

## :mod:`configparser` --- Configuration file parser

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 1); [backlink](#)**  
Unknown interpreted text role "mod".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 4)**  
Unknown directive type "module".  
  
.. module:: configparser  
    :synopsis: Configuration file parser.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 7)**  
Unknown directive type "moduleauthor".  
  
.. moduleauthor:: Ken Manheimer <klm@zope.com>

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 8)**  
Unknown directive type "moduleauthor".  
  
.. moduleauthor:: Barry Warsaw <bwarshaw@python.org>

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 9)**  
Unknown directive type "moduleauthor".  
  
.. moduleauthor:: Eric S. Raymond <esr@thyrsus.com>

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 10)**  
Unknown directive type "moduleauthor".  
  
.. moduleauthor:: Łukasz Langa <lukasz@langa.pl>

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 11)**  
Unknown directive type "sectionauthor".  
  
.. sectionauthor:: Christopher G. Petrilli <petrilli@amber.org>

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 12)**  
Unknown directive type "sectionauthor".  
  
.. sectionauthor:: Łukasz Langa <lukasz@langa.pl>

Source code: [source:Lib/configparser.py](#)

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 14); [backlink](#)**  
Unknown interpreted text role "source".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 16)**  
Unknown directive type "index".  
  
.. index::  
    pair: .ini; file  
    pair: configuration; file  
    single: ini file  
    single: Windows ini file

This module provides the `class:ConfigParser` class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 24); [backlink](#)**  
Unknown interpreted text role "class".

**Note**  
This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 34)**  
Unknown directive type "seealso".  
  
.. seealso::  
  
    Module :mod:`shlex`  
        Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.  
  
    Module :mod:`json`  
        The json module implements a subset of JavaScript syntax which can also be used for this purpose.

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 45)
```

Unknown directive type "testsetup".

```
.. testsetup::

    import configparser
```

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 49)
```

Unknown directive type "testcleanup".

```
.. testcleanup::

    import os
    os.remove("example.ini")
```

## Quick Start

Let's take a very basic configuration file that looks like this:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described in the following section. Essentially, the file consists of sections, each of which contains keys with values. `mod:configparser` classes can read and write such files. Let's start by creating the above configuration file programmatically.

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 75); backlink
```

Unknown interpreted text role "mod".

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 81)
```

Unknown directive type "doctest".

```
.. doctest::

    >>> import configparser
    >>> config = configparser.ConfigParser()
    >>> config['DEFAULT'] = {'ServerAliveInterval': '45',
    ...                   'Compression': 'yes',
    ...                   'CompressionLevel': '9'}
    >>> config['bitbucket.org'] = {}
    >>> config['bitbucket.org']['User'] = 'hg'
    >>> config['topsecret.server.com'] = {}
    >>> topsecret = config['topsecret.server.com']
    >>> topsecret['Port'] = '50022' # mutates the parser
    >>> topsecret['ForwardX11'] = 'no' # same here
    >>> config['DEFAULT']['ForwardX11'] = 'yes'
    >>> with open('example.ini', 'w') as configfile:
    ...     config.write(configfile)
    ...
```

As you can see, we can treat a config parser much like a dictionary. There are differences, outlined later, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 106)
```

Unknown directive type "doctest".

```
.. doctest::

    >>> config = configparser.ConfigParser()
    >>> config.sections()
    []
    >>> config.read('example.ini')
    ['example.ini']
    >>> config.sections()
    ['bitbucket.org', 'topsecret.server.com']
    >>> 'bitbucket.org' in config
    True
    >>> 'bytebong.com' in config
    False
    >>> config['bitbucket.org']['User']
    'hg'
    >>> config['DEFAULT']['Compression']
    'yes'
    >>> topsecret = config['topsecret.server.com']
    >>> topsecret['ForwardX11']
    'no'
    >>> topsecret['Port']
    '50022'
    >>> for key in config['bitbucket.org']: # doctest: +SKIP
    ...     print(key)
    user
    compressionlevel
    serveraliveinterval
    compression
    forwardx11
    >>> config['bitbucket.org']['ForwardX11']
    'yes'
```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections [1]. Note also that keys in sections are case-insensitive and stored in lowercase [1].

It is possible to read several configurations into a single `class: ConfigParser`, where the most recently added configuration has the

highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained.

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 143); backlink
```

Unknown interpreted text role "class".

