the literal value(s), then returns the matched value.
`Enum`	Input must match one of the enum values (the name or value).
`Sequence`	Unmarshals a sequence type from a sequence input. Should work for all sequence types. If the sequence type is one of the abstract sequence classes in *collections.abc* then a suitable replacement type is implicitly used instead.
`defaultdict`	Unmarshals an input mapping into a defaultdict. If no default value is given in the type, all values default to `None`. An attempt is made to initialize the default value with a zero-value (zero, empty sequence, empty mapping, `None` e.t.c). This is not always guaranteed to work if the default type requires specific arguments to be initialized. Use with caution.
`Mapping`	Unmarshals a mapping type from a mapping input. Should work for all mapping types. If the mapping type is one of the abstract mapping classes in *collections.abc* then a suitable replacement type is implicitly used instead.
`datetime.datetime`, `datetime.time`, `datetime.date`	For string inputs the format must be compatible with `dateparser.parse()`. The date order is D M Y, not M D Y. A float value can also be used as input in which case it is treated as a unix timestamp. A mapping or sequence as input will be passed directly to datetime as keyword or positional arguments.
`dataclass()`	Expects a mapping input with the dataclass field names as keys.

CHAPTER

SIX

PYTHON API

6.1 marsh

Root of the marsh framework.

Exposes the basic functionality and modules of the framework.

marshal(*value*)

Serialize a value to a JSON-like object.

Parameters
value (Any) – The input value to serialize.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The JSON representation of the object.

unmarshal(*type_: Type[_T]*, *element: Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]] = marsh.MISSING*) → _T

unmarshal(*type_: Type[_T]*, *element: Mapping[str, ElementType] = marsh.MISSING*, ***kwargs*) → _T

Unmarshal a JSON-like object to some type.

Note: Static type checkers may give an error for abstract types (issue).

Some type aliases are also not recognized as types. For the moment there is not better solution than using a simple `# type: ignore` comment.

Parameters

- **type** – The type to cast to.
- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The JSON-like object.

Return type
TypeVar(_T)

Returns
The unmarshaled value.

cast(*type_*, *value='???'*)

Cast a value to some type. The input value is serialized to a JSON-like object and then used for constructing an instance of the specified type.

Note: Static type checkers may give an error for abstract types (issue).

Some type aliases are also not recognized as types. For the moment there is not better solution than using a simple `# type: ignore` comment.

Parameters

- **type** – The type to cast to.
- **value** (Any) – The value to cast.
- **type_** (*Type[_T]*) –

Return type
TypeVar(_T)

Returns
The casted value.

unmarshal_args(*callable_*, *element='???'*)

Return the unmarshaled arguments of the given function.

Parameters

- **callable** – The function whos arguments to unmarshal.
- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input to the unmarshaler.
- **callable_** (*Callable*) –

Return type
Arguments

Returns
Tuple of positional arguments and keyword arguments.

cast_args(*callable_*, *value='???'*)

Return the casted arguments of the given function.

The given input will be marshaled before being unmarshaled into the function arguments.

Parameters

- **callable** – The function whos arguments to unmarshal.
- **value** (Any) – The input.
- **callable_** (*Callable*) –

Return type
Arguments

Returns
Tuple of positional arguments and keyword arguments.

main(**, setup_logging: bool = False, prog: Optional[str] = None, description: Optional[str] = None, epilog: Optional[str] = None, help_depth: int = 1*) → Callable[[_C], _C]

main(*fn: _C*, */*, ***, *setup_logging: bool = False*, *prog: Optional[str] = None*, *description: Optional[str] = None*, *epilog: Optional[str] = None*, *help_depth: int = 1*) → _C

main(***, *config: Type[_T]*, *setup_logging: bool = False*, *prog: Optional[str] = None*, *description: Optional[str] = None*, *epilog: Optional[str] = None*, *help_depth: int = 1*) → Callable[[Callable[[_T], _R]], Callable[[_T], _R]]

main(*fn: Callable[[_T], _R]*, */*, ***, *config: Type[_T]*, *setup_logging: bool = False*, *prog: Optional[str] = None*, *description: Optional[str] = None*, *epilog: Optional[str] = None*, *help_depth: int = 1*) → Callable[[_T], _R]

Decorates a function, allowing it to be called as an entry point for a python script being run from the command line.

If a config class is supplied, the decorated function will be passed an instance of that class as a single positional argument. If no config class is supplied, the arguments for the decorated function are filled out and passed to the function.

Parameters

- **fn** – The callable function to decorate. When decorating, this argument should not be explicitly specified.
- **config** (Optional[Any]) – An optional config class.
- **setup_loggin** – Add console line arguments for logging level and format. Initiallizes logging through `logging.basicConfig()`.
- **prog** (Optional[str]) – Optional program name. Defaults to the first value of `sys.argv`.
- **description** (Optional[str]) – Optional description shown in the help message. Defaults to the docstring of the decorated function (or the config class if passed to the decorator). Can be set to `None` to disable completely.
- **epilog** (Optional[str]) – Optional epilog shown in the help message.
- **help_depth** (int) – How many subfields of a config to flatten for `--help [PATH]`.

Returns

The wrapped function.

MISSING

Represents the missing value, i.e. when a value is unset (which is not equivalent to a value being `None`).

Useful for when a default value is required by python even though there should not be one. `marsh` treats this the same as if there were no default value.

The concrete value of `MISSING` is the string `'???'`.

See *marsh.MISSING* for more info.

namespaces: *marsh.schema.namespace.Namespaces*

Gives access to all namespaces. Also allows for new namespaces to be created through its `new(...)` method.

__version__: str

The current version of `marsh`.

6.2 marsh.MISSING

This value is treated as the unset value by *marsh* (not to be confused with None which may be a valid value).

This can be useful when a default value is required by python even though the programmer wants the value to be required (missing).

```
import dataclasses
import marsh

@dataclasses.dataclass
class Config:
    a: int = 3

@dataclasses.dataclass
class ExpandedConfig(Config):
    b: str
```

The code above will fail to run, throwing `TypeError: non-default argument 'b' follows default argument`

Instead we can use *marsh.MISSING*

```
...

@dataclasses.dataclass
class ExpandedConfig(Config):
    b: str  = marsh.MISSING
```

Trying to unmarshal this code we get the correct behavior

```
...

# fails with marsh.errors.MissingValueError
marsh.unmarshal(ExpandedConfig, {'a': 2})
```

Note: Using *marsh.MISSING* as input when unmarshaling is sometimes allowed. For example, None accepts *marsh.MISSING*. Mappings and sequences will also default to an empty instance when the input is *marsh.MISSING*.

The value of *marsh.MISSING* is the same as `omegaconf.MISSING`, represented by the string `'???'`.

6.3 marsh.schema.core

The core functionality powering marshaling and unmarshaling.

class Schema(*value*, **args*, ***kwargs*)

Base class for schemas.

A schema is a class that can be matched against a value and then initialized with that value and any other arguments.

The base class works as a factory when its called as a constructor, selecting the correctly matched schema and returning an instance of it.

Parameters

- **value** (Any) – The value to match.
- **args** – Unused but caught positional arguments.
- **kwargs** – Unused but caught keyword arguments.

classmethod match(*value*)

Match this class against a value.

Parameters
value (Any) – The value to match against.

Return type
bool

Returns
True if matched, else False.

static doc_static_type()

Get a static string representation of the type supported by this schema.

Return type
Optional[str]

Returns
The name of the type.

static doc_static_description()

Get a static string description of how the type supported by this schema is used.

Return type
Optional[str]

Returns
The description.

class SchemaMeta

Metaclass for schemas.

