jsonargparse Documentation

Mauricio Villegas

Jul 27, 2023

CONTENTS

Docs: https://jsonargparse.readthedocs.io/

Source: 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 hints for automatic creation of parsers and minimal boilerplate command line interfaces.
- Non-intrusive/decoupled parsing logic. No need to inherit from a special class or add decorators or use custom type hints.
- Easy to implement configurable dependency injection (object composition).
- Support for nested namespaces making 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.

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
Classes, methods and functions					✓
Parsing URLs	✓				
Json schemas				✓	
Jsonnet files			✓		
Tab completion		✓			

CHAPTER

THREE

BASIC USAGE

There are multiple ways of using jsonargparse. One is to construct low level parsers (see *Parsers*) being almost a drop in replacement of argparse. However, argparse is too verbose and leads to unnecessary duplication. The simplest and recommended way of using jsonargparse is by using the *CLI()* function, which has the benefit of minimizing boilerplate code. A simple example is:

```
from jsonargparse import CLI


def command(name: str, prize: int = 100):
    """Prints the prize won by a person.

    Args:
        name: Name of winner.
        prize: Amount won.
    """
    print(f"{name} won {prize}€!")


if __name__ == "__main__":
    CLI(command)
```

Note that the `name` and `prize` parameters have type hints and are described in the docstring. These are shown in the help of the command line tool. In a shell you could see the help and run a command as follows:

```
$ python example.py --help
...
Prints the prize won by a person:
  name                  Name of winner. (required, type: str)
  --prize PRIZE         Amount won. (type: int, default: 100)

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

Note: Parsing of docstrings is an optional feature. For this example to work as shown, jsonargparse needs to be installed with the `signatures` extras require as explained in section *Installation*.

When *CLI()* receives a single class, the first arguments are for parameters to instantiate the class, then a method name is expected (i.e. methods become *Sub-commands*) and the remaining arguments are for parameters of this method. An example would be:

```python
from random import randint
from jsonargparse import CLI


class Main:
    def __init__(self, max_prize: int = 100):
        """
        Args:
            max_prize: Maximum prize that can be awarded.
        """
        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 the class given does not have any methods, there will be no sub-commands and *CLI()* will return an instance of the class. For example:

```python
from dataclasses import dataclass
from jsonargparse import CLI


@dataclass
class Settings:
    name: str
    prize: int = 100


if __name__ == "__main__":
    print(CLI(Settings, as_positional=False))
```

Then in a shell you could run:

```
$ python example.py --name=Lucky
Settings(name='Lucky', prize=100)
```

Note the use of `as_positional=False` to make required arguments as non-positional.

If more than one function is given to *CLI()*, then any of them can be run 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]`.

Note: The examples above are extremely simple, only defining parameters with `str` and `int` type hints. The true power of jsonargparse is its support for a wide range of types, see *Type hints*. It is even possible to use general classes as type hints, allowing to easily implement configurable dependency injection (object composition), see *Class type and sub-classes*.

3.1 Writing configuration files

All tools implemented with the *CLI()* function have the `--config` option to provide settings in a config file (more details in *Configuration files*). This becomes very useful when the number of configurable parameters is large. To ease the writing of config files, there is also the option `--print_config` which prints to standard output a yaml with all settings that the tool supports with their default values. Users of the tool can be advised to follow the following steps:

```
# Dump default configuration to have as reference
python example.py --print_config > config.yaml
# Modify the config as needed (all default settings can be removed)
nano config.yaml
# Run the tool using the adapted config
python example.py --config config.yaml
```

3.2 Comparison to Fire

The *CLI()* feature is similar to 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 expected to have type hints, and the given values will be validated according to these. Third, the return values of the functions are not automatically printed. *CLI()* returns the value and it is up to the developer to decide what to do with it.

CHAPTER

FOUR

TUTORIALS

- "jsonargparse - Say goodbye to configuration hassles" by Marianne Stecklina at PyCon DE & PyData Berlin 2022
  - Presentation video: https://youtu.be/2gDf2S0nHKg
  - GitHub repository: https://github.com/stecklin/pycon22-jsonargparse

CHAPTER

FIVE

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 `exit_on_error=False` in which case an *ArgumentError* is raised.

5.1 Override order

Final parsed values depend on different sources, namely: source code, command line arguments, *Configuration files* and *Environment variables*. Values are overridden based on the following precedence:

1. Defaults defined in the source code.
2. Existing default config files in the order defined in `default_config_files`, e.g. `~/.config/myapp.yaml`.
3. Full config environment variable, e.g. `APP_CONFIG`.
4. Individual key environment variables, e.g. `APP_OPT1`.
5. Command line arguments in order left to right (might include config files).

Depending on the parse method used (see *ArgumentParser*) and how the parser was built, some of the options above might not apply. Parsing of environment variables must be explicitly enabled, except if using *ArgumentParser.parse_env()*. If the parser does not have an *ActionConfigFile* argument, then there is no parsing of a full config environment variable or a way to provide a config file from command line.

5.2 Capturing parsers

It can be common practice to have a function that implements an entire CLI or a function that constructs a parser conditionally based on some parameters and then parses. For example, one might have:

```
from jsonargparse import ArgumentParser


def main_cli():
    parser = ArgumentParser()
    ...
    cfg = parser.parse_args()
    ...


if __name__ == "__main__":
    main_cli()
```

For some use cases it is necessary to get an instance of the parser object, without doing any parsing. For instance sphinx-argparse can be used to include the help of CLIs in automatically generated documentation of a package. To use sphinx-argparse it is necessary to have a function that returns the parser. Having a CLI function this could be easily implemented with *capture_parser()* as follows:

```
from jsonargparse import capture_parser


def get_parser():
    return capture_parser(main_cli)
```

Note: The official way to obtain the parser for command line tools based on *CLI()* is by using *capture_parser()*.

CHAPTER

SIX

TYPE HINTS

An important feature of jsonargparse is a wide support the argument types and respective validation. This extended support makes use of Python's type hint syntax. 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 types in *jsonargparse.typing* are included for convenience since they are useful in argument parsing use cases and not available in standard python. However, there is no need to use jsonargparse specific types.