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 148)
```

Unknown directive type "doctest".

```
.. doctest::

    >>> another_config = configparser.ConfigParser()
    >>> another_config.read('example.ini')
    ['example.ini']
    >>> another_config['topsecret.server.com']['Port']
    '50022'
    >>> another_config.read_string("[topsecret.server.com]\nPort=48484")
    >>> another_config['topsecret.server.com']['Port']
    '48484'
    >>> another_config.read_dict({"topsecret.server.com": {"Port": 21212}})
    >>> another_config['topsecret.server.com']['Port']
    '21212'
    >>> another_config['topsecret.server.com']['ForwardX11']
    'no'
```

This behaviour is equivalent to a `meth:~ConfigParser.read` call with several files passed to the *filenames* parameter.

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 164); backlink
```

Unknown interpreted text role "meth".

## Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 175)
```

Unknown directive type "doctest".

```
.. doctest::

    >>> int(topsecret['Port'])
    50022
    >>> float(topsecret['CompressionLevel'])
    9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide `meth:~ConfigParser.getboolean`. This method is case-insensitive and recognizes Boolean values from 'yes'/'no', 'on'/'off', 'true'/'false' and '1'/'0' [1]. For example:

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 182); backlink
```

Unknown interpreted text role "meth".

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 190)
```

Unknown directive type "doctest".

```
.. doctest::

    >>> topsecret.getboolean('ForwardX11')
    False
    >>> config['bitbucket.org'].getboolean('ForwardX11')
    True
    >>> config.getboolean('bitbucket.org', 'Compression')
    True
```

Apart from `meth:~ConfigParser.getboolean`, config parsers also provide equivalent `meth:~ConfigParser.getint` and `meth:~ConfigParser.getfloat` methods. You can register your own converters and customize the provided ones. [1]

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 199); backlink
```

Unknown interpreted text role "meth".

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 199); backlink
```

Unknown interpreted text role "meth".

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 199); backlink
```

Unknown interpreted text role "meth".

## Fallback Values

As with a dictionary, you can use a section's `meth:~get` method to provide fallback values:

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 207); backlink
```

Unknown interpreted text role "meth".

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 210)
```

Unknown directive type "doctest".

```
.. doctest::
```

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the 'CompressionLevel' key was specified only in the 'DEFAULT' section. If we try to get it from the section 'topsecret.server.com', we will always get the default, even if we specify a fallback:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 226)**

Unknown directive type "doctest".

```
.. doctest::

>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level `meth.get` method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 231); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 236)**

Unknown directive type "doctest".

```
.. doctest::

>>> config.get('bitbucket.org', 'monster',
...             fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the `meth.getint`, `meth.getfloat` and `meth.getboolean` methods, for example:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 242); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 242); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 242); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 246)**

Unknown directive type "doctest".

```
.. doctest::

>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

## Supported INI File Structure

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or : by default [1]). By default, section names are case sensitive but keys are not [1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted if the parser is configured to allow it [1], in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

By default, a valid section name can be any string that does not contain `'\n'` or `'\r'`. To change this, see `attr: ConfigParser.SECTCRE`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main) [Doc] [library] configparser.rst, line 270); [backlink](#)**

Unknown interpreted text role "attr".

