14.2. `ConfigParser` — Configuration file parser¶

Note

The `ConfigParser` module has been renamed to `configparser` in Python 3.0. The *2to3* tool will automatically adapt imports when converting your sources to 3.0.

This module defines the class `ConfigParser`. The `ConfigParser` class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Note

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

The configuration file consists of sections, led by a `[section]` header and followed by `name: value` entries, with continuations in the style of **RFC 822** (see section 3.1.1, “LONG HEADER FIELDS”); `name=value` is also accepted. Note that leading whitespace is removed from values. The optional values can contain format strings which refer to other values in the same section, or values in a special `DEFAULT` section. Additional defaults can be provided on initialization and retrieval. Lines beginning with `'#'` or `';'` are ignored and may be used to provide comments.

For example:

```
[My Section]
foodir: %(dir)s/whatever
dir=frob
long: this value continues
  in the next line
```

would resolve the `%(dir)s` to the value of `dir` (`frob` in this case). All reference expansions are done on demand.

Default values can be specified by passing them into the `ConfigParser` constructor as a dictionary. Additional defaults may be passed into the `get()` method which will override all others.

Sections are normally stored in a built-in dictionary. An alternative dictionary type can be passed to the `ConfigParser` constructor. For example, if a dictionary type is passed that sorts its keys, the sections will be sorted on write-back, as will be the keys within each section.

class `ConfigParser.RawConfigParser`([*defaults*[, *dict_type*]])¶

The basic configuration object. 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. This class does not support the magical interpolation behavior.

New in version 2.3.

Changed in version 2.6: *dict_type* was added.

class `ConfigParser.ConfigParser`([*defaults*[, *dict_type*]])¶

Derived class of `RawConfigParser` that implements the magical interpolation feature and adds optional arguments to the `get()` and `items()` methods. The values in *defaults* must be appropriate for the `%()s` string interpolation. Note that *__name__* is an intrinsic default; its value is the section name, and will override any value provided in *defaults*.

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

class `ConfigParser.SafeConfigParser`([*defaults*[, *dict_type*]])¶

Derived class of `ConfigParser` that implements a more-sane variant of the magical interpolation feature. This implementation is more predictable as well. New applications should prefer this version if they don’t need to be compatible with older versions of Python.

New in version 2.3.

exception `ConfigParser.NoSectionError`¶

Exception raised when a specified section is not found.

exception `ConfigParser.DuplicateSectionError`¶

Exception raised if `add_section()` is called with the name of a section that is already present.

exception `ConfigParser.NoOptionError`¶

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

exception `ConfigParser.InterpolationError`¶

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

exception `ConfigParser.InterpolationDepthError`¶

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

exception `ConfigParser.InterpolationMissingOptionError`¶

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

New in version 2.3.

exception `ConfigParser.InterpolationSyntaxError`¶

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

New in version 2.3.

exception `ConfigParser.MissingSectionHeaderError`¶

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

exception `ConfigParser.ParsingError`¶

Exception raised when errors occur attempting to parse a file.

`ConfigParser.MAX_INTERPOLATION_DEPTH`¶

The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only for the `ConfigParser` class.

See also

Module `shlex`

Support for a creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

14.2.1. RawConfigParser Objects¶

`RawConfigParser` instances have the following methods:

`RawConfigParser.defaults`()¶

Return a dictionary containing the instance-wide defaults.

`RawConfigParser.sections`()¶

Return a list of the sections available; `DEFAULT` is not included in the list.

`RawConfigParser.add_section`(*section*)¶

Add a section named *section* to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised. If the name `DEFAULT` (or any of it's case-insensitive variants) is passed, `ValueError` is raised.

`RawConfigParser.has_section`(*section*)¶

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

`RawConfigParser.options`(*section*)¶

Returns a list of options available in the specified *section*.

`RawConfigParser.has_option`(*section*, *option*)¶

If the given section exists, and contains the given option, return `True`; otherwise return `False`.

New in version 1.6.

`RawConfigParser.read`(*filenames*)¶

Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If *filenames* is a string or Unicode string, 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 a list 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 list will be read. If none of the named files exist, the `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 `readfp()` before calling `read()` for any optional files:

```
import ConfigParser, os

config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
```

```
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')])
```

Changed in version 2.4: Returns list of successfully parsed filenames.

