jsonargparse Documentation

Release 2.32.2

Mauricio Villegas

Nov 13, 2020

CONTENTS

1 Features 3

2 Basic usage 5

3 Nested namespaces 7

4 Environment variables 9

5 Configuration files 11

6 Json schemas 13

7 Jsonnet files 15

8 Parsing paths 17

9 Parsing URLs 19

10 Comparison operators 21

11 Boolean arguments 23

12 Parsers as arguments 25

13 Sub-commands 27

14 Logging 29

15 Contributing 31

16 API Reference 33

17 License 47

18 Indices and tables 49

Python Module Index 51

Index 53

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

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

The aim is similar to other projects such as configargparse, yconf and confuse. The obvious question is, why yet another module 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 create a new module.

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 yaml or jsonnet configuration files and not necessarily from a command line tool.
- Support for nested namespaces which makes it possible to parse config files with non-flat hierarchies.
- Support for two popular supersets of json, making config files more versatile and powerful.
- Parsing of relative paths within config files and path lists.
- Several convenient action classes to ease common parsing use cases (paths, comparison operators, json schemas, ...).
- Two mechanisms to define parsers in a modular way: parsers as arguments and sub-commands.
- Default behavior is not identical to argparse, though it is possible to configure it to be identical. The main differences are:
  - When parsing fails `ParserError` is raised, instead of printing usage and program exit.
  - To modify the behavior for parsing errors (e.g. print usage) an error handler function can be provided.
- 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

BASIC USAGE

A parser is created just like it is done with 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 *jsonargparse.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 a *ParserError* is raised, so depending on the use case it might be necessary to catch it.

```
>>> try:
...     cfg = parser.parse_args(['--opt2', 'four'])
... except jsonargparse.ParserError as ex:
...     print('parser error: '+str(ex))
...
parser error: argument --opt2: invalid float value: 'four'
```

To get the default behavior of argparse the ArgumentParser can be initialized as follows:

```
parser = ArgumentParser(
    prog='app',
    error_handler='usage_and_exit_error_handler',
    description='Description for my app.')
```

CHAPTER

THREE

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

FOUR

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 *jsonargparse.ArgumentParser.parse_args()* does not check environment variables, so it has to be enabled explicitly.

There is also the *jsonargparse.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

FIVE

CONFIGURATION FILES

An important feature of this module 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.

When creating the *ArgumentParser* the `default_config_files` argument 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. If 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'
```

The config file could 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 *jsonargparse.ArgumentParser.parse_path()* and *jsonargparse.ArgumentParser.parse_string()* to parse a config file or a config contained in a string respectively.

CHAPTER

SIX

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 the module with the *all* extras requires as:

```
$ pip3 install jsonargparse[all]
```

Check out the jsonschema documentation to learn how to write a schema. The current version of jsonargparse uses Draft4Validator. 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

SEVEN

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 *all* extras requires as:

```
$ pip3 install jsonargparse[all]
```

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

EIGHT

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 the *ActionPath* and the *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` exists and is readable, the following could be done:

```
>>> from jsonargparse import ArgumentParser, ActionPath
>>> parser = ArgumentParser()
>>> parser.add_argument('--databases.info', action=ActionPath(mode='fr'))
>>> cfg = parser.parse_path('app/config.yaml')
```

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:

```
>>> cfg.databases.info(absolute=False)
'data/info.db'
>>> cfg.databases.info()
'/YOUR_CWD/app/data/info.db'
```

Likewise directories can also be parsed by including in the mode the `'d'` flag, e.g. `ActionPath(mode='drw')`.

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

An argument with *ActionPath* 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 provided.

CHAPTER

NINE

PARSING URLS

The *ActionPath* and *ActionPathList* classes also support URLs which after parsing the *jsonargparse.Path.get_content()* 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 the *all* extras requires is chosen:

```
$ pip3 install jsonargparse[all]
```

Then the `'u'` flag can be used to parse URLs. For example if it is desired that an argument can be either a readable file or URL the action would be initialized as `ActionPath(mode='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 *jsonargparse.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: *jsonargparse.ArgumentParser.parse_path()*, *jsonargparse.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

TEN

COMPARISON OPERATORS

It is quite common that when parsing a number, its range should be limited. To ease these cases the module includes the *ActionOperators*. Some examples of arguments that can be added using this action are the following:

```
from jsonargparse import ActionOperators
# Larger than zero
parser.add_argument('--op1', action=ActionOperators(expr=('>', 0)))
# Between 0 and 10
parser.add_argument('--op2', action=ActionOperators(expr=[('>=', 0), ('<=', 10)]))
# Either larger than zero or 'off' string
def int_or_off(x): return x if x == 'off' else int(x)
parser.add_argument('--op3', action=ActionOperators(expr=[('>', 0), ('==', 'off')],␣
↪join='or', type=int_or_off))
```

CHAPTER

ELEVEN

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)
```

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

TWELVE

PARSERS AS ARGUMENTS

As parsers get more complex, being able to define them in a modular way becomes important. Two mechanisms are available to define parsers in a modular way, both explained in this and the next section respectively.

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. So they should be instantiated exclusively for the *ActionParser* and not used standalone.

CHAPTER

THIRTEEN

SUB-COMMANDS

A second 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 *jsonargparse.ArgumentParser.add_subcommands()* method is used. Then an existing parser is added as a sub-command using *jsonargparse.ActionSubCommands.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 *jsonargparse.ArgumentParser.parse_path()* or *jsonargparse.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`.

CHAPTER

FOURTEEN

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"}`.

CHAPTER

FIFTEEN

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 optional requirements are stored in section `[options.extras_require]` in the `all` entry. Also there are `test`, `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 would be:

```
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 --editable .[test,dev,doc,all]
```

CHAPTER

SIXTEEN

API REFERENCE

Classes:

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.
ActionJsonSchema(**kwargs)	Action to parse option as json validated by a json-schema.
ActionJsonnet(**kwargs)	Action to parse a jsonnet, optionally validating against a jsonschema.
ActionJsonnetExtVars(**kwargs)	Action to be used for jsonnet ext_vars.
ActionOperators(**kwargs)	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.
ActionSubCommands(option_strings, prog, ...)	Extension of argparse._SubParsersAction to modify sub-commands functionality.
ActionYesNo(**kwargs)	Paired options –{yes_prefix}opt, –{no_prefix}opt to set True or False respectively.
ArgumentParser(*args[, env_prefix, ...])	Parser for command line, yaml/jsonnet files and environment variables.
DefaultHelpFormatter(prog[, ...])	Help message formatter with namespace key, env var and default values in argument help.
LoggerProperty()	Class designed to be inherited by other classes to add a logger property.
Path(path, mode, cwd, skip_check)	Stores a (possibly relative) path and the corresponding absolute path.
SimpleNamespace	A simple attribute-based namespace.
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.

Data:

Dict	The central part of internal API.
List	The central part of internal API.
Set	The central part of internal API.

Functions:

contextmanager(func)	@contextmanager decorator.
deepcopy(x[, memo, _nil])	Deep copy operation on arbitrary Python objects.
dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
import_jsonnet(importer)	
import_jsonschema(importer)	
import_requests(importer)	
import_url_validator(importer)	
namespace_to_dict(cfg_ns)	Converts a nested namespace into a nested dictionary.
set_url_support(enabled)	Enables/disables URL support for config read mode.
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.

jsonargparse.**import_jsonschema**(*importer*)

jsonargparse.**import_jsonnet**(*importer*)

jsonargparse.**import_url_validator**(*importer*)

jsonargparse.**import_requests**(*importer*)

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

exception jsonargparse.**ParserError**
Bases: Exception

Error raised when parsing a value fails.

class jsonargparse.**DefaultHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)
Bases: argparse.ArgumentDefaultsHelpFormatter

Help message formatter with namespace key, env var and default values in argument help.