Configuration files may include comments, prefixed by specific characters (`#` and `;` by default [1]). Comments may appear on their own on an otherwise empty line, possibly indented. [1]

For example:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
      I sleep all night and I work all day
```

```
[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

[Sections Can Be Indented]
    can_values_be_as_well = True
    does_that_mean_anything_special = False
    purpose = formatting for readability
    multiline_values = are
        handled just fine as
        long as they are indented
        deeper than the first line
        of a value
    # Did I mention we can indent comments, too?
```

## Interpolation of values

On top of the core functionality, `xclass:ConfigParser` supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 327); [backlink](#)**  
Unknown interpreted text role "class".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 331)**  
Unknown directive type "index".  
  
.. index:: single: % (percent); interpolation in configuration files

The default implementation used by `xclass:ConfigParser`. It enables values to contain format strings which refer to other values in the same section, or values in the special default section [1]. Additional default values can be provided on initialization.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 335); [backlink](#)**  
Unknown interpreted text role "class".

For example:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a % to escape the % sign (% is the only character that needs to be escaped):
gain: 80%
```

In the example above, `xclass:ConfigParser` with `interpolation` set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 353); [backlink](#)**  
Unknown interpreted text role "class".

With interpolation set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 364)**  
Unknown directive type "index".  
  
.. index:: single: \$ (dollar); interpolation in configuration files

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `$(section:option)` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character that needs to be escaped):
cost: $$80
```

Values from other sections can be fetched as well:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

## Mapping Protocol Access

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 413)

Unknown directive type "versionadded".

.. versionadded:: 3.2

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of `mod:configparser`, the mapping interface implementation is using the `parser['section']['option']` notation.

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 415); [backlink](#)

Unknown interpreted text role "mod".

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

`mod:configparser` objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the `class:~collections.abc.MutableMapping` ABC. However, there are a few differences that should be taken into account:

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 426); [backlink](#)

Unknown interpreted text role "mod".

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 426); [backlink](#)

Unknown interpreted text role "class".

- By default, all keys in sections are accessible in a case-insensitive manner [1]. E.g. for `option` in `parser["section"]` yields only `optionxform`'ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `'a'`, both expressions return `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a `KeyError`.

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 439); [backlink](#)

Unknown interpreted text role "exc".

- `DEFAULTSECT` cannot be removed from the parser:
  - trying to delete it raises `ValueError`,

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 448); [backlink](#)

Unknown interpreted text role "exc".

- `parser.clear()` leaves it intact,
- `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the `section-level get()` methods are compatible both with the mapping protocol and the classic `configparser` API.
- `parser.items()` is compatible with the mapping protocol (returns a list of `section_name, section_proxy` pairs including the `DEFAULTSECT`). However, this method can also be invoked with arguments: `parser.items(section, raw, vars)`. The latter call returns a list of `option, value` pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

## Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. `mod:configparser` goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 473); [backlink](#)

Unknown interpreted text role "mod".

The most common way to change the way a specific config parser works is to use the `meth: __init__` options:

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 479); [backlink](#)

Unknown interpreted text role "meth".

- `defaults`, default value: `None`

This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don't specify values which are the same as the documented default.

Hint: if you want to specify default values for a specific section, use `meth:read_dict` before you read the actual file.

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 489); [backlink](#)

Unknown interpreted text role "meth".

- `dict_type`, default value: `class:dict`

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst,

[line 492](#); [backlink](#)

Unknown interpreted text role "class".

This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the standard dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

An alternative dictionary type can be used for example to sort sections and options on write-back.

Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered. For example:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 506)**

Unknown directive type "doctest".

```
.. doctest::

    >>> parser = configparser.ConfigParser()
    >>> parser.read_dict({'section1': {'key1': 'value1',
    ...                             'key2': 'value2',
    ...                             'key3': 'value3'},
    ...                 'section2': {'keyA': 'valueA',
    ...                             'keyB': 'valueB',
    ...                             'keyC': 'valueC'},
    ...                 'section3': {'foo': 'x',
    ...                             'bar': 'y',
    ...                             'baz': 'z'}
    ... })
    >>> parser.sections()
    ['section1', 'section2', 'section3']
    >>> [option for option in parser['section3']]
    ['foo', 'bar', 'baz']
```

- `allow_no_value`, default value: `False`

Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by `mod: configparser`. The `allow_no_value` parameter to the constructor can be used to indicate that such values should be accepted:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 526); [backlink](#)**

Unknown interpreted text role "mod".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 531)**

Unknown directive type "doctest".

