jsonargparse Documentation

Mauricio Villegas

Nov 29, 2021

CONTENTS

https://github.com/omni-us/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

- Great support of type hint annotations for automatic creation of parsers and minimal boilerplate command line interfaces.
- Support for nested namespaces which makes it possible to parse config files with non-flat hierarchies.
- Parsing of relative paths within config files and path lists.
- Support for two popular supersets of json: yaml and jsonnet.
- Parsers can be configured just like with python's argparse, thus it has a gentle learning curve.
- Several convenient types and action classes to ease common parsing use cases (paths, comparison operators, json schemas, enums, regex strings, ...).
- 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, several optional features 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 to enable all optional features. Installing jsonargparse with extras require is as follows:

```
pip install "jsonargparse[signatures,urls]"  # Enable signatures and URLs features
pip install "jsonargparse[all]"              # Enable all optional features
```

The following table references sections that describe optional features and the corresponding extras requires that enables them.

	urls/fsspec	argcomplete	jsonnet	jsonschema	signatures
Type hints					✓
Classes, methods and functions					✓
Class type and sub-classes					✓
Parsing URLs	✓				
Json schemas				✓	
Jsonnet files			✓		
Tab completion		✓			

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 for parameters to instantiate the class, then a class method name must be given (i.e. methods become *Sub-commands*) and the remaining arguments are for parameters of the class method. An example would be:

```
from random import randint
from jsonargparse import CLI

class Main:
    def __init__(
        self,
        max_prize: int = 100
    ):
        """
```

(continues on next page)

(continued from previous page)

```
        Args:
            max_prize: Maximum prize that can be awarded.
        """
        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, i.e. `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 the name of the class and the second the name of the method, i.e. `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 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 for modularity, 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 shown 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. 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 1'
>>> cfg.lev1.opt2
'from default 2'
```

A group of nested options can be created by using a dataclass. This has the advantage that the same options can be reused in multiple places of a project. An example analogous to the one above would be:

```
from dataclasses import dataclass

@dataclass
class Level1Options:
    """Level 1 options
    Args:
        opt1: Option 1
        opt2: Option 2
    """
    opt1: str = 'from default 1'
    opt2: str = 'from default 2'

parser = ArgumentParser()
parser.add_argument('--lev1', type=Level1Options, default=Level1Options())
```

The *Namespace* class an extension of the one from argparse. It has some additional features which can be seen in the API. In particular keys can be accessed like a dictionary either with individual keys, e.g. `cfg['lev1']['opt1']`, or a single one, e.g. `cfg['lev1.opt1']`. Also the class has a method *Namespace.as_dict()* that can be used to represent the nested namespace as a nested dictionary useful for example for class instantiation.

CHAPTER

SIX

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.default_config_files* property can be set when creating a parser to specify patterns to search for configuration files. For example if a parser is created as `ArgumentParser(default_config_files=['~/.myapp.yaml', '/etc/myapp.yaml'])`, when parsing if any of those two config files exist it will be parsed and used to override the defaults. All matched config files are parsed and applied in given order. The default config files are always parsed first, this means that any command line arguments will override its values.

It is also possible to add an argument to explicitly provide a configuration file path. Providing a config file as an argument does not disable the parsing of `default_config_files`. The config argument would be parsed in the specific position among the command line arguments. Therefore the arguments found after would override the values from that config file. The config argument can be given multiple times, each overriding the values of the previous. Using the example parser from the *Nested namespaces* section above, 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 config file argument and parsing some dummy 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('--config', action=ActionConfigFile)
>>> cfg = parser.parse_args(['--lev1.opt1', 'from arg 1',
...                          '--config', '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(['--config', '{"lev1":{"opt1":"from string 1"}}'])
>>> cfg.lev1.opt1
'from string 1'
```

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. Any other argument provided through an environment variable would override the config file one.

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

Parsers that have an *ActionConfigFile* also 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. If the ruyaml package is installed, the config can be printed having the help descriptions content as yaml comments by using `--print_config=comments`. Another option is `--print_config=skip_null` which skips entries whose value is `null`.

From within python it is also possible to serialize a config object by using either the *ArgumentParser.dump()* or *ArgumentParser.save()* mehtods.

CHAPTER

SEVEN

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. 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 parsed before all the other environment variables.

CHAPTER

EIGHT

CLASSES, METHODS AND FUNCTIONS

It is good practice to write python code in which parameters have type hints and these 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 annotated parameters as arguments: *SignatureArguments.add_class_arguments()*, *SignatureArguments.add_method_arguments()*, *SignatureArguments.add_function_arguments()* and *SignatureArguments.add_dataclass_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, foo: Dict[str, Union[int, List[int]]], **kwargs):
        """Initializer for MyClass.

        Args:
            foo: Description for foo.
        """
        pass

    def mymethod(self, bar: float, baz: bool = False):
        """Description for mymethod.

        Args:
            bar: Description for bar.
            baz: Description for baz.
        """
        pass
```

