multiconfparse

Jun 22, 2020

Contents:

Multiconfparse is a Python3 library for specifying and reading configuration data from multiple sources, including the command line, environment variables and various config file formats. The API is very similar to the Argparse API.

CHAPTER 1

Installation

1.1 To get the latest stable version

```
python -m pip install multiconfparse
```

1.2 To get the latest unstable version

```
python -m pip install git+https://github.com/jonathanhaigh/multiconfparse
```

CHAPTER 2

Quick start

2.1 Import the `multiconfparse` module

```
import multiconfparse
```

2.2 Create a `ConfigParser` object

```
config_parser = multiconfparse.ConfigParser()
```

`ConfigParser` objects:

- contain the specifications of your configuration items;
- maintain a list of sources that can obtain configuration values in different ways.
- coordinate the parsing done by each source;
- merge configuration values from the sources into a single set of values.

2.3 Add specifications of your config items

```
config_parser.add_config("config_item1", required=True)
config_parser.add_config("config_item2", default="default_value")
config_parser.add_config("config_item3", action="append", type=int)
```

`ConfigParser.add_config()` has a single required parameter - the name of the config item. This is used as the name of the config item's attribute in the object returned by the parse and must be a valid Python identifier. The optional parameters used above are:

- `required` - if a `required` config item cannot be found in any config source then `ConfigParser.parse` will raise an exception.

- `default_value` - the value the config item will take if it is not found in any source.
- `action` - this specifies the way the values found during parsing will be processed. The default action is `store`, which means that the value from the highest priority source will be set as the config item's value in the output of the parse. The `append` action creates a list of all of the values seen from all sources during the parse.
- `type` - this specifies the type of the value that should be returned by the parse.

The `add_config()` method was modeled on the `argparse.ArgumentParser.add_argument()` method and accepts mostly the same options.

2.4 Add config sources

```
config_parser.add_source("simple_argparse")
config_parser.add_source("environment", env_var_prefix="MY_APP_")
config_parser.add_source("json", path="/path/to/config/file.json")
```

`ConfigParser.add_source()`'s first parameter is the name of a source or a class that implements a source. Other parameters are passed on to the class that implements the source.

In the example above, three sources are added:

- `simple_argparse` - a source that reads config values from the command line. The `simple_argparse` source creates an `argparse.ArgumentParser` that will accept `--config-item1` and `--config-item2` options.
- `environment` - a source that reads config values from environment variables. The `environment` source will look for config values in the `MY_APP_CONFIG_ITEM1` and `MY_APP_CONFIG_ITEM2` environment variables.
- `json` - a source that reads config values from a JSON file. In this example it will look for a JSON object in "/path/to/config/file.json" and obtain values from the `"config_item1"` and `"config_item2"` keys.

2.5 Parse config from all sources

```
config = config_parser.parse_config()
```

`ConfigParser.parse_config()` returns a `Namespace` object which is essentially just a plain object with attributes for each config item. If a config item was not found in any source, and was not a `required` option then it will (by default) be given a value of `None` in the returned `Namespace` object.

2.6 Use the config

```
item1 = config.config_item1
item2 = config.config_item2
item3 = config.config_item3
```

CHAPTER 3

Creating a parser

To parse config using *multiconfparse* you must first create a *ConfigParser* object:

```
config_parser = multiconfparse.ConfigParser()
```

The full documentation for creating *ConfigParser* objects is:

class multiconfparse.**ConfigParser**(*config_default=NOT_GIVEN*)

Create a new ConfigParser object. Options are:

- `config_default`: the default value to use in the *Namespace* returned by *parse_config()* for config items for which no value was found.

  The default behaviour (when `config_default` is *NOT_GIVEN*) is to represent these config items with the value `None`.

  Set `config_default` to *SUPPRESS* to prevent these configs from having an attribute set in the *Namespace* at all.

CHAPTER 4

Adding config items

Once you have created your *ConfigParser* object, you can add specifications of your config items using the *ConfigParser.add_config()*:

ConfigParser.**add_config**(*name*, ***kwargs*)

Add a config item to the *ConfigParser*.

The arguments that apply to all config items are:

- name (required, positional): the name of the config item.

  In the *Namespace* object returned by *parse_config()*, the name of the attribute used for this config item will be name and must be a valid Python identifier.

  name is also used by source *Source* classes to generate the strings that will be used to find the config in config sources. The *Source* classes may, use a modified version of name, however. For example, the argparse and simple_argparse sources will convert underscores (_) to hyphens (-) and add a -- prefix, so if a config item had the name "config_item1", the argparse and simple_argparse sources would use the option string "--config-item1".

- action (optional, keyword): the name of the action that should be performed when a config item is found in a config source. The default action is "store", and the built-in actions are described briefly below. See *Actions* for more detailed information about the built-in actions and creating your own actions. The built-in actions are all based on argparse actions so the argparse documentation may also provide useful information.

  - store: this action just stores the highest priority value for config item.
  - store_const: this stores the value specified in the const argument.
  - store_true: this stores the value True and sets the default argument to False. It is a special case of store_const.
  - store_false: this stores the value False and sets the default argument to True. It is a special case of store_const.
  - append: this creates a list containing every value seen (with lower priority values first). When nargs >= 1, nargs == "+" or nargs == "*", each value in the list s iteself a list containing the arguments for a mention of the config item.

  - count: this stores the number of mentions of the config item.
  - extend: this creates a list containing every value seen (with lower priority values first). Unlike append, when nargs >= 1, nargs == "+" or nargs == "*", arguments for mentions of the config item are not placed in separate sublists for each mention.

- default: the default value for this config item. Note that some actions will incorporate the default value into the final value for the config item even if the config item is mentioned in one of the sources (e.g. append, count and extend).

  Note that the default value for all config items can also be set by passing a value for the config_default argument of *ConfigParser*. If both the config_default argument to *ConfigParser* and the default argument to *add_config()* are used then only the default argument to *add_config()* is used.

  If a default value is not provided for the config item by the default argument, the config_default argument or by the action class (like e.g. *store_true* does), then the final value for the config will be None if the config item is not mentioned in any source.

  The special value *SUPPRESS* can be passed as the default argument. In this case, if the config item is not mentioned in any source, it will not be given an attribute in the *Namespace* object returned by *parse_config()*.

- dest (optional, keyword): the name of the attribute for the config item in the *Namespace* object returned by *parse_config()*. By default, dest is set to the name of the config item (name).
- exclude_sources (optional, keyword): a collection of source names or *Source* classes that should ignore this config item. This argument is mutually exclusive with include_sources. If neither exclude_sources nor include_sources is given, the config item will be looked for by all sources added to the *ConfigParser*.
- include_sources (optional, keyword): a collection of source names or *Source* classes that should look for this config item. This argument is mutually exclusive with exclude_sources. If neither exclude_sources nor include_sources is given, the config item will be looked for by all sources added to the *ConfigParser*.
- help: the help text/description for the config item. Set this to *SUPPRESS* to prevent this config item from being mentioned in generated documentation.

The other arguments are all keyword arguments and are passed on to the class that implements the config items action and may have different default values or may not even be valid for all actions. See *Actions* for action specific documentation.

- nargs: specifies the number of arguments that the config item accepts. The values that nargs can take are:
  - None: the config item will take a single argument.
  - 0: the config item will take no arguments. This value is usually not given to *add_config()* but may be implicit for an action (e.g. store_const or count).
  - An int N >= 1: the config item will take N arguments and the value for a mention of the config item will be a list containing each argument. In particular, when nargs == 1 the value for each mention of a config item will be a list containing a single element.
  - "?": The config item will take a single optional argument. When the config item is mentioned without an accompanying value, the value for the mention is the value of the config item's const argument.
  - "*": The config item will take zero or more arguments and the value for a mention of the config item will be a list containing each argument.

  - `"+"` The config item will take one or more arguments and the value for a mention of the config item will be a `list` containing each argument.

