jsonargparse Documentation

Mauricio Villegas

Dec 30, 2020

CONTENTS

https://omni-us.github.io/jsonargparse/

This package is an extension to python's argparse which simplifies parsing of configuration options from command line arguments, json configuration files (yaml or jsonnet supersets), environment variables and hard-coded defaults.

The aim is similar to other projects such as configargparse, yconf, confuse, typer, OmegaConf, Fire and click. The obvious question is, why yet another package similar to many already existing ones? The answer is simply that none of the existing projects had the exact features we wanted and after analyzing the alternatives it seemed simpler to start a new project.

CHAPTER

ONE

FEATURES

- Parsers are configured just like with python's argparse, thus it has a gentle learning curve.
- Not exclusively intended for parsing command line arguments. The main focus is parsing configuration files and not necessarily from a command line tool.
- Support for two popular supersets of json: yaml and jsonnet.
- Support for nested namespaces which makes it possible to parse config files with non-flat hierarchies.
- Three mechanisms to define parsers in a modular way: arguments from classes, methods and functions; sub-commands and parsers as arguments.
- Parsing of relative paths within config files and path lists.
- Several convenient action classes and types to ease common parsing use cases (paths, comparison operators, json schemas, enums . . .).
- Support for command line tab argument completion using argcomplete.
- Configuration values are overridden based on the following precedence.
  - **Parsing command line:** command line arguments (might include config file) > environment variables > default config file > defaults.
  - **Parsing files:** config file > environment variables > default config file > defaults.
  - **Parsing environment:** environment variables > default config file > defaults.

CHAPTER

TWO

INSTALLATION

You can install using pip as:

```
pip install jsonargparse
```

By default the only dependency that jsonargparse installs is PyYAML. However, jsonargparse has several optional features that can be enabled by specifying any of the following extras requires: `signatures`, `jsonschema`, `jsonnet`, `urls`, `argcomplete` and `reconplogger`. There is also the `all` extras require that can be used to enable all optional features. Installing jsonargparse with extras require is as follows:

```
pip install "jsonargparse[signatures]"    # Enable only signatures feature
pip install "jsonargparse[all]"           # Enable all optional features
```

CHAPTER

THREE

BASIC USAGE

There are multiple ways of using jsonargparse. The most simple way which requires to write the least amount of code is by using the `CLI()` function, for example:

```
from jsonargparse import CLI

def command(
    name: str,
    prize: int = 100
):
    """
    Args:
        name: Name of winner.
        prize: Amount won.
    """
    print(f'{name} won {prize}€!')

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

Then in a shell you could run:

```
$ python example.py Lucky --prize=1000
Lucky won 1000€!
```

`CLI()` without arguments searches for functions and classes defined in the same module and in the local context where `CLI()` is called. Giving a single or a list functions/classes as first argument to `CLI()` skips the automatic search and only includes what is given.

When `CLI()` receives a single class, the first arguments are used to instantiate the class, then a class method name must be given (see *Sub-commands*) and the remaining arguments are used to run the class method. An example would be:

```
from random import randint
from jsonargparse import CLI

class Main:
    def __init__(
        self,
        max_prize: int = 100
    ):
        """
        Args:
            max_prize: Maximum prize that can be awarded.
        """
```

(continues on next page)

(continued from previous page)

```
        self.max_prize = max_prize

    def person(
        self,
        name: str
    ):
        """
        Args:
            name: Name of winner.
        """
        return f'{name} won {randint(0, self.max_prize)}€!'

if __name__ == '__main__':
    print(CLI(Main))
```

Then in a shell you could run:

```
$ python example.py --max_prize=1000 person Lucky
Lucky won 632€!
```

If more than one function is given to `CLI()`, then any of them can be executed via *Sub-commands* similar to the single class example above, e.g. `python example.py function [arguments]` where `function` is the name of the function to execute.

If multiple classes or a mixture of functions and classes is given to `CLI()`, to execute a method of a class, two levels of *Sub-commands* are required. The first sub-command would be name of the class and the second the name of the method, i.e. `python example.py class [init_arguments] method [arguments]`. For more details about the automatic adding of arguments from classes and functions and the use of configuration files refer to section *Classes, methods and functions*.

This simple way of usage is similar and inspired by Fire. However, there are fundamental differences. First, the purpose is not allowing to call any python object from the command line. It is only intended for running functions and classes specifically written for this purpose. Second, the arguments are required to have type hints, and the values will be validated according to these. Third, the return values of the functions are not automatically printed. `CLI()` returns its value and it is up to the developer to decide what to do with it. Finally, jsonargparse has many features designed to help in creating convenient argument parsers such as: *Nested namespaces*, *Configuration files*, additional type hints (*Parsing paths*, *Restricted numbers*, *Restricted strings*) and much more.

The next section explains how to create an argument parser in a very low level argparse-style. However, as parsers get more complex, being able to define them in a modular way becomes important. Three mechanisms are available to define parsers in a modular way, see respective sections *Classes, methods and functions*, *Sub-commands* and *Parsers as arguments*.

CHAPTER

FOUR

PARSERS

An argument parser is created just like it is done with python's argparse. You import the module, create a parser object and then add arguments to it. A simple example would be:

```
from jsonargparse import ArgumentParser
parser = ArgumentParser(
    prog='app',
    description='Description for my app.')

parser.add_argument('--opt1',
    type=int,
    default=0,
    help='Help for option 1.')

parser.add_argument('--opt2',
    type=float,
    default=1.0,
    help='Help for option 2.')
```

After creating the parser, you can use it to parse command line arguments with the *ArgumentParser.parse_args()* function, after which you get an object with the parsed values or defaults available as attributes. For illustrative purposes giving to `parse_args()` a list of arguments (instead of automatically getting them from the command line arguments), with the parser from above you would observe:

```
>>> cfg = parser.parse_args(['--opt2', '2.3'])
>>> cfg.opt1, type(cfg.opt1)
(0, <class 'int'>)
>>> cfg.opt2, type(cfg.opt2)
(2.3, <class 'float'>)
```

If the parsing fails the standard behavior is that the usage is printed and the program is terminated. Alternatively you can initialize the parser with `error_handler=None` in which case a *ParserError* is raised.

CHAPTER

FIVE

NESTED NAMESPACES

A difference with respect to the basic argparse is that it by using dot notation in the argument names, you can define a hierarchy of nested namespaces. So for example you could do the following:

```
>>> parser = ArgumentParser(prog='app')
>>> parser.add_argument('--lev1.opt1', default='from default 1')
>>> parser.add_argument('--lev1.opt2', default='from default 2')
>>> cfg = parser.get_defaults()
>>> cfg.lev1.opt1
'from default 2'
>>> cfg.lev1.opt2
'from default 2'
```

CHAPTER

SIX

ENVIRONMENT VARIABLES

The jsonargparse parsers can also get values from environment variables. The parser checks existing environment variables whose name is of the form `[PREFIX_][LEV__]*OPT`, that is all in upper case, first a prefix (set by `env_prefix`, or if unset the `prog` without extension) followed by underscore and then the argument name replacing dots with two underscores. Using the parser from the *Nested namespaces* section above, in your shell you would set the environment variables as:

```
export APP_LEV1__OPT1='from env 1'
export APP_LEV1__OPT2='from env 2'
```

Then in python the parser would use these variables, unless overridden by the command line arguments, that is:

```
>>> parser = ArgumentParser(env_prefix='APP', default_env=True)
>>> parser.add_argument('--lev1.opt1', default='from default 1')
>>> parser.add_argument('--lev1.opt2', default='from default 2')
>>> cfg = parser.parse_args(['--lev1.opt1', 'from arg 1'])
>>> cfg.lev1.opt1
'from arg 1'
>>> cfg.lev1.opt2
'from env 2'
```

Note that when creating the parser, `default_env=True` was given as argument. By default *`ArgumentParser.parse_args()`* does not check environment variables, so it has to be enabled explicitly.

There is also the *`ArgumentParser.parse_env()`* function to only parse environment variables, which might be useful for some use cases in which there is no command line call involved.

If a parser includes an *`ActionConfigFile`* argument, then the environment variable for this config file will be checked before all the other environment variables.

CHAPTER

SEVEN

CONFIGURATION FILES

An important feature of jsonargparse is the parsing of yaml/json files. The dot notation hierarchy of the arguments (see *Nested namespaces*) are used for the expected structure in the config files.

The *ArgumentParser* class accepts a `default_config_files` argument that can be given to specify patterns to search for configuration files. Only the first matched config file is parsed.

When parsing command line arguments, it is possible to add a configuration file path argument. The config file would be read and parsed in the specific position among the command line arguments, so the arguments after would override the values from the configuration file. The config argument can be given multiple times, each overriding the values of the previous. Again using the parser from the *Nested namespaces* section above, for example we could have the following config file in yaml format:

```
# File: example.yaml
lev1:
  opt1: from yaml 1
  opt2: from yaml 2
```

Then in python adding a yaml file argument and parsing some example arguments, the following would be observed:

```
>>> from jsonargparse import ArgumentParser, ActionConfigFile
>>> parser = ArgumentParser()
>>> parser.add_argument('--lev1.opt1', default='from default 1')
>>> parser.add_argument('--lev1.opt2', default='from default 2')
>>> parser.add_argument('--cfg', action=ActionConfigFile)
>>> cfg = parser.parse_args(['--lev1.opt1', 'from arg 1',
                            '--cfg', 'example.yaml',
                            '--lev1.opt2', 'from arg 2'])
>>> cfg.lev1.opt1
'from yaml 1'
>>> cfg.lev1.opt2
'from arg 2'
```

Instead of providing a path to a configuration file, a string with the configuration content can also be provided.