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. By nesting it is meant child types inside `List`, `Dict`, etc. There is no limit in nesting depth.
- Postponed evaluation of types PEP 563 (i.e. `from __future__ import annotations`) is supported. Also supported on `python<=3.9` are PEP 585 (i.e. `list[<type>], dict[<type>], ...` instead of `List[<type>], Dict[<type>], ...`) and 604 (i.e. `<type> | <type>` instead of `Union[<type>, <type>]`).
- Fully supported types are: `str`, `bool` (more details in *Booleans*), `int`, `float`, `complex`, `bytes`/`bytearray` (Base64 encoding), `range`, `List` (more details in *List append*), `Iterable`, `Sequence`, `Any`, `Union`, `Optional`, `Type`, `Enum`, `PathLike`, `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. For more details see *Dict items*.
- `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.
- 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`.
- Normal classes can be used as a type, which are specified with a dict containing `class_path` and optionally `init_args`. *ArgumentParser.instantiate_classes()* can be used to instantiate all classes in a config object. For more details see *Class type and sub-classes*.
- `dataclasses` are supported even when nested. Final classes, attrs' `define` decorator, and pydantic's `dataclass` decorator and `BaseModel` classes are supported and behave like standard dataclasses. For more

details see *Dataclass-like classes*. If a dataclass is mixed inheriting from a normal class, it is considered a subclass type instead of a dataclass.

- Pydantic types are supported. There might be edge cases which don't work as expected. Please report any encountered issues.
- `Callable` is supported by either giving a dot import path to a callable object or by giving a dict with a `class_path` and optionally `init_args` entries. The specified class must either instantiate into a callable or be a subclass of the return type of the callable. For these cases running *`ArgumentParser.instantiate_classes()`* will instantiate the class or provide a function that returns the instance of the class. For more details see *Callable type*. Currently the callable's argument and return types are not validated.

6.1 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)
```

6.2 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)
```

6.3 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*).

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

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

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

```
from jsonargparse import ArgumentParser
from jsonargparse.typing import Path_fr

parser = ArgumentParser()
parser.add_argument("--databases.info", type=Path_fr)
cfg = parser.parse_path("app/config.yaml")
```

The `fr` in the type are flags 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 add the argument with type `List[<path_type>]` and `enable_path=True`. To read from stdin give the special string `'-'`. Example:

```
from jsonargparse.typing import Path_fr

parser.add_argument("--list", type=List[Path_fr], enable_path=True)
cfg = parser.parse_args(["--list", "paths.lst"])  # File with list of paths
cfg = parser.parse_args(["--list", "-"])  # List of paths from stdin
```

If `nargs='+'` is given to `add_argument` with `List[<path_type>]` and `enable_path=True` then for each argument a list of paths is generated.

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

6.4 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 *requests* python package is 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, 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()* method Scan 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
```

Note: Relative paths inside a remote path are parsed as remote. For example, for a relative path `model/state_dict.pt` found inside `s3://bucket/config.yaml`, its parsed absolute path becomes `s3://bucket/model/state_dict.pt`.

6.5 Booleans

Parsing boolean arguments is very common, however, the original argparse only has a limited support for them, via `store_true` and `store_false`. Furthermore 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'}`.

6.6 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>)
```

6.7 List append

As detailed before, arguments with `List` type are supported. By default when specifying an argument value, the previous value is replaced, and this also holds for lists. Thus, a parse such as `parser.parse_args(['--list=[1]', '--list=[2, 3]'])` would result in a final value of `[2, 3]`. However, in some cases it might be decided to append to the list instead of replacing. This can be achieved by adding `+` as suffix to the argument key, for example:

```
>>> parser.add_argument("--list", type=List[int])
>>> parser.parse_args(["--list=[1]", "--list+=[2, 3]"])
Namespace(list=[1, 2, 3])
>>> parser.parse_args(["--list=[4]", "--list+=5"])
Namespace(list=[4, 5])
```

Append is also supported in config files. For instance the following two config files would first assign a list and then append to this list:

```
# config1.yaml
list:
- 1
```

```
# config2.yaml
list+:
```

(continues on next page)

(continued from previous page)

```
- 2
- 3
```

Appending works for any type for the list elements. Lists with class type elements (see *Class type and sub-classes*) are also supported. To append to the list, first append a new class by using the + suffix. Then `init_args` for this class are specified like if the type wasn't a list, since the arguments are applied to the last class in the list. Take for example that an argument is added to a parser as:

```
parser.add_argument("--list_of_instances", type=List[MyBaseClass])
```

Thanks to the short notation, command line arguments don't require to specify `class_path` and `init_args`. Thus, multiple classes can be appended and its arguments set as follows:

```
python tool.py \
  --list_of_instances+={CLASS_1_PATH} \
  --list_of_instances.{CLASS_1_ARG_1}=... \
  --list_of_instances.{CLASS_1_ARG_2}=... \
  --list_of_instances+={CLASS_2_PATH} \
  --list_of_instances.{CLASS_2_ARG_1}=... \
  ...
  --list_of_instances+={CLASS_N_PATH} \
  --list_of_instances.{CLASS_N_ARG_1}=... \
  ...
```

Once a new class has been appended to the list, it is not possible to modify the arguments of a previous class. This limitation is by intention since it forces classes and its arguments to be defined in order, making the command line call intuitive to write and understand.

6.8 Dict items

When an argument has `Dict` as type, the value can be set using json format, e.g.:

```
>>> parser.add_argument("--dict", type=dict)
>>> parser.parse_args(['--dict={"key1": "val1", "key2": "val2"}'])
Namespace(dict={'key1': 'val1', 'key2': 'val2'})
```

Similar to lists, providing a second argument with value a json dict completely replaces the previous value. Setting individual dict items without replacing can be achieved as follows:

```
>>> parser.parse_args(["--dict.key1=val1", "--dict.key2=val2"])
Namespace(dict={'key1': 'val1', 'key2': 'val2'})
```

6.9 Dataclass-like classes

In contrast to subclasses, which requires the user to provide a `class_path`, in some cases it is not expected to have subclasses. In this case the init args are given directly in a dictionary without specifying a `class_path`. This is the behavior for standard `dataclasses`, `final` classes, attrs' `define` decorator, and pydantic's `dataclass` decorator and `BaseModel` classes.