- `const`: The value to use for a mention of the config item where there is no accompanying argument. This is only ever used when `nargs == 0` or `nargs == "+"`.
- `type`: The type to which each argument of the config item should be converted. This can be any callable object that takes a single argument (an object with a `__call__(self, arg)` method), including classes like `int` and functions that take a single argument. Note that some sources that read typed data may produce config item argument values that aren't always `str` objects.

  The default `type` is `str` unless that doesn't make sense (e.g. when `nargs == 0`.
- `required`: specifies whether an exception should be raised if a value for this config item cannot be found in any source.

  The default `required` is `False`.
- `choices`: specifies a collection of valid values for the arguments of the config item. If `choices` is specified, an exception is raised if the config item is mentioned in a source with an argument that is not in `choices`.

CHAPTER 5

Adding config sources

To add config sources to a *ConfigParser*, use *ConfigParser.add_source()* method:

ConfigParser.**add_source**(*source*, **args*, ***kwargs*)

Add a new config source to the *ConfigParser*.

The only argument required for all sources is the `source` parameter which may be the name of a source or a class that implements a source. Other arguments are passed on to the class that implements the source.

The built-in sources are:

- `argparse`: for getting config values from the command line using an `argparse.ArgumentParser`.
- `simple_argparse`: a simpler version of the `argparse` source that is easier to use but doesn't allow you to add any arguments that aren't also config items.
- `environment`: for getting config values from environment variables.
- `json`: for getting config values from JSON files.
- `dict`: for getting config values from Python dictionaries.

See *Sources* for more information about the built-in sources and creating your own sources.

Return the created config source object.

CHAPTER 6

Running the parse

To parse configuration from all sources for a *ConfigParser*, use the *ConfigParser.parse_config()* method:

ConfigParser.**parse_config**()

Parse the config sources.

Returns: a *Namespace* object containing the parsed values.

CHAPTER 7

Actions

The built-in actions are:

7.1 store

class multiconfparse.**StoreAction**(*const=None*, ***kwargs*)

The `store` action simply stores the value from the highest priority mention of a config item. Its behaviour is based on the `store` `argparse` action and is the default action.

Arguments to *`ConfigParser.add_config()`* have standard behaviour, but note:

- `nargs == 0` is not allowed. The default `nargs` value is `None`.
- The `const` argument is only accepted when `nargs == "?"`.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_source("dict", {"config_item1": "v1"}, priority=2)
parser.add_source("dict", {"config_item1": "v2"}, priority=1)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", nargs=2, type=int, default=[1, 2])
parser.add_source("simple_argparse")
parser.parse_config()
#
# If the command line looks something like:
#   prog some-arg --config-item1 3 4
# parse_config() will return something like:
```

(continues on next page)

(continued from previous page)

```
#   multiconfparse.Namespace {
#       "config_item1": [3, 4],
#   }
#
# If the command line looks something like:
#   prog some-arg
# parse_config() will return something like:
#   multiconfparse.Namespace {
#       "config_item1": [1, 2],
#   }
```

7.2 store_const

class multiconfparse.**StoreConstAction**(*const*, ***kwargs*)

The store_const action stores the value from the const argument whenever a config item is mentioned in a source. Its behaviour is based on the store_const argparse action.

Notes about the arguments to *ConfigParser.add_config()*:

- The const argument is mandatory.
- nargs is not accepted as an argument. nargs is always 0 for store_const actions.
- required is not accepted as an argument. required is always False for store_const actions.
- type is not accepted as an argument - it doesn't make sense for store_const.
- choices is not accepted as an argument - it doesn't make sense for store_const.

Example:

```
parser = multiconfparse.ConfigParser()
parser.add_config(
    "config_item1",
    action="store_const",
    const="yes",
    default="no"
)
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "yes",
# }
```

7.3 store_true

class multiconfparse.**StoreTrueAction**(*default=False*, ***kwargs*)

The store_true action simply stores the value True whenever a config item is mentioned in a source. Its behaviour is based on the store_true argparse action.

Notes about the arguments to *ConfigParser.add_config()*:

- const is not accepted as an argument - const is always True for store_true actions.
- nargs is not accepted as an argument. nargs is always 0 for store_true actions.

- `required` is not accepted as an argument. `required` is always `False` for `store_true` actions.
- `type` is not accepted as an argument - it doesn't make sense for `store_true`.
- `choices` is not accepted as an argument - it doesn't make sense for `store_true`.
- The default value for the `default` argument is `False`.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_true")
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": True,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_true")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": False,
# }
```

7.4 store_false

class multiconfparse.**StoreFalseAction**(*default=True*, ***kwargs*)

The `store_false` action simply stores the value `False` whenever a config item is mentioned in a source. Its behaviour is based on the `store_false` `argparse` action.

Notes about the arguments to *ConfigParser.add_config()*:

- `const` is not accepted as an argument - `const` is always `False` for `store_false` actions.
- `nargs` is not accepted as an argument. `nargs` is always `0` for `store_false` actions.
- `required` is not accepted as an argument. `required` is always `False` for `store_false` actions.
- `type` is not accepted as an argument - it doesn't make sense for `store_false`.
- `choices` is not accepted as an argument - it doesn't make sense for `store_false`.
- The default value for the `default` argument is `True`.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_false")
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": False,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_false")
parser.parse_config()
# -> multiconfparse.Namespace {
```

(continues on next page)

(continued from previous page)

```
#   "config_item1": True,
# }
```

7.5 append

class multiconfparse.**AppendAction**(*const=None*, *default=NOT_GIVEN*, ***kwargs*)

The append action stores the value for each mention of a config item in a list. The list is sorted according to the priorities of the mentions of the config item, lower priorities first. The Behaviour is based on the append argparse action.

Notes about the arguments to *ConfigParser.add_config()*:

- nargs == 0 is not allowed. The default nargs value is None.

  When nargs >= 1, nargs == "+" or nargs == "*", each value in the list for the config item is itself a list containing the arguments for a mention of the config item.

- The const argument is only accepted when nargs == "?".
- The default argument (if it is given and is not *SUPPRESS*) is used as the initial list of values. This means that the default value is incorporated into the final value for the config item, even if the config item is mentioned in a source.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="append", default=["v1"])
parser.add_source("dict", {"config_item1": "v2"}, priority=2)
parser.add_source("dict", {"config_item1": "v3"}, priority=1)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": ["v1", "v3", "v2"],
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config(
    "config_item1",
    action="append",
    nargs="?",
    const="v0",
)
parser.parse_config()
parser.add_source("dict", {"config_item1": "v1"}, priority=2)
parser.add_source("dict", {"config_item1": None}, priority=1)
# -> multiconfparse.Namespace {
#   "config_item1": ["v0", "v1"],
# }
```

7.6 count

class multiconfparse.**CountAction**(***kwargs*)

The count action stores the number of times a config item is mentioned in the config sources. Its behaviour is based on the count argparse action.

Notes about the arguments to *ConfigParser.add_config()*:

- `nargs` is not accepted as an argument. `nargs` is always `0` for `count` actions.
- `const` is not accepted as an argument - it doesn't make sense for `count`.
- `required` is not accepted as an argument. `required` is always `False` for `count` actions.
- `type` is not accepted as an argument - it doesn't make sense for `count`.
- `choices` is not accepted as an argument - it doesn't make sense for `count`.
- If the `default` argument is given and is not *SUPPRESS*, it acts as the initial value for the count. I.e. the final value for the config item will be the number of mentions of the config item in the sources, plus the value of `default`.