```
.. doctest::

    >>> import configparser

    >>> sample_config = """
    ... [mysqld]
    ...     user = mysql
    ...     pid-file = /var/run/mysqld/mysqld.pid
    ...     skip-external-locking
    ...     old_passwords = 1
    ...     skip-bdb
    ...     # we don't need ACID today
    ...     skip-innodb
    ... """
    >>> config = configparser.ConfigParser(allow_no_value=True)
    >>> config.read_string(sample_config)

    >>> # Settings with values are treated as before:
    >>> config["mysqld"]["user"]
    'mysql'

    >>> # Settings without values provide None:
    >>> config["mysqld"]["skip-bdb"]

    >>> # Settings which aren't specified still raise an error:
    >>> config["mysqld"]["does-not-exist"]
    Traceback (most recent call last):
    ...
    KeyError: 'does-not-exist'
```

- `delimiters`, default value: `('=', ':')`

Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

See also the `space_around_delimiters` argument to `meth: ConfigParser.write`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 567); [backlink](#)**

Unknown interpreted text role "meth".

- `comment_prefixes`, default value: `(';', ':')`
- `inline_comment_prefixes`, default value: `None`

Comment prefixes are strings that indicate the start of a valid comment within a config file. `comment_prefixes` are used only on otherwise empty lines (optionally indented) whereas `inline_comment_prefixes` can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `'#'` and `':'` are used as prefixes for whole line comments.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 581)**

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.2
    In previous versions of :mod:`configparser` behaviour matched
    ``comment_prefixes=(';', ':')`` and ``inline_comment_prefixes=(';', ':')``.
```

Please note that config parsers don't support escaping of comment prefixes so using `inline_comment_prefixes` may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting

*inline\_comment\_prefixes*. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
... ${hash}!/usr/bin/env python
... ${hash} -*- coding: utf-8 -*-
...
... extensions =
... enabled_extension
... another_extension
... #disabled_by_comment
... yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
... line #2
... line #3
... """)
>>> print(parser['hashes']['shebang'])
<BLANKLINE>
#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])
<BLANKLINE>
enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, default value: `True`

When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using `meth: read_file`, `meth: read_string` or `meth: read_dict`). It is recommended to use strict parsers in new applications.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 633); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 633); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 633); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 638)**

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.2
   In previous versions of :mod:`configparser` behaviour matched
   ``strict=False``.
```

- *empty\_lines\_in\_values*, default value: `True`

In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

```
[Section]
key = multiline
    value with a gotcha

this = is still a part of the multiline value of 'key'
```

This can be especially problematic for the user to see if she's using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

- *default\_section*, default value: `configparser.DEFAULTSECT` (that is: `"DEFAULT"`)

The convention of allowing a special section of default values for other sections or interpolation purposes is a powerful concept of this library, letting users create complex declarative configurations. This section is normally called `"DEFAULT"` but this can be customized to point to any other valid section name. Some typical values include: `"general"` or `"common"`. The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

- *interpolation*, default value: `configparser.BasicInterpolation`

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the [dedicated documentation section](#). `xclass: RawConfigParser` has a default value of `None`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 681); [backlink](#)**

Unknown interpreted text role "class".

- *converters*, default value: not set

Config parsers provide option value getters that perform type conversion. By default `meth: ~ConfigParser.getint`, `meth: ~ConfigParser.getfloat`, and `meth: ~ConfigParser.getboolean` are implemented. Should other getters be desirable,



users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing `{'decimal': decimal.Decimal}` would add `meth: getdecimal` on both the parser object and all section proxies. In other words, it will be possible to write both `parser_instance.getdecimal('section', 'key', fallback=0)` and `parser_instance['section'].getdecimal('key', 0)`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 690); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 690); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 690); [backlink](#)**

Unknown interpreted text role "meth".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 690); [backlink](#)**

Unknown interpreted text role "meth".

If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with `get`, it will be available on all section proxies, in the dict-compatible form (see the `getdecimal()` example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 710)**

Unknown directive type "attribute".