As an example, take a class that is decorated with *`final()`*, meaning that it shouldn't be subclassed. The code below would accept the corresponding yaml structure.

```
from jsonargparse.typing import final


@final
class FinalClass:
    def __init__(self, number: int = 0, accepted: bool = False):
        ...


parser = ArgumentParser()
parser.add_argument("--data", type=FinalClass)
cfg = parser.parse_path("config.yaml")
```

```
data:
  number: 8
  accepted: true
```

6.10 Callable type

When using `Callable` as type, the parser accepts several options. The first option is the import path of a callable object, for example:

```
parser.add_argument("--callable", type=Callable)
parser.parse_args(["--callable=time.sleep"])
```

A second option is a class that once instantiated becomes callable:

```
class OffsetSum:
    def __init__(self, offset: int):
        self.offset = offset

    def __call__(self, value: int):
        return self.offset + value
```

```
>>> value = {
...     "class_path": "__main__.OffsetSum",
...     "init_args": {
...         "offset": 3,
...     },
... }
```

(continues on next page)

(continued from previous page)

```
>>> cfg = parser.parse_args(["--callable", str(value)])
>>> cfg.callable
Namespace(class_path='__main__.OffsetSum', init_args=Namespace(offset=3))
>>> init = parser.instantiate_classes(cfg)
>>> init.callable(5)
8
```

The third option is only applicable when the type is a callable that has a class as return type or a `Union` including a class. This is useful to support dependency injection for classes that require a parameter that is only available after injection. The parser supports this automatically by providing a function that receives this parameter and returns the instance of the class. Take for example the classes:

```
class Optimizer:
    def __init__(self, params: Iterable):
        self.params = params


class SGD(Optimizer):
    def __init__(self, params: Iterable, lr: float):
        super().__init__(params)
        self.lr = lr
```

A possible parser and callable behavior would be:

```
>>> value = {
...     "class_path": "SGD",
...     "init_args": {
...         "lr": 0.01,
...     },
... }

>>> parser.add_argument("--optimizer", type=Callable[[Iterable], Optimizer])
>>> cfg = parser.parse_args(["--optimizer", str(value)])
>>> cfg.optimizer
Namespace(class_path='__main__.SGD', init_args=Namespace(lr=0.01))
>>> init = parser.instantiate_classes(cfg)
>>> optimizer = init.optimizer([1, 2, 3])
>>> isinstance(optimizer, SGD)
True
>>> optimizer.params, optimizer.lr
([1, 2, 3], 0.01)
```

Note: When the `Callable` has a class return type, it is possible to specify the `class_path` giving only its name if imported before parsing, as explained in *Command line*.

If the same type above is used as type hint of a parameter of another class, a default can be set using a lambda, for example:

```
class Model:
    def __init__(
        self,
```

(continues on next page)

(continued from previous page)

```
        optimizer: Callable[[Iterable], Optimizer] = lambda p: SGD(p, lr=0.05),
    ):
        self.optimizer = optimizer
```

Then a parser and behavior could be:

```
>>> parser.add_class_arguments(Model, 'model')
>>> cfg = parser.get_defaults()
>>> cfg.model.optimizer
Namespace(class_path='__main__.SGD', init_args=Namespace(lr=0.05))
>>> init = parser.instantiate_classes(cfg)
>>> optimizer = init.model.optimizer([1, 2, 3])
>>> optimizer.params, optimizer.lr
([1, 2, 3], 0.05)
```

See *AST resolver* for limitations of lambda defaults.

6.11 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

SEVEN

NESTED NAMESPACES

A difference with respect to basic argparse is, that 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 is an extension of the one from argparse, having some additional features. 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. This is useful for example for class instantiation.

CHAPTER

EIGHT

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 the given order. The default config files are always parsed first, this means that any command line argument 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:

```yaml
# 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.

8.1 Serialization

Parsers that have an *ActionConfigFile* argument 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()* methods. Three formats with a particular style are supported: `yaml`, `json` and `json_indented`. It is possible to add more dumping formats by using the *set_dumper()* function. For example to allow dumping using PyYAML's `default_flow_style` do the following:

```
import yaml
from jsonargparse import set_dumper


def custom_yaml_dump(data):
    return yaml.safe_dump(data, default_flow_style=True)


set_dumper("yaml_custom", custom_yaml_dump)
```

8.2 Custom loaders

The `yaml` parser mode (see *ArgumentParser.__init__()*) uses for loading a subclass of yaml.SafeLoader with two modifications. First, it supports float's scientific notation, e.g. `'1e-3' => 0.001` (unlike default PyYAML which considers `'1e-3'` a string). Second, text within curly braces is considered a string, e.g. `'{text}' (unlike default PyYAML which parses this as ``{'text': None}`).

It is possible to replace the yaml loader or add a loader as a new parser mode via the *set_loader()* function. For example if you need a custom PyYAML loader it can be registered and used as follows:

```
import yaml
from jsonargparse import ArgumentParser, set_loader


class CustomLoader(yaml.SafeLoader):
    ...


def custom_yaml_load(stream):
    return yaml.load(stream, Loader=CustomLoader)


```

(continues on next page)

(continued from previous page)

```
set_loader("yaml_custom", custom_yaml_load)

parser = ArgumentParser(parser_mode="yaml_custom")
```

When setting a loader based on a library different from PyYAML, the `exceptions` that it raises when there are failures should be given to *set_loader()*.

CHAPTER

NINE

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, see *SignatureArguments*.

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.
        """
        super().__init__(**kwargs)
        ...

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

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

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, sets it as required since it lacks a default value. When parsed, it is validated according to the 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 several classes added 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. For convenience it is also possible that the values for each of the argument groups created by the calls to add signatures 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 include the settings for the instantiation of the class and the call to the method respectively.

A wide range of type hints are supported for the signature parameters. For exact details go to section *Type hints*. Some notes about the add signature methods are:

- All positional only parameters must have a type, otherwise the add arguments functions raise an exception.
- Keyword parameters are ignored if they don't have at least one type that is supported.
- Parameters whose name starts with _ are considered internal 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: The signatures support is intended to be non-intrusive. It is by design that there is no need to inherit from a class, add decorators, or use special type hints and default values. This has several advantages. For example it is possible to use classes from third party libraries which is not possible for developers to modify.

9.1 Docstring parsing

To get parameter docstrings in the parser help, the docstring-parser package is required. This package is included when installing jsonargparse with the `signatures` extras require as explained in section *Installation*.

A couple of options can be configured, both related to docstring parsing speed. By default docstrings are parsed used with `docstring_parser.DocstringStyle.AUTO`, which means that it is attempted to parse docstrings with all supported styles. If the relevant codebase uses a single style, this is inefficient. A single style can be configured as follows:

```
from docstring_parser import DocstringStyle
from jsonargparse import set_docstring_parse_options

set_docstring_parse_options(style=DocstringStyle.REST)
```

The second option that can be configured is the support for attribute docstrings (i.e. literal strings in the line after an attribute is defined). By default this feature is disabled and enabling it makes the parsing slower even for classes that don't have attribute docstrings. To enable, do as follows:

```
from dataclasses import dataclass
from jsonargparse import set_docstring_parse_options

set_docstring_parse_options(attribute_docstrings=True)


@dataclass
class Options:
    """Options for a competition winner."""

    name: str
    """Name of winner."""
    prize: int = 100
    """Amount won."""
```

9.2 Classes from functions

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 above 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")
```

Note: *class_from_function()* requires the input function to have a return type annotation that must be the class type it returns.