This class is an extension of argparse.ArgumentDefaultsHelpFormatter. The main difference is that optional arguments are preceded by 'ARG:', the nested namespace key in dot notation is included preceded by 'NSKEY:', and if the ArgumentParser's default_env=True, the environment variable name is included preceded by 'ENV:'.

class jsonargparse.**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 current logger.

__init__()
Initializer for LoggerProperty class.

property logger
The current logger.

class jsonargparse.**ArgumentParser**(**args*, *env_prefix=None*, *error_handler=None*, *formatter_class='default'*, *logger=None*, *version=None*, *parser_mode='yaml'*, *default_config_files: List[str] = []*, *default_env: bool = False*, *default_meta: bool = True*, ***kwargs*)
Bases: jsonargparse._ActionsContainer, argparse.ArgumentParser, *jsonargparse.LoggerProperty*

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

Methods:

__init__(*args[, env_prefix, error_handler, ...])	Initializer for ArgumentParser instance.
add_argument_group(*args[, name])	Adds a group to the parser.
add_subcommands([required, dest])	Adds sub-command parsers to the ArgumentParser.
add_subparsers(**kwargs)	Raises a NotImplementedError.
check_config(cfg[, skip_none, branch])	Checks that the content of a given configuration object conforms with the parser.
dump(cfg[, format, skip_none, skip_check])	Generates a yaml or json string for the given configuration object.
error(message)	Logs error message if a logger is set, calls the error handler and raises a ParserError.
get_config_files(cfg)	Returns a list of loaded config file paths.
get_defaults([nested])	Returns a namespace with all default values.
merge_config(cfg_from, cfg_to)	Merges the first configuration into the second configuration.
parse_args([args, namespace, env, defaults, ...])	Parses command line argument strings.
parse_env([env, defaults, nested, ...])	Parses environment variables.
parse_known_args([args, namespace])	parse_known_args not implemented to dissuade its use, since typos in configs would go unnoticed.
parse_object(cfg_obj[, cfg_base, env, ...])	Parses configuration given as an object.
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.
save(cfg, path[, format, skip_none, ...])	Generates a yaml or json string for the given configuration object.
strip_unknown(cfg)	Removes all unknown keys from a configuration object.

Attributes:

default_env	The current value of the default_env.
default_meta	The current value of the default_meta.
env_prefix	The current value of the env_prefix.

continues on next page

Table 8 – continued from previous page

error_handler	The current error_handler.

__init__(**args*, *env_prefix=None*, *error_handler=None*, *formatter_class='default'*, *logger=None*, *version=None*, *parser_mode='yaml'*, *default_config_files: List[str] = []*, *default_env: bool = False*, *default_meta: bool = True*, ***kwargs*)

Initializer for ArgumentParser instance.

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

Parameters

- **env_prefix** (*str*) – Prefix for environment variables.
- **error_handler** (*Callable*) – Handler for parsing errors (default=None). For same behavior as argparse use *usage_and_exit_error_handler()*.
- **formatter_class** (*argparse.HelpFormatter or str*) – Class for printing help messages or one of {"default", "default_argparse"}.
- **logger** (*logging.Logger or bool or int or str or None*) – A logger to use or True/int(log level)/str(logger name) to use the default logger or False/None to disable logging.
- **version** (*str*) – Program version string to add –version argument.
- **parser_mode** (*str*) – Mode for parsing configuration files, either "yaml" or "jsonnet".
- **default_config_files** (*list[str]*) – List of strings defining default config file locations. For example: `['~/.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.

property error_handler

The current error_handler.

parse_known_args(*args=None*, *namespace=None*)

parse_known_args not implemented to dissuade its use, since typos in configs would go unnoticed.

add_subparsers(***kwargs*)

Raises a NotImplementedError.

add_subcommands(*required=True*, *dest='subcommand'*, ***kwargs*)

Adds sub-command parsers to the ArgumentParser.