If calling the constructor on the base schema class it will instead match the input with all registered schemas and return an instance of a subclass.

registry: *SchemaRegistry*[_S] = <marsh.schema.core.base.SchemaRegistry object>

The registered schema types.

class SchemaRegistry(*error=<class 'marsh.errors.MarshError'>*)

Holds schema classes in a prioritized order.

Parameters
error (Type[Exception]) – Type of the error to throw when no schema could be matched during selection.

match(*value*)

Match a value to one or more schemas types.

Selected schema types are returned in the order they were match. The matching is canceled and the results are returned when the first non-wrapper schema type has been matched.

Raises an error if no non-wrapper schema could be matched.

Parameters
value (Any) – Value to match against schema types.

Return type
SchemaSelection[TypeVar(_S, bound= Schema)]

Returns
Selection of schema types.

class SchemaSelection(*iterable=()*, */*)

A sequence of schema types that were matched against a value.

Contains one or more schema types, with all but the last one being wrapper classes.

build(**args*, ***kwargs*)

Build the selected schema(s) into a single schema.

Parameters

- **args** – Any positional arguments to pass on to the schema constructor.
- **kwargs** – Any keyword arguments to pass on to the schema constructor.

Return type
TypeVar(_S, bound= Schema)

Returns
Schema instance.

6.4 marsh.schema

This module houses the mechanics that powers marshaling and unmarshaling.

class MarshalSchema(*value*, **args*, ***kwargs*)

Marshals its value input into a JSON-like object.

Parameters
value (Any) – The value to match/marshal.

marshal()

Marshal the value held by this instance into a JSON-like object.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The JSON-like marshaled value.

static doc_static_description()

Get a static string description of how the type supported by this schema is used.

Return type
Optional[str]

Returns
The description.

static doc_static_type()

Get a static string representation of the type supported by this schema.

Return type
Optional[str]

Returns
The name of the type.

classmethod match(*value*)

Match this class against a value.

Parameters
value (Any) – The value to match against.

Return type
bool

Returns
True if matched, else False.

class UnmarshalSchema(**args*, *default='???'*, *default_factory='???'*, ***kwargs*)

Unmarshals JSON-like inputs into its value type.

Note: Static type checkers may give an error for abstract types (issue).

Some type aliases are also not recognized as types. For the moment there is not better solution than using a simple `# type: ignore` comment.

Parameters

- **value** (*Type[_T]*) – The value to match/unmarshal.
- **default** (TypeVar(_T)) – An immutable default value. Mutually exclusive with default_factory.
- **default_factory** (Callable[..., TypeVar(_T)]) – Factory function for a mutable default value. Mutually exclusive with default.

class Doc(*type=None*, *default=None*, *description=None*, *fields=None*, *special_fields=None*)

Structured unmarshal documentation for a type.

Parameters

- **type** (*Optional[str]*) –
- **default** (*Optional[str]*) –
- **description** (*Optional[str]*) –
- **fields** (*Optional[Mapping[str, Field]]*) –
- **special_fields** (*Optional[Sequence[SpecialField]]*) –

class Field(*doc*, *type=None*, *description=None*)

Unmarshal documentation for a field/attribute of a type.

Parameters
- **doc** (Doc) –
- **type** (*Optional[str]*) –
- **description** (*Optional[str]*) –

doc: *Doc*

The field unmarshal documentation.

type: Optional[str] = None

Type string as should be displayed when this is a field of a parent.

description: Optional[str] = None

Description for field by parent.

class SpecialField(*value*, *description=None*)

Unmarshal documentation for a special field.

A special field may document any value.

Parameters

- **value** (*str*) –
- **description** (*Optional[str]*) –

value: str

The name of the value being documented.

description: Optional[str] = None

Its description.

type: Optional[str] = None

String representation of the type.

default: Optional[str] = None

Default value.

description: Optional[str] = None

Description for type.

fields: Optional[Mapping[str, *Field*]] = None

Named internal fields of the documented type.

special_fields: Optional[Sequence[*SpecialField*]] = None

Any other fields that do not follow the name/type/default style.

has_default()

Evaluates if this schema holds a default value.

Return type

bool

Returns

`True` if there is a default value, else `False`.

get_default()

Get the default value of this schema.

If default value exists *marsh.MISSING* is returned instead.

Return type

TypeVar(_T)

Returns

The default value or *marsh.MISSING*.

select(*path*)

Select a nested schema from this schema based on a dot-separated path. If not all of the path is consumed by this schema it is forwarded to the *select* method of the nested schema.

Parameters

path (str) – The path to traverse.

Return type
UnmarshalSchema

Returns
The schema at the end of the path.

doc(*depth=0*)

Get the documentation for the type unmarshaled by this schema.

Parameters
depth (int) – How deep should documentation be fetched in nested schemas. If `0`, only the fields and info for the type of this schema is returned.

Return type
Doc

Returns
The documentation.

doc_type()

The name of the type unmarshaled by this schema.

Return type
str

Returns
The type name.

doc_field_type()

Alternative name of the type unmarshaled by this schema.

This corresponds to the type that should be display in parent documentation when this is one of its fields.

Return type
str

Returns
Alternative type name.

doc_default()

Document default value.

The default implementation of this method returns None if no default value exists and `'...'` if the default value is mutable. If the default value is immutable, an attempt is first made to marshal the default value before returning a string of its marshaled state. If this fails, a string of the default is returned.

Return type
Optional[str]

Returns
String of default.

doc_description()

A description for the type unmarshaled by this schema.

Return type
Optional[str]

Returns
The description.

doc_fields(*depth*)

Documentation for the fields of the type unmarshaled by this schema.

No documentation is returned if no fields exist.

Return type
Optional[Mapping[str, *Field*]]

Returns
Field documentation.

Parameters
depth (*int*) –

doc_special_fields()

Documentation for the special fields of the type unmarshaled by this schema.

No documentation is returned if no special fields exist.

Return type
Optional[Sequence[*SpecialField*]]

Returns
Special field documentation.

unmarshal(*element*)

Construct the object of this schema.

Parameters
element (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input JSON-style data.

Return type
TypeVar(_T)

Returns
The object initialized with the input data.

static doc_static_description()

Get a static string description of how the type supported by this schema is used.

Return type
Optional[str]

Returns
The description.

static doc_static_type()

Get a static string representation of the type supported by this schema.

Return type
Optional[str]

Returns
The name of the type.

classmethod match(*value*)

Match this class against a value.

Parameters
value (Any) – The value to match against.

Return type
bool

Returns
True if matched, else False.

register(*schema: Type[_S], /, *, priority: int = 0, lower_priority: Union[None, Type[Schema], int, Iterable[Type[Schema]], Iterable[int]] = None, higher_priority: Union[None, Type[Schema], int, Iterable[Type[Schema]], Iterable[int]] = None, replace: bool = False*) → Type[_S]

register(*, *priority: int = 0, lower_priority: Union[None, Type[Schema], int, Iterable[Type[Schema]], Iterable[int]] = None, higher_priority: Union[None, Type[Schema], int, Iterable[Type[Schema]], Iterable[int]] = None, replace: bool = False*) → Callable[[Type[_S]], Type[_S]]

Register a new schema (decorator).

Relative priorities can be given as higher and lower, where any schema in higher means that the schema being registered should have a lower priority than that, the opposite is true for lower.

The relative priority (if given) takes precedence over the base priority.

Parameters

- **schema** (Optional[Type[TypeVar(_S, bound= *Schema*)]]) – The schema to register.
- **priority** (int) – Base priority level.
- **lower_priority** (Union[None, Type[*Schema*], int, Iterable[Type[*Schema*]], Iterable[int]]) – Any other schemas that should have lower priority compared to the new schema.
- **higher_priority** (Union[None, Type[*Schema*], int, Iterable[Type[*Schema*]], Iterable[int]]) – Any other schemas that should have higher priority compared to the new schema.
- **replace** (bool) – Allow replacing existing registered schema with same name.