```
>>> cfg = parser.parse_args(['--cfg', '{"lev1":{"opt1":"from string 1"}}'])
>>> cfg.lev1.opt1
'from string 1'
```

All parsers include a `--print_config` option. This is useful particularly for command line tools with a large set of options to create an initial config file including all default values. This option by default all entries, including the ones with `null` values. If the argument is given as `--print_config=skip_null`, then the entries with `null` values will not be included.

The config file can also be provided as an environment variable as explained in section *Environment variables*. The

configuration file environment variable is the first one to be parsed. So any other argument provided through environment variables would override the config file one.

A configuration file or string can also be parsed without parsing command line arguments. The functions for this are *ArgumentParser.parse_path()* and *ArgumentParser.parse_string()* to parse a config file or a config contained in a string respectively.

CHAPTER

EIGHT

CLASSES, METHODS AND FUNCTIONS

It is good practice to write python code in which parameters have type hints and are described in the docstrings. To make this well written code configurable, it wouldn't make sense to duplicate information of types and parameter descriptions. To avoid this duplication, jsonargparse includes methods to automatically add their arguments: *SignatureArguments.add_class_arguments()*, *SignatureArguments.add_method_arguments()* and *SignatureArguments.add_function_arguments()*.

Take for example a class with its init and a method with docstrings as follows:

```
from typing import Dict, Union, List

class MyClass(MyBaseClass):
    def __init__(self, items: Dict[str, Union[int, List[int]]], **kwargs):
        """Initializer for MyClass.

        Args:
            items: Description for items.
        """
        pass

    def mymethod(self, value: float, flag: bool = False):
        """Description for mymethod.

        Args:
            value: Description for value.
            flag: Description for flag.
        """
        pass
```

Both `MyClass` and `mymethod` can easily be made configurable, the class initialized and the method executed as follows:

```
from jsonargparse import ArgumentParser, namespace_to_dict

parser = ArgumentParser()
parser.add_class_arguments(MyClass, 'myclass.init')
parser.add_method_arguments(MyClass, 'mymethod', 'myclass.method')

cfg = parser.parse_args()
myclass = MyClass(**namespace_to_dict(cfg.myclass.init))
myclass.mymethod(**namespace_to_dict(cfg.myclass.method))
```

The `add_class_arguments()` call adds to the *myclass.init* key the `items` argument with description as in the docstring, it is set as required since it does not have a default value, and when parsed it is validated according to its type hint, i.e., a dict with values ints or list of ints. Also since the init has the `**kwargs` argument, the keyword

arguments from `MyBaseClass` are also added to the parser. Similarly the `add_method_arguments()` call adds to the *myclass.method* key the arguments `value` as a required float and `flag` as an optional boolean with default value false.

Instead of using `namespace_to_dict()` to convert the namespaces to a dictionary, the *`ArgumentParser`* object can be instantiated with `parse_as_dict=True` to get directly a dictionary from the parsing methods.

When parsing from a configuration file (see *Configuration files*) all the values can be given in a single config file. However, for convenience it is also possible that the values for each of the groups created by the calls to the add signature methods can be parsed from independent files. This means that for the example above there could be one general config file with contents:

```
myclass:
  init: myclass.yaml
  method: mymethod.yaml
```

Then the files `myclass.yaml` and `mymethod.yaml` would only include the settings for each of the instantiation of the class and the call to the method respectively.

A wide range of type hints are supported. For exact details go to section *Type hints*. Some notes about the support for automatic adding of arguments are:

- All positional arguments must have a type, otherwise the add arguments functions raise an exception.
- Keyword arguments are ignored if they don't have at least one type that is supported.
- Recursive adding of arguments from base classes only considers the presence of `*args` and `**kwargs`. It does not check the code to identify if `super().__init__` is called or with which arguments.

Since keyword arguments with unsupported types are ignored, during development it might be desired to know which arguments are ignored and the specific reason. This can be done by initializing *`ArgumentParser`* with `logger={'level': 'DEBUG'}`. For more details about logging go to section *Logging*.

For all features described above to work, two optional packages are required: jsonschema to support validation of complex type hints and docstring-parser to get the argument descriptions from the docstrings. Both these packages are included when jsonargparse is installed using the `signatures` extras require as explained in section *Installation*.

CHAPTER

NINE

TYPE HINTS

As explained in section *Classes, methods and functions* type hints are required to automatically add arguments from signatures to a parser. Additional to this feature, a type hint can also be used independently when adding a single argument to the parser. For example, an argument that can be `None` or a float in the range `(0, 1)` or a positive int could be added using type hints as follows:

```
from typing import Optional, Union
from jsonargparse.typing import PositiveInt, OpenUnitInterval
parser.add_argument('--op', type=Optional[Union[PositiveInt, OpenUnitInterval]])
```

The support of type hints is designed to not require developers to change their types or default values. In other words, the idea is to support type hints whatever they may be, as opposed to requiring to be changed some jsonargparse specific types for the parsers to work. The types included in `jsonargparse.typing` are completely generic and could even be useful independent of the argument parsers.

A wide range of type hints are supported and with arbitrary complexity/nesting. Some notes about this support are:

- Nested types are supported as long as at least one child type is supported.
- Fully supported types are: `str`, `bool`, `int`, `float`, `List`, `Iterable`, `Sequence`, `Any`, `Union`, `Optional`, `Enum`, restricted types as explained in sections *Restricted numbers* and *Restricted strings* and paths and URLs as explained in sections *Parsing paths* and *Parsing URLs*.
- `Dict` is supported but only with `str` or `int` keys.
- `Tuple` and `Set` are supported even though they can't be represented in json distinguishable from a list. Each `Tuple` element position can have its own type and will be validated as such. In command line arguments, config files and environment variables, tuples and sets are represented as a list.
- To set a value to `None` it is required to use `null` since this is how json/yaml requires it. To avoid confusion in the help, `NoneType` is displayed as `null`. For example `Optional[str] = None` would be shown as `type: Union[str, null], default: null`.

CHAPTER

TEN

CLASSES AS TYPE

Using an arbitrary class as a type is also possible, though it requires a bit explanation. In the config file or environment variable or command line argument, a class is represented by a dictionary with a `class_path` entry indicating the dot notation expression to import the class, and optionally some `init_args` that would be used to instantiate it. When parsing it will be checked that the class can be imported, that it is a subclass of the type and that `init_args` values correspond to valid arguments to instantiate. After parsing, the config object will include the `class_path` and `init_args` entries. To get a config object with all subclasses instantiated, the *`ArgumentParser.instantiate_subclasses()`* method is used.

A simple example would be having some config file `config.yaml` as:

```
calendar:
  class_path: calendar.Calendar
  init_args:
    firstweekday: 1
```

Then in python:

```
>>> from calendar import Calendar
>>> parser = ArgumentParser(parse_as_dict=True)
>>> parser.add_argument('--calendar', type=Calendar)
>>> cfg = parser.parse_path('config.yaml')
>>> cfg['calendar']
{'class_path': 'calendar.Calendar', 'init_args': {'firstweekday': 1}}
>>> cfg = parser.instantiate_subclasses(cfg)
>>> cfg['calendar']
<calendar.Calendar object at 0x7ffa559aa940>
```

In the example the `class_path` points to the same class used for the type. But a subclass of `Calendar` with an extended list of init parameters would also work.

CHAPTER

ELEVEN

SUB-COMMANDS

A way to define parsers in a modular way is what in argparse is known as sub-commands. However, to promote modularity, in jsonargparse sub-commands work a bit different than in argparse. To add sub-commands to a parser, the *ArgumentParser.add_subcommands()* method is used. Then an existing parser is added as a sub-command using `add_subcommand()`. In a parsed config object the sub-command will be stored in the `subcommand` entry (or whatever `dest` was set to), and the values of the sub-command will be in an entry with the same name as the respective sub-command. An example of defining a parser with sub-commands is the following:

```
from jsonargparse import ArgumentParser
...
parser_subcomm1 = ArgumentParser()
parser_subcomm1.add_argument('--op1')
...
parser_subcomm2 = ArgumentParser()
parser_subcomm2.add_argument('--op2')
...
parser = ArgumentParser(prog='app')
parser.add_argument('--op0')
subcommands = parser.add_subcommands()
subcommands.add_subcommand('subcomm1', parser_subcomm1)
subcommands.add_subcommand('subcomm2', parser_subcomm2)
```

Then some examples of parsing are the following:

```
>>> parser.parse_args(['subcomm1', '--op1', 'val1'])
Namespace(op0=None, subcomm1=Namespace(op1='val1'), subcommand='subcomm1')
>>> parser.parse_args(['--op0', 'val0', 'subcomm2', '--op2', 'val2'])
Namespace(op0='val0', subcomm2=Namespace(op2='val2'), subcommand='subcomm2')
```

Parsing config files with *ArgumentParser.parse_path()* or *ArgumentParser.parse_string()* is also possible. Though there can only be values for one of the sub-commands. The config file is not required to specify a value for `subcommand`. For the example parser above a valid yaml would be:

```
# File: example.yaml
op0: val0
subcomm1:
  op1: val1
```

Parsing of environment variables works similar to *ActionParser*. For the example parser above, all environment variables for `subcomm1` would have as prefix `APP_SUBCOMM1_` and likewise for `subcomm2` as prefix `APP_SUBCOMM2_`. The sub-command to use could be chosen by setting environment variable `APP_SUBCOMMAND`.

It is possible to have multiple levels of sub-commands. With multiple levels there is one basic requirement: the sub-commands must be added in the order of the levels. This is, first call `add_subcommands()` and `add_subcommand()` for the first level. Only after do the same for the second level, and so on.

CHAPTER

TWELVE

JSON SCHEMAS

The *ActionJsonSchema* class is provided to allow parsing and validation of values using a json schema. This class requires the jsonschema python package. Though note that jsonschema is not a requirement of the minimal jsonargparse install. To enable this functionality install with the `jsonschema` extras require as explained in section *Installation*.

Check out the jsonschema documentation to learn how to write a schema. The current version of jsonargparse uses Draft7Validator. Parsing an argument using a json schema is done like in the following example:

```
>>> schema = {
...     "type" : "object",
...     "properties" : {
...         "price" : {"type" : "number"},
...         "name" : {"type" : "string"},
...     },
... }

>>> from jsonargparse import ActionJsonSchema
>>> parser.add_argument('--op', action=ActionJsonSchema(schema=schema))

>>> parser.parse_args(['--op', '{"price": 1.5, "name": "cookie"}'])
Namespace(op=Namespace(name='cookie', price=1.5))
```

Instead of giving a json string as argument value, it is also possible to provide a path to a json/yaml file, which would be loaded and validated against the schema. If the schema defines default values, these will be used by the parser to initialize the config values that are not specified. When adding an argument with the *ActionJsonSchema* action, you can use "%s" in the `help` string so that in that position the schema will be printed.

CHAPTER

THIRTEEN

JSONNET FILES

The Jsonnet support requires jsonschema and jsonnet python packages which are not included with minimal jsonargparse install. To enable this functionality install jsonargparse with the `jsonnet` extras require as explained in section *Installation*.

By default an *ArgumentParser* parses configuration files as yaml. However, if instantiated giving as argument `parser_mode='jsonnet'`, then `parse_args()`, `parse_path()` and `parse_string()` will expect config files to be in jsonnet format instead. Example:

```
>>> from jsonargparse import ArgumentParser, ActionConfigFile
>>> parser = ArgumentParser(parser_mode='jsonnet')
>>> parser.add_argument('--cfg', action=ActionConfigFile)
>>> cfg = parser.parse_args(['--cfg', 'example.jsonnet'])
```

Jsonnet files are commonly parametrized, thus requiring external variables for parsing. For these cases, instead of changing the parser mode away from yaml, the *ActionJsonnet* class can be used. This action allows to define an argument which would be a jsonnet string or a path to a jsonnet file. Moreover, another argument can be specified as the source for any external variables required, which would be either a path to or a string containing a json dictionary of variables. Its use would be as follows:

```
from jsonargparse import ArgumentParser, ActionJsonnet, ActionJsonnetExtVars
parser = ArgumentParser()
parser.add_argument('--in_ext_vars',
    action=ActionJsonnetExtVars())
parser.add_argument('--in_jsonnet',
    action=ActionJsonnet(ext_vars='in_ext_vars'))
```

For example, if a jsonnet file required some external variable `param`, then the jsonnet and the external variable could be given as:

```
cfg = parser.parse_args(['--in_ext_vars', '{"param": 123}',
                         '--in_jsonnet', 'path_to_jsonnet'])
```

Note that the external variables argument must be provided before the jsonnet path so that this dictionary already exists when parsing the jsonnet.

The *ActionJsonnet* class also accepts as argument a json schema, in which case the jsonnet would be validated against this schema right after parsing.

CHAPTER

FOURTEEN

PARSING PATHS

For some use cases it is necessary to parse file paths, checking its existence and access permissions, but not necessarily opening the file. Moreover, a file path could be included in a config file as relative with respect to the config file's location. After parsing it should be easy to access the parsed file path without having to consider the location of the config file. To help in these situations jsonargparse includes a type generator *path_type()*, some predefined types (e.g. *Path_fr*) and the *ActionPath* and *ActionPathList* classes.

For example suppose you have a directory with a configuration file `app/config.yaml` and some data `app/data/info.db`. The contents of the yaml file is the following:

```
# File: config.yaml
databases:
  info: data/info.db
```

To create a parser that checks that the value of `databases.info` is a file that exists and is readable, the following could be done:

```
from jsonargparse import ArgumentParser
from jsonargparse.typing import Path_fr
parser = ArgumentParser()
parser.add_argument('--databases.info', type=Path_fr)
cfg = parser.parse_path('app/config.yaml')
```

The `fr` in the type are flags stand for file and readable. After parsing the value of `databases.info` will be an instance of the *Path* class that allows to get both the original relative path as included in the yaml file, or the corresponding absolute path:

```
>>> str(cfg.databases.info)
'data/info.db'
>>> cfg.databases.info()
'/YOUR_CWD/app/data/info.db'
```

Likewise directories can be parsed for example using as type the *Path_dw* type, would require a directory to exist and be writeable. New path types can be created using the *path_type()* function. For example to create a type for files that must exist and be both readable and writeable, the command would be `Path_frw = path_type('frw')`. If the file `app/config.yaml` is not writeable, then usig the type to cast `Path_frw('app/config.yaml')` would raise a *TypeError: File is not writeable* exception. For more information of all the mode flags supported, refer to the documentation of the *Path* class.

The content of a file that a *Path* instance references can be read by using the *Path.get_content()* method. For the previous example would be `info_db = cfg.databases.info.get_content()`.

Adding arguments with path types is equivalent to adding using for example `action=ActionPath(mode='fr')` instead of a `type=Path_fr`. However, the type option is preferred.

An argument with a path type can be given `nargs='+'` to parse multiple paths. But it might also be wanted to parse a list of paths found in a plain text file or from stdin. For this the *ActionPathList* is used and as argument either the path to a file listing the paths is given or the special `'-'` string for reading the list from stdin. For for example:

```
from jsonargparse import ActionPathList
parser.add_argument('--list', action=ActionPathList(mode='fr'))
cfg = parser.parse_args(['--list', 'paths.lst')  # Text file with paths
cfg = parser.parse_args(['--list', '-')          # List from stdin
```

If `nargs='+'` is given to `add_argument` then a single list is generated including all paths in all lists is provided.

Note: the *Path* class is currently not fully supported in windows.

CHAPTER

FIFTEEN

PARSING URLS

The *path_type()* function also supports URLs which after parsing the *Path.get_content()* method can be used to perform a GET request to the corresponding URL and retrieve its content. For this to work the *validators* and *requests* python packages are required which will be installed along with jsonargparse if installed with the `urls` extras require as explained in section *Installation*.

The `'u'` flag is used to parse URLs. For example if it is desired that an argument can be either a readable file or URL, the type would be created as `Path_fur = path_type('fur')`. If the value appears to be a URL according to `validators.url.url()` then a HEAD request would be triggered to check if it is accessible, and if so, the parsing succeeds. To get the content of the parsed path, without needing to care if it is a local file or a URL, the *Path.get_content()* can be used.

If after importing jsonargparse you run `jsonargparse.set_url_support(True)`, the following functions and classes will also support loading from URLs: *ArgumentParser.parse_path()*, *ArgumentParser.get_defaults()* (`default_config_files` argument), *ActionConfigFile*, *ActionJsonSchema*, *ActionJsonnet* and *ActionParser*. This means for example that a tool that can receive a configuration file via *ActionConfigFile* is able to get the config file from a URL, that is something like the following would work:

```
$ my_tool.py --cfg http://example.com/config.yaml
```

CHAPTER

SIXTEEN

RESTRICTED NUMBERS

It is quite common that when parsing a number, its range should be limited. To ease these cases the module `jsonargparse.typing` includes some predefined types and a function *restricted_number_type()* to define new types. The predefined types are: *PositiveInt*, *NonNegativeInt*, *PositiveFloat*, *NonNegativeFloat*, *ClosedUnitInterval* and *OpenUnitInterval*. Examples of usage are:

```
from jsonargparse.typing import PositiveInt, PositiveFloat, restricted_number_type
# float larger than zero
parser.add_argument('--op1', type=PositiveFloat)
# between 0 and 10
from_0_to_10 = restricted_number_type('from_0_to_10', int, [('>=', 0), ('<=', 10)])
parser.add_argument('--op2', type=from_0_to_10)
# either int larger than zero or 'off' string
def int_or_off(x): return x if x == 'off' else PositiveInt(x)
parser.add_argument('--op3', type=int_or_off))
```

CHAPTER

SEVENTEEN

RESTRICTED STRINGS

Similar to the restricted numbers, there is a function to create string types that are restricted to match a given regular expression: *restricted_string_type()*. A predefined type is *Email* which is restricted so that it follows the normal email pattern. For example to add an argument required to be exactly four uppercase letters:

```
from jsonargparse.typing import Email, restricted_string_type
CodeType = restricted_string_type('CodeType', '^[A-Z]{4}$')
parser.add_argument('--code', type=CodeType)
parser.add_argument('--email', type=Email)
```

CHAPTER

EIGHTEEN

ENUM ARGUMENTS

Another case of restricted values is string choices. In addition to the common `choices` given as a list of strings, it is also possible to provide as type an `Enum` class. This has the added benefit that strings are mapped to some desired values. For example:

```
>>> class MyEnum(enum.Enum):
...     choice1 = -1
...     choice2 = 0
...     choice3 = 1
>>> parser.add_argument('--op', type=MyEnum)
>>> parser.parse_args(['--op=choice1'])
Namespace(op=<MyEnum.choice1: -1>)
```

CHAPTER

NINETEEN

BOOLEAN ARGUMENTS

Parsing boolean arguments is very common, however, the original argparse only has a limited support for them, via `store_true` and `store_false`. Futhermore unexperienced users might mistakenly use `type=bool` which would not provide the intended behavior.

With jsonargparse adding an argument with `type=bool` the intended action is implemented. If given as values `{'yes', 'true'}` or `{'no', 'false'}` the corresponding parsed values would be `True` or `False`. For example:

```
>>> parser.add_argument('--op1', type=bool, default=False)
>>> parser.add_argument('--op2', type=bool, default=True)
>>> parser.parse_args(['--op1', 'yes', '--op2', 'false'])
Namespace(op1=True, op2=False)
```

To use `type=bool` jsonargparse needs to be installed with the `jsonschema` extras require as explained in section *Installation*.

Sometimes it is also useful to define two paired options, one to set `True` and the other to set `False`. The *ActionYesNo* class makes this straightforward. A couple of examples would be:

```
from jsonargparse import ActionYesNo
# --opt1 for true and --no_opt1 for false.
parser.add_argument('--op1', action=ActionYesNo)
# --with-opt2 for true and --without-opt2 for false.
parser.add_argument('--with-op2', action=ActionYesNo(yes_prefix='with-', no_prefix=
↪'without-'))
```

If the *ActionYesNo* class is used in conjunction with `nargs='?'` the options can also be set by giving as value any of `{'true', 'yes', 'false', 'no'}`.

CHAPTER

TWENTY

PARSERS AS ARGUMENTS

Sometimes it is useful to take an already existing parser that is required standalone in some part of the code, and reuse it to parse an inner node of another more complex parser. For these cases an argument can be defined using the *ActionParser* class. An example of how to use this class is the following:

```
from jsonargparse import ArgumentParser, ActionParser
inner_parser = ArgumentParser(prog='app1')
inner_parser.add_argument('--op1')
...
outer_parser = ArgumentParser(prog='app2')
outer_parser.add_argument('--inner.node',
    action=ActionParser(parser=inner_parser))
```

When using the *ActionParser* class, the value of the node in a config file can be either the complex node itself, or the path to a file which will be loaded and parsed with the corresponding inner parser. Naturally using *ActionConfigFile* to parse a complete config file will parse the inner nodes correctly.

From the command line the help of the inner parsers can be shown by calling the tool with a prefixed help command, that is, for the example above it would be `--inner.node.help`.

Regarding environment variables, the prefix of the outer parser will be used to populate the leaf nodes of the inner parser. In the example above, if `inner_parser` is used to parse environment variables, then as normal `APP1_OP1` would be checked to populate option `op1`. But if `outer_parser` is used, then `APP2_INNER__NODE__OP1` would be checked to populate `inner.node.op1`.

An important detail to note is that the parsers that are given to *ActionParser* are internally modified. Therefore, to use the parser both as standalone and an inner node, it is necessary to implement a function that instantiates the parser. This function would be used in one place to get an instance of the parser for standalone parsing, and in some other place use the function to provide an instance of the parser to *ActionParser*.

CHAPTER

TWENTYONE

TAB COMPLETION

Tab completion is available for jsonargparse parsers by using the argcomplete package. There is no need to implement completer functions or to call `argcomplete.autocomplete()` since this is done automatically by *`ArgumentParser.parse_args()`*. The only requirement to enable tab completion is to install argcomplete either directly or by installing jsonargparse with the `argcomplete` extras require as explained in section *Installation*. Then the tab completion can be enabled globally for all argcomplete compatible tools or for each individual tool. A simple `example.py` tool would be:

```
#!/usr/bin/env python3

from typing import Optional
from jsonargparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument('--bool', type=Optional[bool])

parser.parse_args()
```

Then in a bash shell you can add the executable bit to the script, activate tab completion and use it as follows:

```
$ chmod +x example.py
$ eval "$(register-python-argcomplete example.py)"

$ ./example.py --bool <TAB><TAB>
false  null   true
$ ./example.py --bool f<TAB>
$ ./example.py --bool false
```

CHAPTER

TWENTYTWO

LOGGING

The parsers from jsonargparse log some basic events, though by default this is disabled. To enable it the `logger` argument should be set when creating an *`ArgumentParser`* object. The intended use is to give as value an already existing logger object which is used for the whole application. Though for convenience to enable a default logger the `logger` argument can also receive `True` or a string which sets the name of the logger or a dictionary that can include the name and the level, e.g. `{"name": "myapp", "level": "ERROR"}`. If reconplogger is installed, setting `logger` to `True` or a dictionary without specifying a name, then the reconplogger is used.

CHAPTER

TWENTYTHREE

CONTRIBUTING

Contributions to the jsonargparse package are very welcome, be it just to create issues for reporting bugs and proposing enhancements, or more directly by creating pull requests.

If you intend to work with the source code, note that this project does not include any `requirements.txt` file. This is by intention. To make it very clear what are the requirements for different use cases, all the requirements of the project are stored in the file `setup.cfg`. The basic runtime requirements are defined in section `[options]` in the `install_requires` entry. All extras requires for optional features listed in *Installation* are stored in section `[options.extras_require]`. Also there are `test`, `test_no_urls`, `dev` and `doc` entries in the same `[options.extras_require]` section which lists requirements for testing, development and documentation building.

The recommended way to work with the source code is the following. First clone the repository, then create a virtual environment, activate it and finally install the development requirements. More precisely the steps are:

```
git clone https://github.com/omni-us/jsonargparse.git
cd jsonargparse
virtualenv -p python3 venv
. venv/bin/activate
```

The crucial step is installing the requirements which would be done by running:

```
pip install -e ".[dev,all]"
```

Running the unit tests can be done either using using tox or the `setup.py` script. The unit tests are also installed with the package, thus can be used to in a production system.