  Note that if the config item is not found in any sources and `default` is not given, it is *not* assumed to be `0`. The final value for the config item would be `None` in this case.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="count")
parser.add_source("dict", {"config_item1": None})
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": 2,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="count", default=10)
parser.add_source("dict", {"config_item1": None})
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": 12,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="count")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": None,
# }
```

7.7 extend

class multiconfparse.**ExtendAction**(***kwargs*)

The `extend` action stores the value for each argument of each mention of a config item in a `list`. The `list` is sorted according to the priorities of the mentions of the config item, lower priorities first. The Behaviour is based on the `extend` `argparse` action, although the behaviour when `nargs == None` or `nargs == "?"` is different.

Notes about the arguments to *ConfigParser.add_config()*:

- `nargs == 0` is not allowed. The default `nargs` value is "+".

Unlike the `append` action, when `nargs >= 1`, `nargs == "+"` or `nargs == "*"`, each value in the `list` for the config item is *not* itself a `list` containing the arguments for a mention of the config item. Each argument of each mention is added separately to the `list` that makes the final value for the config item.

Unlike the `argparse` `extend` action, when `nargs == None` or `nargs == "?"`, the *`multiconfparse`* `extend` action behaves exactly like the `append` action.

- The `const` argument is only accepted when `nargs == "?"`.
- The `default` argument (if it is given and is not *`SUPPRESS`*) is used as the initial `list` of values. This means that the `default` value is incorporated into the final value for the config item, even if the config item is mentioned in a source.

Example:

```
parser = multiconfparse.ConfigParser()
parser.add_config(
    "config_item1",
    action="extend",
    default=[["v1", "v2"]]
)
parser.add_source("dict", {"config_item1": ["v3", "v4"]}, priority=2)
parser.add_source("dict", {"config_item1": ["v5"]}, priority=1)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": ["v1", "v2", "v5", "v3", "v4"],
# }
```

7.8 Creating your own action classes

To create your own action class, create a subclass of *`Action`*:

class multiconfparse.**Action**(*name*, *dest=None*, *nargs=None*, *type=<class 'str'>*, *required=False*, *default=NOT_GIVEN*, *choices=None*, *help=None*, *include_sources=None*, *exclude_sources=None*)

Abstract base class config actions.

Classes to support actions should:

- Inherit from *`Action`*.
- Implement the *`__call__()`* method documented below.
- Have an `action_name` class attribute set to the name of the action that the class implements.
- Have an `__init__()` method that accepts arguments passed to *`ConfigParser.add_config()`* calls by the user (except the `action` argument) and which calls `Action.__init__()` with any of those arguments which are not specific to the action handled by the class. I.e.:
  - `name`;
  - `nargs`;
  - `type`;
  - `required`;
  - `default`;
  - `choices`;

- help;
- dest;
- include_sources;
- exclude_sources.

These arguments will be assigned to attributes of the *Action* object being created (perhaps after some processing or validation) that are available for access by subclasses. The names of the attributes are the same as the argument names.

It is recommended that passing the arguments to `Action.__init__()` is done by the subclass `__init__` method accepting a `**kwargs` argument to collect any arguments that are not used or modified by the action class, then passing that `**kwargs` argument to `Action.__init__()`. The action class may also want pass some arguments that aren't specified by the user if the value of those arguments is implied by the action. For example, the `store_const` action class has the following `__init__` method:

```
def __init__(self, const, **kwargs):
    super().__init__(
        nargs=0,
        type=None,
        required=False,
        choices=None,
        **kwargs,
    )
    self.const = const
```

This ensures that an exception is raised if the user specifies `nargs`, `type`, `required`, or `choices` arguments when adding a `store_const` action because if the user specifies those arguments they will be given twice in the call to `Action.__init__()`.

`__call__`(*namespace*, *args*)

Combine the arguments from a mention of this config with any existing value.

This method is called once for each mention of the config item in the sources in order to combine the arguments from the mention with any existing value.

`namespace` will be the same *Namespace* object for all calls to this function during a *ConfigParser.parse_config()* call, and it is used to hold the so-far-accumulated value for the config item mentions.

This method's purpose is to combine the current value for this config item in `namespace` with the values in the `new` argument, and write the combined value into back into `namespace`.

The first time this method is called, if the config item has a default value, `namespace` will have an attribute for the config item and it will contain the default value; otherwise `namespace` will not have an attribute for the config item.

After the first call to this method, `namespace` should have an attribute value for the config item set by the previous call.

Notes:

- The name of the attribute in `namespace` for this config item is given by this object's `dest` attribute.
- The calls to this method are made in order of the priorities of the config item mentions in the sources, lowest priority first.
- The values in `new` have already been coerced to the config item's `type`.
- The values in `new` have already been checked for `choices` validity.

- The number of values in `new` has already been checked for `nargs` validity.
- If no arguments for the config item were given, `new` will just be an empty `list`.

For example, the *`__call__()`* method for the `append` action is:

```
def __call__(self, namespace, args):
    current = getattr(namespace, self.dest, [])
    if self.nargs == "?" and not args:
        current.append(self.const)
    elif self.nargs is None or self.nargs == "?":
        assert len(args) == 1
        current.extend(args)
    else:
        current.append(args)
    setattr(namespace, self.dest, current)
```

The full example of the class for the `store_const` action is:

```
class StoreConstAction(Action):
    action_name = "store_const"

    def __init__(self, const, **kwargs):
        super().__init__(
            nargs=0,
            type=None,
            required=False,
            choices=None,
            **kwargs,
        )
        self.const = const

        def __call__(self, namespace, args):
            assert not args
            setattr(namespace, self.dest, self.const)
```

CHAPTER 8

Sources

The built-in sources are:

8.1 argparse

class multiconfparse.**ArgparseSource**(*actions*, *priority=20*)

Obtains config values from an argparse.ArgumentParser.

Do not create *ArgparseSource* objects directly, add them to a *ConfigParser* object using *ConfigParser.add_source()*. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")
argparse_source = parser.add_source("argparse")

argparse_parser = argparse.ArgumentParser()
argparse_parser.add_argument("arg1")
argparse_parser.add_argument("--opt1", type=int, action="append")
argparse_source.add_configs_to_argparse_parser(argparse_parser)

args = argparse_parser.parse_args((
    "arg1_value --config-item1 v1 --config-item2 1 2 --opt1 opt1_v1 "
    "--config-item-3 --opt1 opt1_v2"
).split())

argparse_source.notify_parsed_args(args)

config_parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
```

(continues on next page)

(continued from previous page)

```
#   "config_item3": True,
# }
```

The `argparse` source does not create an `argparse.ArgumentParser` for you. This is to allow extra command line arguments to be added to an `argparse.ArgumentParser` that are not config items. Instead *`ArgparseSource`*, which implements the `argparse` source provides two methods to provide communication with the `argparse.ArgumentParser`:

`add_configs_to_argparse_parser`(*argparse_parser*)
Add arguments to an `argparse.ArgumentParser` for config items.

`notify_parsed_args`(*argparse_namespace*)
Notify the `argparse` source of the `argparse.Namespace` object returned by `argparse.ArgumentParser.parse_args()`.

If you don't need to add command line arguments other than for config items, see *`SimpleArgparseSource`* which implements the `simple_argparse` source.

The arguments of *`ConfigParser.add_source()`* for the `argparse` source are:

- `source` (required, positional): `"argparse"`
- `priority` (optional, keyword): The priority for the source. The default priority for an `argparse` source is `20`.

Note that:

- The name of the command line argument for a config item is the config item's name with underscores (_) converted to hyphens (-) and prefixed with `--`.

8.2 simple_argparse

class multiconfparse.**SimpleArgparseSource**(*actions*, *argument_parser_class=<class 'argparse.ArgumentParser'>*, *priority=20*, ***kwargs*)

Obtains config values from the command line.

The `simple_argparse` source is simpler to use than the `argparse` source but it doesn't allow adding arguments that are not config items.