```
.. attribute:: ConfigParser.BOOLEAN_STATES
```

By default when using `meth:~ConfigParser.getboolean`, config parsers consider the following values `True`: `'1'`, `'yes'`, `'true'`, `'on'` and the following values `False`: `'0'`, `'no'`, `'false'`, `'off'`. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```
.. doctest::
```

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include `'accept'/'reject'` or `'enabled'/'disabled'`.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 733)**

Unknown directive type "method".

```
.. method:: ConfigParser.optionxform(option)
:nomindex:
```

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example:

```
.. doctest::
```

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
...
...
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

```
.. note::
```

The `optionxform` function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 771)**

Unknown directive type "attribute".

```
.. attribute:: ConfigParser.SECTCRE
```

A compiled regular expression used to parse section headers. The default matches `'[section]'` to the name `'section'`. Whitespace is considered

```
part of the section name, thus ``[ larch ]`` will be read as a section of
name ``" larch "``. Override this attribute if that's unsuitable. For
example:
```

```
.. doctest::

>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [ Section 2 ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', ' Section 2 ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^\]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']

.. note::

While ConfigParser objects also use an ``OPTCRE`` attribute for recognizing
option lines, it's not recommended to override it because that would
interfere with constructor options *allow_no_value* and *delimiters*.
```

## Legacy API Examples

Mainly because of backwards compatibility concerns, `mod:configparser` provides also a legacy API with explicit `get/set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 809); [backlink](#)**  
Unknown interpreted text role "mod".

An example of writing to a configuration file:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use `class:ConfigParser`:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library]configparser.rst, line 856); [backlink](#)**  
Unknown interpreted text role "class".

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False)) # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))  # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
# -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
# -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
# -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
# -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser
```

```
# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))    # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))    # -> "Life is hard!"
```

## ConfigParser Objects

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 909)**

Invalid class attribute value for "class" directive: "ConfigParser(defaults=None, dict\_type=dict, allow\_no\_value=False, delimiters=('=', ':'), comment\_prefixes=(';', '#'), inline\_comment\_prefixes=None, strict=True, empty\_lines\_in\_values=True, default\_section=configparser.DEFAULTSECT, interpolation=BasicInterpolation(), converters={})".

```
.. class:: ConfigParser(defaults=None, dict_type=dict, allow_no_value=False, delimiters=('=', ':'), comment_prefixes=(';', '#'), in
```

The main configuration parser. When `*defaults*` is given, it is initialized into the dictionary of intrinsic defaults. When `*dict_type*` is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When `*delimiters*` is given, it is used as the set of substrings that divide keys from values. When `*comment_prefixes*` is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When `*inline_comment_prefixes*` is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When `*strict*` is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising `exc:DuplicateSectionError` or `exc:DuplicateOptionError`. When `*empty_lines_in_values*` is `False` (default: `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When `*allow_no_value*` is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When `*default_section*` is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `DEFAULT`). This value can be retrieved and changed on runtime using the `default_section` instance attribute.

Interpolation behaviour may be customized by providing a custom handler through the `*interpolation*` argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the `dedicated documentation section <#interpolation-of-values>`.

All option names used in interpolation will be passed through the `:meth:'optionxform'` method just like any other option name reference. For example, using the default implementation of `:meth:'optionxform'` (which converts option names to lower case), the values `'foo %(bar)s'` and `'foo %(BAR)s'` are equivalent.

When `*converters*` is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `:meth:'get*()'` method on the parser object and section proxies.

```
.. versionchanged:: 3.1
   The default *dict_type* is :class:'collections.OrderedDict'.
```

```
.. versionchanged:: 3.2
   *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*,
   *empty_lines_in_values*, *default_section* and *interpolation* were
   added.
```

```
.. versionchanged:: 3.5
   The *converters* argument was added.
```

```
.. versionchanged:: 3.7
   The *defaults* argument is read with :meth:'read_dict()',
   providing consistent behavior across the parser: non-string
   keys and values are implicitly converted to strings.
```

```
.. versionchanged:: 3.8
   The default *dict_type* is :class:'dict', since it now preserves
   insertion order.
```

```
.. method:: defaults()

   Return a dictionary containing the instance-wide defaults.
```

```
.. method:: sections()

   Return a list of the sections available; the *default section* is not
   included in the list.
```

