configobj Documentation

Release 5.1.0

Michael Foord, Nicola Larosa, Rob Dennis, Eli Courtwright

Mar 23, 2018

Contents

This manual consists of two parts, the first shows you how to *read and write config files*, the second covers *using a validation schema*, to verify that a given config file adheres to defined rules.

Contents

CHAPTER 1

ConfigObj 5 Introduction and Reference

Authors Michael Foord, Nicola Larosa, Rob Dennis, Eli Courtwright

Version ConfigObj 5.0.6

Date 2014/08/25

PyPI Entry ConfigObj on PyPI

Homepage Github Page

License BSD License

Support Mailing List

ConfigObj Manual

Note: The best introduction to working with ConfigObj, including the powerful configuration validation system, is the article:

- An Introduction to ConfigObj

1.1 Introduction

ConfigObj is a simple but powerful config file reader and writer: an *ini file round tripper*. Its main feature is that it is very easy to use, with a straightforward programmer's interface and a simple syntax for config files. It has lots of other features though :

- Nested sections (subsections), to any level
- List values
- Multiple line values
- String interpolation (substitution)
- Integrated with a powerful validation system
    - including automatic type checking/conversion

  - repeated sections
  - and allowing default values
- When writing out config files, ConfigObj preserves all comments and the order of members and sections
- Many useful methods and options for working with configuration files (like the 'reload' method)
- Full Unicode support

For support and bug reports please use the ConfigObj Github Page.

1.2 Downloading

The current version is **5.0.6**, dated 25th August 2014. ConfigObj 5 is stable and mature. We still expect to pick up a few bugs along the way though, particularly with respect to Python 3 compatibility[1].

We recommend downloading and installing using pip:

```
pip install configobj
```

1.2.1 Development Version

It's possible to get the latest *development version* of ConfigObj from the Git Repository maintained on the Github Page.

1.3 ConfigObj in the Real World

ConfigObj is widely used. Projects using it include:

- Bazaar.

  Bazaar is a Python distributed {acro;VCS;Version Control System}. ConfigObj is used to read `bazaar.conf` and `branches.conf`.

- Chandler

  A Python and wxPython Personal Information Manager, being developed by the OSAFoundation.

- matplotlib

  A 2D plotting library.

- IPython

  IPython is an enhanced interactive Python shell. IPython uses ConfigObj in a module called 'TConfig' that combines it with enthought Traits: tconfig.

- Elisa - the Fluendo Mediacenter

  Elisa is an open source cross-platform media center solution designed to be simple for people not particularly familiar with computers.

[1] And if you discover any bugs, let us know. We'll fix them quickly.

1.4 Getting Started

The outstanding feature of using ConfigObj is simplicity. Most functions can be performed with single line commands.

1.4.1 Reading a Config File

The normal way to read a config file, is to give ConfigObj the filename :

```
from configobj import ConfigObj
config = ConfigObj(filename)
```

You can also pass the config file in as a list of lines, or a `StringIO` instance, so it doesn't matter where your config data comes from.

You can then access members of your config file as a dictionary. Subsections will also be dictionaries.

```
from configobj import ConfigObj
config = ConfigObj(filename)
#
value1 = config['keyword1']
value2 = config['keyword2']
#
section1 = config['section1']
value3 = section1['keyword3']
value4 = section1['keyword4']
#
# you could also write
value3 = config['section1']['keyword3']
value4 = config['section1']['keyword4']
```

1.4.2 Writing a Config File

Creating a new config file is just as easy as reading one. You can specify a filename when you create the ConfigObj, or do it later[2].

If you *don't* set a filename, then the `write` method will return a list of lines instead of writing to file. See the *write* method for more details.

Here we show creating an empty ConfigObj, setting a filename and some values, and then writing to file :

```
from configobj import ConfigObj
config = ConfigObj()
config.filename = filename
#
config['keyword1'] = value1
config['keyword2'] = value2
#
config['section1'] = {}
config['section1']['keyword3'] = value3
config['section1']['keyword4'] = value4
#
section2 = {
    'keyword5': value5,
    'keyword6': value6,
```

[2] If you specify a filename that doesn't exist, ConfigObj will assume you are creating a new one. See the *create_empty* and *file_error* options.

```
    'sub-section': {
        'keyword7': value7
        }
}
config['section2'] = section2
#
config['section3'] = {}
config['section3']['keyword 8'] = [value8, value9, value10]
config['section3']['keyword 9'] = [value11, value12, value13]
#
config.write()
```

Caution: Keywords and section names can only be strings[3]. Attempting to set anything else will raise a `ValueError`.