Classes created with *class_from_function()* can be selected using `class_path` for *Class type and sub-classes*. For example, if *class_from_function()* is run in a module `my_module` as:

```
class_from_function(instantiate_myclass, name="MyClass")
```

Then the `class_path` for the created class would be `my_module.MyClass`.

9.3 Parameter resolvers

Three techniques are implemented for resolving signature parameters. One makes use of python's Abstract Syntax Trees (AST) library and the second is based on assumptions of class inheritance. The AST resolver is used first and only when AST fails, the assumptions resolver is run as fallback. The third resolver uses stub files `*.pyi` and is applied on top of both the AST and assumptions resolvers.

9.3.1 Unresolved parameters

The parameter resolvers make a best effort to determine the correct names and types that the parser should accept. However, there can be cases not yet supported or cases for which it would be impossible to support. To somewhat overcome these limitations, there is a special key `dict_kwargs` that can be used to provide arguments that will not be validated during parsing, but will be used for class instantiation. It is called `dict_kwargs` because there are use cases in which `**kwargs` is used just as a dict, thus it also serves that purpose.

Take for example the following parsing and instantiation:

```
from jsonargparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument("--myclass", type=MyClass)
cfg = parser.parse_args()
cfg_init = parser.instantiate_classes(cfg)
```

If `MyClass.__init__` has `**kwargs` with some unresolved parameters, the following could be a valid config file:

```
class_path: MyClass
init_args:
  foo: 1
dict_kwargs:
  bar: 2
```

The value for `bar` will not be validated, but the class will be instantiated as `MyClass(foo=1, bar=2)`.

9.3.2 Assumptions resolver

The assumptions resolver only considers classes. Whenever the `__init__` method has `*args` and/or `**kwargs`, the resolver assumes that these are directly forwarded to the next parent class, i.e. `__init__` includes a line like `super().__init__(*args, **kwargs)`. Thus, it blindly collects the `__init__` parameters of parent classes. The collected parameters will be incorrect if the code does not follow this pattern. This is why it is only used as fallback when the AST resolver fails.

9.3.3 AST resolver

The AST resolver analyzes the source code and tries to figure out how the `*args` and `**kwargs` are used to further find more accepted parameters. This type of resolving is limited to a few specific cases since there are endless possibilities for what code can do. The supported cases are illustrated below. Bear in mind that the code does not need to be exactly like this. The important detail is how `*args` and `**kwargs` are used, not other parameters, or the names of variables, or the complexity of the code that is unrelated to these variables.

Cases for statements in functions or methods

```
def calls_a_function(*args, **kwargs):
    a_function(*args, **kwargs)


def calls_a_method(*args, **kwargs):
    an_instance = SomeClass()
    an_instance.a_method(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
def calls_a_static_method(*args, **kwargs):
    an_instance = SomeClass()
    an_instance.a_static_method(*args, **kwargs)


def calls_a_class_method(*args, **kwargs):
    SomeClass.a_class_method(*args, **kwargs)


def pops_from_kwargs(**kwargs):
    val = kwargs.pop("name", "default")


def gets_from_kwargs(**kwargs):
    val = kwargs.get("name", "default")


def constant_conditional(**kwargs):
    if global_boolean_1:
        first_function(**kwargs)
    elif not global_boolean_2:
        second_function(**kwargs)
    else:
        third_function(**kwargs)
```

Cases for classes

```
class PassThrough(BaseClass):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)


class CallMethod:
    def __init__(self, *args, **kwargs):
        self.a_method(*args, **kwargs)


class AttributeUseInMethod:
    def __init__(self, **kwargs):
        self._kwargs = kwargs

    def a_method(self):
        a_callable(**self._kwargs)


class AttributeUseInProperty:
    def __init__(self, **kwargs):
        self._kwargs = kwargs

    @property
    def a_property(self):
```

(continues on next page)

(continued from previous page)

```
        return a_callable(**self._kwargs)


class DictUpdateUseInMethod:
    def __init__(self, **kwargs):
        self._kwargs = dict(p1=1)  # Can also be: self._kwargs = {'p1': 1}
        self._kwargs.update(**kwargs)  # Can also be: self._kwargs = dict(p1=1, **kwargs)

    def a_method(self):
        a_callable(**self._kwargs)


class InstanceInClassmethod:
    @classmethod
    def get_instance(cls, **kwargs):
        return cls(**kwargs)


class NonImmediateSuper(BaseClass):
    def __init__(self, *args, **kwargs):
        super(BaseClass, self).__init__(*args, **kwargs)
```

Cases for class instance defaults

```
# Class instance: only keyword arguments with ``ast.Constant` value
class_instance: SomeClass = SomeClass(param=1)

# Lambda returning class instance: only keyword arguments with ``ast.Constant` value
class_instance: Callable[[type], BaseClass] = lambda a: ChildClass(a, param=2.3)
```

There can be other parameters apart from `*args` and `**kwargs`, thus in the cases above, the signatures can be for example like `name(p1: int, k1: str = 'a', **kws)`. Also when internally calling some function or instantiating a class, there can be additional parameters. For example in:

```
def calls_a_function(*args, **kwargs):
    a_function(*args, param=1, **kwargs)
```

The `param` parameter would be excluded from the resolved parameters because it is internally hard coded.

A special case which is supported but with caveats, is multiple calls that use `**kwargs`. For example:

```
def conditional_calls(**kwargs):
    if condition_1:
        first_function(**kwargs)
    elif condition_2:
        second_function(**kwargs)
    else:
        third_function(**kwargs)
```

The resolved parameters that have the same type hint and default accross all calls are supported normally. When there is a discrepancy between the calls, the parameters behave differently and are shown in the help in special "Conditional arguments" sections. The main difference is that these arguments are not included in *ArgumentParser.get_defaults()* or the output of `--print_config`. This is necessary because the parser does not know which of

the calls will be used at runtime, and adding them would cause *ArgumentParser.instantiate_classes()* to fail due to unexpected keyword arguments.

Note: The parameter resolvers log messages of failures and unsupported cases. To view these logs, set the environment variable `JSONARGPARSE_DEBUG` to any value. The supported cases are limited and it is highly encouraged that people create issues requesting the support for new ones. However, note that when a case is highly convoluted it could be a symptom that the respective code is in need of refactoring.

9.3.4 Stubs resolver

The stubs resolver makes use of the typeshed-client package to identify parameters and their type hints from stub files `*.pyi`. To enable this resolver, install jsonargparse with the `signatures` extras require as explained in section *Installation*.