Return type
Union[Callable[[Type[TypeVar(_S, bound= *Schema*)]], Type[TypeVar(_S, bound= *Schema*)]], Type[TypeVar(_S, bound= *Schema*)]]

Returns
Decorator if schema was not given, else schema.

caches

An instance of *marsh.utils.CachePool* containing the caches used throughout *marsh.schema*

6.5 marsh.schema.template

A set of template implementations of schemas.

These templates are useful as base classes when extending the framework to support more types.

class CallableUnmarshalSchema(**args*, ***kwargs*)

Unmarshals the arguments to a callable and uses them to call it.

Parameters
value (*Type[_T]*) –

schemas: Mapping[str, *UnmarshalSchema*]

The schemas for the fixed attributes.

construct(**args*, ***kwargs*)

Takes the unmarshaled arguments to the type as input and initializes an instance of the type using these arguments.

Return type
TypeVar(_T)

Returns
The initialized value.

class MappingUnmarshalSchema(**args*, *default='???'*, *default_factory='???'*, ***kwargs*)

A schema for a type with dynamic keys and values.

Parameters
value (*Type[_T]*) –

key_schema: *UnmarshalSchema*

The schema for the key type.

value_schema: *UnmarshalSchema*

The schema for the value type.

construct(*value*)

Build an instance of the type from unmarshaled keys and values.

Parameters
value (Mapping) – The unmarshaled keys and values.

Return type
TypeVar(_T)

Returns
An instance of the type.

class SequenceUnmarshalSchema(**args*, *default='???'*, *default_factory='???'*, ***kwargs*)

A schema for a dynamic sequence of values.

Parameters
value (*Type[_T]*) –

value_schema: *UnmarshalSchema*

The schema for the value type.

construct(*value*)

Build an instance of the type from unmarshaled values.

Parameters
value (Sequence) – The unmarshaled values.

Return type
TypeVar(_T)

Returns
An instance of the type.

class StructuredUnmarshalSchema(**args*, *default='???'*, *default_factory='???'*, ***kwargs*)

A schema for types with fixed attributes that are recursively unmarshaled.

Supports both positional arguments and keyword arguments. Variable versions of these arguments (typically `*args`/`**kwargs`) are also supported and are marked by * or ** infront of their names in the *schemas* attribute of this class.

Parameters
value (*Type[_T]*) –

schemas: `Mapping[str, UnmarshalSchema]`

The schemas for the fixed attributes.

construct(**args*, ***kwargs*)

Takes the unmarshaled arguments to the type as input and initializes an instance of the type using these arguments.

Return type
TypeVar(_T)

Returns
The initialized value.

class UnionUnmarshalSchema(**args*, *default='???'*, *default_factory='???'*, ***kwargs*)

A schema for a union of types.

The order of the types is reflected in the order of attempted unmarshals. The value of the first type that is successfully unmarshaled is returned.

If all types failed to unmarshal an error is raised.

Parameters
value (*Type[_T]*) –

schemas: `Sequence[UnmarshalSchema]`

The schemas for the union of types.

6.6 marsh.schema.namespace

This module creates support for inheritance-based type matching.

For all types in a namespace each subclass of a type being that is being unmarshaled is considered.

class Namespace(*name*, *base*)

Holds a group of subclasses for a base class which are each associated with a name.

This class should be initialized via *Namespaces.new()*.

Includes functionality for matching a type against the classes held by this namespace. This functionality is cached for improved performance.

Parameters
base (Type[TypeVar(_T)]) – The base class for the namespace.

class CacheInfo(**args*, ***kwargs*)

cache_clear()

Clear the cache.

Return type
None

cache_info()

Get info for the cache.

Return type
CacheInfo

Returns
The cache info.

`find_class`(*cls*)

Find the name of a class maybe held by this namespace.

Parameters
cls (Type[TypeVar(_T)]) – The type to find a name for.

Return type
Optional[str]

Returns
None if no class was matched, else the name associated with the type

`find_subclasses_iter`(*cls*)

Yield names of components in this namespace that are subclasses of the input type.

This method provides results for the same but cached method *find_subclasses()*.

Parameters
cls (Type[TypeVar(_T)]) – The input type.

Return type
Iterator[str]

Returns
Iterator of string names.

`find_subclasses`(*cls*)

Get the names of all classes held by this namespace that are subclasses of the input.

Parameters
cls (Type[TypeVar(_T)]) – The type to find subclasses for.

Return type
IterableFromIterator[str]

Returns
The names of the subclasses.

`register`(*component: Type[_T]*, *, *name: str*, *replace: bool = False*) → Type[_T]

`register`(*, *name: str*, *replace: bool = False*) → Callable[[Type[_T]], Type[_T]]

Register a new type for this namespace.

Parameters

- **name** (str) – The name to associate with the type.
- **replace** (bool) – Allow replacing any existing value with the same name.

Return type
Union[Callable[[Type[TypeVar(_T)]], Type[TypeVar(_T)]], Type[TypeVar(_T)]]

Returns
A decorator for registering the type if no type was given else the type itself.

`build`(*element: Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]] = marsh.MISSING*, *, *name: str*) → _T

`build`(*element: Optional[Sequence[ElementType]] = marsh.MISSING*, **args*, *name: str*) → _T

build(*element: Optional[Mapping[str, ElementType]] = marsh.MISSING*, *, *name: str*, ***kwargs*) → _T

Unmarshal a type held by this namespace.

Return type
TypeVar(_T)

class Namespaces

Singleton class holding all namespaces.

Includes functionality for matching a type against the classes of all namespaces. These matching functions are cached for improved performance.

class CacheInfo(**args*, ***kwargs*)

class FullCacheInfo(**args*, ***kwargs*)

find_subclasses_iter(*cls*)

Yield results from finding subclasses of an input type in all namespaces.

This method provides results for the same but cached method *find_subclasses()*.

Parameters
cls (Any) – The input type.

Return type
Iterator[Tuple[str, *Namespace*]]

Returns
Iterator of result.

find_namespaces_iter(*cls*)

Yield results from finding namespaces matching an input type.

This method provides results for the same but cached method *find_namespaces()*.

Parameters
cls (Any) – The input type.

Return type
Iterator[*Namespace*]

Returns
Iterator of result.

new(*name*, *base*)

Create a new namespace.

The created namespace has functions for registering new items and building them.

Parameters

- **name** (str) – Name of namespace.
- **base** (Type[TypeVar(_T)]) – Base class for namespace.

Return type
Namespace[TypeVar(_T)]

Returns
The namespace.

cache_clear(*full=False*)

Clear the cache of this class.

Parameters
full (bool) – If True, also clear the cache of all namespaces.

Return type
None

cache_info(*full: Literal[True]*) → *Namespaces.FullCacheInfo*

cache_info() → *Namespaces.CacheInfo*

Get info for the cache of this class.

Parameters
full (bool) – If True, return the cache info for all namespaces as well.

Return type
Union[*CacheInfo*, *FullCacheInfo*]

Returns
The cache info.

find_class(*cls*)

Search through all namespaces and return the name of the first class that equals the given type.

Parameters
cls (Any) – The type to find a name for.

Return type
Optional[str]

Returns
A name if found, else None.

find_subclasses(*cls*)

Search for all subclasses of the given type.

Argument:
cls: The type to find subclasses for.

Return type
IterableFromIterator[Tuple[str, *Namespace*]]

Returns
Iterable of names and namespaces.