Do not create objects of this class directly - create them via *`ConfigParser.add_source()`* instead. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")
parser.add_source("simple_argparse")
config_parser.parse_config()
# If the command line looks something like:
#    PROG_NAME --config-item1 v1 --config-item2 1 2 --config-item-3
# The result would be:
# multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for the `simple_argparse` source are:

- `source` (required, positional): `"simple_argparse"`
- `argument_parser_class` (optional, keyword): a class derived from `argparse.ArgumentParser` to use instead of `ArgumentParser` itself. This can be useful if you want to override `argparse.ArgumentParser.exit()` or `argparse.ArgumentParser.error()`.
- `priority` (optional, keyword): The priority for the source. The default priority for a `simple_argparse` source is `20`.
- Extra keyword arguments to pass to `argparse.ArgumentParser`. E.g. `prog`, `allow_help`. Don't use the `argument_default` option though - the `simple_argparse` sources sets this internally. See the `config_default` option for *ConfigParser* instead.

Note that:

- The name of the command line argument for a config item is the config item's name with underscores (_) converted to hyphens (-) and prefixed with `--`.

8.3 environment

class multiconfparse.**EnvironmentSource**(*actions*, *none_values=None*, *priority=10*, *env_var_prefix=''*, *env_var_force_upper=True*)

Obtains config values from the environment.

Do not create *EnvironmentSource* objects directly, add them to a *ConfigParser* object using *ConfigParser.add_source()*. For example:

```
# For demonstration purposes, set some config values in environment
# variables
os.environ["MY_APP_CONFIG_ITEM1"] = "v1"
os.environ["MY_APP_CONFIG_ITEM2"] = "1 2"
os.environ["MY_APP_CONFIG_ITEM3"] = ""

parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")
parser.add_source("environment", env_var_prefix="MY_APP_")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for the `environment` source are:

- `source` (required, positional): `"environment"`
- `none_values` (optional, keyword): a list of values that, when seen in environment variables, should be treated as if they were not present (i.e. values for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict).

  The default `none_values` is `[""]`. using a different value for `none_values` is useful you want the empty string to be treated as a valid config value.
- `priority` (optional, keyword): The priority for the source. The default priority for an `environment` source is `10`.

- `env_var_prefix` (optional, keyword): a string prefixed to the environment variable names that the source will look for. The default value is `""`.
- `env_var_force_upper` (optional, keyword): force the environment variable name to be in upper case. Default is `True`.

Note that:

- Values in environment variables for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the environment variable) should be values from the `none_values` list described above.
- Values in environment variables for config items with `nargs >= 2`, `nargs == "+"` or `nargs == "*"` are split into arguments by `shlex.split()` (i.e. like arguments given on a command line via a shell). See the `shlex` documentation for full details.

8.4 json

class multiconfparse.**JsonSource**(*actions*, *path=None*, *fileobj=None*, *none_values=None*, *json_none_values=None*, *priority=0*)

Obtains config values from a JSON file.

Do not create objects of this class directly - create them via *ConfigParser.add_source()*. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")

fileobj = io.StringIO('''
    {
        "config_item1": "v1",
        "config_item2": [1, 2],
        "config_item3": null
    }
''')
parser.add_source("json", fileobj=fileobj)

config_parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for `json` sources are:

- `source` (required, positional): `"json"`.
- `priority` (optional, keyword): The priority for the source. The default priority for a `json` source is `0`.
- `path` (optional, keyword): path to the JSON file to parse. Exactly one of the `path` and `fileobj` options must be given.
- `fileobj` (optional keyword): a file object representing a stream of JSON data. Exactly one of the `path` and `fileobj` options must be given.
- `none_values` (optional, keyword): a list of python values that, when seen as config item values after JSON decoding, should be treated as if they were not present (i.e. values for config items with `nargs ==`

`0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict). The default `none_values` is `[]`.

- `json_none_values` (optional, keyword): a list of JSON values (as strings) that are decoded into Python values and added to `none_values`. The default `json_none_values` is `["null"]`.

Notes:

- The data in the JSON file should be a JSON object. Each config item value should be assigned to a field of the object that has the same name as the config item.
- Fields in the JSON object for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict) should either have values from the `json_none_values` list or should decode to values in the `none_values` list.
- Fields in the JSON object for config items with `nargs >= 2`, `nargs == "+"` or `nargs == "*"` should be JSON arrays with an element for each argument of the config item.

  In the special case where `nargs == "+"` or `nargs == "*"` and there is a single argument for the config item, the value may be given without the enclosing JSON array, unless the argument is itself an array.

8.5 dict

class multiconfparse.**DictSource**(*actions*, *values_dict*, *none_values=None*, *priority=0*)

Obtains config values from a Python `dict` object.

Do not create *DictSource* objects directly, add them to a *ConfigParser* object using *ConfigParser.add_source()*. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")

values_dict = {
    "config_item1": "v1",
    "config_item2": [1, 2],
    "config_item3": None,
}
parser.add_source("dict", values_dict)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for the `dict` source are:

- `source` (required, positional): `"dict"`.
- `values_dict` (required, positional): the `dict` containing the config values.

  Note that:

  - Values in `values_dict` for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict) should be values from the `none_values` list described below.

  - Values in `values_dict` for config items with `nargs >= 2`, `nargs == "+"` or `nargs == "*"` should be `list` objects with an element for each argument of the config item.

    In the special case where `nargs == "+"` or `nargs == "*"` and there is a single argument for the config item, the value may be given without the enclosing `list`, unless the argument is itself a `list`.

- `none_values` (optional, keyword): a list of values that, when seen in `values_dict`, should be treated as if they were not present (i.e. values for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict).

  The default `none_values` is `[None]`. Using a different `none_values` is useful if you want `None` to be treated as a valid config value.

- `priority` (optional, keyword): The priority for the source. The default priority for a `dict` source is `0`.

8.6 Creating your own source classes

To create your own source class, create a subclass of *`Source`*:

class multiconfparse.**Source**(*actions*, *priority=0*)
Abstract base for classes that parse config sources.

All config source classes should:

- Inherit from *`Source`*.
- Have a `source_name` class attribute containing the name of the source.
- Provide an implementation for the *`parse_config()`* method.
- Have an `__init__()` method that forwards its `actions` and `priority` arguments to `Source.__init__()`. `Source.__init__()` will create `actions` and `priority` attributes to make them available to subclass methods.

  `actions` is a `dict` with config item names as the keys and *`Action`* objects as the values. The *`Action`* attributes that are most useful for source classes to use are:

  - `name`: the name of the config item to which the *`Action`* applies. The source class should use this to determine which *`Action`* object corresponds with each config item mention in the source. The `name` atribute of an *`Action`* has the same value as the key in the `actions` `dict`.
  - `nargs`: this specifies the number of arguments/values that a config item should have when mentioned in the source.

parse_config()
Read the values of config items for this source.

This is an abstract method that subclasses must implement to return a `list` containing a *`ConfigMention`* element for each config item mentioned in the source, in the order in which they appear (unless order makes no sense for the source).

The implementation of this method will need to make use of the `actions` and `priority` attributes created by the *`Action`* base class.

class ConfigMention(*action*, *args*, *priority*)
A *`ConfigMention`* object represents a single mention of a config item in a source.

The arguments are:
- `action`: the *`Action`* object that corresponds with the config item being mentioned.
- `args`: a `list` of arguments that accompany the config item mention.

- `priority`: the priority of the mention. Generally, this is the same as the `priority` of the *`Source`* object that found the mention.

CHAPTER 9

API reference

class multiconfparse.**Action**(*name*, *dest=None*, *nargs=None*, *type=<class 'str'>*, *required=False*, *default=NOT_GIVEN*, *choices=None*, *help=None*, *include_sources=None*, *exclude_sources=None*)

Abstract base class config actions.

Classes to support actions should:

- Inherit from *Action*.
- Implement the *__call__()* method documented below.
- Have an `action_name` class attribute set to the name of the action that the class implements.
- Have an `__init__()` method that accepts arguments passed to *ConfigParser.add_config()* calls by the user (except the `action` argument) and which calls `Action.__init__()` with any of those arguments which are not specific to the action handled by the class. I.e.:
  - `name`;
  - `nargs`;
  - `type`;
  - `required`;
  - `default`;
  - `choices`;
  - `help`;
  - `dest`;
  - `include_sources`;
  - `exclude_sources`.

  These arguments will be assigned to attributes of the *Action* object being created (perhaps after some processing or validation) that are available for access by subclasses. The names of the attributes are the same as the argument names.

It is recommended that passing the arguments to `Action.__init__()` is done by the subclass `__init__` method accepting a `**kwargs` argument to collect any arguments that are not used or modified by the action class, then passing that `**kwargs` argument to `Action.__init__()`. The action class may also want pass some arguments that aren't specified by the user if the value of those arguments is implied by the action. For example, the `store_const` action class has the following `__init__` method:

```
def __init__(self, const, **kwargs):
    super().__init__(
        nargs=0,
        type=None,
        required=False,
        choices=None,
        **kwargs,
    )
    self.const = const
```

This ensures that an exception is raised if the user specifies `nargs`, `type`, `required`, or `choices` arguments when adding a `store_const` action because if the user specifies those arguments they will be given twice in the call to `Action.__init__()`.

`__call__`(*namespace*, *args*)

Combine the arguments from a mention of this config with any existing value.

This method is called once for each mention of the config item in the sources in order to combine the arguments from the mention with any existing value.

`namespace` will be the same *`Namespace`* object for all calls to this function during a *`ConfigParser.parse_config()`* call, and it is used to hold the so-far-accumulated value for the config item mentions.

This method's purpose is to combine the current value for this config item in `namespace` with the values in the `new` argument, and write the combined value into back into `namespace`.

The first time this method is called, if the config item has a default value, `namespace` will have an attribute for the config item and it will contain the default value; otherwise `namespace` will not have an attribute for the config item.

After the first call to this method, `namespace` should have an attribute value for the config item set by the previous call.

Notes:

- The name of the attribute in `namespace` for this config item is given by this object's `dest` attribute.
- The calls to this method are made in order of the priorities of the config item mentions in the sources, lowest priority first.
- The values in `new` have already been coerced to the config item's `type`.
- The values in `new` have already been checked for `choices` validity.
- The number of values in `new` has already been checked for `nargs` validity.
- If no arguments for the config item were given, `new` will just be an empty `list`.

For example, the *`__call__()`* method for the `append` action is:

```
def __call__(self, namespace, args):
    current = getattr(namespace, self.dest, [])
    if self.nargs == "?" and not args:
        current.append(self.const)
```

(continues on next page)

(continued from previous page)

```
    elif self.nargs is None or self.nargs == "?":
        assert len(args) == 1
        current.extend(args)
    else:
        current.append(args)
    setattr(namespace, self.dest, current)
```

The full example of the class for the `store_const` action is:

```
class StoreConstAction(Action):
    action_name = "store_const"

    def __init__(self, const, **kwargs):
        super().__init__(
            nargs=0,
            type=None,
            required=False,
            choices=None,
            **kwargs,
        )
        self.const = const

        def __call__(self, namespace, args):
            assert not args
            setattr(namespace, self.dest, self.const)
```

__call__(*namespace*, *args*)

Combine the arguments from a mention of this config with any existing value.

This method is called once for each mention of the config item in the sources in order to combine the arguments from the mention with any existing value.

`namespace` will be the same *`Namespace`* object for all calls to this function during a *`ConfigParser.parse_config()`* call, and it is used to hold the so-far-accumulated value for the config item mentions.

This method's purpose is to combine the current value for this config item in `namespace` with the values in the `new` argument, and write the combined value into back into `namespace`.

The first time this method is called, if the config item has a default value, `namespace` will have an attribute for the config item and it will contain the default value; otherwise `namespace` will not have an attribute for the config item.

After the first call to this method, `namespace` should have an attribute value for the config item set by the previous call.

Notes:

- The name of the attribute in `namespace` for this config item is given by this object's `dest` attribute.
- The calls to this method are made in order of the priorities of the config item mentions in the sources, lowest priority first.
- The values in `new` have already been coerced to the config item's `type`.
- The values in `new` have already been checked for `choices` validity.
- The number of values in `new` has already been checked for `nargs` validity.
- If no arguments for the config item were given, `new` will just be an empty `list`.

For example, the *__call__()* method for the `append` action is:

```
def __call__(self, namespace, args):
    current = getattr(namespace, self.dest, [])
    if self.nargs == "?" and not args:
        current.append(self.const)
    elif self.nargs is None or self.nargs == "?":
        assert len(args) == 1
        current.extend(args)
    else:
        current.append(args)
    setattr(namespace, self.dest, current)
```

class multiconfparse.**AppendAction**(*const=None*, *default=NOT_GIVEN*, ***kwargs*)

The `append` action stores the value for each mention of a config item in a `list`. The `list` is sorted according to the priorities of the mentions of the config item, lower priorities first. The Behaviour is based on the `append` `argparse` action.

Notes about the arguments to *ConfigParser.add_config()*:

- `nargs == 0` is not allowed. The default `nargs` value is `None`.

  When `nargs >= 1`, `nargs == "+"` or `nargs == "*"`, each value in the `list` for the config item is itself a `list` containing the arguments for a mention of the config item.

- The `const` argument is only accepted when `nargs == "?"`.
- The `default` argument (if it is given and is not *SUPPRESS*) is used as the initial `list` of values. This means that the `default` value is incorporated into the final value for the config item, even if the config item is mentioned in a source.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="append", default=["v1"])
parser.add_source("dict", {"config_item1": "v2"}, priority=2)
parser.add_source("dict", {"config_item1": "v3"}, priority=1)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": ["v1", "v3", "v2"],
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config(
    "config_item1",
    action="append",
    nargs="?",
    const="v0",
)
parser.parse_config()
parser.add_source("dict", {"config_item1": "v1"}, priority=2)
parser.add_source("dict", {"config_item1": None}, priority=1)
# -> multiconfparse.Namespace {
#   "config_item1": ["v0", "v1"],
# }
```

class multiconfparse.**ArgparseSource**(*actions*, *priority=20*)

Obtains config values from an `argparse.ArgumentParser`.

Do not create *ArgparseSource* objects directly, add them to a *ConfigParser* object using *ConfigParser.add_source()*. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")
argparse_source = parser.add_source("argparse")

argparse_parser = argparse.ArgumentParser()
argparse_parser.add_argument("arg1")
argparse_parser.add_argument("--opt1", type=int, action="append")
argparse_source.add_configs_to_argparse_parser(argparse_parser)

args = argparse_parser.parse_args((
    "arg1_value --config-item1 v1 --config-item2 1 2 --opt1 opt1_v1 "
    "--config-item-3 --opt1 opt1_v2"
).split())

argparse_source.notify_parsed_args(args)