`RawConfigParser.readfp`(*fp*[, *filename*])¶
Read and parse configuration data from the file or file-like object in *fp* (only the `readline()` method is used). If *filename* is omitted and *fp* has a `name` attribute, that is used for *filename*; the default is <???>.

`RawConfigParser.get`(*section*, *option*)¶
Get an *option* value for the named *section*.

`RawConfigParser.getint`(*section*, *option*)¶
A convenience method which coerces the *option* in the specified *section* to an integer.

`RawConfigParser.getfloat`(*section*, *option*)¶
A convenience method which coerces the *option* in the specified *section* to a floating point number.

`RawConfigParser.getboolean`(*section*, *option*)¶
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 `ValueError`.

`RawConfigParser.items`(*section*)¶
Return a list of `(name, value)` pairs for each option in the given *section*.

`RawConfigParser.set`(*section*, *option*, *value*)¶

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. While it is possible to use `RawConfigParser` (or `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.

New in version 1.6.

`RawConfigParser.write`(*fileobject*)¶

Write a representation of the configuration to the specified file object. This representation can be parsed by a future `read()` call.

New in version 1.6.

`RawConfigParser.remove_option`(*section*, *option*)¶

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

New in version 1.6.

`RawConfigParser.remove_section`(*section*)¶
Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

`RawConfigParser.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 necessarily need to subclass a ConfigParser to use this method, you can also re-set it on an instance, to a function that takes a string argument. 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 are stripped before `optionxform()` is called.

14.2.2. ConfigParser Objects¶

The `ConfigParser` class extends some methods of the `RawConfigParser` interface, adding some optional arguments.

`ConfigParser.get`(*section*, *option*[, *raw*[, *vars*]])¶
Get an *option* value for the named *section*. All the `'%'` interpolations are expanded in the return values, based on the defaults passed into the constructor, as well as the options *vars* provided, unless the *raw* argument is true.

`ConfigParser.items`(*section*[, *raw*[, *vars*]])¶

Return a list of `(name, value)` pairs for each option in the given *section*. Optional arguments have the same meaning as for the `get()` method.

New in version 2.3.

14.2.3. SafeConfigParser Objects¶

The SafeConfigParser class implements the same extended interface as ConfigParser, with the following addition:

SafeConfigParser.set(*section*, *option*, *value*)¶

If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. *value* must be a string (str or unicode); if not, TypeError is raised.

New in version 2.4.

14.2.4. Examples¶

An example of writing to a configuration file:

```
import ConfigParser

config = ConfigParser.RawConfigParser()

# When adding sections or items, add them in the reverse order of
# how you want them to be displayed in the actual file.
# In addition, please note that using RawConfigParser's and the raw
# mode of ConfigParser's respective 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. SafeConfigParser does not allow such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'int', '15')
config.set('Section1', 'bool', 'true')
config.set('Section1', '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', 'wb') 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
float = config.getfloat('Section1', 'float')
int = config.getint('Section1', 'int')
print float + 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', 'bool'):
   print config.get('Section1', 'foo')
```

To get interpolation, you will need to use a ConfigParser or SafeConfigParser:

```
import ConfigParser

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

# Set the third, optional argument of get to 1 if you wish to use raw mode.
print config.get('Section1', 'foo', 0) # -> "Python is fun!"
print config.get('Section1', 'foo', 1) # -> "%(bar)s is %(baz)s!"

# The optional fourth argument is a dict with members that will take
# precedence in interpolation.
print config.get('Section1', 'foo', 0, {'bar': 'Documentation',
```

```
                                       'baz': 'evil'})
```

Defaults are available in all three 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.SafeConfigParser({'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!"
```

The function `opt_move` below can be used to move options between sections:

```
def opt_move(config, section1, section2, option):
   try:
       config.set(section2, option, config.get(section1, option, 1))
   except ConfigParser.NoSectionError:
       # Create non-existent section
       config.add_section(section2)
       opt_move(config, section1, section2, option)
   else:
       config.remove_option(section1, option)
```


Previous topic

14.1. `csv` — CSV File Reading and Writing

Next topic

14.3. `robotparser` — Parser for robots.txt

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 14. File Formats »