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

```
from jsonargparse import ArgumentParser

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

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

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.

Instantiation of classes added as argument groups with `add_class_arguments()` can be done more simply for an entire config object using *ArgumentParser.instantiate_classes()*. For the example above running `cfg = parser.instantiate_classes(cfg)` would result in `cfg['myclass']['init']` containing an instance of `MyClass` initialized with whatever command line arguments were parsed.

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.

In some cases there are functions which return an instance of a class. To add this to a parser such that *ArgumentParser.instantiate_classes()* calls this function, the example would change to:

```
from jsonargparse import ArgumentParser, class_from_function

parser = ArgumentParser()
dynamic_class = class_from_function(instantiate_myclass)
parser.add_class_arguments(dynamic_class, 'myclass.init')
```

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.
- Arguments whose name starts with `_` are considered for internal use and ignored.
- The signature methods have a `skip` parameter which can be used to exclude adding some arguments, e.g. `parser.add_method_arguments(MyClass, 'mymethod', skip={'flag'})`.

Note: 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*.

Note: For all features described above to work, one optional package is required: docstring-parser to get the argument descriptions from the docstrings. This package is included when jsonargparse is installed using the `signatures` extras require as explained in section *Installation*.

CHAPTER

NINE

ARGUMENT LINKING

Some use cases could require adding arguments from multiple classes and be desired that some parameters get a value automatically computed from other arguments. This behavior can be obtained by using the *ArgumentParser.link_arguments()* method.

There are two types of links each defined with `apply_on='parse'` and `apply_on='instantiate'`. As the names suggest the former are set when calling one of the parse methods and the latter are set when calling *ArgumentParser.instantiate_classes()*.

For parsing links, source keys can be individual arguments or nested groups. The target key has to be a single argument. The keys can be inside init_args of a subclass. The compute function should accept as many positional arguments as there are sources and return a value of type compatible with the target. An example would be the following:

```
class Model:
    def __init__(self, batch_size: int):
        self.batch_size = batch_size

class Data:
    def __init__(self, batch_size: int = 5):
        self.batch_size = batch_size

parser = ArgumentParser()
parser.add_class_arguments(Model, 'model')
parser.add_class_arguments(Data, 'data')
parser.link_arguments('data.batch_size', 'model.batch_size', apply_on='parse')
```

As argument and in config files only `data.batch_size` should be specified. Then whatever value it has will be propagated to `model.batch_size`.

For instantiation links, only a single source key is supported. The key can be for a class group created using *SignatureArguments.add_class_arguments()* or a subclass action created using *SignatureArguments.add_subclass_arguments()*. If the key is only the class group or subclass action, then a compute function is required which takes the source class instance and returns the value to set in target. Alternatively the key can specify a class attribute. The target key has to be a single argument and can be inside init_args of a subclass. The order of instantiation used by *ArgumentParser.instantiate_classes()* is automatically determined based on the links. The instantiation links must be a directed acyclic graph. An example would be the following:

```
class Model:
    def __init__(self, num_classes: int):
        self.num_classes = num_classes

class Data:
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.num_classes = get_num_classes()

parser = ArgumentParser()
parser.add_class_arguments(Model, 'model')
parser.add_class_arguments(Data, 'data')
parser.link_arguments('data.num_classes', 'model.num_classes', apply_on='instantiate')
```

This link would imply that *ArgumentParser.instantiate_classes()* instantiates `Data` first, then use the `num_classes` attribute to instantiate `Model`.

CHAPTER

TEN

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 a type hint 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 jsonargparse specific types. 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`, `complex`, `List`, `Iterable`, `Sequence`, `Any`, `Union`, `Optional`, `Type`, `Enum`, `UUID`, `timedelta`, 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`, `Mapping`, and `MutableMapping` are supported but only with `str` or `int` keys.
- `Tuple`, `Set` and `MutableSet` 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. `Tuple` with ellipsis (`Tuple[type, ...]`) is also supported. In command line arguments, config files and environment variables, tuples and sets are represented as an array.
- `dataclasses` are supported as a type but without any nesting and for pure data classes. By pure it is meant that it only inherits from data classes, not a mixture of normal classes and data classes.
- To set a value to `None` it is required to use `null` since this is how json/yaml defines it. To avoid confusion in the help, `NoneType` is displayed as `null`. For example a function argument with type and default `Optional[str] = None` would be shown in the help as `type: Union[str, null], default: null`.
- `Callable` has an experimental partial implementation and not officially supported yet.

CHAPTER

ELEVEN

REGISTERING TYPES

With the *register_type()* function it is possible to register additional types for use in jsonargparse parsers. If the type class can be instantiated with a string representation and casting the instance to `str` gives back the string representation, then only the type class is given to *register_type()*. For example in the `jsonargparse.typing` package this is how complex numbers are registered: `register_type(complex)`. For other type classes that don't have these properties, to register it might be necessary to provide a serializer and/or deserializer function. Including the serializer and deserializer functions, the registration of the complex numbers example is equivalent to `register_type(complex, serializer=str, deserializer=complex)`.

A more useful example could be registering the `datetime` class. This case requires to give both a serializer and a deserializer as seen below.