```
.. method:: add_section(section)

   Add a section named *section* to the instance. If a section by the given
   name already exists, exc:DuplicateSectionError is raised. If the
   *default section* name is passed, exc:ValueError is raised. The name
   of the section must be a string; if not, exc:TypeError is raised.
```

```
.. versionchanged:: 3.2
   Non-string section names raise exc:TypeError.
```

```
.. method:: has_section(section)

   Indicates whether the named *section* is present in the configuration.
   The *default section* is not acknowledged.
```

```
.. method:: options(section)

   Return a list of options available in the specified *section*.
```

```
.. method:: has_option(section, option)

   If the given *section* exists, and contains the given *option*, return
   :const:'True'; otherwise return :const:'False'. If the specified
   *section* is :const:'None' or an empty string, DEFAULT is assumed.
```

```
.. method:: read(filenames, encoding=None)

   Attempt to read and parse an iterable of filenames, returning a list of
```

filenames which were successfully parsed.

If `*filenames*` is a string, a `:class:`bytes`` object or a `:term:`path-like object``, it is treated as a single filename. If a file named in `*filenames*` cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the `:class:`ConfigParser`` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `:meth:`read_file`` before calling `:meth:`read`` for any optional files::

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

.. versionadded:: 3.2  
The `*encoding*` parameter. Previously, all files were read using the default encoding for `:func:`open``.

.. versionadded:: 3.6.1  
The `*filenames*` parameter accepts a `:term:`path-like object``.

.. versionadded:: 3.7  
The `*filenames*` parameter accepts a `:class:`bytes`` object.

.. method:: read\_file(f, source=None)

Read and parse configuration data from `*f*` which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument `*source*` specifies the name of the file being read. If not given and `*f*` has a `:attr:`name`` attribute, that is used for `*source*`; the default is `''<???'>''`.

.. versionadded:: 3.2  
Replaces `:meth:`readfp``.

.. method:: read\_string(string, source=<string>')

Parse configuration data from a string.

Optional argument `*source*` specifies a context-specific name of the string passed. If not given, `''<string>''` is used. This should commonly be a filesystem path or a URL.

.. versionadded:: 3.2

.. method:: read\_dict(dictionary, source=<dict>')

Load configuration from any object that provides a dict-like ```items()``` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument `*source*` specifies a context-specific name of the dictionary passed. If not given, `''<dict>''` is used.

This method can be used to copy state between parsers.

.. versionadded:: 3.2

.. method:: get(section, option, \*, raw=False, vars=None[, fallback])

Get an `*option*` value for the named `*section*`. If `*vars*` is provided, it must be a dictionary. The `*option*` is looked up in `*vars*` (if provided), `*section*`, and in `DEFAULTSECT` in that order. If the key is not found and `*fallback*` is provided, it is used as a fallback value. ```None``` can be provided as a `*fallback*` value.

All the ```%``` interpolations are expanded in the return values, unless the `*raw*` argument is true. Values for interpolation keys are looked up in the same manner as the option.

.. versionchanged:: 3.2  
Arguments `*raw*`, `*vars*` and `*fallback*` are keyword only to protect users from trying to use the third argument as the `*fallback*` fallback (especially when using the mapping protocol).

.. method:: getint(section, option, \*, raw=False, vars=None[, fallback])

A convenience method which coerces the `*option*` in the specified `*section*` to an integer. See `:meth:`get`` for explanation of `*raw*`, `*vars*` and `*fallback*`.

.. method:: getfloat(section, option, \*, raw=False, vars=None[, fallback])

A convenience method which coerces the `*option*` in the specified `*section*` to a floating point number. See `:meth:`get`` for explanation of `*raw*`, `*vars*` and `*fallback*`.

.. method:: getboolean(section, option, \*, raw=False, vars=None[, fallback])

A convenience method which coerces the `*option*` in the specified `*section*` to a Boolean value. Note that the accepted values for the option are ```1```, ```yes```, ```true```, and ```on```, which cause this method to return ```True```, and ```0```, ```no```, ```false```, and ```off```, which cause it to return ```False```. These string values are checked in a case-insensitive manner. Any other value will cause it to raise `:exc:`ValueError``. See `:meth:`get`` for explanation of `*raw*`, `*vars*` and `*fallback*`.

.. method:: items(raw=False, vars=None)

items(section, raw=False, vars=None)

When `*section*` is not given, return a list of `*section_name*`, `*section_proxy*` pairs, including `DEFAULTSECT`.

Otherwise, return a list of `*name*`, `*value*` pairs for the options in the given `*section*`. Optional arguments have the same meaning as for the `:meth:`get`` method.

.. versionchanged:: 3.8  
Items present in `*vars*` no longer appear in the result. The previous