Parameters
cls (*Any*) –

find_namespaces(*cls*)

Find the namespaces with base classes that are superclasses of the given type.

Parameters
cls (Any) – The type to find namespaces for.

Return type
IterableFromIterator[*Namespace*]

Returns
Iterable of namespaces found.

6.7 marsh.element

Tools for working with the raw JSON-like data.

This type of data is refered to as *element* in the marsh framework. It is what is produced when marshaling a python value and it is given as input when unmarshaling a python type.

ElementType: TypeAlias = Union[None, int, float, bool, str, Sequence[ElementType], Mapping[str, ElementType]]

A type alias for any element value.

SequenceElementType: TypeAlias = Sequence[ElementType]

A type alias for a sequence element value.

MappingElementType: TypeAlias = Mapping[str, ElementType]

A type alias for a mapping element value.

TerminalElementType: TypeAlias = Union[None, int, float, bool, str]

A type alias for a terminal element value.

merge(*element*, **elements*, *concatenate=False*)

Combine two or more elements.

Maps are combined recursively. All other types of values are replaced with the last element in the merge (with an exception for sequences if `concatenate` is `True`). The final element in the inputs will be given the highest priority in the merge, i.e. if it contains a value for a specific path that exists in other elements in the input its value will be the one that exists in the final merged element (output).

Parameters

- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The first element.
- **elements** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – Remaining elements to merge into the first element.
- **concatenate** (bool) – Concatenate sequences with eachother when merging instead of replacing the old sequence with the new sequence.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
A new element which is the combination of all input elements.

override(*element*, *value*, *path*, *combine=False*)

Set the value of a specific path in an element.

If the path is already occupied by a value, it will be replaced. A copy of the input element is returned with the value set for the specified path.

Parameters

- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input element.
- **value** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The value to set for the path.
- **path** (str) – The path for the new value.

- **combine** (bool) – If a previous value exists for the path, attempt to merge it with the new value. This only applies when both values are maps or sequences. For maps, the new element takes precedence when a key exists in both values. For sequences, the old value is concatenated with the new value (*old* + *new*).

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
A copy of the element with a new value set for the specified path.

remove(*element*, *path*)

Remove a subelement in the input specified by a path.

A copy of the input element is returned with the final subelement referenced by the path removed.

Parameters

- **element** (Union[Sequence[Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]], Mapping[str, Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]]]) – The element containing a value to be removed.
- **path** (*str*) –

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
A copy of the element with the value for the specified path removed.

select(*element*, *path*)

Retrieve the value for a specific path in an element.

Parameters

- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input element to retrieve a value from.
- **path** (str) – The path to retrieve the value from.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
Value for the specified path.

iterative_select(*element*, *path*)

Iterate through all elements in a specified path.

At each step in the path a selection is yielded containing the current element, the path traversed and the remaining untraversed path.

Parameters

- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input element to yield subelements from based on the specified path.
- **path** (str) – The path to traverse.

Return type
Iterator[*ElementSelection*]

Returns
Selection iterator.

standardize(*element*, *, *mapping_type=<class 'dict'>*, *sequence_type=<class 'tuple'>*)

Standardize the sequence/mapping types through an entire element.

Copy the element and replace all sequences with the specified sequence type and all mappings with the specified mapping type recursively.

Parameters

- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The element to standardize sequence and mapping types for.
- **sequence_type** (Type[Sequence]) – The type to use for sequences.
- **mapping_type** (Type[Mapping]) – The type to use for mappings.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The standardized element.

has_missing(*element*)

Recursively look for missing values in an element.

Returns `True` if the element is missing or contains any missing values, otherwise `False` is returned. A missing value is represented by `marsh.MISSING` or `dataclasses.MISSING`.

Parameters
element (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – Element to check for missing values in.

Return type
bool

Returns
Boolean representing if the element is missing or contains missing values.

resolve(*element*)

Resolve any variable interpolation using omegaconf resolvers.

Parameters
element (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The element to resolve any variable interpolations for.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The element with variable interpolations performed.

class ElementSelection(*element*, *path*, *remaining_path*)

Parameters

- **element** (*Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]*) –
- **path** (*str*) –
- **remaining_path** (*str*) –

6.8 marsh.annotation

Annotations that can be used with `typing.Annotated`.

Annotations may be used to customize the unmarshaling behavior through transforms or validations.

```
from typing import Annotated
import marsh

UnsignedInt = Annotated[int, marsh.annotation.Unsigned]
value = marsh.unmarshal(UnsignedInt, 5)  # works
value = marsh.unmarshal(UnsignedInt, -5)  # fails
```

Create your own annotations by inheriting *Annotation*.

```
from typing import Annotated
import marsh

class Abs(marsh.annotation.Annotation):

    def __call__(
        self,
        value
    ):
        return abs(value)

AbsInt = Annotated[int, Abs]
assert marsh.unmarshal(AbsInt, -5) == 5
```

class Annotation

Base class for annotations recognized by *marsh*.

These can be used with `typing.Annotated` to include validation or transformation of the unmarshal result.

class Positive

Validate that a number is non-zero and positive.

class Negative

Validate that a number is non-zero and negative.

class Unsigned

Validate that a number is zero or higher.

class Populated

Validate that a collection contains at least one item.

6.9 marsh.utils

Collection of general utilities used throughout the framework.

class SequenceProtocol(**args*, ***kwargs*)

Protocol matching a sequence class.

class MappingProtocol(**args*, ***kwargs*)

Protocol matching a mapping class.

class NamedTupleProtocol(**args*, ***kwargs*)

Protocol matching a namedtuple class.

class SingletonMeta

Converts a class into a singleton.

When the constructor of the class is called the first time, an instance is constructed. After this, the constructor will always return that instance unless it is garbage collected.

To allow for garbage collection of unused instances a weak reference is kept of the instance.

class CacheInfo(*hits=0*, *misses=0*, *maxsize=None*, *currsize=0*)

Tuple with information for a cache.

Parameters

- **hits** (*int*) –
- **misses** (*int*) –
- **maxsize** (*Optional[int]*) –
- **currsize** (*int*) –

hits: int

Alias for field number 0

misses: int

Alias for field number 1

maxsize: Optional[int]

Alias for field number 2

currsize: int

Alias for field number 3

classmethod from_info(*info*)

Create an instance of this class from any cache info that follows the same attribute pattern.

Parameters
info (_CacheInfoInputType) – The cache info to use as source

Return type
CacheInfo

Returns
An instance of this class containing the same information as the input.

class CacheType(**args*, ***kwargs*)

Represents the protocol for a cache in *marsh*.

cache_info()

Get info from the cache.

Return type
CacheInfo

Returns
The cache info.

cache_clear()

Clear the content of the cache.

Return type
None

cache_disable()

Disable the cache.

Return type
None

cache_enable()

Enable the cache.

Return type
None

cache(*, *maxsize: Optional[int] = None*, *typed: bool = False*, *safe: bool = True*, *binding: Optional[Literal['weak', 'ignore']] = None*) → Callable[[_C], _C]

cache(*fn: _C*, */*, *, *maxsize: Optional[int] = None*, *typed: bool = False*, *safe: bool = True*, *binding: Optional[Literal['weak', 'ignore']] = None*) → _C

Extended LRU cache decorator for functions.

Allows for safe usage of lru cache, where the TypeError produced by non-hashable types in the input arguments are caught before a new attempt is made to call the function, this time bypassing the cache compmletely.

Note: The bypass is triggered for TypeError. If the function itself can raise this error it will be called twice before the final error is propagated.

The wrapped function fulfills the *CacheType* protocol.