```
from datetime import datetime
from jsonargparse import ArgumentParser
from jsonargparse.typing import register_type

def serializer(v):
    return v.isoformat()

def deserializer(v):
    return datetime.strptime(v, '%Y-%m-%dT%H:%M:%S')

register_type(datetime, serializer, deserializer)

parser = ArgumentParser()
parser.add_argument('--datetime', type=datetime)
parser.parse_args(['--datetime=2008-09-03T20:56:35'])
```

Note: The registering of types is only intended for simple types. By default any class used as a type hint is considered a sub-class (see *Class type and sub-classes*) which might be good for many use cases. If a class is registered with *register_type()* then the sub-class option is no longer available.

CHAPTER

TWELVE

CLASS TYPE AND SUB-CLASSES

It is also possible to use an arbitrary class as a type such that the argument accepts this class or any derived subclass. In the config file 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 given type and that `init_args` values correspond to valid arguments to instantiate it. After parsing, the config object will include the `class_path` and `init_args` entries. To get a config object with all sub-classes instantiated, the *`ArgumentParser.instantiate_classes()`* method is used. The `skip` parameter of the signature methods can also be used to exclude arguments within subclasses. This is done by giving its relative destination key, i.e. as `param.init_args.subparam`.

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

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

Then in python:

```
>>> from calendar import Calendar

>>> class MyClass:
...     def __init__(self, calendar: Calendar):
...         self.calendar = calendar

>>> parser = ArgumentParser()
>>> parser.add_class_arguments(MyClass, 'myclass')

>>> cfg = parser.parse_path('config.yaml')
>>> cfg.myclass.calendar.as_dict()
{'class_path': 'calendar.Calendar', 'init_args': {'firstweekday': 1}}

>>> cfg = parser.instantiate_classes(cfg)
>>> cfg.myclass.calendar.getfirstweekday()
1
```

In this 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. The help of the parser does not show details for a type class since this depends on the subclass. To get help details for a particular subclass there is a specific help option that receives the import path. If there is some subclass of `Calendar` which can be imported from `mycode.MyCalendar`, then it would be possible to see the corresponding `init_args` details by running the tool from the command line as:

```
python tool.py --myclass.calendar.help mycode.MyCalendar
```

An individual argument can also be added having as type a class, i.e. `parser.add_argument('--calendar', type=Calendar)`. There is also another method *SignatureArguments.add_subclass_arguments()* which does the same as `add_argument`, but has some added benefits: 1) the argument is added in a new group automatically; 2) the argument values can be given in an independent config file by specifying a path to it; and 3) by default sets a useful `metavar` and `help` strings.

12.1 Default values

For a parameter that has a class as type it might also be wanted to set a default value for it. Special care must be taken when doing this, could be considered bad practice and be a good idea to avoid in most cases. The issue is that classes are normally mutable. Depending on how the parameter value is used, its default class instance in the signature could be changed. This goes against what a default value is expected to be and lead to bugs which are difficult to debug.

Since there are some legitimate use cases for class instances in defaults, they are supported with a particular behavior and recommendations. The first approach is using a normal class instance, for example:

```
class MyClass:
    def __init__(
        self,
        calendar: Calendar = Calendar(firstweekday=1),
    ):
        self.calendar = calendar
```

Adding this class to a parser will work without issues. Parsing would also work and if not overridden the default class instance will be found in the respective key of the config object. If `--print_config` is used, the class instance is just cast to a string. This means that the generated config file must be modified to become a valid input to the parser. Due to the limitations and the risk of mutable default this approach is discouraged.

The second approach which is the recommended one is to use the special function *lazy_instance()* to instantiate the default. Continuing with the same example above this would be:

```
from jsonargparse import lazy_instance

class MyClass:
    def __init__(
        self,
        calendar: Calendar = lazy_instance(Calendar, firstweekday=1),
    ):
        self.calendar = calendar
```

In this case the default value will still be an instance of `Calendar`. The difference is that the parsing methods would provide a dict with `class_path` and `init_args` instead of the class instance. Furthermore, if *ArgumentParser.instantiate_classes()* is used a new instance of the class is created thereby avoiding issues related to the mutability of the default.

12.2 Final classes

When a class is decorated with *final()* there shouldn't be any derived subclass. Using a final class as a type hint works similar to subclasses. The difference is that the init args are given directly in a dictionary without specifying a `class_path`. Therefore, the code below would accept the corresponding yaml structure.

```
from jsonargparse.typing import final

@final
class FinalCalendar(Calendar):
    pass

parser = ArgumentParser()
parser.add_argument('--calendar', type=FinalCalendar)
cfg = parser.parse_path('config.yaml')
```

```
calendar:
  firstweekday: 1
```

CHAPTER

THIRTEEN

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. 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

FOURTEEN

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:

```
>>> from jsonargparse import ActionJsonSchema

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

>>> parser = ArgumentParser()
>>> parser.add_argument('--json', action=ActionJsonSchema(schema=schema))

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

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 is printed.

CHAPTER

FIFTEEN

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 `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('--config', action=ActionConfigFile)
cfg = parser.parse_args(['--config', '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', 'example.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

SIXTEEN

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 *ActionPathList* class.

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:

```yaml
# 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 that 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()
'/.../app/data/info.db'
```

Likewise directories can be parsed using the *Path_dw* type, which 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 using 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()`.