In contrast to argparse.ArgumentParser.add_subparsers a required argument is accepted, dest by default is 'subcommand' and the values of the sub-command are stored using the sub-command's name as base key.

parse_args(*args=None*, *namespace=None*, *env: bool = None*, *defaults: bool = True*, *nested: bool = True*, *with_meta: bool = None*)

Parses command line argument strings.

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

Parameters

- **env** (*bool or None*) – Whether to merge with the parsed environment. None means use the ArgumentParser's default.
- **defaults** (*bool*) – Whether to merge with the parser's defaults.
- **nested** (*bool*) – Whether the namespace should be nested.

- **with_meta** (*bool*) – Whether to include metadata in config object.

Returns An object with all parsed values as nested attributes.

Return type types.SimpleNamespace

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

parse_path(*cfg_path: str*, *ext_vars: dict = {}*, *env: bool = None*, *defaults: bool = True*, *nested: bool = True*, *with_meta: bool = None*, *_skip_check: bool = False*, *_base=None*) → types.SimpleNamespace

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

Parameters

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

Returns An object with all parsed values as nested attributes.

Return type types.SimpleNamespace

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

parse_string(*cfg_str: str*, *cfg_path: str = ''*, *ext_vars: dict = {}*, *env: bool = None*, *defaults: bool = True*, *nested: bool = True*, *with_meta: bool = None*, *_skip_logging: bool = False*, *_skip_check: bool = False*, *_base=None*) → types.SimpleNamespace

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** (*dict*) – Optional external variables used for parsing jsonnet.
- **env** (*bool or None*) – Whether to merge with the parsed environment. None means use the ArgumentParser's default.
- **defaults** (*bool*) – Whether to merge with the parser's defaults.
- **nested** (*bool*) – Whether the namespace should be nested.
- **with_meta** (*bool*) – Whether to include metadata in config object.

Returns An object with all parsed values as attributes.

Return type types.SimpleNamespace

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

parse_object(*cfg_obj: dict*, *cfg_base=None*, *env: bool = None*, *defaults: bool = True*, *nested: bool = True*, *with_meta: bool = None*, *_skip_check: bool = False*) → types.SimpleNamespace

Parses configuration given as an object.

Parameters

- **cfg_obj** (*dict*) – The configuration object.
- **env** (*bool or None*) – Whether to merge with the parsed environment. None means use the ArgumentParser's default.
- **defaults** (*bool*) – Whether to merge with the parser's defaults.
- **nested** (*bool*) – Whether the namespace should be nested.
- **with_meta** (*bool*) – Whether to include metadata in config object.

Returns An object with all parsed values as attributes.

Return type types.SimpleNamespace

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

dump(*cfg: Union[types.SimpleNamespace, dict]*, *format: str = 'parser_mode'*, *skip_none: bool = True*, *skip_check: bool = False*) → str

Generates a yaml or json string for the given configuration object.

Parameters

- **cfg** (*types.SimpleNamespace or dict*) – The configuration object to dump.
- **format** (*str*) – The output format: "yaml", "json", "json_indented" or "parser_mode".
- **skip_none** (*bool*) – Whether to exclude checking values that are None.
- **skip_check** (*bool*) – Whether to skip parser checking.

Returns The configuration in yaml or json format.

Return type str

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

save(*cfg: Union[types.SimpleNamespace, dict]*, *path: str*, *format: str = 'parser_mode'*, *skip_none: bool = True*, *skip_check: bool = False*, *overwrite: bool = False*, *multifile: bool = True*, *branch=None*) → None

Generates a yaml or json string for the given configuration object.