Many of the types defined in stub files use the latest syntax for type hints, that is, bitwise or operator `|` for unions and generics, e.g. `list[<type>]` instead of `typing.List[<type>]`, see PEPs 604 and 585. On python>=3.10 these are fully supported. On python<=3.9 backporting these types is attempted and in some cases it can fail. On failure the type annotation is set to `Any`.

Most of the types in the Python standard library have their types in stubs. An example from the standard library would be:

```
>>> from random import uniform

>>> parser = ArgumentParser()
>>> parser.add_function_arguments(uniform, "uniform")
>>> parser.parse_args(["--uniform.a=0.7", "--uniform.b=3.4"])
Namespace(uniform=Namespace(a=0.7, b=3.4))
```

Without the stubs resolver, the *SignatureArguments.add_function_arguments()* call requires the `fail_untyped=False` option. This has the disadvantage that type `Any` is given to the `a` and `b` arguments, instead of `float`. And this means that the parser would not fail if given an invalid value, for instance a string.

CHAPTER

TEN

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 set of init parameters would also work.

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.

Note: Classes will be parsed and instantiated when given as value a dict with `class_path` and `init_args` if the corresponding parameter has type `Any`, or when `fail_untyped=False` which defaults to type `Any`.

10.1 Command line

The help of the parser does not show details for a type class since this depends on the subclass. To get details for a particular subclass there is a specific help option that receives the import path. Take for example a parser defined as:

```
from calendar import Calendar
from jsonargparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument("--calendar", type=Calendar)
```

The help for a corresponding subclass could be printed as:

```
python tool.py --calendar.help calendar.TextCalendar
```

In the command line, a subclass can be specified through multiple command line arguments:

```
python tool.py \
  --calendar.class_path calendar.TextCalendar \
  --calendar.init_args.firstweekday 1
```

For convenience, the arguments can be somewhat shorter by omitting `.class_path` and `.init_args` and only specifying the name of the subclass instead of the full import path.

```
python tool.py --calendar TextCalendar --calendar.firstweekday 1
```

Specifying the name of the subclass works for subclasses in modules that have been imported before parsing. Abstract classes and private classes (module or name starting with `'_'`) are not considered. All the subclasses resolvable by its name can be seen in the general help `python tool.py --help`.

10.2 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. An example is:

```
class MyClass:
    def __init__(
        self,
```

(continues on next page)

(continued from previous page)

```
        calendar: Calendar = Calendar(firstweekday=1),
    ):
        self.calendar = calendar
```

Adding this class to a parser will work without issues. The *AST resolver* in limited cases determines how to instantiate the original default. 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.

Since the *AST resolver* only supports limited cases, or when the source code is not available, a second approach 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
```

Like this, the parsed default will be a dict with `class_path` and `init_args`, again avoiding the risk of mutability.

Note: In python there can be some classes or functions for which it is not possible to determine its import path from the object alone. When using one of these as a default would cause a failure when serializing because what gets saved in the config file is the import path. To overcome this problem use the *`register_unresolvable_import_paths()`* function giving it the module from where the respective object can be imported.

CHAPTER

ELEVEN

ARGUMENT LINKING

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

There are two types of links, defined with `apply_on='parse'` or `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, sources can be class groups (added with *SignatureArguments.add_class_arguments()*) or subclass arguments (see *Class type and sub-classes*). The source key can be the entire instantiated object or an attribute of the object. 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 set of all 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:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        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

TWELVE

VARIABLE INTERPOLATION

One of the possible reasons to add a parser mode (see *Custom loaders*) can be to have support for variable interpolation in yaml files. Any library could be used to implement a loader and configure a mode for it. Without needing to implement a loader function, an `omegaconf` parser mode is available out of the box when this package is installed.

Take for example a yaml file as:

```yaml
server:
  host: localhost
  port: 80
client:
  url: http://${server.host}:${server.port}/
```

This yaml could be parsed as follows:

```
>>> @dataclass
... class ServerOptions:
...     host: str
...     port: int
...

>>> @dataclass
... class ClientOptions:
...     url: str
...

>>> parser = ArgumentParser(parser_mode="omegaconf")
>>> parser.add_argument("--server", type=ServerOptions)
>>> parser.add_argument("--client", type=ClientOptions)
>>> parser.add_argument("--config", action=ActionConfigFile)

>>> cfg = parser.parse_args(["--config=example.yaml"])
>>> cfg.client.url
'http://localhost:80/'
```

Note: The `parser_mode='omegaconf'` provides support for OmegaConf's variable interpolation in a single yaml file. It is not possible to do interpolation across multiple yaml files or in an isolated individual command line argument.

CHAPTER

THIRTEEN

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 or none if set to False) 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 parse environment variables. If `default_env` is left unset, environment variable parsing can also be enabled by setting in your shell `JSONARGPARSE_DEFAULT_ENV=true`.

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

FOURTEEN

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, subcommand='subcomm1', subcomm1=Namespace(op1='val1'))
>>> parser.parse_args(["--op0", "val0", "subcomm2", "--op2", "val2"])
Namespace(op0='val0', subcommand='subcomm2', subcomm2=Namespace(op2='val2'))
```

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

FIFTEEN

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

SIXTEEN

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

SEVENTEEN

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

EIGHTEEN

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

NINETEEN

TROUBLESHOOTING AND LOGGING

The standard behavior for the parse methods, when they fail, is to print a short message and terminate the process with a non-zero exit code. This is problematic during development since there is not enough information to track down the root of the problem. Without the need to change the source code, this default behavior can be changed such that in case of failure, a ParseError exception is raised and the full stack trace is printed. This is done by setting the `JSONARGPARSE_DEBUG` environment variable to any value.

The parsers from jsonargparse log some basic events, though by default this is disabled. To enable, 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. 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. If reconplogger is installed and the `JSONARGPARSE_DEBUG` environment variable is set, then the logging level becomes `DEBUG`.

CHAPTER

TWENTY

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 `pyproject.toml`. The basic runtime requirements are defined in `dependencies`. Requirements for optional features stored in `[project.optional-dependencies]`. Also in the same section there are requirements for testing, development and documentation building: `test`, `test-no-urls`, `dev` and `doc`.

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]"
```

Please also install the pre-commit git hook scripts so that unit tests and code checks are automatically run locally. This is done as follows:

```
pre-commit install
```

To build the documentation run:

```
sphinx-build sphinx sphinx/_build sphinx/index.rst
```

Then to see it, open the file `sphinx/_build/index.html` in a browser.

Running the unit tests can be done either using using pytest or tox. For convenience, the `setup.py` script can run the tests and create an html coverage report. The tests are also installed with the package, thus can be run in a production system.