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. 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` with *`ActionPathList`* then a single list is generated including all paths in all provided lists.

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

CHAPTER

SEVENTEEN

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. Alternatively, *path_type()* can also be used for fsspec supported file systems. The respective optional package(s) will be installed along with jsonargparse if installed with the `urls` or `fsspec` extras require as explained in section *Installation*.

The `'u'` flag is used to parse URLs using requests and the flag `'s'` to parse fsspec file systems. 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. 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 you import `from jsonargparse import set_config_read_mode` and then run `set_config_read_mode(urls_enabled=True)` or `set_config_read_mode(fsspec_enabled=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 that a tool that can receive a configuration file via *ActionConfigFile* is able to get the content from a URL, thus something like the following would work:

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

CHAPTER

EIGHTEEN

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

NINETEEN

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

TWENTY

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:

```
>>> import enum
>>> 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

TWENTYONE

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

TWENTYTWO

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',
    title='Inner node title',
    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.

Note that when adding `inner_parser` a title was given. In the help, the added parsers are shown as independent groups starting with the given `title`. It is also possible to provide a `description`.

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 as 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

TWENTYTHREE

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

TWENTYFOUR

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

TWENTYFIVE

CONTRIBUTING

Contributions to jsonargparse 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 run 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

TWENTYSIX

API REFERENCE

Even though jsonargparse has several internal modules, users are expected to only import from the main `jsonargparse` or `jsonargparse.typing`. This allows doing internal refactorings without affecting dependants. Only objects explicity exposed in `jsonargparse.__init__.__all__` and `jsonargparse.typing.__all__` can be considered public.

26.1 jsonargparse

Functions:

CLI([components, args, config_help, ...])	Function for simple creation of command line interfaces.
class_from_function(func)	Creates a dynamic class which if instantiated is equivalent to calling func.
compose_dataclasses(*args)	Returns a pure dataclass inheriting all given dataclasses and properly handling __post_init__.
lazy_instance(class_type, **kwargs)	Instantiates a lazy instance of the given type.
namespace_to_dict(namespace)	Returns a deepcopy of a nested namespace converted into a nested dictionary.
dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
strip_meta()	Removes all metadata keys from a configuration object.
get_config_read_mode()	Returns the current config reading mode.
set_config_read_mode([urls_enabled, ...])	Enables/disables optional config read modes.
usage_and_exit_error_handler(parser, message)	Error handler to get the same behavior as in argparse.
set_url_support(enabled)	Enables/disables URL support for config read mode.

Classes:

ArgumentParser(*args[, env_prefix, ...])	Parser for command line, yaml/jsonnet files and environment variables.
SignatureArguments()	Methods to add arguments based on signatures to an ArgumentParser instance.
ActionTypeHint([typehint, enable_path])	Action to parse a type hint.
ActionJsonSchema([schema, enable_path, ...])	Action to parse option as json validated by a jsonschema.
ActionJsonnetExtVars(**kwargs)	Action to add argument to provide ext_vars for jsonnet parsing.
ActionJsonnet([ext_vars, schema])	Action to parse a jsonnet, optionally validating against a jsonschema.

continues on next page

Table 2 – continued from previous page

ActionConfigFile(**kwargs)	Action to indicate that an argument is a configuration file or a configuration string.
ActionYesNo([yes_prefix, no_prefix])	Paired options --[yes_prefix]opt, --[no_prefix]opt to set True or False respectively.
ActionParser([parser])	Action to parse option with a given parser optionally loading from file if string value.
ActionPathList([mode, rel])	Action to check and store a list of file paths read from a plain text file or stream.
Namespace(*args, **kwargs)	Extension of argparse's Namespace to support nesting and subscript access.
DefaultHelpFormatter(prog[, ...])	Help message formatter that includes types, default values and env var names.
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.
ActionEnum(**kwargs)	An action based on an Enum that maps to-from strings and enum values.
ActionOperators(**kwargs)	Action to restrict a value with comparison operators.
ActionPath(mode[, skip_check])	Action to check and store a path.

Exceptions:

ParserError	Error raised when parsing a value fails.

jsonargparse.**CLI**(*components=None*, *args=None*, *config_help='Path to a configuration file in json or yaml format.'*, *set_defaults=None*, *as_positional=True*, *return_parser=False*, ***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.
- **set_defaults** (Optional[Dict[str, Any]]) – Dictionary of values to override components defaults.
- **as_positional** (bool) – Whether to add required parameters as positional arguments.
- **return_parser** (bool) – Whether to return the parser instead of parsing and running.
- ****kwargs** – Used to instantiate ArgumentParser.

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

class jsonargparse.**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'*, *default_config_files=None*, *default_env=False*, *default_meta=True*, ***kwargs*)

Bases: jsonargparse.core._ActionsContainer, argparse.ArgumentParser

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, with_meta, ...])	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.
link_arguments(source, target[, compute_fn, ...])	Makes an argument value be derived from the values of other arguments.