Parameters

- **fn** (Optional[TypeVar(_C, bound= Callable)]) – If None, the wrapper is returned, otherwise this argument is wrapped and returned.
- **maxsize** (Optional[int]) – Maximum size of lru cache.
- **typed** (bool) – Differentiate between different types of values that produce the same hash.
- **safe** (bool) – Make a second attempt at calling the function and bypassing the cache if the first attempt fails with a TypeError.
- **binding** (Optional[Literal[‘weak’, ‘ignore’]]) – Should only be used when a method to a class is decorated with this function. `'weak'` will make sure that a weak reference to `self/cls/metacls` is stored in the cache instead of a strong reference. This allows for objects with lru_cache to be garbage collected when no longer used. `'ignore'` allows the first argument to bypass the cache, only performing a lookup on the other arguments.

Return type
Union[TypeVar(_C, bound= Callable), Callable[[TypeVar(_C, bound= Callable)], TypeVar(_C, bound= Callable)]]

Returns
The wrapper, unless `fn` is given, which instead wraps the argument and returns it.

make_hash_key(**args*, ***kwargs*)

Make a single cached hashable key from arguments.

All arguments must be hashable.

Return type
Any

Returns
Hashable object that caches its hash, only caclulating it once.

make_typed_hash_key(**args*, ***kwargs*)

Make a single cached hashable key from arguments.

Includes the type of each value.

All arguments must be hashable.

Return type
Any

Returns
Hashable object that caches its hash, only caclulating it once.

class SafeDict

Dictionary that works with unhashable keys.

Just like built-in `dict` the order is kept.

class ValueCache(*typed=False*)

A dict-like cache that allows for non-hashable keys.

The hash for cachable keys are also hashed to improve lookup speed.

Parameters
typed (bool) – The key type is also taken into consideration when matching keys.

cache_clear()

Clear the cache, removing all stored items.

Return type
None

cache_info()

Get statistics for the cache.

Return type
CacheInfo

cache_disable()

Disable the cache.

Getting a value for a key raise KeyError. Setting a value for a key is a no-op (the value is not stored). Deleting a value in the cache is still permitted.

Return type
None

cache_enable()

Enable the cache.

Return type
None

class CachePool

A collection of caches.

add(*name*, *cache*)

Add a cache to the collection.

Parameters

- **name** (str) – A name for the cache being added.
- **cached** – The cache to add.
- **cache** (CacheType) –

Return type
None

new_callable_cache(*, *name: str*, *maxsize: Optional[int] = None*, *typed: bool = False*, *safe: bool = True*, *binding: Optional[Literal['weak', 'ignore']] = None*) → Callable[[_C], _C]

new_callable_cache(*fn: _C*, *, *name: str*, *maxsize: Optional[int] = None*, *typed: bool = False*, *safe: bool = True*, *binding: Optional[Literal['weak', 'ignore']] = None*) → _C

Wrap a function with *cache()* while simultaneously adding it to this pool.

See *cache()* for descriptions of arguments forwarded to the wrapper.

Parameters
name (str) – The name associated with the new cache.

Return type
Union[TypeVar(_C, bound= Callable), Callable[[TypeVar(_C, bound= Callable)], TypeVar(_C, bound= Callable)]]

Returns
The cache wrapper.

new_value_cache(*name*)

Create a new value cache and add it to this pool before returning it.

Parameters
name (str) – The name associated with the new cache.

Return type
ValueCache

Returns
The value cache.

cache_clear()

Clear every cache in this container.

This does not remove the caches from the container, it only calls the `clear_cache()` function on each cache.

Return type
None

`cache_info()`

Get the cache info for all caches in this container.

Return type
Mapping[str, *CacheInfo*]

Returns
A mapping of the name of each cache as key and its respective cache info as value.

`cache_disable()`

Disable all caches in this container.

Return type
None

`cache_enable()`

Enable all caches in this container.

Return type
None

`class IterableFromIterator(iterator)`

Lazily stores results of an iterator.

Allows for multiple iterations over the wrapped iterator while only consuming it once.

Parameters
iterator (Iterator[TypeVar(_T)]) – The iterator to cache.

`class PriorityOrder`

A mutable sequence of values ordered by priority.

Higher priority items are moved forward while lower priority items are moved backward in the sequence.

`RelativePriority`

Relative priority input type.

alias of Union[None, _V, int, Iterable[_V], Iterable[int]]

`reload()`

Reshuffle values in this container based on priority.

Return type
None

`class WeakTypeCache(skip_types=())`

A type cache that holds its values as weak references.

When iterated over the types are yielded in reverse order, the most recently added is yielded first.

Parameters
skip_types (Iterable[Any]) – Any types to ignore. If attempting to store one of these types it becomes a no-op.

`add(type_)`

Add a type to the cache.

Parameters

- **type** – The type to add.

- **type_** (*Any*) –

Return type
None

clear()

Clear the cache, removing all stored types.

Return type
None

is_missing(*value*)

Determine if a value is missing.

Supports *marsh.MISSING*, omegaconf.MISSING and dataclasses.MISSING.

Parameters
value (Any) – Any value to test for missing.

Return type
bool

Returns
True if missing, else False.

is_sequence(*value*)

Determine if a value is a sequence (excluding string, bytes).

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Sequence]

Returns
True if value is a sequence, else False.

is_sequence_type(*value*)

Determine if a value is a sequence type (excluding string, bytes types).

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Type[Sequence]]

Returns
True if value is a sequence type, else False.

is_empty_tuple_type(*value*)

Determine if a value is an empty tuple type (typing.Tuple[()], tuple[()]).

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Type[Tuple]]

Returns
True if value is a an empty tuple type, else False.

is_fixed_size_tuple_type(*value*)

Determine if a value is fixed size tuple type.

Parameters
value – Any value to evaluate.

Return type
bool

Returns
True if value is a a fixed size tuple type, else False.

is_mapping(*value*)

Determine if a value is a mapping.

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Mapping]

Returns
True if value is a mapping, else False.

is_mapping_type(*value*)

Determine if a value is a mapping type.

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Type[Mapping]]

Returns
True if value is a mapping, else False.

is_primitive(*value*)

Determine if a value is primitive.

```
PrimitiveElement:  int | float | bool | str
```

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Union[int, float, bool, str]]

Returns
True if value is primitive, else False.

is_primitive_type(*value*)

Determine if a value is primitive type.

```
PrimitiveElement:  int | float | bool | str
```

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Union[Type[int], Type[float], Type[bool], Type[str]]]

Returns
True if value is a primitive type, else False.

`is_namedtuple`(*value*)

Determine if a value is a namedtuple.

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[*NamedTupleProtocol*]

Returns
True if value is a namedtuple, else False.

`is_namedtuple_type`(*value*)

Determine if a value is a namedtuple type.

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Type[*NamedTupleProtocol*]]

Returns
True if value is a namedtuple type, else False.

`is_typed_namedtuple`(*value*)

Determine if a value is a typed namedtuple.

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[*NamedTupleProtocol*]

Returns
True if value is a namedtuple, else False.

`is_typed_namedtuple_type`(*value*)

Determine if a value is a typed namedtuple type.

Parameters
value (Any) – Any value to evaluate.

Return type
TypeGuard[Type[*NamedTupleProtocol*]]

Returns
True if value is a typed namedtuple type, else False.

`is_typeddict_type`(*value*)

Determine if a value is a typed dict type.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a typed dict type, else False.

`is_defaultdict_type`(*value*)

Determine if a value is a default dict type.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a default dict type, else False.

`is_callable`(*value*)

Determine if a value is callable.

Always returns False if the value is a protocol or typing alias.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a callable, else False.

`is_obj_instance`(*value*)

Determine if a value is an instantiated object.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is an iinstantiated object, else False.