Parameters

- **cfg** (*types.SimpleNamespace or dict*) – 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 checking values that are 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.

property default_env

The current value of the default_env.

property default_meta

The current value of the default_meta.

property env_prefix

The current value of the env_prefix.

parse_env(*env: Dict[str, str] = None*, *defaults: bool = True*, *nested: bool = True*, *with_meta: bool = None*, *_skip_logging: bool = False*, *_skip_check: bool = False*) → types.SimpleNamespace

Parses environment variables.

Parameters

- **env** (*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** (*bool*) – Whether to include metadata in config object.

Returns An object with all parsed values as attributes.

Return type types.SimpleNamespace

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

get_defaults(*nested: bool = True*) → types.SimpleNamespace

Returns a namespace with all default values.

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

Returns An object with all default values as attributes.

Return type types.SimpleNamespace

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

error(*message*)

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

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

Adds a group to the parser.

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

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

Returns The group object.

check_config(*cfg: Union[types.SimpleNamespace, dict]*, *skip_none: bool = True*, *branch=None*)

Checks that the content of a given configuration object conforms with the parser.

Parameters

- **cfg** (*types.SimpleNamespace or dict*) – The configuration object to check.
- **skip_none** (*bool*) – Whether to skip checking of values that are None.
- **branch** (*str or None*) – 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.

strip_unknown(*cfg*)

Removes all unknown keys from a configuration object.

Parameters **cfg** (*types.SimpleNamespace or dict*) – The configuration object to strip.

Returns The stripped configuration object.

Return type types.SimpleNamespace

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

Parameters **cfg** (*types.SimpleNamespace or dict*) – The configuration object.

Returns Paths to loaded config files.

Return type list

static merge_config(*cfg_from: types.SimpleNamespace*, *cfg_to: types.SimpleNamespace*) → types.SimpleNamespace
Merges the first configuration into the second configuration.

Parameters

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

Returns The merged configuration.

Return type types.SimpleNamespace

jsonargparse.**dict_to_namespace**(*cfg_dict: Dict[str, Any]*) → types.SimpleNamespace
Converts a nested dictionary into a nested namespace.

Parameters **cfg_dict** (*dict*) – The configuration to process.

Returns The nested configuration namespace.

Return type types.SimpleNamespace

jsonargparse.**namespace_to_dict**(*cfg_ns: types.SimpleNamespace*) → Dict[str, Any]
Converts a nested namespace into a nested dictionary.

Parameters **cfg_ns** (*types.SimpleNamespace*) – The configuration to process.

Returns The nested configuration dictionary.

Return type dict

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

Parameters **cfg** (*types.SimpleNamespace or dict*) – The configuration object to strip.

Returns The stripped configuration object.

Return type types.SimpleNamespace

class jsonargparse.**ActionConfigFile**(***kwargs*)
Bases: argparse.Action

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

Methods:

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

__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.**ActionYesNo**(***kwargs*)
Bases: argparse.Action

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

Methods:

__call__(*args, **kwargs)	Sets the corresponding key to True or False depending on the option string used.
__init__(**kwargs)	Initializer for ActionYesNo instance.

__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.

class jsonargparse.**ActionJsonSchema**(***kwargs*)
Bases: argparse.Action

Action to parse option as json validated by a jsonschema.

Methods:

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

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

Parameters

- **schema** (*str or object*) – Schema to validate values against.
- **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.

class jsonargparse.**ActionJsonnetExtVars**(***kwargs*)
Bases: *jsonargparse.ActionJsonSchema*

Action to be used for jsonnet ext_vars.

Methods:

__init__(**kwargs)	Initializer for ActionJsonSchema instance.

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

Parameters

- **schema** (*str or object*) – Schema to validate values against.
- **with_meta** (*bool*) – Whether to include metadata (def.=True).

Raises

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

class jsonargparse.**ActionJsonnet**(***kwargs*)
Bases: argparse.Action

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

Methods:

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

__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** (*dict*) – External variables. Values can be strings or any other basic type.

parse (*jsonnet*, *ext_vars={}*, *with_meta=False*)
Method that can be used to parse jsonnet independent from an ArgumentParser.