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, ...])	Writes to file(s) the yaml or json 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([skip_check])	Returns a namespace with all default values.
error(message[, ex])	Logs error message if a logger is set, calls the error handler and raises a ParserError.
check_config(cfg[, skip_none, ...])	Checks that the content of a given configuration object conforms with the parser.
instantiate_classes(cfg[, instantiate_groups])	Recursively instantiates all subclasses defined by 'class_path' and 'init_args' and class groups.
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.
instantiate_subclasses(cfg)	Calls instantiate_classes with instantiate_groups=False.

Attributes:

error_handler	Property for the error_handler function that is called when there are parsing errors.
default_config_files	Default config file locations.
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'*, *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[[*ArgumentParser*, str], None]]) – Handler for parsing errors, set to None to simply raise exception.
- **formatter_class** (Type[*DefaultHelpFormatter*]) – 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".
- **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*, *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[Sequence[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.

- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type *Namespace*

Returns A config object with all parsed values.

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

parse_object(*cfg_obj*, *cfg_base=None*, *env=None*, *defaults=True*, *with_meta=None*, *_skip_check=False*, *_skip_required=False*)

Parses configuration given as an object.

Parameters

- **cfg_obj** (Union[*Namespace*, 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.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type *Namespace*

Returns A config object with all parsed values.

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

parse_env(*env=None*, *defaults=True*, *with_meta=None*, *_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.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type *Namespace*

Returns A config object with all parsed values.

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

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

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.
- **with_meta** (Optional[bool]) – Whether to include metadata in config object, None to use parser's default.

Return type *Namespace*

Returns A config object with all parsed values.

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*, *with_meta=None*, *_skip_check=False*, *_fail_no_subcommand=True*)

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** – 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 *Namespace*

Returns A config object with all parsed values.

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

link_arguments(*source*, *target*, *compute_fn=None*, *apply_on='parse'*)

Makes an argument value be derived from the values of other arguments.

Refer to *Argument linking* for a detailed explanation and examples-

Parameters

- **source** (Union[str, Tuple[str, ...]]) – Key(s) from which the target value is derived.
- **target** (str) – Key to where the value is set.
- **compute_fn** (Optional[Callable]) – Function to compute target value from source.
- **apply_on** (str) – At what point to set target value, 'parse' or 'instantiate'.

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

add_subparsers(***kwargs*)

Raises a NotImplementedError since jsonargparse uses add_subcommands.

Return type NoReturn

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 _ActionSubCommands

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

Parameters

- **cfg** (*Namespace*) – 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.
- **yaml_comments** (bool) – Whether to add help content as comments.

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*)
Writes to file(s) the yaml or json for the given configuration object.

Parameters

- **cfg** (*Namespace*) – 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.

Return type None

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.

Return type None

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 or its default not defined in the parser.

Return type Any

get_defaults(*skip_check=False*)
Returns a namespace with all default values.

Parameters

- **nested** – Whether the namespace should be nested.
- **skip_check** (bool) – Whether to skip check if configuration is valid.

Return type *Namespace*

Returns An object with all default values as attributes.

error(*message*, *ex=None*)
Logs error message if a logger is set, calls the error handler and raises a ParserError.

Return type NoReturn

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

Parameters

- **cfg** (*Namespace*) – The configuration object to check.
- **skip_none** (bool) – Whether to skip checking of values that are None.
- **skip_required** (bool) – Whether to skip checking required arguments.
- **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.

Return type None

instantiate_classes(*cfg*, *instantiate_groups=True*)
Recursively instantiates all subclasses defined by 'class_path' and 'init_args' and class groups.

Parameters

- **cfg** (*Namespace*) – The configuration object to use.
- **instantiate_groups** (bool) – Whether class groups should be instantiated.

Return type *Namespace*

Returns A configuration object with all subclasses and class groups instantiated.

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

Parameters **cfg** (*Namespace*) – 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** (*Namespace*) – 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 A new object with the merged configuration.

instantiate_subclasses(*cfg:* jsonargparse.namespace.Namespace) → *jsonargparse.namespace.Namespace*
Calls instantiate_classes with instantiate_groups=False.

Args: cfg: The configuration object to use.

Returns: A configuration object with all subclasses instantiated.

> **Warning:** instantiate_subclasses was deprecated in v4.0.0 and will be removed in v5.0.0.

Return type *Namespace*

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_config_files
Default config file locations.

Getter Returns the current default config file locations.

Setter Sets new default config file locations, e.g. `['~/.config/myapp/*.yaml']`.