`is_annotated`(*value*)

Determine if a value originates from typing.Annotated.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value originates from typing.Annotated, else False.

`get_annotations`(*value*)

Extract the additional arguments of an instance of typing.Annotated

Parameters
value (Any) – The typing.Annotated instance.

Return type
Sequence

Returns
The annotations.

is_protocol(*value*)

Determine if a value is a protocol.

Any class that has typing.Protocol in its immediate bases is considered a protocol.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a protocol, else False.

is_literal(*value*)

Determine if a value is a typing.Literal.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a typing.Literal, else False.

is_literal_string(*value*)

Determine if a value is a typing.LiteralString.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a typing.LiteralString, else False.

is_typing_alias(*value*)

Determine if a value is a typing alias.

If the class name ends with '_GenericAlias', '_SpecialForm' or '_AnnotatedAlias' the value is considered a typing alias.

For python 3.9+ a check is made against types.GenericAlias.

Parameters
value (Any) – Any value to evaluate.

Return type
bool

Returns
True if value is a typing alias, else False.

is_optional(*type_*)

Determine if a type is a optional.

Parameters

- **type** – The type to evaluate.
- **type_** (*Any*) –

Return type
bool

Returns
True if value is typing.Union and None is in its arguments, else False.

get_type(*value*)

Attempt to retrieve the type of an object.

If the value is a generic typing alias, the origin type is returned. I.e. get_type(typing.List[str]) returns the builtins list type.

If the value is an instance of a class, the class is returned.

If the value is a class, it is directly returned.

```
>>> assert get_type([1, 2, 3]) == get_type(list)
```

Parameters
value (Any) – Value to retrieve type from.

Return type
Any

Returns
The type.

get_optional_type(*type_*)

Get the non-optional version of an optional type.

Parameters

- **type** – The optional type.
- **type_** (*Any*) –

Return type
Any

Returns
The type as non-optional.

get_type_name(*value*)

Get a string representation for the type of a value.

Parameters
value (Any) – A type or instance of a type to get a string representation for.

Return type
str

Returns
The string representation of the value.

bytes_to_base64(*value*)

Encode bytes to a base64 string.

Follows the format used by protobuf.

Parameters
value (bytes) – The bytes to encode.

Return type
str

Returns
The base64 string representing the bytes argument.

`base64_to_bytes`(*value*)

Decode a base64 string to bytes.

Follows the format used by protobuf.

Parameters
value (str) – The base64 string to decode.

Return type
bytes

Returns
The bytes representation of the base64 string argument.

`primitive_to_bool`(*value*)

Convert a primitive value to a bool.

Follows the YAML convention instead of using `__bool__` function of the object, see *str_to_bool()* for more info.

Raises ValueError on failure.

Parameters
value (Union[int, float, bool, str]) – The primitive value to convert into bool.

Return type
bool

Returns
The bool representation of the value argument.

`str_to_bool`(*value*)

Cast a string to a boolean.

```
True:  1 | [Tt][Rr][Uu][Ee] | [Yy] | [Yy][Ee][Ss] | [Oo][Nn]
```

```
False:  0 | [Ff][Aa][Ll][Ss][Ee] | [Nn] | [Nn][Oo] | [Oo][Ff][Ff]
```

Inspired by https://yaml.org/type/bool.html

Raises ValueError if the string does not conform to this format.

Parameters
value (str) – String to cast.

Return type
bool

Returns
The input string parsed as a bool.

`str_to_int`(*value*)

Cast a string to an int.

Compared to using int to cast the value this function allows values such as `"0.0"` by first parsing the value as a float before casting it to an int (unless it contains decimals).

Raises ValueError if the input string can not be parsed as an int.

Parameters
value (str) – String input to parse as int.

Return type
int

Returns
The parsed int.

float_to_int(*value*)

Cast a float to an int.

Raises ValueError if the input float is not an integer (contains decimal values). This differs from using int to cast a float since decimal values produce an error instead of being dropped.

Raises ValueError on failure.

Parameters
value (float) – Float input to cast as int.

Return type
int

Returns
Integer of the input value.

cast_primitive(*type_*, *value*)

Cast a primitive value to a specified primitive type.

Uses *primitive_to_bool()*, *str_to_int()* and *float_to_int()* when applicable.

Raises ValueError if unsuccessful.

Parameters

- **primitive** – The primitive type (int, float, bool or str).
- **value** (Union[int, float, bool, str]) – A primitive value to cast.
- **type_** (*Type[_P]*) –

Return type
TypeVar(_P, int, float, bool, str)

Returns
The casted value.

cast_literal(*literal*, *value*)

Attempts to cast a primtive value to the same type as a primitive literal and match it.

Raises ValueError if not able to match.

Parameters

- **literal** (TypeVar(_P, int, float, bool, str)) – The primitive literal to cast to.
- **value** (Union[int, float, bool, str]) – The value to cast.

Return type
TypeVar(_P, int, float, bool, str)

Returns
The literal.

match_literal(*literals*, *value*)

Attempts to cast and match a value to a set of primitive literals.

Raises ValueError if not able to match.

Parameters

- **literals** (Iterable[Union[int, float, bool, str]]) – The primitive literals to match.
- **value** (Union[int, float, bool, str]) – The value to cast and match.

Return type
Union[int, float, bool, str]

Returns
The literal that was matched.

cast_none(*value*)

Attempt to cast a value to None.

Allowed values are case insensitive strings "null" and "none" or a missing value.

Raises ValueError on failure.

Parameters
value (Any) – The value to cast to None

Return type
None

Returns
The casted value.

extract_description(*doc*)

Omit all but the main text in a docstring.

Attempts to omit things such as arguments and return value from a docstring.

Parameters
doc (str) – Docstring.

Return type
str

Returns
The description if found, else the entire docstring.

get_description(*cls*)

Get the docstring for a class.

An attempt is made to omit all but the main text in the docstring. This means that the return value should be missing things such as arguments and return value descriptions.

Parameters
cls – Class to retrieve description from.

Return type
Optional[str]

Returns
The description if found, else None.

`get_attribute_description`(*cls*, *name*)

Get the docstring for an attribute of a class.

Parameters

- **cls** – Class for which an attribute description should be fetched.
- **name** (`str`) – The name of the attribute.

Return type
`Optional`[`str`]

Returns
The attribute description if found, else `None`.

`as_dataclass`(*cls*)

Safely convert a class into a dataclass.

This function can be called multiple times on the same class without it raising an error the second time.

Parameters
cls (`Type`[`TypeVar`(_T)]) – Class to convert into dataclass.

Return type
`Type`[`TypeVar`(_T)]

Returns
Dataclass.

`get_closest`(*value*, *candidates*)

Get the closest value to some input from a set of candidates.

If no close value is found, `None` is returned.

Parameters

- **value** (`str`) – The input value.
- **candidates** (`Iterable`[`str`]) – The values to evaluate closeness to.

Return type
`Optional`[`str`]

Returns
The closest candidate. `None` if no candidate is close.

`get_closest_error_message`(*value*, *candidates*, *key='key'*)

Get an error message for a key.

The error message can take two forms, the first being `'unknown "{key}"'` and the second being `'unknown "{key}", did you mean "{closest}"?'` which is only returned when a close match is found among the candidates.

Parameters

- **value** (`str`) – The input value.
- **candidates** (`Iterable`[`str`]) – The values to evaluate closeness to.
- **key** (`str`) – A name for the key value.

Return type
`str`

Returns
The error message.

inspect_sequence_type(*sequence_type*)

Get the type of the content of a sequence type.

Expects a single content type. If no content type is found or there are multiple content types `typing.Any` is returned.