```
tox  # Run tests using tox
./setup.py test_coverage  # Run tests and generate coverage report
python3 -m jsonargparse_tests  # Run tests for installed package
```

CHAPTER

TWENTYFOUR

API REFERENCE

24.1 jsonargparse.cli

Simple creation of command line interfaces.

Functions:

CLI([components, args, config_help, ...])	Function for simple creation of command line interfaces.

Classes:

ActionConfigFile(**kwargs)	Action to indicate that an argument is a configuration file or a configuration string.
ArgumentParser(*args[, env_prefix, ...])	Parser for command line, yaml/jsonnet files and environment variables.

jsonargparse.cli.**CLI**(*components=None*, *args=None*, *config_help='Path to a configuration file in json or yaml format.'*, *as_positional=True*, ***kwargs*)

Function for simple creation of command line interfaces.

Creates an argument parser from one or more functions/classes, parses arguments and runs one of the functions or class methods depending on what was parsed. If the components argument is not given, then the components will be all the locals in the context and defined in the same module as from where CLI is called.

Parameters

- **components** (Union[Callable, Type, List[Union[Callable, Type]], None]) – One or more functions/classes to include in the command line interface.
- **args** (Optional[List[str]]) – List of arguments to parse or None to use sys.argv.
- **config_help** (str) – Help string for config file option in help.
- **as_positional** (bool) – Whether to add required parameters as positional arguments.
- ****kwargs** – Used to instantiate *ArgumentParser*.

Returns The value returned by the executed function or class method.

24.2 jsonargparse.core

Classes:

ArgumentParser(*args[, env_prefix, ...])	Parser for command line, yaml/jsonnet files and environment variables.
Action(option_strings, dest[, nargs, const, ...])	Information about how to convert command line strings to Python objects.
ActionConfigFile(**kwargs)	Action to indicate that an argument is a configuration file or a configuration string.
ActionEnum(**kwargs)	An action based on an Enum that maps to-from strings and enum values.
ActionJsonSchema(**kwargs)	Action to parse option as json validated by a jsonschema.
ActionJsonnet(**kwargs)	Action to parse a jsonnet, optionally validating against a jsonschema.
ActionJsonnetExtVars(**kwargs)	Action to add argument to provide ext_vars for jsonnet parsing.
ActionParser(**kwargs)	Action to parse option with a given parser optionally loading from file if string value.
ActionPath(**kwargs)	Action to check and store a path.
ActionYesNo(**kwargs)	Paired options –{yes_prefix}opt, –{no_prefix}opt to set True or False respectively.
DefaultHelpFormatter(prog[, ...])	Help message formatter that includes types, default values and env var names.
Enum(value)	Generic enumeration.
FilesCompleterMethod()	Completer method for Action classes that should complete files.
LoggerProperty()	Class designed to be inherited by other classes to add a logger property.
Namespace(**kwargs)	Simple object for storing attributes.
Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.
SignatureArguments()	Methods to add arguments based on signatures to an ArgumentParser instance.
TypeCastCompleterMethod()	Completer method for Action classes with a casting type.
redirect_stderr(new_target)	Context manager for temporarily redirecting stderr to another file.

Exceptions:

ArgumentError(argument, message)	An error from creating or using an argument (optional or positional).
ParserError	Error raised when parsing a value fails.
yamlParserError	alias of yaml.parser.ParserError
yamlScannerError	alias of yaml.scanner.ScannerError

Functions:

deepcopy(x[, memo, _nil])	Deep copy operation on arbitrary Python objects.
dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
get_config_read_mode()	Returns the current config reading mode.
import_argcomplete(importer)	
import_jsonnet(importer)	
namespace_to_dict(cfg_ns)	Converts a nested namespace into a nested dictionary.
strip_meta(cfg)	Removes all metadata keys from a configuration object.
type_in(obj, types_set)	
usage_and_exit_error_handler(self, message)	Error handler to get the same behavior as in argparse.

class jsonargparse.core.**ArgumentParser**(**args*, *env_prefix=None*, *error_handler=<function usage_and_exit_error_handler>*, *formatter_class=<class 'jsonargparse.formatters.DefaultHelpFormatter'>*, *logger=None*, *version=None*, *print_config='--print_config'*, *parser_mode='yaml'*, *parse_as_dict=False*, *default_config_files=None*, *default_env=False*, *default_meta=True*, ***kwargs*)

Bases: *jsonargparse.signatures.SignatureArguments*, jsonargparse.core._ActionsContainer, argparse.ArgumentParser, *jsonargparse.util.LoggerProperty*

Parser for command line, yaml/jsonnet files and environment variables.

Methods:

__init__(*args[, env_prefix, error_handler, ...])	Initializer for ArgumentParser instance.
parse_known_args([args, namespace])	Raises NotImplementedError to dissuade its use, since typos in configs would go unnoticed.
parse_args([args, namespace, env, defaults, ...])	Parses command line argument strings.
parse_object(cfg_obj[, cfg_base, env, ...])	Parses configuration given as an object.
parse_env([env, defaults, nested, ...])	Parses environment variables.
parse_path(cfg_path[, ext_vars, env, ...])	Parses a configuration file (yaml or jsonnet) given its path.
parse_string(cfg_str[, cfg_path, ext_vars, ...])	Parses configuration (yaml or jsonnet) given as a string.
add_argument_group(*args[, name])	Adds a group to the parser.
add_subparsers(**kwargs)	Raises a NotImplementedError since jsonargparse uses add_subcommands.
add_subcommands([required, dest])	Adds sub-command parsers to the ArgumentParser.
dump(cfg[, format, skip_none, skip_check])	Generates a yaml or json string for the given configuration object.
save(cfg, path[, format, skip_none, ...])	Generates a yaml or json string for the given configuration object.
set_defaults(*args, **kwargs)	Sets default values from dictionary or keyword arguments.
get_default(dest)	Gets a single default value for the given destination key.
get_defaults([nested])	Returns a namespace with all default values.
error(message)	Logs error message if a logger is set, calls the error handler and raises a ParserError.

continues on next page

Table 6 – continued from previous page

check_config(cfg[, skip_none, branch])	Checks that the content of a given configuration object conforms with the parser.
instantiate_subclasses(cfg)	**rtype** Union[Namespace, Dict[str, Any]]
strip_unknown(cfg)	Removes all unknown keys from a configuration object.
get_config_files(cfg)	Returns a list of loaded config file paths.
merge_config(cfg_from, cfg_to)	Merges the first configuration into the second configuration.

Attributes:

error_handler	Property for the error_handler function that is called when there are parsing errors.
default_env	Whether by default environment variables parsing is enabled.
default_meta	Whether by default metadata is included in config objects.
env_prefix	The environment variables prefix property.

__init__(**args*, *env_prefix=None*, *error_handler=<function usage_and_exit_error_handler>*, *formatter_class=<class 'jsonargparse.formatters.DefaultHelpFormatter'>*, *logger=None*, *version=None*, *print_config='--print_config'*, *parser_mode='yaml'*, *parse_as_dict=False*, *default_config_files=None*, *default_env=False*, *default_meta=True*, ***kwargs*)

Initializer for ArgumentParser instance.

All the arguments from the initializer of argparse.ArgumentParser are supported. Additionally it accepts:

Parameters

- **env_prefix** (Optional[str]) – Prefix for environment variables.
- **error_handler** (Optional[Callable[[Type, str], None]]) – Handler for parsing errors, set to None to simply raise exception.
- **formatter_class** (Type[HelpFormatter]) – Class for printing help messages.
- **logger** (Union[bool, Dict[str, str], Logger, None]) – Configures the logger, see *LoggerProperty*.
- **version** (Optional[str]) – Program version string to add –version argument.
- **print_config** (Optional[str]) – Add this as argument to print config, set None to disable.
- **parser_mode** (str) – Mode for parsing configuration files, either "yaml" or "jsonnet".
- **parse_as_dict** (bool) – Whether to parse as dict instead of Namespace.
- **default_config_files** (Optional[List[str]]) – Default config file locations, e.g. ['~/.config/myapp/*.yaml'].
- **default_env** (bool) – Set the default value on whether to parse environment variables.
- **default_meta** (bool) – Set the default value on whether to include metadata in config objects.

parse_known_args(*args=None*, *namespace=None*)
Raises NotImplementedError to dissuade its use, since typos in configs would go unnoticed.

parse_args(*args=None*, *namespace=None*, *env=None*, *defaults=True*, *nested=True*, *with_meta=None*, *_skip_check=False*)
Parses command line argument strings.

All the arguments from argparse.ArgumentParser.parse_args are supported. Additionally it accepts:

Parameters

- **args** (Optional[List[str]]) – List of arguments to parse or None to use sys.argv.
- **env** (Optional[bool]) – Whether to merge with the parsed environment, None to use parser's default.
- **defaults** (bool) – Whether to merge with the parser's defaults.
- **nested** (bool) – Whether the namespace should be nested.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type Union[Namespace, Dict[str, Any]]

Returns An object with all parsed values as nested attributes.

Raises ***ParserError*** – If there is a parsing error and error_handler=None.

parse_object(*cfg_obj*, *cfg_base=None*, *env=None*, *defaults=True*, *nested=True*, *with_meta=None*, *_skip_check=False*)
Parses configuration given as an object.

Parameters

- **cfg_obj** (Dict[str, Any]) – The configuration object.
- **env** (Optional[bool]) – Whether to merge with the parsed environment, None to use parser's default.
- **defaults** (bool) – Whether to merge with the parser's defaults.
- **nested** (bool) – Whether the namespace should be nested.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type Union[Namespace, Dict[str, Any]]

Returns An object with all parsed values as attributes.

Raises ***ParserError*** – If there is a parsing error and error_handler=None.

parse_env(*env=None*, *defaults=True*, *nested=True*, *with_meta=None*, *_skip_logging=False*, *_skip_check=False*, *_skip_subcommands=False*)
Parses environment variables.

Parameters

- **env** (Optional[Dict[str, str]]) – The environment object to use, if None *os.environ* is used.
- **defaults** (bool) – Whether to merge with the parser's defaults.
- **nested** (bool) – Whether the namespace should be nested.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type Union[Namespace, Dict[str, Any]]

Returns An object with all parsed values as attributes.

Raises ***ParserError*** – If there is a parsing error and error_handler=None.

parse_path(*cfg_path*, *ext_vars=None*, *env=None*, *defaults=True*, *nested=True*, *with_meta=None*, *_skip_check=False*, *_base=None*)

Parses a configuration file (yaml or jsonnet) given its path.

Parameters

- **cfg_path** (str) – Path to the configuration file to parse.
- **ext_vars** (Optional[dict]) – Optional external variables used for parsing jsonnet.
- **env** (Optional[bool]) – Whether to merge with the parsed environment, None to use parser's default.
- **defaults** (bool) – Whether to merge with the parser's defaults.
- **nested** (bool) – Whether the namespace should be nested.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type Union[Namespace, Dict[str, Any]]

Returns An object with all parsed values as nested attributes.

Raises ***ParserError*** – If there is a parsing error and error_handler=None.