Parameters

- **jsonnet** (*str*) – Either a path to a jsonnet file or the jsonnet content.
- **ext_vars** (*dict*) – External variables. Values can be strings or any other basic type.

Returns The parsed jsonnet object.

Return type SimpleNamespace

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

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

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

Methods:

__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.

__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.**ActionSubCommands**(*option_strings*, *prog*, *parser_class*, *dest='==SUPPRESS=='*, *required=False*, *help=None*, *metavar=None*)
Bases: argparse._SubParsersAction

Extension of argparse._SubParsersAction to modify sub-commands functionality.

Methods:

__call__(parser, namespace, values[, ...])	Adds sub-command dest and parses sub-command arguments.
add_parser(**kwargs)	Raises a NotImplementedError.
add_subcommand(name, parser, **kwargs)	Adds a parser as a sub-command parser.
handle_subcommands(parser, cfg, env, defaults)	Adds sub-command dest if missing and parses defaults and environment variables.

add_parser(***kwargs*)
Raises a NotImplementedError.

add_subcommand(*name*, *parser*, ***kwargs*)
Adds a parser as a sub-command parser.

In contrast to argparse.ArgumentParser.add_subparsers add_parser requires to be given a parser as argument.

__call__(*parser*, *namespace*, *values*, *option_string=None*)
Adds sub-command dest and parses sub-command arguments.

static handle_subcommands(*parser*, *cfg*, *env*, *defaults*)
Adds sub-command dest if missing and parses defaults and environment variables.

class jsonargparse.**ActionOperators**(***kwargs*)
Bases: argparse.Action

Action to restrict a value with comparison operators.

Methods:

__call__(*args, **kwargs)	Parses an argument restricted by the operators and if valid sets the parsed value to the corresponding key.
__init__(**kwargs)	Initializer for ActionOperators instance.

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

Parameters

- **expr** (*tuple or list[tuple]*) – Pairs of operators (> >= < <= == !=) and reference values, e.g. [('>=', 1),...].
- **join** (*str*) – How to combine multiple comparisons, must be 'or' or 'and' (default='and').
- **type** (*type*) – The value type (default=int).

Raises **ValueError** – If any of the parameters (expr, join or type) are invalid.

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

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

class jsonargparse.**ActionPath**(***kwargs*)
Bases: argparse.Action

Action to check and store a path.

Methods:

__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.

__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.**ActionPathList**(***kwargs*)
Bases: argparse.Action

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

Methods:

__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.

__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.

class jsonargparse.**Path**(*path*, *mode: str = 'fr'*, *cwd: str = None*, *skip_check: bool = 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:

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

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

Parameters

- **path** (*str or Path*) – The path to check and store.
- **mode** (*str*) – The required type and access permissions among [fdrwxcuFDRWX].
- **cwd** (*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.

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

jsonargparse.**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.

CHAPTER

SEVENTEEN

LICENSE

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

j

jsonargparse, 33

INDEX

Symbols

__call__() (*jsonargparse.ActionConfigFile method*), 41
__call__() (*jsonargparse.ActionJsonSchema method*), 41
__call__() (*jsonargparse.ActionJsonnet method*), 42
__call__() (*jsonargparse.ActionOperators method*), 44
__call__() (*jsonargparse.ActionParser method*), 43
__call__() (*jsonargparse.ActionPath method*), 45
__call__() (*jsonargparse.ActionPathList method*), 45
__call__() (*jsonargparse.ActionSubCommands method*), 44
__call__() (*jsonargparse.ActionYesNo method*), 41
__call__() (*jsonargparse.Path method*), 46
__init__() (*jsonargparse.ActionConfigFile method*), 40
__init__() (*jsonargparse.ActionJsonSchema method*), 41
__init__() (*jsonargparse.ActionJsonnet method*), 42
__init__() (*jsonargparse.ActionJsonnetExtVars method*), 42
__init__() (*jsonargparse.ActionOperators method*), 44
__init__() (*jsonargparse.ActionParser method*), 43
__init__() (*jsonargparse.ActionPath method*), 44
__init__() (*jsonargparse.ActionPathList method*), 45
__init__() (*jsonargparse.ActionYesNo method*), 41
__init__() (*jsonargparse.ArgumentParser method*), 36
__init__() (*jsonargparse.LoggerProperty method*), 35
__init__() (*jsonargparse.Path method*), 45