Parameters

- **type** – The type variable of a sequence.
- **sequence_type** (*Any*) –

Return type
Any

Returns
The type for the content of the input sequence type.

```
>>> get_sequence_type(typing.List[int])
<class 'int'>
>>> get_sequence_type(typing.Tuple[str, ...])
<class 'str'>
>>> get_sequence_type(list)
typing.Any
>>> get_sequence_type(typing.Tuple[str, int, float])
typing.Any
```

inspect_mapping_type(*mapping_type*)

Get the key and value type from a mapping type.

Returns `typing.Any` for types that could not be inferred.

Parameters

- **type** – The type variable of a mapping.
- **mapping_type** (*Any*) –

Return type
_MappingKeyValueTypesInfo

Returns
The key and value type.

```
>>> get_sequence_type(typing.Dict[str, int])
(<class 'str'>, <class 'int'>)
>>> get_sequence_type(typing.Mapping[str, typing.List[int]])
(<class 'str'>, typing.List[int])
>>> get_sequence_type(dict)
(typing.Any, typing.Any)
```

is_testing()

Discover if the current runtime is for testing with Pytest.

Return type
bool

get_terminal_width()

Attempt to get the width of the current terminal.

The terminal size is preprocessed in a similar way to `argparse`.

Return type
int

6.10 marsh.errors

Collection of common errors used throughout the framework.

exception MarshError(**args*, *path=''*)

Base class for marsh errors.

Allows tracking of the path that produced the error.

Parameters

- ***args** – General exception arguments.
- **path** (str) – The path that produced the error.

Return type
None

append(*field*)

Append a field to the path of this error.

Return type
None

Parameters
field (*str*) –

prepend(*field*)

Prepend a field to the path of this error.

Return type
None

Parameters
field (*str*) –

exception PathError(**args*, *path=''*)

Accessing non-existing fields or indices.

Also used for path formatting errors.

Parameters
path (*str*) –

Return type
None

exception UnmarshalError(*msg=''*, *path=''*, *element='???'*, *type='???'*)

Failure to unmarshal an element to a specific type.

Parameters

- **msg** (*str*) –
- **path** (*str*) –
- **element** (*marsh.element.ElementType*) –
- **type** (*Any*) –

Return type
None

exception MissingValueError(*msg=''*, *path=''*, *element='???'*, *type='???'*)

Required value is missing.

Parameters

- **msg** (*str*) –
- **path** (*str*) –
- **element** (*marsh.element.ElementType*) –
- **type** (*Any*) –

Return type
None

exception MarshalError(**args*, *path=''*)

Failure to serialize a value.

Parameters
path (*str*) –

Return type
None

exception ConfigFileError(**args*, *path=''*)

Base error for config files.

Parameters
path (*str*) –

Return type
None

maybe_handle_error(*callback: Callable[[*MarshError*], Any]*) → contextlib.AbstractContextManager

maybe_handle_error(*callback: Callable[[_E], Any]*, *error: Type[_E]*) → contextlib.AbstractContextManager

Context manager that invokes a callback when catching an error.

The error is propagated and the callback ignored if the environmental variable `MARSH_FULL_ERROR` is set and not empty.

The error is passed to the callback.

Parameters
callback (`Callable`[[`Any`], `Any`]) – Callback function which may be called with the caught error.

Return type
`Iterator`[`None`]

Returns
Context manager.

prepend(*field*)

Catches any marsh error, prepends a field to it and then re-raises the error.

Parameters
field (`Any`) – The field to prepend to the marsh error path attribute.

Return type
`Iterator`[`None`]

Returns
Context manager.

6.11 marsh.config

Tools for reading and writing config files.

Support for new file extensions may be added by registering more config schemas.

meta_key: `str = '_meta_'`

This key is used to store any metadata in a config.

The metadata is typically dropped after variable interpolations have been performed.

tree(*path*)

Get a tree representation of a directory recursively searched for configs.

Parameters
path (str) – The path to a directory.

Return type
str

Returns
A tree representation of all available configs in that directory.

load(*name*, **names*, *root=None*, *resolve=True*, *keep_meta=False*, *concatenate=False*)

Load one or more config files.

When more than one config is loaded their contents are merged and the values of the succeeding config takes priority for fields where a merge can not be performed.

File extensions may be omitted.

Parameters

- **name** (str) – Name of a config file.
- ***names** (str) – Any additional config files.
- **root** (Optional[str]) – A root working directory for the config files. Defaults to the current working directory.
- **resolve** (bool) – Resolve omegaconf interpolations.
- **keep_meta** (bool) – Keep meta field if it exists.
- **concatenate** (bool) – Concatenate sequences when multiple configs are loaded and merged instead of replacing the previous value.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The loaded config.

write(*config*, *path='-'*)

Write a config to a file.

By default the config is written to stdout in YAML format.

Parameters

- **config** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The config to write.
- **path** (str) – A filepath to write to. If not specified, stdout is used.

Return type
None

drop_meta(*config*)

Drop the meta key with all its values if present.

Parameters
config (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – Config to drop meta data from.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
Input config without meta data.

6.12 marsh.path

Path tools that work with any separator character.

iterative_split(*path*, *delimiter='.'*)

Iterate through the fields in a path.

Escaped or quoted (unescaped quotes) delimiters will become part of a field instead of splitting it into more fields. Quotes are removed from the final fields.

Parameters
- **path** (str) – The path to split into fields and iterate through.
- **delimiter** (str) – The delimiter between fields in the path.

Return type
Iterator[str]

Returns
Field iterator.

split(*path*, *delimiter='.'*)

Split the fields of a path.

Escaped or quoted (unescaped quotes) delimiters will become part of a field instead of splitting it into more fields. Quotes are removed from the final fields.

Parameters
- **path** (str) – The path to split into fields.
- **delimiter** (str) – The delimiter between fields in the path.

Return type
Sequence[str]

Returns
The fields.

escape_field(*field*, *delimiter='.'*)

Escape all delimiters and quotes found in a field.

This may be done to maintain the original field when it becomes part of a path that is later split back into fields.

Parameters

- **field** (str) – The field to escape characters in.
- **delimiter** (str) – The delimiter used.

Return type
str

Returns
The escaped field.

strip_delimiter(*field_or_path*, *delimiter='.'*)

Remove any delimiters from the start and end of a field or path.

Escaped delimiters are not removed.

Parameters

- **field_or_path** (str) – The field or path to strip delimiters from.
- **delimiter** (str) – The delimiter to strip.

Return type
str

Returns
The field or path with stripped delimiters.

prepend(*path*, *field*, *delimiter='.'*)

Prepend a field to a path.

Parameters

- **field** (str) – The field to prepend to a path.
- **path** (str) – The path to prepend the field to.
- **delimiter** (str) – The delimiter used between fields in the path.

Return type
str

Returns
The prepended path.

append(*path*, *field*, *delimiter='.'*)

Append a field to a path.

Parameters

- **field** (str) – The field to append to a path.
- **path** (str) – The path to append the field to.
- **delimiter** (str) – The delimiter used between fields in the path.

Return type
str

Returns
The appended path.

join_fields(*fields*, *delimiter='.'*)

Create a path from fields.

Parameters

- **fields** (Iterable[str]) – The fields to create a path from.
- **delimiter** (str) – The delimiter to use in the path between fields.

Return type
str

Returns
The path.

head(*path*, *delimiter='.'*)

Split the first field in a path from the path.

Parameters

- **path** (str) – The path to split.
- **delimiter** (str) – The delimiter used between fields in the path.

Return type
_head_return_value

Returns
A tuple of the first field in the path and the remaining path.

tail(*path*, *delimiter='.'*)

Split the last field in a path from the path.