```

behaviour mixed actual parser options with variables provided for
interpolation.

.. method:: set(section, option, value)

    If the given section exists, set the given option to the specified value;
    otherwise raise :exc:`NoSectionError`. *option* and *value* must be
    strings; if not, :exc:`TypeError` is raised.

.. method:: write(fileobject, space_around_delimiters=True)

    Write a representation of the configuration to the specified :term:`file
    object`, which must be opened in text mode (accepting strings). This
    representation can be parsed by a future :meth:`read` call. If
    *space_around_delimiters* is true, delimiters between
    keys and values are surrounded by spaces.

.. note::

    Comments in the original configuration file are not preserved when
    writing the configuration back.
    What is considered a comment, depends on the given values for
    *comment_prefix* and *inline_comment_prefix*.

.. method:: remove_option(section, option)

    Remove the specified *option* from the specified *section*. If the
    section does not exist, raise :exc:`NoSectionError`. If the option
    existed to be removed, return :const:`True`; otherwise return
    :const:`False`.

.. method:: remove_section(section)

    Remove the specified *section* from the configuration. If the section in
    fact existed, return ``True``. Otherwise return ``False``.

.. method:: optionxform(option)

    Transforms the option name *option* as found in an input file or as passed
    in by client code to the form that should be used in the internal
    structures. The default implementation returns a lower-case version of
    *option*; subclasses may override this or client code can set an attribute
    of this name on instances to affect this behavior.

    You don't need to subclass the parser to use this method, you can also
    set it on an instance, to a function that takes a string argument and
    returns a string. Setting it to ``str``, for example, would make option
    names case sensitive:

        cfgparser = ConfigParser()
        cfgparser.optionxform = str

    Note that when reading configuration files, whitespace around the option
    names is stripped before :meth:`optionxform` is called.

.. method:: readfp(fp, filename=None)

.. deprecated:: 3.2
    Use :meth:`read_file` instead.

.. versionchanged:: 3.2
    :meth:`readfp` now iterates on *fp* instead of calling ``fp.readline()``.

    For existing code calling :meth:`readfp` with arguments which don't
    support iteration, the following generator may be used as a wrapper
    around the file-like object::

        def readline_generator(fp):
            line = fp.readline()
            while line:
                yield line
                line = fp.readline()

    Instead of ``parser.readfp(fp)`` use
    ``parser.read_file(readline_generator(fp))``.

```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1231)**

Unknown directive type "data".

```

.. data:: MAX_INTERPOLATION_DEPTH

    The maximum depth for recursive interpolation for :meth:`get` when the *raw*
    parameter is false. This is relevant only when the default *interpolation*
    is used.

```

## RawConfigParser Objects

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1243)**

Invalid class attribute value for "class" directive: "RawConfigParser(defaults=None, dict\_type=dict, \allow\_no\_value=False, \*, delimiters=('=', ':'), \comment\_prefixes=(';', ':'), \inline\_comment\_prefixes=None, strict=True, \empty\_lines\_in\_values=True, \default\_section=configparser.DEFAULTSECT[, \interpolation])".

```

.. class:: RawConfigParser(defaults=None, dict_type=dict, \
    allow_no_value=False, *, delimiters=('=', ':'), \
    comment_prefixes=(';', ':'), \
    inline_comment_prefixes=None, strict=True, \
    empty_lines_in_values=True, \
    default_section=configparser.DEFAULTSECT[, \
    interpolation])

    Legacy variant of the :class:`ConfigParser`. It has interpolation
    disabled by default and allows for non-string section names, option
    names, and values via its unsafe ``add_section`` and ``set`` methods,
    as well as the legacy ``defaults`` keyword argument handling.