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

property default_env: bool
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.

Return type bool

property default_meta: bool
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.

Return type bool

property env_prefix: Optional[str]
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.

Return type Optional[str]

jsonargparse.**class_from_function**(*func*)
Creates a dynamic class which if instantiated is equivalent to calling func.

Parameters **func** (Callable[..., ~ClassType]) – A function that returns an instance of a class. It must have a return type annotation.

Return type Type[~ClassType]

jsonargparse.**compose_dataclasses**(**args*)
Returns a pure dataclass inheriting all given dataclasses and properly handling __post_init__.

class jsonargparse.**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_dataclass_arguments(theclass, nested_key)	Adds arguments from a dataclass based on its field types and docstrings.
add_subclass_arguments(baseclass, nested_key)	Adds arguments to allow specifying any subclass of the given base class.

add_class_arguments(*theclass*, *nested_key=None*, *as_group=True*, *as_positional=False*, *skip=None*, *instantiate=True*, *fail_untyped=True*, *sub_configs=False*, *linked_targets=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[Set[str]]) – Names of parameters that should be skipped.
- **instantiate** (bool) – Whether the class group should be instantiated by instantiate_classes.
- **fail_untyped** (bool) – Whether to raise exception if a required parameter does not have a type.

- **sub_configs** (bool) – Whether subclass type hints should be loadable from inner config file.

Return type List[str]

Returns The list 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*, *fail_untyped=True*, *sub_configs=False*)

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[Set[str]]) – Names of parameters that should be skipped.
- **fail_untyped** (bool) – Whether to raise exception if a required parameter does not have a type.
- **sub_configs** (bool) – Whether subclass type hints should be loadable from inner config file.

Return type List[str]

Returns The list 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*, *fail_untyped=True*, *sub_configs=False*)

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[Set[str]]) – Names of parameters that should be skipped.
- **fail_untyped** (bool) – Whether to raise exception if a required parameter does not have a type.

- **sub_configs** (bool) – Whether subclass type hints should be loadable from inner config file.

Return type List[str]

Returns The list of arguments added.

Raises

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

add_dataclass_arguments(*theclass*, *nested_key*, *default=None*, *as_group=True*, ***kwargs*)

Adds arguments from a dataclass based on its field types and docstrings.

Parameters

- **theclass** (Type) – Class from which to add arguments.
- **nested_key** (str) – Key for nested namespace.
- **default** (Union[Type, dict, None]) – Value for defaults. Must be instance of or kwargs for theclass.
- **as_group** (bool) – Whether arguments should be added to a new argument group.

Return type List[str]

Returns The list of arguments added.

Raises

- **ValueError** – When not given a dataclass.
- **ValueError** – When default is not instance of or kwargs for theclass.

add_subclass_arguments(*baseclass*, *nested_key*, *as_group=True*, *skip=None*, *instantiate=True*, *required=False*, *metavar='{"class_path":...[,"init_args":...]}'*, *help='Dictionary with "class_path" and "init_args" for any subclass of %(baseclass_name)s.'*, ***kwargs*)

Adds arguments to allow specifying any subclass of the given base class.

This adds an argument that requires a dictionary with a "class_path" entry which must be a import dot notation expression. Optionally any init arguments for the class can be given in the "init_args" entry. Since subclasses can have different init arguments, the help does not show the details of the arguments of the base class. Instead a help argument is added that will print the details for a given class path.

Parameters

- **baseclass** (Union[Type, Tuple[Type, ...]]) – Base class or classes to use to check subclasses.
- **nested_key** (str) – Key for nested namespace.
- **as_group** (bool) – Whether arguments should be added to a new argument group.
- **skip** (Optional[Set[str]]) – Names of parameters that should be skipped.
- **required** (bool) – Whether the argument group is required.
- **metavar** (str) – Variable string to show in the argument's help.
- **help** (str) – Description of argument to show in the help.
- ****kwargs** – Additional parameters like in add_class_arguments.

Raises **ValueError** – When given an invalid base class.

class jsonargparse.**ActionTypeHint**(*typehint=None*, *enable_path=False*, ***kwargs*)
Bases: argparse.Action

Action to parse a type hint.

Methods:

__init__([typehint, enable_path])	Initializer for ActionTypeHint instance.
is_supported_typehint(typehint[, full])	Whether the given type hint is supported.
__call__(*args, **kwargs)	Parses an argument validating against the corresponding type hint.
completer(prefix, **kwargs)	Used by argcomplete, validates value and shows expected type.

__init__(*typehint=None*, *enable_path=False*, ***kwargs*)
Initializer for ActionTypeHint instance.

Parameters

- **typehint** (Optional[Type]) – The type hint to use for parsing.
- **enable_path** (bool) – Whether to try to load parsed value from path.

Raises ValueError – If a parameter is invalid.

static is_supported_typehint(*typehint*, *full=False*)
Whether the given type hint is supported.

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

Raises TypeError – If the argument is not valid.

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

jsonargparse.**lazy_instance**(*class_type*, ***kwargs*)
Instantiates a lazy instance of the given type.

By lazy it is meant that the __init__ is delayed unit the first time that a method of the instance is called. It also provides a *lazy_get_init_data* method useful for serializing.

Parameters

- **class_type** (Type[~ClassType]) – The class to instantiate.
- ****kwargs** – Any keyword arguments to use for instantiation.

Return type ~ClassType

class jsonargparse.**ActionJsonSchema**(*schema=None*, *enable_path=True*, *with_meta=True*, ***kwargs*)
Bases: argparse.Action