Parameters

- **path** (str) – The path to split.
- **delimiter** (str) – The delimiter used between fields in the path.

Return type
_tail_return_value

Returns
A tuple of the last field in the path and the remaining path.

6.13 marsh.parse

Tools for parsing strings into elements and/or commands.

string(*source*, *terminal_chars=''*)

Parse a single string.

Reserved characters are allowed by being escaped (`) *or for the string (or part of the string) to be enclosed in quotes (* `"` or `'`).

Quotes are discarded before returning the result.

Parameters

- **source** (Union[str, MutableSequence[str]]) – The string source to parse
- **terminal_chars** (str) – Any characters that should terminate the parsing process (unless escaped or within a set of quotes). If none are given the entire input is consumed.

Return type
str

Returns
The parsed string.

terminal(*source*, *terminal_chars=''*)

Parse a terminal value.

None | int | float | string | bool

Parameters

- **source** (Union[str, MutableSequence[str]]) – The string source to parse
- **terminal_chars** (str) – Any characters that should terminate the parsing process (unless escaped or within a set of quotes). If none are given the entire input is consumed.

Return type
Union[None, int, float, bool, str]

Returns
The parsed value.

sequence(*source*, *terminal_chars=''*, *element_parser=<function element>*)

Parse a single sequence of elements.

See *element()* for defenition of an element.

Elements should be separated by `,`. The sequence should be enclosed in `()` or `[]`.

Parameters

- **source** (Union[str, MutableSequence[str]]) – The string source to parse
- **terminal_chars** (str) – Expected characters that can succeed the part of the input string that contains the sequence. If not given, an exception is raised if there is anything other than whitespace after the sequence.
- **element_parser** (_ParseFunction) – Each element of the sequence will be parsed using this parser.

Return type
Sequence

Returns
Sequence of elements.

mapping(*source*, *terminal_chars=''*, *key_parser=<function string>*, *value_parser=<function element>*)

Parse a single mapping of strings to elements.

See *element()* for defenition of an element.

Keys and values should be separated by `:` Key-value pairs should be separated by `,`. The mapping should be enclosed in `{}`.

Parameters

- **source** (Union[str, MutableSequence[str]]) – The string source to parse
- **terminal_chars** (str) – Expected characters that can succeed the part of the input string that contains the mapping. If not given, an exception is raised if there is anything other than whitespace after the mapping.
- **key_parser** (_ParseFunction) – Each key of the mapping will be parsed using this parser.

- **value_parser** (_ParseFunction) – Each value of the mapping will be parsed using this parser.

Return type
Mapping

Returns
The mapping of strings to elements.

element(*source*, *terminal_chars=''*)

Parse a single element.

Returns a mapping, sequence or terminal element.

See *mapping()*, *sequence()* and *terminal()* for more information parsing rules.

Parameters

- **source** (Union[str, MutableSequence[str]]) – The string source to parse
- **terminal_chars** (str) – Expected characters that can succeed the part of the input string that contains the element. If not given, an exception is raised if there is anything other than whitespace after the element.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The parsed element.

6.14 marsh.testing

Tools that can be used for testing purposes.

catch_cause(*base=<class 'marsh.errors.MarshError'>*, *cause=None*)

Assert that an error is thrown in this context.

Similar to `pytest.raises()` but allows for inspecting and finding a specific cause for the error raised by traversing the __cause__ attribute chain.

Parameters

- **base** (Type[Exception]) – The base exception class to catch. If not caught, the test fails.
- **cause** (Optional[Type[Exception]]) – If not None, at least one of the exceptions in the cause chain must be an instance of this class.

Return type
Iterator[None]

Returns
Context manager.

marshal_succeeds(*value*, *element*)

Assert that a value can be marshaled.

The element is also compared to the expected output for the marshal.

Parameters

- **value** (Any) – The value to marshal.

- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The expected marshal value.

Return type
Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]

Returns
The marshaled value if successful.

marshal_fails(*value*, *exception=None*)

Assert that a value can not be marshaled.

Requires that marshaling of the input value raises *marsh.errors.MarshalError*.

Parameters

- **value** (Any) – Value to marshal.
- **exception** (Optional[Type[Exception]]) – Optional cause that must exist in the exception chain if given.

Return type
None

unmarshal_succeeds(*type_*, *element*, *value*)

Assert that a value is unmarshaled correctly.

Parameters

- **type** – Unmarshal into this type.
- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input that should be unmarshaled.
- **value** (TypeVar(_T)) – The expected result.
- **type_** (*Any*) –

Return type
TypeVar(_T)

Returns
The unmarshaled result if successful.

unmarshal_fails(*type_*, *element*, *exception=None*)

Assert that a value can not be unmarshaled.

Requires that unmarshaling raises *marsh.errors.UnmarshalError*.

Parameters

- **type** – Attempt to unmarshal into this type.
- **element** (Union[None, int, float, bool, str, Sequence[Any], Mapping[str, Any]]) – The input that should be unmarshaled.
- **exception** (Optional[Type[Exception]]) – Optional cause that must exist in the exception chain if given.
- **type_** (*Any*) –

Return type
None

6.15 marsh.doc

Tools for producing documentation.

6.15.1 marsh.doc.terminal

format_terminal_value(*value*, *width=None*, *description=None*, *indent=2*, *description_indent=24*, *description_margin=4*)

Create a help message in the same style as `argparse.ArgumentParser` for a value and its description

Parameters

- **value** (`str`) – The string value.
- **width** (`Optional[int]`) – The width of the message.
- **description** (`Optional[str]`) – A text description for the value.
- **indent** (`int`) – The indentation of the value in number of blank spaces.
- **description_indent** (`int`) – The indentation of the description in number of blank spaces.
- **description_margin** (`int`) – The minimum number of blank spaces between the description and the value. The description will be started on the next line if the resulting margin would be smaller than this on the line containing the argument.

Return type
`str`

Returns
The help message.

6.15.2 marsh.doc.markdown

format_types(*source*, *heading=3*)

Return a markdown-formatted string containing registered schema types that have static documention.

A description below each type is included when available.

Parameters

- **source** (`Union[Type[Schema], SchemaRegistry, Iterable[Type[Schema]]]`) – The source of the schemas.
- **heading** (`int`) – The heading number used for each types.

Return type
`str`

Returns
The formatted markdown.

6.15.3 marsh.doc.restructuredtext

format_table(*headers*, *rows*, *title=None*, *widths=None*)

Create a RST list-table.

Any newlines in the headers or rows are replaced with whitespace.

Parameters

- **headers** (Sequence) – The header names.
- **rows** (Iterable[Sequence]) – Iterable of rows, each row containing the same number of values as there are headers.
- **title** (Optional[str]) – A title for the table.
- **widths** (Optional[Sequence]) – The width of columns. If given, must contain the same number of values as there are headers.

Return type
str

Returns
The table in RST format.

format_types(*source*, *section='^'*)

Return a RST-formatted string containing registered schema types that have static documention.

A description below each type is included when available.

Parameters

- **source** (Union[Type[*Schema*], *SchemaRegistry*, Iterable[Type[*Schema*]]]) – The source of the schemas.
- **section** (Literal['#', '*', '=', '-', '^', '"']) – The section character.

Return type
str

Returns
The formatted RST.

format_types_table(*source*, *title=None*)

Return a RST-formatted table containing registered schema types that have static documention.

A description for each type is included when available.

Parameters

- **source** (Union[Type[*Schema*], *SchemaRegistry*, Iterable[Type[*Schema*]]]) – The source of the schemas.
- **title** (*Optional[str]*) –

Return type
str

Returns
The formatted RST table.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W