.. versionchanged:: 3.8
    The default *dict_type* is :class:`dict`, since it now preserves
    insertion order.

.. note::
    Consider using :class:`ConfigParser` instead which checks types of
    the values to be stored internally. If you don't want interpolation, you
    can use ``ConfigParser(interpolation=None)``.

```

```
.. method:: add_section(section)

Add a section named *section* to the instance. If a section by the given
name already exists, :exc:`DuplicateSectionError` is raised. If the
*default section* name is passed, :exc:`ValueError` is raised.

Type of *section* is not checked which lets users create non-string named
sections. This behaviour is unsupported and may cause internal errors.

.. method:: set(section, option, value)

If the given section exists, set the given option to the specified value;
otherwise raise :exc:`NoSectionError`. While it is possible to use
:class:`RawConfigParser` (or :class:`ConfigParser` with *raw* parameters
set to true) for *internal* storage of non-string values, full
functionality (including interpolation and output to files) can only be
achieved using string values.

This method lets users assign non-string values to keys internally. This
behaviour is unsupported and will cause errors when attempting to write
to a file or get it in non-raw mode. **Use the mapping protocol API**
which does not allow such assignments to take place.
```

## Exceptions

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1294)**

Unknown directive type "exception".

```
.. exception:: Error

Base class for all other :mod:`configparser` exceptions.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1299)**

Unknown directive type "exception".

```
.. exception:: NoSectionError

Exception raised when a specified section is not found.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1304)**

Unknown directive type "exception".

```
.. exception:: DuplicateSectionError

Exception raised if :meth:`add_section` is called with the name of a section
that is already present or in strict parsers when a section is found more
than once in a single input file, string or dictionary.

.. versionadded:: 3.2
Optional ``source`` and ``lineno`` attributes and arguments to
:meth:`__init__` were added.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1315)**

Unknown directive type "exception".

```
.. exception:: DuplicateOptionError

Exception raised by strict parsers if a single option appears twice during
reading from a single file, string or dictionary. This catches misspellings
and case sensitivity-related errors, e.g. a dictionary may have two keys
representing the same case-insensitive configuration key.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1323)**

Unknown directive type "exception".

```
.. exception:: NoOptionError

Exception raised when a specified option is not found in the specified
section.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1329)**

Unknown directive type "exception".

```
.. exception:: InterpolationError

Base class for exceptions raised when problems occur performing string
interpolation.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\cpython-main [Doc] [library] configparser.rst, line 1335)**

Unknown directive type "exception".

```
.. exception:: InterpolationDepthError

Exception raised when string interpolation cannot be completed because the
number of iterations exceeds :const:`MAX_INTERPOLATION_DEPTH`. Subclass of
:exc:`InterpolationError`.
```

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-**

main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 1342)

Unknown directive type "exception".

```
.. exception:: InterpolationMissingOptionError
```

```
Exception raised when an option referenced from a value does not exist.
Subclass of :exc:`InterpolationError`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 1348)

Unknown directive type "exception".

```
.. exception:: InterpolationSyntaxError
```

```
Exception raised when the source text into which substitutions are made does
not conform to the required syntax. Subclass of :exc:`InterpolationError`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 1354)

Unknown directive type "exception".

```
.. exception:: MissingSectionHeaderError
```

```
Exception raised when attempting to parse a file which has no section
headers.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main] [Doc] [library] configparser.rst, line 1360)

Unknown directive type "exception".

```
.. exception:: ParsingError
```

```
Exception raised when errors occur attempting to parse a file.
```

```
.. versionchanged:: 3.2
```

```
The ``filename`` attribute and :meth:`~__init__` argument were renamed to
``source`` for consistency.
```

## Footnotes

[1](1,2,3,4,5,6,7,8,9,10,11) Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the [Customizing Parser Behaviour](#) section.