```
tox                          # Run tests using tox on available python versions
pytest                       # Run tests using pytest on the python of the environment
pytest --cov                 # Run tests and generate coverage report
python3 -m jsonargparse_tests  # Run tests for installed package (requires pytest and␣
↪pytest-subtests)
```

CHAPTER

TWENTYONE

API REFERENCE

Even though jsonargparse has several internal modules, users are expected to only import from `jsonargparse` or `jsonargparse.typing`. This allows doing internal refactoring without affecting dependants. Only objects explicitly exposed in `jsonargparse.__init__.__all__` and in `jsonargparse.typing.__all__` are included in this API reference and is what can be considered public.

21.1 jsonargparse

Exceptions:

ArgumentError(argument, message)	An error from creating or using an argument (optional or positional).
ParserError	alias of *ArgumentError*

Functions:

CLI([components, args, config_help, ...])	Simple creation of command line interfaces.
compose_dataclasses(*args)	Returns a 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 copy of a nested namespace converted into a nested dictionary.
dict_to_namespace(cfg_dict)	Converts a nested dictionary into a nested namespace.
strip_meta(cfg)	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.
set_docstring_parse_options([style, ...])	Sets options for docstring parsing.
set_loader(mode, loader_fn[, exceptions])	Sets the value loader function to be used when parsing with a certain mode.
set_dumper(format_name, dumper_fn)	Sets the dumping function for a given format name.
capture_parser(function, *args, **kwargs)	Returns the parser object used within the execution of a function.
class_from_function(func[, func_return, name])	Creates a dynamic class which if instantiated is equivalent to calling func.
register_unresolvable_import_paths(*modules)	Saves import paths of module objects for which its import path is unresolvable from the object alone.
set_url_support(enabled)	Enables/disables URL support for config read mode.
usage_and_exit_error_handler(parser, message)	Prints the usage and exits with error code 2 (same behavior as argparse).

Classes:

ActionsContainer(*args, **kwargs)	Extension of argparse._ActionsContainer to support additional functionalities.
ArgumentParser(*args[, env_prefix, ...])	Parser for command line, configuration files and environment variables.
SignatureArguments(*args[, logger])	Methods to add arguments based on signatures to an ArgumentParser instance.
ArgumentLinking()	Method for linking arguments.
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.
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.
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.
LoggerProperty(*args[, logger])	Class designed to be inherited by other classes to add a logger property.
Path(path[, mode, cwd])	Stores a (possibly relative) path and the corresponding absolute path.
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.
ActionPathList([mode, rel])	Action to check and store a list of file paths read from a plain text file or stream.

exception jsonargparse.**ArgumentError**(*argument*, *message*)

Bases: Exception

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

__init__(*argument*, *message*)

jsonargparse.**CLI**(*components=None*, *args=None*, *config_help='Path to a configuration file.'*, *set_defaults=None*, *as_positional=True*, *fail_untyped=True*, *parser_class=<class 'jsonargparse._core.ArgumentParser'>*, ***kwargs*)

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' parameter 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.
- **fail_untyped** (bool) – Whether to raise exception if a required parameter does not have a type.
- **parser_class** (Type[*ArgumentParser*]) – The ArgumentParser class to use.
- ****kwargs** – Used to instantiate *ArgumentParser*.

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

class jsonargparse.**ActionsContainer**(**args*, ***kwargs*)

Bases: *SignatureArguments*, _ActionsContainer

Extension of argparse._ActionsContainer to support additional functionalities.

Methods:

__init__(*args, **kwargs)	Initializer for LoggerProperty class.
add_argument(*args[, enable_path])	Adds an argument to the parser or argument group.
add_argument_group(*args[, name])	Adds a group to the parser.

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

Initializer for LoggerProperty class.

add_argument(**args*, *enable_path=False*, ***kwargs*)

Adds an argument to the parser or argument group.

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

Parameters
enable_path (bool) – Whether to try parsing path/subconfig when argument is a complex type.

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

Adds a group to the parser.

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

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

Return type
_ArgumentGroup

Returns
The group object.

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

class jsonargparse.**ArgumentParser**(**args*, *env_prefix=True*, *formatter_class=<class 'jsonargparse._formatters.DefaultHelpFormatter'>*, *exit_on_error=True*, *logger=False*, *version=None*, *print_config='--print_config'*, *parser_mode='yaml'*, *dump_header=None*, *default_config_files=None*, *default_env=False*, *default_meta=True*, ***kwargs*)

Bases: ParserDeprecations, *ActionsContainer*, *ArgumentLinking*, ArgumentParser

Parser for command line, configuration files and environment variables.

Methods:

__init__(*args[, env_prefix, ...])	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 given its path.
parse_string(cfg_str[, cfg_path, ext_vars, ...])	Parses configuration given as a string.
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_default, ...])	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 and exits or raises an ArgumentError.
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:

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.
parser_mode	`'yaml'`, `'jsonnet'` or ones added via *set_loader()*.
dump_header	Header to include as comment when dumping a config object.

__init__(**args*, *env_prefix=True*, *formatter_class=<class 'jsonargparse._formatters.DefaultHelpFormatter'>*, *exit_on_error=True*, *logger=False*, *version=None*, *print_config='--print_config'*, *parser_mode='yaml'*, *dump_header=None*, *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** (Union[bool, str]) – Prefix for environment variables. `True` to derive from `prog`.
- **formatter_class** (Type[*DefaultHelpFormatter*]) – Class for printing help messages.
- **logger** (Union[bool, str, dict, Logger]) – Configures the logger, see *LoggerProperty*.
- **version** (Optional[str]) – Program version which will be printed by the –version argument.
- **print_config** (Optional[str]) – Add this as argument to print config, set None to disable.
- **parser_mode** (str) – Mode for parsing config files: `'yaml'`, `'jsonnet'` or ones added via *set_loader()*.
- **dump_header** (Optional[List[str]]) – Header to include as comment when dumping a config object.
- **default_config_files** (Optional[List[Union[str, PathLike]]]) – 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*, ***kwargs*)

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
ArgumentError – If the parsing fails error and exit_on_error=True.

parse_object(*cfg_obj*, *cfg_base=None*, *env=None*, *defaults=True*, *with_meta=None*, ***kwargs*)

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
ArgumentError – If the parsing fails error and exit_on_error=True.

parse_env(*env=None*, *defaults=True*, *with_meta=None*, ***kwargs*)

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
ArgumentError – If the parsing fails error and exit_on_error=True.

parse_path(*cfg_path*, *ext_vars=None*, *env=None*, *defaults=True*, *with_meta=None*, ***kwargs*)

Parses a configuration file given its path.

Parameters

- **cfg_path** (Union[str, PathLike]) – 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