parse_string(*cfg_str*, *cfg_path=''*, *ext_vars=None*, *env=None*, *defaults=True*, *nested=True*, *with_meta=None*, *_skip_logging=False*, *_skip_check=False*, *_base=None*)

Parses configuration (yaml or jsonnet) given as a string.

Parameters

- **cfg_str** (str) – The configuration content.
- **cfg_path** (str) – Optional path to original config path, just for error printing.
- **ext_vars** (Optional[dict]) – Optional external variables used for parsing jsonnet.
- **env** (Optional[bool]) – Whether to merge with the parsed environment, None to use parser's default.
- **defaults** (bool) – Whether to merge with the parser's defaults.
- **nested** (bool) – Whether the namespace should be nested.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type Union[Namespace, Dict[str, Any]]

Returns An object with all parsed values as attributes.

Raises ***ParserError*** – If there is a parsing error and error_handler=None.

add_argument_group(**args*, *name=None*, ***kwargs*)

Adds a group to the parser.

All the arguments from argparse.ArgumentParser.add_argument_group are supported. Additionally it accepts:

Parameters **name** (Optional[str]) – Name of the group. If set the group object will be included in the parser.groups dict.

Returns The group object.

Raises **ValueError** – If group with the same name already exists.

add_subparsers(***kwargs*)
Raises a NotImplementedError since jsonargparse uses add_subcommands.

add_subcommands(*required=True*, *dest='subcommand'*, ***kwargs*)
Adds sub-command parsers to the ArgumentParser.

The aim is the same as argparse.ArgumentParser.add_subparsers the difference being that dest by default is 'subcommand' and the parsed values of the sub-command are stored in a nested namespace using the sub-command's name as base key.

Parameters

- **required** (bool) – Whether the subcommand must be provided.
- **dest** (str) – Destination key where the chosen subcommand name is stored.
- ****kwargs** – All options that *argparse.ArgumentParser.add_subparsers* accepts.

Return type Action

dump(*cfg*, *format='parser_mode'*, *skip_none=True*, *skip_check=False*)
Generates a yaml or json string for the given configuration object.

Parameters

- **cfg** (Union[Namespace, Dict[str, Any]]) – The configuration object to dump.
- **format** (str) – The output format: "yaml", "json", "json_indented" or "parser_mode".
- **skip_none** (bool) – Whether to exclude entries whose value is None.
- **skip_check** (bool) – Whether to skip parser checking.

Return type str

Returns The configuration in yaml or json format.

Raises **TypeError** – If any of the values of cfg is invalid according to the parser.

save(*cfg*, *path*, *format='parser_mode'*, *skip_none=True*, *skip_check=False*, *overwrite=False*, *multifile=True*, *branch=None*)
Generates a yaml or json string for the given configuration object.

Parameters

- **cfg** (Union[Namespace, Dict[str, Any]]) – The configuration object to save.
- **path** (str) – Path to the location where to save config.
- **format** (str) – The output format: "yaml", "json", "json_indented" or "parser_mode".
- **skip_none** (bool) – Whether to exclude entries whose value is None.
- **skip_check** (bool) – Whether to skip parser checking.
- **overwrite** (bool) – Whether to overwrite existing files.
- **multifile** (bool) – Whether to save multiple config files by using the __path__ metas.

Raises **TypeError** – If any of the values of cfg is invalid according to the parser.

set_defaults(**args*, ***kwargs*)
Sets default values from dictionary or keyword arguments.

Parameters

- ***args** (*dict*) – Dictionary defining the default values to set.
- ****kwargs** – Sets default values based on keyword arguments.

Raises **KeyError** – If key not defined in the parser.

get_default(*dest*)
Gets a single default value for the given destination key.

Parameters **dest** (str) – Destination key from which to get the default.

Raises **KeyError** – If key not defined in the parser.

get_defaults(*nested=True*)
Returns a namespace with all default values.

Parameters **nested** (bool) – Whether the namespace should be nested.

Return type Namespace

Returns An object with all default values as attributes.

error(*message*)
Logs error message if a logger is set, calls the error handler and raises a ParserError.

check_config(*cfg*, *skip_none=True*, *branch=None*)
Checks that the content of a given configuration object conforms with the parser.

Parameters

- **cfg** (Union[Namespace, Dict[str, Any]]) – The configuration object to check.
- **skip_none** (bool) – Whether to skip checking of values that are None.
- **branch** (Optional[str]) – Base key in case cfg corresponds only to a branch.

Raises

- **TypeError** – If any of the values are not valid.
- **KeyError** – If a key in cfg is not defined in the parser.

instantiate_subclasses(*cfg*)

Return type Union[Namespace, Dict[str, Any]]

strip_unknown(*cfg*)
Removes all unknown keys from a configuration object.

Parameters **cfg** (Union[Namespace, Dict[str, Any]]) – The configuration object to strip.

Return type Namespace

Returns The stripped configuration object.

get_config_files(*cfg*)
Returns a list of loaded config file paths.

Parameters **cfg** (Union[Namespace, Dict[str, Any]]) – The configuration object.

Return type List[str]

Returns Paths to loaded config files.

static merge_config(*cfg_from*, *cfg_to*)
Merges the first configuration into the second configuration.

Parameters

- **cfg_from** (Namespace) – The configuration from which to merge.
- **cfg_to** (Namespace) – The configuration into which to merge.

Return type Namespace

Returns The merged configuration.

property error_handler
Property for the error_handler function that is called when there are parsing errors.

Getter Returns the current error_handler function.

Setter Sets a new error_handler function (Callable[self, message:str] or None).

Raises **ValueError** – If an invalid value is given.

property default_env
Whether by default environment variables parsing is enabled.

Getter Returns the current default environment variables parsing setting.

Setter Sets the default environment variables parsing setting.

Raises **ValueError** – If an invalid value is given.

property default_meta
Whether by default metadata is included in config objects.

Getter Returns the current default metadata setting.

Setter Sets the default metadata setting.

Raises **ValueError** – If an invalid value is given.

property env_prefix
The environment variables prefix property.

Getter Returns the current environment variables prefix.

Setter Sets the environment variables prefix.

Raises **ValueError** – If an invalid value is given.

24.3 jsonargparse.signatures

Metods to add arguments based on class/method/function signatures.

Classes:

SignatureArguments()	Methods to add arguments based on signatures to an ArgumentParser instance.
ActionEnum(**kwargs)	An action based on an Enum that maps to-from strings and enum values.
ActionJsonSchema(**kwargs)	Action to parse option as json validated by a jsonschema.
Enum(value)	Generic enumeration.

Functions:

import_dataclasses(importer)	
import_docstring_parse(importer)	
is_optional(annotation, ref_type)	Checks whether a type annotation is an optional for one type class.

class jsonargparse.signatures.**SignatureArguments**

Bases: object

Methods to add arguments based on signatures to an ArgumentParser instance.

Methods:

add_class_arguments(theclass[, nested_key, ...])	Adds arguments from a class based on its type hints and docstrings.
add_method_arguments(theclass, themethod[, ...])	Adds arguments from a class based on its type hints and docstrings.
add_function_arguments(function[, ...])	Adds arguments from a function based on its type hints and docstrings.

add_class_arguments(*theclass*, *nested_key=None*, *as_group=True*, *as_positional=False*, *skip=None*)

Adds arguments from a class based on its type hints and docstrings.

Note: Keyword arguments without at least one valid type are ignored.

Parameters

- **theclass** (Type) – Class from which to add arguments.
- **nested_key** (Optional[str]) – Key for nested namespace.
- **as_group** (bool) – Whether arguments should be added to a new argument group.
- **as_positional** (bool) – Whether to add required parameters as positional arguments.
- **skip** (Optional[Container[str]]) – Names of parameters that should be skipped.

Return type int

Returns Number of arguments added.

Raises

- **ValueError** – When not given a class.
- **ValueError** – When there are required parameters without at least one valid type.

add_method_arguments(*theclass*, *themethod*, *nested_key=None*, *as_group=True*, *as_positional=False*, *skip=None*)

Adds arguments from a class based on its type hints and docstrings.

Note: Keyword arguments without at least one valid type are ignored.

Parameters

- **theclass** (Type) – Class which includes the method.
- **themethod** (str) – Name of the method for which to add arguments.
- **nested_key** (Optional[str]) – Key for nested namespace.
- **as_group** (bool) – Whether arguments should be added to a new argument group.

- **as_positional** (bool) – Whether to add required parameters as positional arguments.
- **skip** (Optional[Container[str]]) – Names of parameters that should be skipped.

Return type int

Returns Number of arguments added.

Raises

- **ValueError** – When not given a class or the name of a method of the class.
- **ValueError** – When there are required parameters without at least one valid type.

add_function_arguments(*function*, *nested_key=None*, *as_group=True*, *as_positional=False*, *skip=None*)

Adds arguments from a function based on its type hints and docstrings.

Note: Keyword arguments without at least one valid type are ignored.

Parameters

- **function** (Callable) – Function from which to add arguments.
- **nested_key** (Optional[str]) – Key for nested namespace.
- **as_group** (bool) – Whether arguments should be added to a new argument group.
- **as_positional** (bool) – Whether to add required parameters as positional arguments.
- **skip** (Optional[Container[str]]) – Names of parameters that should be skipped.

Return type int

Returns Number of arguments added.

Raises

- **ValueError** – When not given a callable.
- **ValueError** – When there are required parameters without at least one valid type.

24.4 jsonargparse.typing

Collection of types and type generators.

Functions:

restricted_number_type(name, base_type, ...)	Creates or returns an already registered restricted number type class.
restricted_string_type(name, regex[, docstring])	Creates or returns an already registered restricted string type class.
path_type(mode[, docstring])	Creates or returns an already registered path type class.
annotation_to_schema(annotation)	Generates a json schema from a type annotation if possible.
is_optional(annotation, ref_type)	Checks whether a type annotation is an optional for one type class.
register_type(register_key, new_type)	
type_in(obj, types_set)	
type_to_str(obj)	

Classes:

PositiveInt(v)	int restricted to be >0
NonNegativeInt(v)	int restricted to be 0
PositiveFloat(v)	float restricted to be >0
NonNegativeFloat(v)	float restricted to be 0
ClosedUnitInterval(v)	float restricted to be 0 and 1
OpenUnitInterval(v)	float restricted to be >0 and <1
NotEmptyStr(v)	str restricted to not-empty pattern ^.*[^].*$
Email(v)	str restricted to the email pattern ^[^@]+@[^@]+.[^@]+$
Path_fr(v)	str pointing to a file that exists and is readable
Path_fc(v)	str pointing to a file that can be created if it does not exist
Path_dw(v)	str pointing to a directory that exists and is writeable
Enum(value)	Generic enumeration.
Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.

jsonargparse.typing.**restricted_number_type**(*name*, *base_type*, *restrictions*, *join='and'*, *docstring=None*)

Creates or returns an already registered restricted number type class.

Parameters

- **name** (Optional[str]) – Name for the type or None for an automatic name.
- **base_type** (Type) – One of {int, float}.
- **restrictions** (Union[Tuple, List[Tuple]]) – Tuples of pairs (comparison, reference), e.g. ('>', 0).
- **join** (str) – How to combine multiple comparisons, one of {'or', 'and'}.
- **docstring** (Optional[str]) – Docstring for the type class.

Return type Type

Returns The created or retrieved type class.

jsonargparse.typing.**restricted_string_type**(*name*, *regex*, *docstring=None*)

Creates or returns an already registered restricted string type class.

Parameters

- **name** (str) – Name for the type or None for an automatic name.
- **regex** (Union[str, Pattern]) – Regular expression that the string must match.
- **docstring** (Optional[str]) – Docstring for the type class.

Return type Type

Returns The created or retrieved type class.

jsonargparse.typing.**path_type**(*mode*, *docstring=None*)

Creates or returns an already registered path type class.

Parameters

- **mode** (str) – The required type and access permissions among [fdrwxcuFDRWX].
- **docstring** (Optional[str]) – Docstring for the type class.

Return type Type

Returns The created or retrieved type class.

class jsonargparse.typing.**PositiveInt**(*v*)
Bases: jsonargparse.typing.restricted_number_type.<locals>.RestrictedNumber, int

int restricted to be >0

class jsonargparse.typing.**NonNegativeInt**(*v*)
Bases: jsonargparse.typing.restricted_number_type.<locals>.RestrictedNumber, int

int restricted to be 0

class jsonargparse.typing.**PositiveFloat**(*v*)
Bases: jsonargparse.typing.restricted_number_type.<locals>.RestrictedNumber, float

float restricted to be >0

class jsonargparse.typing.**NonNegativeFloat**(*v*)
Bases: jsonargparse.typing.restricted_number_type.<locals>.RestrictedNumber, float

float restricted to be 0

class jsonargparse.typing.**ClosedUnitInterval**(*v*)
Bases: jsonargparse.typing.restricted_number_type.<locals>.RestrictedNumber, float

float restricted to be 0 and 1

class jsonargparse.typing.**OpenUnitInterval**(*v*)
Bases: jsonargparse.typing.restricted_number_type.<locals>.RestrictedNumber, float

float restricted to be >0 and <1

class jsonargparse.typing.**NotEmptyStr**(*v*)
Bases: jsonargparse.typing.restricted_string_type.<locals>.RestrictedString, str

str restricted to not-empty pattern ^.*[^].*$

class jsonargparse.typing.**Email**(*v*)
Bases: jsonargparse.typing.restricted_string_type.<locals>.RestrictedString, str

str restricted to the email pattern ^[^@]+@[^@]+.[^@]+$

class jsonargparse.typing.**Path_fr**(*v*)
Bases: jsonargparse.typing.path_type.<locals>.PathType, str

str pointing to a file that exists and is readable

class jsonargparse.typing.**Path_fc**(*v*)
Bases: jsonargparse.typing.path_type.<locals>.PathType, str

str pointing to a file that can be created if it does not exist

class jsonargparse.typing.**Path_dw**(*v*)
Bases: jsonargparse.typing.path_type.<locals>.PathType, str

str pointing to a directory that exists and is writeable

24.5 jsonargparse.jsonschema

Action to support jsonschema and type hint annotations.

Classes:

ActionJsonSchema(**kwargs)	Action to parse option as json validated by a jsonschema.
Action(option_strings, dest[, nargs, const, ...])	Information about how to convert command line strings to Python objects.
Enum(value)	Generic enumeration.
Namespace(**kwargs)	Simple object for storing attributes.
Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.

Exceptions:

ModuleNotFound	alias of ModuleNotFoundError
ParserError	Error raised when parsing a value fails.
yamlParserError	alias of yaml.parser.ParserError
yamlScannerError	alias of yaml.scanner.ScannerError

Functions:

annotation_to_schema(annotation)	Generates a json schema from a type annotation if possible.
argcomplete_warn_redraw_prompt(prefix, message)	
import_jsonschema(importer)	
import_object(name)	Returns an object in a module given its dot import path.
is_optional(annotation, ref_type)	Checks whether a type annotation is an optional for one type class.
namespace_to_dict(cfg_ns)	Converts a nested namespace into a nested dictionary.
strip_meta(cfg)	Removes all metadata keys from a configuration object.
type_to_str(obj)	

class jsonargparse.jsonschema.**ActionJsonSchema**(***kwargs*)

Bases: argparse.Action

Action to parse option as json validated by a jsonschema.

Methods:

__init__(**kwargs)	Initializer for ActionJsonSchema instance.
__call__(*args, **kwargs)	Parses an argument validating against the corresponding jsonschema.
completer(prefix, **kwargs)	Used by argcomplete, validates value and shows expected type.

__init__(***kwargs*)
Initializer for ActionJsonSchema instance.

Parameters

- **schema** (*str or dict*) – Schema to validate values against.
- **annotation** (*type*) – Type object from which to generate schema.
- **enable_path** (*bool*) – Whether to try to load json from path (def.=True).
- **with_meta** (*bool*) – Whether to include metadata (def.=True).

Raises

- **ValueError** – If a parameter is invalid.
- **jsonschema.exceptions.SchemaError** – If the schema is invalid.

__call__(**args*, ***kwargs*)
Parses an argument validating against the corresponding jsonschema.

Raises **TypeError** – If the argument is not valid.

completer(*prefix*, ***kwargs*)
Used by argcomplete, validates value and shows expected type.

24.6 jsonargparse.jsonnet

Actions to support jsonnet.

Classes:

ActionJsonnetExtVars(**kwargs)	Action to add argument to provide ext_vars for jsonnet parsing.
ActionJsonnet(**kwargs)	Action to parse a jsonnet, optionally validating against a jsonschema.
Action(option_strings, dest[, nargs, const, ...])	Information about how to convert command line strings to Python objects.
ActionJsonSchema(**kwargs)	Action to parse option as json validated by a jsonschema.
Namespace(**kwargs)	Simple object for storing attributes.
Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.

Exceptions:

ParserError	Error raised when parsing a value fails.
yamlParserError	alias of yaml.parser.ParserError
yamlScannerError	alias of yaml.scanner.ScannerError

Functions:

dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
get_config_read_mode()	Returns the current config reading mode.
import_jsonnet(importer)	

continues on next page

Table 19 – continued from previous page

import_jsonschema(importer)	
namespace_to_dict(cfg_ns)	Converts a nested namespace into a nested dictionary.

class jsonargparse.jsonnet.**ActionJsonnetExtVars**(***kwargs*)

Bases: *jsonargparse.jsonschema.ActionJsonSchema*

Action to add argument to provide ext_vars for jsonnet parsing.

Methods:

__init__(**kwargs)	Initializer for ActionJsonnetExtVars instance.
__call__(*args, **kwargs)	Parses an argument validating against the corresponding jsonschema.

__init__(***kwargs*)

Initializer for ActionJsonnetExtVars instance.

__call__(**args*, ***kwargs*)

Parses an argument validating against the corresponding jsonschema.

Raises **TypeError** – If the argument is not valid.

class jsonargparse.jsonnet.**ActionJsonnet**(***kwargs*)

Bases: argparse.Action

Action to parse a jsonnet, optionally validating against a jsonschema.

Methods:

__init__(**kwargs)	Initializer for ActionJsonnet instance.
__call__(*args, **kwargs)	Parses an argument as jsonnet using ext_vars if defined.
split_ext_vars(ext_vars)	Splits an ext_vars dict into the ext_codes and ext_vars required by jsonnet.
parse(jsonnet[, ext_vars, with_meta])	Method that can be used to parse jsonnet independent from an ArgumentParser.

__init__(***kwargs*)

Initializer for ActionJsonnet instance.

Parameters

- **ext_vars** (*str or None*) – Key where to find the external variables required to parse the jsonnet.
- **schema** (*str or object or None*) – Schema to validate values against. Keyword argument required even if schema=None.

Raises

- **ValueError** – If a parameter is invalid.
- **jsonschema.exceptions.SchemaError** – If the schema is invalid.

__call__(**args*, ***kwargs*)

Parses an argument as jsonnet using ext_vars if defined.

Raises **TypeError** – If the argument is not valid.

static split_ext_vars(*ext_vars*)
Splits an ext_vars dict into the ext_codes and ext_vars required by jsonnet.

Parameters **ext_vars** (Union[Dict[str, Any], Namespace, None]) – External variables. Values can be strings or any other basic type.

Return type Tuple[Dict[str, Any], Dict[str, Any]]

parse(*jsonnet*, *ext_vars=None*, *with_meta=False*)
Method that can be used to parse jsonnet independent from an ArgumentParser.

Parameters

- **jsonnet** (Union[str, *Path*]) – Either a path to a jsonnet file or the jsonnet content.
- **ext_vars** (Union[Dict[str, Any], Namespace, None]) – External variables. Values can be strings or any other basic type.
- **with_meta** (bool) – Whether to include metadata in config object.

Return type Namespace

Returns The parsed jsonnet object.

Raises **TypeError** – If the input is neither a path to an existent file nor a jsonnet.

24.7 jsonargparse.actions

Collection of useful actions to define arguments.

Classes:

ActionConfigFile(**kwargs)	Action to indicate that an argument is a configuration file or a configuration string.
ActionYesNo(**kwargs)	Paired options –{yes_prefix}opt, –{no_prefix}opt to set True or False respectively.
ActionEnum(**kwargs)	An action based on an Enum that maps to-from strings and enum values.
ActionOperators(**kwargs)	DEPRECATED: Action to restrict a value with comparison operators.
ActionParser(**kwargs)	Action to parse option with a given parser optionally loading from file if string value.
ActionPath(**kwargs)	Action to check and store a path.
ActionPathList(**kwargs)	Action to check and store a list of file paths read from a plain text file or stream.
Action(option_strings, dest[, nargs, const, ...])	Information about how to convert command line strings to Python objects.
Enum(value)	Generic enumeration.
FilesCompleterMethod()	Completer method for Action classes that should complete files.
Namespace(**kwargs)	Simple object for storing attributes.
Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.

Exceptions:

ParserError	Error raised when parsing a value fails.
yamlParserError	alias of yaml.parser.ParserError
yamlScannerError	alias of yaml.scanner.ScannerError

Functions:

dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
get_config_read_mode()	Returns the current config reading mode.
namespace_to_dict(cfg_ns)	Converts a nested namespace into a nested dictionary.
restricted_number_type(name, base_type, ...)	Creates or returns an already registered restricted number type class.

class jsonargparse.actions.**ActionConfigFile**(***kwargs*)

Bases: argparse.Action, jsonargparse.optionals.FilesCompleterMethod

Action to indicate that an argument is a configuration file or a configuration string.

Methods:

__init__(**kwargs)	Initializer for ActionConfigFile instance.
__call__(parser, namespace, values[, ...])	Parses the given configuration and adds all the corresponding keys to the namespace.

__init__(***kwargs*)

Initializer for ActionConfigFile instance.

__call__(*parser*, *namespace*, *values*, *option_string=None*)

Parses the given configuration and adds all the corresponding keys to the namespace.

Raises **TypeError** – If there are problems parsing the configuration.

class jsonargparse.actions.**ActionYesNo**(***kwargs*)

Bases: argparse.Action

Paired options –{yes_prefix}opt, –{no_prefix}opt to set True or False respectively.

Methods:

__init__(**kwargs)	Initializer for ActionYesNo instance.
__call__(*args, **kwargs)	Sets the corresponding key to True or False depending on the option string used.
completer(**kwargs)	Used by argcomplete to support tab completion of arguments.

__init__(***kwargs*)

Initializer for ActionYesNo instance.

Parameters

- **yes_prefix** (*str*) – Prefix for yes option (default='').
- **no_prefix** (*str or None*) – Prefix for no option (default='no_').

Raises **ValueError** – If a parameter is invalid.

__call__(**args*, ***kwargs*)

Sets the corresponding key to True or False depending on the option string used.

completer(***kwargs*)
Used by argcomplete to support tab completion of arguments.

class jsonargparse.actions.**ActionEnum**(***kwargs*)
Bases: argparse.Action

An action based on an Enum that maps to-from strings and enum values.

Methods:

__init__(**kwargs)	Initializer for ActionEnum instance.
__call__(*args, **kwargs)	Parses an argument mapping a string to its Enum value.
completer(**kwargs)	Used by argcomplete to support tab completion of arguments.

__init__(***kwargs*)
Initializer for ActionEnum instance.

Parameters **enum** (*Enum*) – An Enum class.

Raises **ValueError** – If a parameter is invalid.

__call__(**args*, ***kwargs*)
Parses an argument mapping a string to its Enum value.

Raises **TypeError** – If value not present in the Enum.

completer(***kwargs*)
Used by argcomplete to support tab completion of arguments.

class jsonargparse.actions.**ActionOperators**(***kwargs*)
Bases: object

DEPRECATED: Action to restrict a value with comparison operators.

The new alternative is explained in *Restricted numbers*.

Methods:

__init__(**kwargs)	Initialize self.
__call__(*args, **kwargs)	Call self as a function.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__call__(**args*, ***kwargs*)
Call self as a function.

class jsonargparse.actions.**ActionParser**(***kwargs*)
Bases: argparse.Action

Action to parse option with a given parser optionally loading from file if string value.

Methods:

__init__(**kwargs)	Initializer for ActionParser instance.

continues on next page

Table 29 – continued from previous page

__call__(*args, **kwargs)	Parses an argument with the corresponding parser and if valid, sets the parsed value to the corresponding key.

__init__(***kwargs*)
Initializer for ActionParser instance.

Parameters **parser** (ArgumentParser) – A parser to parse the option with.

Raises **ValueError** – If the parser parameter is invalid.

__call__(**args*, ***kwargs*)
Parses an argument with the corresponding parser and if valid, sets the parsed value to the corresponding key.

Raises **TypeError** – If the argument is not valid.

class jsonargparse.actions.**ActionPath**(***kwargs*)
Bases: argparse.Action, jsonargparse.optionals.FilesCompleterMethod

Action to check and store a path.

Methods:

__init__(**kwargs)	Initializer for ActionPath instance.
__call__(*args, **kwargs)	Parses an argument as a Path and if valid sets the parsed value to the corresponding key.

__init__(***kwargs*)
Initializer for ActionPath instance.

Parameters

- **mode** (*str*) – The required type and access permissions among [fdrwxcuFDRWX] as a keyword argument, e.g. ActionPath(mode='drw').
- **skip_check** (*bool*) – Whether to skip path checks (def.=False).

Raises **ValueError** – If the mode parameter is invalid.

__call__(**args*, ***kwargs*)
Parses an argument as a Path and if valid sets the parsed value to the corresponding key.

Raises **TypeError** – If the argument is not a valid Path.

class jsonargparse.actions.**ActionPathList**(***kwargs*)
Bases: argparse.Action, jsonargparse.optionals.FilesCompleterMethod

Action to check and store a list of file paths read from a plain text file or stream.

Methods:

__init__(**kwargs)	Initializer for ActionPathList instance.
__call__(*args, **kwargs)	Parses an argument as a PathList and if valid sets the parsed value to the corresponding key.

__init__(***kwargs*)
Initializer for ActionPathList instance.

Parameters

- **mode** (*str*) – The required type and access permissions among [fdrwxcuFDRWX] as a keyword argument (uppercase means not), e.g. ActionPathList(mode='fr').
- **skip_check** (*bool*) – Whether to skip path checks (def.=False).
- **rel** (*str*) – Whether relative paths are with respect to current working directory 'cwd' or the list's parent directory 'list' (default='cwd').

Raises **ValueError** – If any of the parameters (mode or rel) are invalid.

__call__(**args*, ***kwargs*)
Parses an argument as a PathList and if valid sets the parsed value to the corresponding key.

Raises **TypeError** – If the argument is not a valid PathList.

24.8 jsonargparse.formatters

Formatter classes.

Classes:

DefaultHelpFormatter(prog[, ...])	Help message formatter that includes types, default values and env var names.
ActionConfigFile(**kwargs)	Action to indicate that an argument is a configuration file or a configuration string.
ActionEnum(**kwargs)	An action based on an Enum that maps to-from strings and enum values.
ActionJsonSchema(**kwargs)	Action to parse option as json validated by a jsonschema.
ActionParser(**kwargs)	Action to parse option with a given parser optionally loading from file if string value.
ActionYesNo(**kwargs)	Paired options –{yes_prefix}opt, –{no_prefix}opt to set True or False respectively.
Enum(value)	Generic enumeration.
HelpFormatter(prog[, indent_increment, ...])	Formatter for generating usage messages and argument help strings.

Functions:

type_to_str(obj)

class jsonargparse.formatters.**DefaultHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: argparse.HelpFormatter

Help message formatter that includes types, default values and env var names.

This class is an extension of argparse.HelpFormatter. Default values are always included. Furthermore, if the parser is configured with default_env=True command line options are preceded by 'ARG:' and the respective environment variable name is included preceded by 'ENV:'.

24.9 jsonargparse.optionals

Code related to optional dependencies.

Functions:

set_url_support(enabled)	Enables/disables URL support for config read mode.
get_config_read_mode()	Returns the current config reading mode.
argcomplete_warn_redraw_prompt(prefix, message)	
find_spec(name[, package])	Return the spec for the specified module.
import_argcomplete(importer)	
import_dataclasses(importer)	
import_docstring_parse(importer)	
import_jsonnet(importer)	
import_jsonschema(importer)	
import_requests(importer)	
import_url_validator(importer)	

Classes:

FilesCompleterMethod()	Completer method for Action classes that should complete files.
TypeCastCompleterMethod()	Completer method for Action classes with a casting type.

Exceptions:

ModuleNotFound	alias of ModuleNotFoundError

jsonargparse.optionals.**set_url_support**(*enabled*)
Enables/disables URL support for config read mode.

jsonargparse.optionals.**get_config_read_mode**()
Returns the current config reading mode.

Return type str

24.10 jsonargparse.util

Collection of general functions and classes.

Exceptions:

ParserError	Error raised when parsing a value fails.
ModuleNotFound	alias of ModuleNotFoundError
yamlParserError	alias of yaml.parser.ParserError
yamlScannerError	alias of yaml.scanner.ScannerError

Functions:

dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
namespace_to_dict(cfg_ns)	Converts a nested namespace into a nested dictionary.
strip_meta(cfg)	Removes all metadata keys from a configuration object.
usage_and_exit_error_handler(self, message)	Error handler to get the same behavior as in argparse.
contextmanager(func)	@contextmanager decorator.
deepcopy(x[, memo, _nil])	Deep copy operation on arbitrary Python objects.
get_config_read_mode()	Returns the current config reading mode.
import_object(name)	Returns an object in a module given its dot import path.
import_requests(importer)	
import_url_validator(importer)	

Classes:

Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.
LoggerProperty()	Class designed to be inherited by other classes to add a logger property.
Namespace(**kwargs)	Simple object for storing attributes.
redirect_stderr(new_target)	Context manager for temporarily redirecting stderr to another file.

exception jsonargparse.util.**ParserError**
Bases: Exception

Error raised when parsing a value fails.

jsonargparse.util.**dict_to_namespace**(*cfg_dict*)
Converts a nested dictionary into a nested namespace.

Parameters **cfg_dict** (Dict[str, Any]) – The configuration to process.

Return type Namespace

Returns The nested configuration namespace.

jsonargparse.util.**namespace_to_dict**(*cfg_ns*)
Converts a nested namespace into a nested dictionary.

Parameters **cfg_ns** (Namespace) – The configuration to process.

Return type Dict[str, Any]

Returns The nested configuration dictionary.

jsonargparse.util.**strip_meta**(*cfg*)
Removes all metadata keys from a configuration object.

Parameters **cfg** (Union[Namespace, Dict]) – The configuration object to strip.

Returns The stripped configuration object.

Return type argparse.Namespace

jsonargparse.util.**usage_and_exit_error_handler**(*self*, *message*)
Error handler to get the same behavior as in argparse.

Parameters

- **self** (ArgumentParser) – The ArgumentParser object.

- **message** (str) – The message describing the error being handled.

class jsonargparse.util.**Path**(*path*, *mode='fr'*, *cwd=None*, *skip_check=False*)
Bases: object

Stores a (possibly relative) path and the corresponding absolute path.

When a Path instance is created it is checked that: the path exists, whether it is a file or directory and whether has the required access permissions (f=file, d=directory, r=readable, w=writeable, x=executable, c=creatable, u=url or in uppercase meaning not, i.e., F=not-file, D=not-directory, R=not-readable, W=not-writeable and X=not-executable). The absolute path can be obtained without having to remember the working directory from when the object was created.

Methods:

__init__(path[, mode, cwd, skip_check])	Initializer for Path instance.
__call__([absolute])	Returns the path as a string.
get_content([mode])	Returns the contents of the file or the response of a GET request to the URL.

__init__(*path*, *mode='fr'*, *cwd=None*, *skip_check=False*)
Initializer for Path instance.

Parameters

- **path** (Union[str, *Path*]) – The path to check and store.
- **mode** (str) – The required type and access permissions among [fdrwxcuFDRWX].
- **cwd** (Optional[str]) – Working directory for relative paths. If None, then os.getcwd() is used.
- **skip_check** (bool) – Whether to skip path checks.

Raises

- **ValueError** – If the provided mode is invalid.
- **TypeError** – If the path does not exist or does not agree with the mode.

__call__(*absolute=True*)
Returns the path as a string.

Parameters **absolute** (bool) – If false returns the original path given, otherwise the corresponding absolute path.

Return type str

get_content(*mode='r'*)
Returns the contents of the file or the response of a GET request to the URL.

Return type str

class jsonargparse.util.**LoggerProperty**
Bases: object

Class designed to be inherited by other classes to add a logger property.

Methods:

__init__()	Initializer for LoggerProperty class.

Attributes:

logger	The logger property for the class.

__init__()
Initializer for LoggerProperty class.

property logger
The logger property for the class.

Getter Returns the current logger.

Setter Sets the given logging.Logger as logger or sets the default logger if given True/str(logger name)/dict(name, level), or disables logging if given False/None.

Raises **ValueError** – If an invalid logger value is given.

CHAPTER

TWENTYFIVE

LICENSE

CHAPTER

TWENTYSIX

INDEX

- changelog
- genindex

PYTHON MODULE INDEX

j

INDEX

Symbols

A

C

D

E

G

I

J

L

M

N

O

P

R

S

U