config_parser.parse_config()
# -> multiconfparse.Namespace {
#    "config_item1": "v1",
#    "config_item2": [1, 2],
#    "config_item3": True,
# }
```

The `argparse` source does not create an `argparse.ArgumentParser` for you. This is to allow extra command line arguments to be added to an `argparse.ArgumentParser` that are not config items. Instead *ArgparseSource*, which implements the `argparse` source provides two methods to provide communication with the `argparse.ArgumentParser`:

add_configs_to_argparse_parser(*argparse_parser*)
Add arguments to an `argparse.ArgumentParser` for config items.

notify_parsed_args(*argparse_namespace*)
Notify the `argparse` source of the `argparse.Namespace` object returned by `argparse.ArgumentParser.parse_args()`.

If you don't need to add command line arguments other than for config items, see *SimpleArgparseSource* which implements the `simple_argparse` source.

The arguments of *ConfigParser.add_source()* for the `argparse` source are:

- `source` (required, positional): `"argparse"`
- `priority` (optional, keyword): The priority for the source. The default priority for an `argparse` source is `20`.

Note that:

- The name of the command line argument for a config item is the config item's name with underscores (_) converted to hyphens (-) and prefixed with `--`.

add_configs_to_argparse_parser(*argparse_parser*)
Add arguments to an `argparse.ArgumentParser` for config items.

notify_parsed_args(*argparse_namespace*)
Notify the `argparse` source of the `argparse.Namespace` object returned by `argparse.ArgumentParser.parse_args()`.

class multiconfparse.**ConfigMention**(*action*, *args*, *priority*)

A *ConfigMention* object represents a single mention of a config item in a source.

The arguments are:

- action: the *Action* object that corresponds with the config item being mentioned.
- args: a list of arguments that accompany the config item mention.
- priority: the priority of the mention. Generally, this is the same as the priority of the *Source* object that found the mention.

class multiconfparse.**ConfigParser**(*config_default=NOT_GIVEN*)

Create a new ConfigParser object. Options are:

- config_default: the default value to use in the *Namespace* returned by *parse_config()* for config items for which no value was found.

  The default behaviour (when config_default is *NOT_GIVEN*) is to represent these config items with the value None.

  Set config_default to *SUPPRESS* to prevent these configs from having an attribute set in the *Namespace* at all.

add_config(*name*, ***kwargs*)

Add a config item to the *ConfigParser*.

The arguments that apply to all config items are:

- name (required, positional): the name of the config item.

  In the *Namespace* object returned by *parse_config()*, the name of the attribute used for this config item will be name and must be a valid Python identifier.

  name is also used by source *Source* classes to generate the strings that will be used to find the config in config sources. The *Source* classes may, use a modified version of name, however. For example, the argparse and simple_argparse sources will convert underscores (_) to hyphens (-) and add a -- prefix, so if a config item had the name "config_item1", the argparse and simple_argparse sources would use the option string "--config-item1".

- action (optional, keyword): the name of the action that should be performed when a config item is found in a config source. The default action is "store", and the built-in actions are described briefly below. See *Actions* for more detailed information about the built-in actions and creating your own actions. The built-in actions are all based on argparse actions so the argparse documentation may also provide useful information.
  - store: this action just stores the highest priority value for config item.
  - store_const: this stores the value specified in the const argument.
  - store_true: this stores the value True and sets the default argument to False. It is a special case of store_const.
  - store_false: this stores the value False and sets the default argument to True. It is a special case of store_const.
  - append: this creates a list containing every value seen (with lower priority values first). When nargs >= 1, nargs == "+" or nargs == "*", each value in the list s iteself a list containing the arguments for a mention of the config item.
  - count: this stores the number of mentions of the config item.

  - `extend`: this creates a `list` containing every value seen (with lower priority values first). Unlike `append`, when `nargs >= 1`, `nargs == "+"` or `nargs == "*"`, arguments for mentions of the config item are not placed in separate sublists for each mention.

- `default`: the default value for this config item. Note that some actions will incorporate the `default` value into the final value for the config item even if the config item is mentioned in one of the sources (e.g. `append`, `count` and `extend`).

  Note that the default value for all config items can also be set by passing a value for the `config_default` argument of *`ConfigParser`*. If both the `config_default` argument to *`ConfigParser`* and the `default` argument to *`add_config()`* are used then only the `default` argument to *`add_config()`* is used.

  If a default value is not provided for the config item by the `default` argument, the `config_default` argument or by the action class (like e.g. *store_true* does), then the final value for the config will be `None` if the config item is not mentioned in any source.

  The special value *`SUPPRESS`* can be passed as the `default` argument. In this case, if the config item is not mentioned in any source, it will not be given an attribute in the *`Namespace`* object returned by *`parse_config()`*.

- `dest` (optional, keyword): the name of the attribute for the config item in the *`Namespace`* object returned by *`parse_config()`*. By default, `dest` is set to the name of the config item (`name`).

- `exclude_sources` (optional, keyword): a collection of source names or *`Source`* classes that should ignore this config item. This argument is mutually exclusive with `include_sources`. If neither `exclude_sources` nor `include_sources` is given, the config item will be looked for by all sources added to the *`ConfigParser`*.

- `include_sources` (optional, keyword): a collection of source names or *`Source`* classes that should look for this config item. This argument is mutually exclusive with `exclude_sources`. If neither `exclude_sources` nor `include_sources` is given, the config item will be looked for by all sources added to the *`ConfigParser`*.

- `help`: the help text/description for the config item. Set this to *`SUPPRESS`* to prevent this config item from being mentioned in generated documentation.

The other arguments are all keyword arguments and are passed on to the class that implements the config items action and may have different default values or may not even be valid for all actions. See *Actions* for action specific documentation.

- `nargs`: specifies the number of arguments that the config item accepts. The values that `nargs` can take are:

  - `None`: the config item will take a single argument.

  - `0`: the config item will take no arguments. This value is usually not given to *`add_config()`* but may be implicit for an action (e.g. `store_const` or `count`).

  - An `int` `N >= 1`: the config item will take `N` arguments and the value for a mention of the config item will be a `list` containing each argument. In particular, when `nargs == 1` the value for each mention of a config item will be a `list` containing a single element.

  - `"?"`: The config item will take a single optional argument. When the config item is mentioned without an accompanying value, the value for the mention is the value of the config item's `const` argument.

  - `"*"`: The config item will take zero or more arguments and the value for a mention of the config item will be a `list` containing each argument.

  - `"+"` The config item will take one or more arguments and the value for a mention of the config item will be a `list` containing each argument.

- `const`: The value to use for a mention of the config item where there is no accompanying argument. This is only ever used when `nargs == 0` or `nargs == "+"`.
- `type`: The type to which each argument of the config item should be converted. This can be any callable object that takes a single argument (an object with a `__call__(self, arg)` method), including classes like `int` and functions that take a single argument. Note that some sources that read typed data may produce config item argument values that aren't always `str` objects.

  The default `type` is `str` unless that doesn't make sense (e.g. when `nargs == 0`.
- `required`: specifies whether an exception should be raised if a value for this config item cannot be found in any source.

  The default `required` is `False`.
- `choices`: specifies a collection of valid values for the arguments of the config item. If `choices` is specified, an exception is raised if the config item is mentioned in a source with an argument that is not in `choices`.

add_source(*source*, **args*, ***kwargs*)
Add a new config source to the *ConfigParser*.

The only argument required for all sources is the `source` parameter which may be the name of a source or a class that implements a source. Other arguments are passed on to the class that implements the source.

The built-in sources are:

- `argparse`: for getting config values from the command line using an `argparse.ArgumentParser`.
- `simple_argparse`: a simpler version of the `argparse` source that is easier to use but doesn't allow you to add any arguments that aren't also config items.
- `environment`: for getting config values from environment variables.
- `json`: for getting config values from JSON files.
- `dict`: for getting config values from Python dictionaries.

See *Sources* for more information about the built-in sources and creating your own sources.

Return the created config source object.

parse_config()
Parse the config sources.

Returns: a *Namespace* object containing the parsed values.

partially_parse_config()
Parse the config sources, but don't raise a RequiredConfigNotFoundError exception if a required config is not found in any config source.

Returns: a Namespace object containing the parsed values.

class multiconfparse.**CountAction**(***kwargs*)
The `count` action stores the number of times a config item is mentioned in the config sources. Its behaviour is based on the `count` `argparse` action.

Notes about the arguments to *ConfigParser.add_config()*:

- `nargs` is not accepted as an argument. `nargs` is always `0` for `count` actions.
- `const` is not accepted as an argument - it doesn't make sense for `count`.
- `required` is not accepted as an argument. `required` is always `False` for `count` actions.

- `type` is not accepted as an argument - it doesn't make sense for `count`.
- `choices` is not accepted as an argument - it doesn't make sense for `count`.
- If the `default` argument is given and is not *`SUPPRESS`*, it acts as the initial value for the count. I.e. the final value for the config item will be the number of mentions of the config item in the sources, plus the value of `default`.

  Note that if the config item is not found in any sources and `default` is not given, it is *not* assumed to be `0`. The final value for the config item would be `None` in this case.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="count")
parser.add_source("dict", {"config_item1": None})
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": 2,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="count", default=10)
parser.add_source("dict", {"config_item1": None})
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": 12,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="count")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": None,
# }
```

class multiconfparse.**DictSource**(*actions*, *values_dict*, *none_values=None*, *priority=0*)

Obtains config values from a Python `dict` object.