ArgumentError – If the parsing fails error and exit_on_error=True.

parse_string(*cfg_str*, *cfg_path=''*, *ext_vars=None*, *env=None*, *defaults=True*, *with_meta=None*, ***kwargs*)

Parses configuration given as a string.

Parameters

- **cfg_str** (str) – The configuration content.
- **cfg_path** (Union[str, PathLike]) – 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.
- **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
ArgumentError – If the parsing fails error and exit_on_error=True.

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_default=False*, *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', 'parser_mode' or ones added via *set_dumper()*.
- **skip_none** (bool) – Whether to exclude entries whose value is None.
- **skip_default** (bool) – Whether to exclude entries whose value is the same as the default.
- **skip_check** (bool) – Whether to skip parser checking.
- **yaml_comments** (bool) – Whether to add help content as comments. yaml_comments=True implies format='yaml'.

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** (Union[str, PathLike]) – Path to the location where to save config.
- **format** (str) – The output format: 'yaml', 'json', 'json_indented', 'parser_mode' or ones added via *set_dumper()*.
- **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[str, Any]) – Dictionary defining the default values to set.
- ****kwargs** (Any) – 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
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 and exits or raises an ArgumentError.

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.

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.

property default_config_files: List[str]

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.

instantiate_subclasses(*cfg*)

Calls instantiate_classes with instantiate_groups=False. :rtype: *Namespace*

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.

property default_env: bool

Whether by default environment variables parsing is enabled.

If the JSONARGPARSE_DEFAULT_ENV environment variable is set to true or false, that value will take precedence.

Getter
Returns the current default environment variables parsing setting.

Setter
Sets the default environment variables parsing setting.

Raises
ValueError – If an invalid value is given.

property default_meta: 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.

property env_prefix: Union[bool, 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.

property parser_mode: str

'yaml', 'jsonnet' or ones added via *set_loader()*.

Getter
Returns the current parser mode.

Setter
Sets the parser mode.

Raises
ValueError – If an invalid value is given.

Type
Mode for parsing configuration files

property dump_header: Optional[List[str]]

Header to include as comment when dumping a config object.

Getter
Returns the current dump header.

Setter
Sets the dump header.

Raises
ValueError – If an invalid value is given.

jsonargparse.**compose_dataclasses**(**args*)

Returns a dataclass inheriting all given dataclasses and properly handling __post_init__.

class jsonargparse.**SignatureArguments**(**args*, *logger=False*, ***kwargs*)

Bases: *LoggerProperty*

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*, *default=None*, *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.
- **default** (Optional[LazyInitBaseClass]) – Default value used to override parameter defaults. Must be lazy_instance.
- **skip** (Optional[Set[Union[str, int]]]) – Names of parameters or number of positionals 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[Union[str, int]]]) – Names of parameters or number of positionals 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[Union[str, int]]]) – Names of parameters or number of positionals 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*, *fail_untyped=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.
- **fail_untyped** (bool) – Whether to raise exception if a required parameter does not have a type.

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='CONFIG | CLASS_PATH_OR_NAME | .INIT_ARG_NAME VALUE'*, *help='One or more arguments specifying "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.

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

Return type

TypeVar(ClassType)

class jsonargparse.**ArgumentLinking**

Bases: object

Method for linking arguments.

Methods:

link_arguments(source, target[, compute_fn, ...])	Makes an argument value be derived from the values of other arguments.

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.

class jsonargparse.**ActionJsonSchema**(*schema=None*, *enable_path=True*, *with_meta=True*, ***kwargs*)

Bases: 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: *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: 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: Action, 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: 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 (*Optional[ArgumentParser]*) – A parser to parse the option with.

Raises
ValueError – If the parser parameter is invalid.

class jsonargparse.**Namespace**(**args*, ***kwargs*)

Bases: Namespace

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

Methods:

__init__(*args, **kwargs)	Initializer for Namespace objects.
as_dict()	Converts the nested namespaces into nested dictionaries.
as_flat()	Converts the nested namespaces into a single argparse flat namespace.
items([branches])	Returns a generator of all leaf (key, value) items, optionally including branches.
keys([branches])	Returns a generator of all leaf keys, optionally including branches.
values([branches])	Returns a generator of all leaf values, optionally including branches.
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*)

Initializer for Namespace objects.

Instantiating a Namespace with initial values most commonly is done by providing keyword arguments, e.g. `Namespace(name1=value1, name2=value2)`. Alternatively a single positional `Namespace` or `dict` object can be given.

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(*branches=False*)

Returns a generator of all leaf (key, value) items, optionally including branches.

Return type
Iterator[Tuple[str, Any]]

keys(*branches=False*)

Returns a generator of all leaf keys, optionally including branches.

Return type
Iterator[str]

values(*branches=False*)

Returns a generator of all leaf values, optionally including branches.

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 namespace.
- **only_unset** (bool) – Whether to only set the value if not set in namespace.

Return type
Namespace

jsonargparse.**namespace_to_dict**(*namespace*)

Returns a copy 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 copy of the configuration object excluding all metadata keys.

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

Bases: 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.

jsonargparse.**set_docstring_parse_options**(*style=None*, *attribute_docstrings=None*)

Sets options for docstring parsing.

Parameters

- **style** (*docstring_parser.DocstringStyle*) – The docstring style to expect.
- **attribute_docstrings** (Optional[bool]) – Whether to parse attribute docstrings (slower).

jsonargparse.**set_loader**(*mode*, *loader_fn*, *exceptions=(<class 'yaml.error.YAMLError'>,)*)

Sets the value loader function to be used when parsing with a certain mode.

The loader_fn function must accept as input a single str type parameter and return any of the basic types {str, bool, int, float, list, dict, None}. If this function is not based on PyYAML for things to work correctly the exceptions types that can be raised when parsing a value fails should be provided.

Parameters

- **mode** (str) – The parser mode for which to set its loader function. Example: "yaml".
- **loader_fn** (Callable[[str], Any]) – The loader function to set. Example: yaml.safe_load.
- **exceptions** (Tuple[Type[Exception], ...]) – Exceptions that the loader can raise when load fails. Example: (yaml.parser.ParserError, yaml.scanner.ScannerError).

jsonargparse.**set_dumper**(*format_name*, *dumper_fn*)