A

ActionConfigFile (*class in jsonargparse*), 40
ActionJsonnet (*class in jsonargparse*), 42
ActionJsonnetExtVars (*class in jsonargparse*), 41
ActionJsonSchema (*class in jsonargparse*), 41
ActionOperators (*class in jsonargparse*), 44
ActionParser (*class in jsonargparse*), 43
ActionPath (*class in jsonargparse*), 44
ActionPathList (*class in jsonargparse*), 45
ActionSubCommands (*class in jsonargparse*), 43
ActionYesNo (*class in jsonargparse*), 41
add_argument_group() (*jsonargparse.ArgumentParser method*), 39
add_parser() (*jsonargparse.ActionSubCommands method*), 43
add_subcommand() (*jsonargparse.ActionSubCommands method*), 43
add_subcommands() (*jsonargparse.ArgumentParser method*), 36
add_subparsers() (*jsonargparse.ArgumentParser method*), 36
ArgumentParser (*class in jsonargparse*), 35

C

check_config() (*jsonargparse.ArgumentParser method*), 39

D

default_env() (*jsonargparse.ArgumentParser property*), 38
default_meta() (*jsonargparse.ArgumentParser property*), 38
DefaultHelpFormatter (*class in jsonargparse*), 34
dict_to_namespace() (*in module jsonargparse*), 40
dump() (*jsonargparse.ArgumentParser method*), 38

E

env_prefix() (*jsonargparse.ArgumentParser property*), 38
error() (*jsonargparse.ArgumentParser method*), 39
error_handler() (*jsonargparse.ArgumentParser property*), 36

G

get_config_files() (*jsonargparse.ArgumentParser method*), 40
get_content() (*jsonargparse.Path method*), 46
get_defaults() (*jsonargparse.ArgumentParser method*), 39

H

handle_subcommands() (*jsonargparse.ActionSubCommands static method*), 44

I

import_jsonnet() (*in module jsonargparse*), 34
import_jsonschema() (*in module jsonargparse*), 34
import_requests() (*in module jsonargparse*), 34
import_url_validator() (*in module jsonargparse*), 34

J

jsonargparse
    module, 33

L

logger() (*jsonargparse.LoggerProperty property*), 35
LoggerProperty (*class in jsonargparse*), 34

M

merge_config() (*jsonargparse.ArgumentParser static method*), 40
module
    jsonargparse, 33

N

namespace_to_dict() (*in module jsonargparse*), 40

P

parse() (*jsonargparse.ActionJsonnet method*), 43
parse_args() (*jsonargparse.ArgumentParser method*), 36
parse_env() (*jsonargparse.ArgumentParser method*), 38
parse_known_args() (*jsonargparse.ArgumentParser method*), 36
parse_object() (*jsonargparse.ArgumentParser method*), 37
parse_path() (*jsonargparse.ArgumentParser method*), 37
parse_string() (*jsonargparse.ArgumentParser method*), 37
ParserError, 34
Path (*class in jsonargparse*), 45

S

save() (*jsonargparse.ArgumentParser method*), 38
set_url_support() (*in module jsonargparse*), 34
split_ext_vars() (*jsonargparse.ActionJsonnet static method*), 42
strip_meta() (*in module jsonargparse*), 40
strip_unknown() (*jsonargparse.ArgumentParser method*), 39

U

usage_and_exit_error_handler() (*in module jsonargparse*), 46