Action to parse option as json validated by a jsonschema.

Methods:

__init__([schema, enable_path, with_meta])	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__(*schema=None*, *enable_path=True*, *with_meta=True*, ***kwargs*)
Initializer for ActionJsonSchema instance.

Parameters

- **schema** (Union[str, Dict, None]) – Schema to validate values against.
- **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.

class jsonargparse.**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.**ActionJsonnet**(*ext_vars=None*, *schema=None*, ***kwargs*)
Bases: argparse.Action

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

Methods:

__init__([ext_vars, schema])	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__(*ext_vars=None*, *schema=None*, ***kwargs*)
Initializer for ActionJsonnet instance.

Parameters

- **ext_vars** (Optional[str]) – Key where to find the external variables required to parse the jsonnet.
- **schema** (Union[str, Dict, None]) – Schema to validate values against.

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** (Optional[Dict[str, Any]]) – 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** (Optional[Dict[str, Any]]) – External variables. Values can be strings or any other basic type.
- **with_meta** (bool) – Whether to include metadata in config object.

Return type Dict

Returns The parsed jsonnet object.

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

class jsonargparse.**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, cfg, values[, option_string])	Parses the given configuration and adds all the corresponding keys to the namespace.

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

__call__(*parser*, *cfg*, *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**(*yes_prefix=''*, *no_prefix='no_'*, ***kwargs*)
Bases: argparse.Action

Paired options –[yes_prefix]opt, –[no_prefix]opt to set True or False respectively.

Methods:

__init__([yes_prefix, no_prefix])	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__(*yes_prefix=''*, *no_prefix='no_'*, ***kwargs*)
Initializer for ActionYesNo instance.

Parameters

- **yes_prefix** (str) – Prefix for yes option.
- **no_prefix** (str) – Prefix for no option.

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.**ActionParser**(*parser=None*)
Bases: object

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

Methods:

__init__([parser])	Initializer for ActionParser instance.

__init__(*parser=None*)
Initializer for ActionParser instance.

Parameters **parser** (None) – A parser to parse the option with.

Raises ValueError – If the parser parameter is invalid.

class jsonargparse.**ActionPathList**(*mode=None*, *rel='cwd'*, ***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__([mode, rel])	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__(*mode=None*, *rel='cwd'*, ***kwargs*)
Initializer for ActionPathList instance.

Parameters

- **mode** (Optional[str]) – The required type and access permissions among [fdrwxcuFDRWX] as a keyword argument (uppercase means not), e.g. ActionPathList(mode='fr').

- **rel** (str) – Whether relative paths are with respect to current working directory 'cwd' or the list's parent directory 'list'.

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.**Namespace**(**args*, ***kwargs*)
Bases: argparse.Namespace

Extension of argparse's Namespace to support nesting and subscript access.

Methods:

__init__(*args, **kwargs)	

as_dict()	Converts the nested namespaces into nested dictionaries.
as_flat()	Converts the nested namespaces into a single argparse flat namespace.
items()	Returns a generator of all nested (key, value) items.
keys()	Returns a generator of all nested keys.
values()	Returns a generator of all nested values.
get_sorted_keys([branches, key_filter])	Returns a list of keys sorted by descending depth.
clone()	Creates an new identical nested namespace.
update(value[, key, only_unset])	Sets or replaces all items from the given nested namespace.

__init__(**args*, ***kwargs*)

as_dict()
Converts the nested namespaces into nested dictionaries.

Return type Dict[str, Any]

as_flat()
Converts the nested namespaces into a single argparse flat namespace.

Return type Namespace

items()
Returns a generator of all nested (key, value) items.

Return type Iterator[Tuple[str, Any]]

keys()
Returns a generator of all nested keys.

Return type Iterator[str]

values()
Returns a generator of all nested values.

Return type Iterator[Any]

get_sorted_keys(*branches=True*, *key_filter=<function is_meta_key>*)
Returns a list of keys sorted by descending depth.

Parameters

- **branches** (bool) – Whether to include branch keys instead of only leaves.
- **key_filter** (Callable) – Function that selects keys to exclude.

Return type List[str]

clone()
Creates an new identical nested namespace.

Return type *Namespace*

update(*value*, *key=None*, *only_unset=False*)
Sets or replaces all items from the given nested namespace.

Parameters

- **value** (Union[*Namespace*, Any]) – A namespace to update multiple values or other type to set in a single key.
- **key** (Optional[str]) – Branch key where to set the value. Required if value is not namepace.
- **only_unset** (bool) – Whether to only set the value if not set in namespace.

Return type *Namespace*

jsonargparse.**namespace_to_dict**(*namespace*)
Returns a deepcopy of a nested namespace converted into a nested dictionary.

Return type Dict[str, Any]

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

Return type *Namespace*

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

Parameters **cfg** – The configuration object to strip.

Returns A deepcopy of the configuration object excluding all metadata keys.

class jsonargparse.**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:'.

Methods:

add_yaml_comments(cfg)	Adds help text as yaml comments.
set_yaml_start_comment(text, cfg)	Sets the start comment to a ruyaml object.
set_yaml_group_comment(text, cfg, key, depth)	Sets the comment for a group to a ruyaml object.
set_yaml_argument_comment(text, cfg, key, depth)	Sets the comment for an argument to a ruyaml object.

add_yaml_comments(*cfg*)
Adds help text as yaml comments.

Return type str

set_yaml_start_comment(*text*, *cfg*)
Sets the start comment to a ruyaml object.

Parameters

- **text** (str) – The content to use for the comment.
- **cfg** (None) – The ruyaml object.

set_yaml_group_comment(*text*, *cfg*, *key*, *depth*)
Sets the comment for a group to a ruyaml object.

Parameters