Sets the dumping function for a given format name.

Parameters

- **format_name** (str) – Name to use for dumping with this function. Example: "yaml_custom".
- **dumper_fn** (Callable[[Any], str]) – The dumper function to set. Example: `yaml.safe_dump`.

jsonargparse.**capture_parser**(*function*, **args*, ***kwargs*)

Returns the parser object used within the execution of a function.

The function execution is stopped on the start of the call to parse_args. No parsing is done or execution of instructions after the parse_args.

Parameters

- **function** (Callable) – A callable that internally creates a parser and calls parse_args.
- ***args** – Positional arguments used to run the function.
- ****kwargs** – Keyword arguments used to run the function.

Raises

CaptureParserException – If the function does not call parse_args.

Return type

None

jsonargparse.**class_from_function**(*func*, *func_return=None*, *name=None*)

Creates a dynamic class which if instantiated is equivalent to calling func.

Parameters

- **func** (Callable[..., TypeVar(ClassType)]) – A function that returns an instance of a class.
- **func_return** (Optional[Type[TypeVar(ClassType)]]) – The return type of the function. Required if func does not have a return type annotation.
- **name** (Optional[str]) – The name of the class. Defaults to function name suffixed with "_class".

Return type

Type[TypeVar(ClassType)]

class jsonargparse.**LoggerProperty**(**args*, *logger=False*, ***kwargs*)

Bases: object

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

Methods:

__init__(*args[, logger])	Initializer for LoggerProperty class.

Attributes:

logger	The logger property for the class.

__init__(**args*, *logger=False*, ***kwargs*)

Initializer for LoggerProperty class.

property logger: 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.

Raises
ValueError – If an invalid logger value is given.

class jsonargparse.**Path**(*path*, *mode='fr'*, *cwd=None*, ***kwargs*)

Bases: PathDeprecations

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

The absolute path can be obtained without having to remember the working directory (or parent remote path) from when the object was created.

When a Path instance is created, it is checked that: the path exists, whether it is a file or directory and whether it 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 creatable flag "c" can be given one or two times. If give once, the parent directory must exist and be writeable. If given twice, the parent directory does not have to exist, but should be allowed to create.

Methods:

__init__(path[, mode, cwd])	Initializer for Path instance.
__call__([absolute])	Returns the path as a string.
get_content([mode])	Returns the contents of the file or the remote path.
open([mode])	Return an opened file object for the path.
relative_path_context()	Context manager to use this path's parent (directory or URL) for relative paths defined within.

__init__(*path*, *mode='fr'*, *cwd=None*, ***kwargs*)

Initializer for Path instance.

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

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 remote path.

Return type
str

open(*mode='r'*)

Return an opened file object for the path.

Return type
Iterator[IO]

relative_path_context()

Context manager to use this path's parent (directory or URL) for relative paths defined within.

Return type
Iterator[str]

jsonargparse.**register_unresolvable_import_paths**(**modules*)

Saves import paths of module objects for which its import path is unresolvable from the object alone.

Objects with unresolvable import paths have the __module__ attribute set to None.

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.

class jsonargparse.**ActionPathList**(*mode=None*, *rel='cwd'*, ***kwargs*)

Bases: Action, FilesCompleterMethod

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

Warning: ActionPathList was deprecated in v4.20.0 and will be removed in v5.0.0. Instead use as type List[<path_type>] with enable_path=True.

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 [fdrwxcuF-DRWX] 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.

jsonargparse.**ParserError**

alias of *ArgumentError*

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.

jsonargparse.**usage_and_exit_error_handler**(*parser*, *message*)

Prints the usage and exits with error code 2 (same behavior as argparse). :rtype: None

Args:
parser: The parser object. message: The message describing the error being handled.

> **Warning:** usage_and_exit_error_handler was deprecated in v4.20.0 and will be removed in v5.0.0. With the removal of error_handler, there is no longer a need for this function.

21.2 jsonargparse.typing

Collection of types and type generators.

Functions:

final(cls)	Decorator to make a class `final`, i.e., it shouldn't be subclassed.
is_final_class(cls)	Checks whether a class is final, i.e. decorated with `typing.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])	Creates or returns an already registered path type class.
`is_subclass`(cls, class_or_tuple)	Extension of issubclass that supports non-class arguments.

Classes:

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

jsonargparse.typing.**final**(*cls*)

Decorator to make a class `final`, i.e., it shouldn't be subclassed.

It is the same as `typing.final` or an equivalent implementation depending on the python version and whether typing-extensions is installed.

jsonargparse.typing.**is_final_class**(*cls*)

Checks whether a class is final, i.e. decorated with `typing.final`.

Return type
bool

jsonargparse.typing.**register_type**(*type_class*, *serializer=<class 'str'>*, *deserializer=None*, *deserializer_exceptions=(<class 'ValueError'>, <class 'TypeError'>, <class 'AttributeError'>)*, *type_check=<function <lambda>>*, *fail_already_registered=True*, *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 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.
- **fail_already_registered** (bool) – Whether to fail if type has already been registered.
- **uniqueness_key** (Optional[Tuple]) – Key to determine uniqueness of type.

Return type
None

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*, ***kwargs*)

Creates or returns an already registered path type class.

Parameters

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

Return type
type

Returns
The created or retrieved type class.

class jsonargparse.typing.**PositiveInt**(*v*)

Bases: TypeCore, int

int restricted to be >0

class jsonargparse.typing.**NonNegativeInt**(*v*)

Bases: TypeCore, int

int restricted to be 0

class jsonargparse.typing.**PositiveFloat**(*v*)

Bases: TypeCore, float

float restricted to be >0

class jsonargparse.typing.**NonNegativeFloat**(*v*)

Bases: TypeCore, float

float restricted to be 0

class jsonargparse.typing.**ClosedUnitInterval**(*v*)

Bases: TypeCore, float

float restricted to be 0 and 1

class jsonargparse.typing.**OpenUnitInterval**(*v*)

Bases: TypeCore, float

float restricted to be >0 and <1

class jsonargparse.typing.**NotEmptyStr**(*v*)

Bases: TypeCore, str

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

class jsonargparse.typing.**Email**(*v*)

Bases: TypeCore, str

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

class jsonargparse.typing.**Path_fr**(*v*, ***k*)

Bases: PathType

path to a file that exists and is readable

class jsonargparse.typing.**Path_fc**(*v*, ***k*)

Bases: PathType

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

class jsonargparse.typing.**Path_dw**(*v*, ***k*)

Bases: PathType

path to a directory that exists and is writeable

class jsonargparse.typing.**Path_dc**(*v*, ***k*)

Bases: PathType

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

class jsonargparse.typing.**Path_drw**(*v*, ***k*)

Bases: PathType

path to a directory that exists and is readable and writeable

CHAPTER

TWENTYTWO

INDEX

- changelog
- license
- genindex

PYTHON MODULE INDEX

j

INDEX

Symbols

A

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

U

V