Do not create *`DictSource`* objects directly, add them to a *`ConfigParser`* object using *`ConfigParser.add_source()`*. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")

values_dict = {
    "config_item1": "v1",
    "config_item2": [1, 2],
    "config_item3": None,
}
parser.add_source("dict", values_dict)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
```

(continues on next page)

(continued from previous page)

```
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for the `dict` source are:

- `source` (required, positional): `"dict"`.
- `values_dict` (required, positional): the `dict` containing the config values.

  Note that:

  – Values in `values_dict` for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict) should be values from the `none_values` list described below.

  – Values in `values_dict` for config items with `nargs >= 2`, `nargs == "+"` or `nargs == "*"` should be `list` objects with an element for each argument of the config item.

  In the special case where `nargs == "+"` or `nargs == "*"` and there is a single argument for the config item, the value may be given without the enclosing `list`, unless the argument is itself a `list`.

- `none_values` (optional, keyword): a list of values that, when seen in `values_dict`, should be treated as if they were not present (i.e. values for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict).

  The default `none_values` is `[None]`. Using a different `none_values` is useful if you want `None` to be treated as a valid config value.

- `priority` (optional, keyword): The priority for the source. The default priority for a `dict` source is `0`.

class multiconfparse.**EnvironmentSource**(*actions*, *none_values=None*, *priority=10*, *env_var_prefix=''*, *env_var_force_upper=True*)

Obtains config values from the environment.

Do not create *EnvironmentSource* objects directly, add them to a *ConfigParser* object using *ConfigParser.add_source()*. For example:

```
# For demonstration purposes, set some config values in environment
# variables
os.environ["MY_APP_CONFIG_ITEM1"] = "v1"
os.environ["MY_APP_CONFIG_ITEM2"] = "1 2"
os.environ["MY_APP_CONFIG_ITEM3"] = ""

parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")
parser.add_source("environment", env_var_prefix="MY_APP_")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for the `environment` source are:

- `source` (required, positional): `"environment"`

- `none_values` (optional, keyword): a list of values that, when seen in environment variables, should be treated as if they were not present (i.e. values for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict).

  The default `none_values` is `[""]`. using a different value for `none_values` is useful you want the empty string to be treated as a valid config value.

- `priority` (optional, keyword): The priority for the source. The default priority for an `environment` source is `10`.

- `env_var_prefix` (optional, keyword): a string prefixed to the environment variable names that the source will look for. The default value is `""`.

- `env_var_force_upper` (optional, keyword): force the environment variable name to be in upper case. Default is `True`.

Note that:

- Values in environment variables for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the environment variable) should be values from the `none_values` list described above.
- Values in environment variables for config items with `nargs >= 2`, `nargs == "+"` or `nargs == "*"` are split into arguments by `shlex.split()` (i.e. like arguments given on a command line via a shell). See the `shlex` documentation for full details.

class multiconfparse.**ExtendAction**(***kwargs*)

The `extend` action stores the value for each argument of each mention of a config item in a `list`. The `list` is sorted according to the priorities of the mentions of the config item, lower priorities first. The Behaviour is based on the `extend` `argparse` action, although the behaviour when `nargs == None` or `nargs == "?"` is different.

Notes about the arguments to *ConfigParser.add_config()*:

- `nargs == 0` is not allowed. The default `nargs` value is "+".

  Unlike the `append` action, when `nargs >= 1`, `nargs == "+"` or `nargs == "*"`, each value in the `list` for the config item is *not* itself a `list` containing the arguments for a mention of the config item. Each argument of each mention is added separately to the `list` that makes the final value for the config item.

  Unlike the `argparse` `extend` action, when `nargs == None` or `nargs == "?"`, the *multiconfparse* `extend` action behaves exactly like the `append` action.

- The `const` argument is only accepted when `nargs == "?"`.

- The `default` argument (if it is given and is not *SUPPRESS*) is used as the initial `list` of values. This means that the `default` value is incorporated into the final value for the config item, even if the config item is mentioned in a source.

Example:

```
parser = multiconfparse.ConfigParser()
parser.add_config(
    "config_item1",
    action="extend",
    default=[["v1", "v2"]]
)
parser.add_source("dict", {"config_item1": ["v3", "v4"]}, priority=2)
parser.add_source("dict", {"config_item1": ["v5"]}, priority=1)
parser.parse_config()
# -> multiconfparse.Namespace {
```

(continues on next page)

(continued from previous page)

```
#   "config_item1": ["v1", "v2", "v5", "v3", "v4"],
# }
```

exception multiconfparse.**InvalidChoiceError**(*action*, *value*)
Exception raised when a config value is not from a specified set of values.

exception multiconfparse.**InvalidNumberOfValuesError**(*action*)
Exception raised when the number of values supplied for a config item is not valid.

exception multiconfparse.**InvalidValueForNargs0Error**(*value*, *none_values*)
Exception raised when a value recieved for an action with nargs=0 is not a "none" value.

class multiconfparse.**JsonSource**(*actions*, *path=None*, *fileobj=None*, *none_values=None*, *json_none_values=None*, *priority=0*)
Obtains config values from a JSON file.

Do not create objects of this class directly - create them via *ConfigParser.add_source()*. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")

fileobj = io.StringIO('''
    {
        "config_item1": "v1",
        "config_item2": [1, 2],
        "config_item3": null
    }
''')
parser.add_source("json", fileobj=fileobj)

config_parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for `json` sources are:

- `source` (required, positional): `"json"`.
- `priority` (optional, keyword): The priority for the source. The default priority for a `json` source is `0`.
- `path` (optional, keyword): path to the JSON file to parse. Exactly one of the `path` and `fileobj` options must be given.
- `fileobj` (optional keyword): a file object representing a stream of JSON data. Exactly one of the `path` and `fileobj` options must be given.
- `none_values` (optional, keyword): a list of python values that, when seen as config item values after JSON decoding, should be treated as if they were not present (i.e. values for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict). The default `none_values` is `[]`.
- `json_none_values` (optional, keyword): a list of JSON values (as strings) that are decoded into Python values and added to `none_values`. The default `json_none_values` is `["null"]`.

Notes:

- The data in the JSON file should be a JSON object. Each config item value should be assigned to a field of the object that has the same name as the config item.
- Fields in the JSON object for config items with `nargs == 0` or `nargs == "?"` (where the `const` value should be used rather than the value from the dict) should either have values from the `json_none_values` list or should decode to values in the `none_values` list.
- Fields in the JSON object for config items with `nargs >= 2`, `nargs == "+"` or `nargs == "*"` should be JSON arrays with an element for each argument of the config item.

  In the special case where `nargs == "+"` or `nargs == "*"` and there is a single argument for the config item, the value may be given without the enclosing JSON array, unless the argument is itself an array.

multiconfparse.**NOT_GIVEN = NOT_GIVEN**

Singleton used to represent that an option or config item is not present.

This is used rather than `None` to distinguish between:

- the case where the user has provided an option with the value :data:None‘ and the user has not provided an option at all;
- the case where a config item has the value `None` and the case where the config item not been mentioned at all in the config.

class multiconfparse.**Namespace**

An object to hold values of config items.

Namespace objects are essentially plain objects used as the return values of *ConfigParser.parse_config()*. Retrieve values with normal attribute accesses:

```
config_values = parser.parse_config()
config_value = config_values.config_name
```

To set values (if, for example, you are implementing a *Source* subclass), use `setattr`:

```
ns = multiconfparse.Namespace()
setattr(ns, config_name, config_value)
```

exception multiconfparse.**ParseError**

Base class for exceptions indicating a configuration error.

exception multiconfparse.**RequiredConfigNotFoundError**

Exception raised when a required config value could not be found from any source.

multiconfparse.**SUPPRESS = SUPPRESS**

Singleton used as a `default` value for a config item to indicate that if no value is found for the config item in any source, it should not be given an attribute in the *Namespace* returned by *ConfigParser.parse_config()*. The default behaviour (when a `default` value is not given by the user) is for the *Namespace* returned by *ConfigParser.parse_config()* to have an attribute with a value of `None`.

class multiconfparse.**SimpleArgparseSource**(*actions*, *argument_parser_class=<class 'argparse.ArgumentParser'>*, *priority=20*, ***kwargs*)

Obtains config values from the command line.

The `simple_argparse` source is simpler to use than the `argparse` source but it doesn't allow adding arguments that are not config items.

Do not create objects of this class directly - create them via *ConfigParser.add_source()* instead. For example:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_config("config_item2", nargs=2, type=int)
parser.add_config("config_item3", action="store_true")
parser.add_source("simple_argparse")
config_parser.parse_config()
# If the command line looks something like:
#    PROG_NAME --config-item1 v1 --config-item2 1 2 --config-item-3
# The result would be:
# multiconfparse.Namespace {
#   "config_item1": "v1",
#   "config_item2": [1, 2],
#   "config_item3": True,
# }
```

The arguments of *ConfigParser.add_source()* for the `simple_argparse` source are:

- `source` (required, positional): `"simple_argparse"`
- `argument_parser_class` (optional, keyword): a class derived from `argparse.ArgumentParser` to use instead of `ArgumentParser` itself. This can be useful if you want to override `argparse.ArgumentParser.exit()` or `argparse.ArgumentParser.error()`.
- `priority` (optional, keyword): The priority for the source. The default priority for a `simple_argparse` source is `20`.
- Extra keyword arguments to pass to `argparse.ArgumentParser`. E.g. `prog`, `allow_help`. Don't use the `argument_default` option though - the `simple_argparse` sources sets this internally. See the `config_default` option for *ConfigParser* instead.

Note that:

- The name of the command line argument for a config item is the config item's name with underscores (_) converted to hyphens (-) and prefixed with `--`.

class `multiconfparse.`**Source**(*actions*, *priority=0*)

Abstract base for classes that parse config sources.

All config source classes should:

- Inherit from *Source*.
- Have a `source_name` class attribute containing the name of the source.
- Provide an implementation for the *parse_config()* method.
- Have an `__init__()` method that forwards its `actions` and `priority` arguments to `Source.__init__()`. `Source.__init__()` will create `actions` and `priority` attributes to make them available to subclass methods.

  `actions` is a `dict` with config item names as the keys and *Action* objects as the values. The *Action* attributes that are most useful for source classes to use are:

  - `name`: the name of the config item to which the *Action* applies. The source class should use this to determine which *Action* object corresponds with each config item mention in the source. The `name` atribute of an *Action* has the same value as the key in the `actions` `dict`.
  - `nargs`: this specifies the number of arguments/values that a config item should have when mentioned in the source.

parse_config()

Read the values of config items for this source.

This is an abstract method that subclasses must implement to return a `list` containing a *`ConfigMention`* element for each config item mentioned in the source, in the order in which they appear (unless order makes no sense for the source).

The implementation of this method will need to make use of the `actions` and `priority` attributes created by the *`Action`* base class.

class ConfigMention(*action*, *args*, *priority*)

A *`ConfigMention`* object represents a single mention of a config item in a source.

The arguments are:

- `action`: the *`Action`* object that corresponds with the config item being mentioned.
- `args`: a `list` of arguments that accompany the config item mention.
- `priority`: the priority of the mention. Generally, this is the same as the `priority` of the *`Source`* object that found the mention.

class multiconfparse.**StoreAction**(*const=None*, ***kwargs*)

The `store` action simply stores the value from the highest priority mention of a config item. Its behaviour is based on the `store` `argparse` action and is the default action.

Arguments to *`ConfigParser.add_config()`* have standard behaviour, but note:

- `nargs == 0` is not allowed. The default `nargs` value is `None`.
- The `const` argument is only accepted when `nargs == "?"`.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1")
parser.add_source("dict", {"config_item1": "v1"}, priority=2)
parser.add_source("dict", {"config_item1": "v2"}, priority=1)
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "v1",
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", nargs=2, type=int, default=[1, 2])
parser.add_source("simple_argparse")
parser.parse_config()
#
# If the command line looks something like:
#   prog some-arg --config-item1 3 4
# parse_config() will return something like:
#   multiconfparse.Namespace {
#       "config_item1": [3, 4],
#   }
#
# If the command line looks something like:
#   prog some-arg
# parse_config() will return something like:
#   multiconfparse.Namespace {
#       "config_item1": [1, 2],
#   }
```

class multiconfparse.**StoreConstAction**(*const*, ***kwargs*)

The `store_const` action stores the value from the `const` argument whenever a config item is mentioned in a source. Its behaviour is based on the `store_const` `argparse` action.

Notes about the arguments to *`ConfigParser.add_config()`*:

- The `const` argument is mandatory.
- `nargs` is not accepted as an argument. `nargs` is always `0` for `store_const` actions.
- `required` is not accepted as an argument. `required` is always `False` for `store_const` actions.
- `type` is not accepted as an argument - it doesn't make sense for `store_const`.
- `choices` is not accepted as an argument - it doesn't make sense for `store_const`.

Example:

```
parser = multiconfparse.ConfigParser()
parser.add_config(
    "config_item1",
    action="store_const",
    const="yes",
    default="no"
)
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": "yes",
# }
```

class multiconfparse.**StoreFalseAction**(*default=True*, ***kwargs*)

The `store_false` action simply stores the value `False` whenever a config item is mentioned in a source. Its behaviour is based on the `store_false` `argparse` action.

Notes about the arguments to *`ConfigParser.add_config()`*:

- `const` is not accepted as an argument - `const` is always `False` for `store_false` actions.
- `nargs` is not accepted as an argument. `nargs` is always `0` for `store_false` actions.
- `required` is not accepted as an argument. `required` is always `False` for `store_false` actions.
- `type` is not accepted as an argument - it doesn't make sense for `store_false`.
- `choices` is not accepted as an argument - it doesn't make sense for `store_false`.
- The default value for the `default` argument is `True`.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_false")
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": False,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_false")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": True,
# }
```

class multiconfparse.**StoreTrueAction**(*default=False*, ***kwargs*)

The `store_true` action simply stores the value `True` whenever a config item is mentioned in a source. Its behaviour is based on the `store_true` `argparse` action.

Notes about the arguments to *ConfigParser.add_config()*:

- `const` is not accepted as an argument - `const` is always `True` for `store_true` actions.
- `nargs` is not accepted as an argument. `nargs` is always `0` for `store_true` actions.
- `required` is not accepted as an argument. `required` is always `False` for `store_true` actions.
- `type` is not accepted as an argument - it doesn't make sense for `store_true`.
- `choices` is not accepted as an argument - it doesn't make sense for `store_true`.
- The default value for the `default` argument is `False`.

Examples:

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_true")
parser.add_source("dict", {"config_item1": None})
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": True,
# }
```

```
parser = multiconfparse.ConfigParser()
parser.add_config("config_item1", action="store_true")
parser.parse_config()
# -> multiconfparse.Namespace {
#   "config_item1": False,
# }
```

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

C

D

E

I

J

M

N

P

R

S