- **text** (str) – The content to use for the comment.
- **cfg** (None) – The parent ruyaml object.
- **key** (str) – The key of the group.
- **depth** (int) – The nested level of the group.

set_yaml_argument_comment(*text*, *cfg*, *key*, *depth*)
Sets the comment for an argument to a ruyaml object.

Parameters

- **text** (str) – The content to use for the comment.
- **cfg** (None) – The parent ruyaml object.
- **key** (str) – The key of the argument.
- **depth** (int) – The nested level of the argument.

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

Return type str

jsonargparse.**set_config_read_mode**(*urls_enabled=False*, *fsspec_enabled=False*)
Enables/disables optional config read modes.

Parameters

- **urls_enabled** (bool) – Whether to read config files from URLs using requests package.
- **fsspec_enabled** (bool) – Whether to read config files from fsspec supported file systems.

exception jsonargparse.**ParserError**
Bases: Exception

Error raised when parsing a value fails.

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

Parameters

- **parser** (None) – The parser object.
- **message** (str) – The message describing the error being handled.

Return type None

class jsonargparse.**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, s=fsspec 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.**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.

class jsonargparse.**ActionEnum**(***kwargs*)
Bases: `object`

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

> **Warning:** ActionEnum was deprecated in v3.9.0 and will be removed in v5.0.0. Enums now should be given directly as a type as explained in *Enum arguments*.

Methods:

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

__init__(***kwargs*)

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

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

Action to restrict a value with comparison operators.

> **Warning:** ActionOperators was deprecated in v3.0.0 and will be removed in v5.0.0. Now types should be used as explained in *Restricted numbers*.

Methods:

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

__init__(***kwargs*)

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

class jsonargparse.**ActionPath**(*mode*, *skip_check=False*)
Bases: object

Action to check and store a path.

> **Warning:** ActionPath was deprecated in v3.11.0 and will be removed in v5.0.0. Paths now should be given directly as a type as explained in *Parsing paths*.

Methods:

__init__(mode[, skip_check])	
__call__(*args, **kwargs)	Call self as a function.

__init__(*mode*, *skip_check=False*)

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

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

> **Warning:** set_url_support was deprecated in v3.12.0 and will be removed in v5.0.0. Optional config read modes should now be set using function set_config_read_mode.

26.2 jsonargparse.typing

Collection of types and type generators.

Functions:

final(cls)	Decorator to make a class *final* meaning that it shouldn't be subclassed.
is_final_class(cls)	Checks whether a class was decorated as *final*.
register_type(type_class[, serializer, ...])	Registers a new type for use in jsonargparse parsers.
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, skip_check])	Creates or returns an already registered path type class.
import_object(name)	Returns an object in a module given its dot import path.

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

continues on next page

Table 24 – continued from previous page

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
Path_dc(v)	str pointing to a directory that can be created if it does not exist
Path_drw(v)	str pointing to a directory that exists and is readable and writeable
Path(path[, mode, cwd, skip_check])	Stores a (possibly relative) path and the corresponding absolute path.
timedelta	Difference between two datetime values.

jsonargparse.typing.**final**(*cls*)
Decorator to make a class *final* meaning that it shouldn't be subclassed.

jsonargparse.typing.**is_final_class**(*cls*)
Checks whether a class was decorated as *final*.

jsonargparse.typing.**register_type**(*type_class*, *serializer=<class 'str'>*, *deserializer=None*, *deserializer_exceptions=(<class 'ValueError'>, <class 'TypeError'>, <class 'AttributeError'>)*, *type_check=<function <lambda>>*, *uniqueness_key=None*)
Registers a new type for use in jsonargparse parsers.

Parameters

- **type_class** (Any) – The type object to be registered.
- **serializer** (Callable) – Function that converts an instance of the class to a basic type.
- **deserializer** (Optional[Callable]) – Function that converts a basic type to an instance of the class. Default None instantiates type_class.
- **deserializer_exceptions** (Union[Type[Exception], Tuple[Type[Exception], ...]]) – Exceptions that deserializer raises when it fails.
- **type_check** (Callable) – Function to check if a value is of type_class. Gets as arguments the value and type_class.
- **uniqueness_key** (Optional[Tuple]) – Key to determine uniqueness of type.

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*, *skip_check=False*)
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.
- **skip_check** (bool) – Whether to skip path checks.

Return type Type

Returns The created or retrieved type class.

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

int restricted to be >0

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

int restricted to be 0

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

float restricted to be >0

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

float restricted to be 0

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

float restricted to be 0 and 1

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

float restricted to be >0 and <1

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

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

class jsonargparse.typing.**Email**(*v*)
Bases: jsonargparse.typing.create_type.<locals>.TypeCore, 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

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

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

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

str pointing to a directory that exists and is readable and writeable

CHAPTER

TWENTYSEVEN

LICENSE

CHAPTER

TWENTYEIGHT

INDEX

- changelog
- genindex

PYTHON MODULE INDEX

j

INDEX

Symbols

A

C

D

U

V