See *String Interpolation and List Values* for an important note on using lists in combination with *String Interpolation*.

1.4.3 Config Files

The config files that ConfigObj will read and write are based on the 'INI' format. This means it will read and write files created for `ConfigParser`[4].

Keywords and values are separated by an `'='`, and section markers are between square brackets. Keywords, values, and section names can be surrounded by single or double quotes. Indentation is not significant, but can be preserved.

Subsections are indicated by repeating the square brackets in the section marker. You nest levels by using more brackets.

You can have list values by separating items with a comma, and values spanning multiple lines by using triple quotes (single or double).

For full details on all these see *the config file format*. Here's an example to illustrate:

```
# This is the 'initial_comment'
# Which may be several lines
keyword1 = value1
'keyword 2' = 'value 2'

[ "section 1" ]
# This comment goes with keyword 3
keyword 3 = value 3
'keyword 4' = value4, value 5, 'value 6'

    [[ sub-section ]]    # an inline comment
    # sub-section is inside "section 1"
    'keyword 5' = 'value 7'
    'keyword 6' = '''A multiline value,
that spans more than one line :-)
The line breaks are included in the value.'''

        [[[ sub-sub-section ]]]
        # sub-sub-section is *in* 'sub-section'
```

[3] They can be byte strings (*ordinary* strings) or Unicode.
[4] Except we don't support the RFC822 style line continuations, nor ':' as a divider.

```
        # which is in 'section 1'
        'keyword 7' = 'value 8'

[section 2]    # an inline comment
keyword8 = "value 9"
keyword9 = value10     # an inline comment
# The 'final_comment'
# Which also may be several lines
```

1.5 ConfigObj specifications

```
config = ConfigObj(infile=None, options=None, configspec=None, encoding=None,
                   interpolation=True, raise_errors=False, list_values=True,
                   create_empty=False, file_error=False, stringify=True,
                   indent_type=None, default_encoding=None, unrepr=False,
                   write_empty_values=False, _inspec=False)
```

Many of the keyword arguments are available as attributes after the config file has been parsed.

Note: New in ConfigObj 4.7.0: Instantiating ConfigObj with an `options` dictionary is now deprecated. To modify code that used to do this simply unpack the dictionary in the constructor call:

```
config = ConfigObj(filename, **options)
```

ConfigObj takes the following arguments (with the default values shown) :

- infile: `None`

  You don't need to specify an infile. If you omit it, an empty ConfigObj will be created. `infile` *can* be :

  - Nothing. In which case the `filename` attribute of your ConfigObj will be `None`. You can set a filename at any time.
  - A filename. What happens if the file doesn't already exist is determined by the options `file_error` and `create_empty`. The filename will be preserved as the `filename` attribute. This can be changed at any time.
  - A list of lines. Any trailing newlines will be removed from the lines. The `filename` attribute of your ConfigObj will be `None`.
  - A `StringIO` instance or file object, or any object with a `read` method. The `filename` attribute of your ConfigObj will be `None`[5].
  - A dictionary. You can initialise a ConfigObj from a dictionary[6]. The `filename` attribute of your ConfigObj will be `None`. All keys must be strings. In this case, the order of values and sections is arbitrary.

- 'raise_errors': `False`

[5] This is a change in ConfigObj 4.2.0. Note that ConfigObj doesn't call the seek method of any file like object you pass in. You may want to call `file_object.seek(0)` yourself, first.

[6] A side effect of this is that it enables you to copy a ConfigObj by using `config2 = ConfigObj(config1)`; be aware this only copies members, but not attributes/comments.

Since ConfigObj 4.7.0 the order of members and sections will be preserved when copying a ConfigObj instance.

When parsing, it is possible that the config file will be badly formed. The default is to parse the whole file and raise a single error at the end. You can set `raise_errors = True` to have errors raised immediately. See the *exceptions* section for more details.

Altering this value after initial parsing has no effect.

- 'list_values': `True`

  If `True` (the default) then list values are possible. If `False`, the values are not parsed for lists.

  If `list_values = False` then single line values are not quoted or unquoted when reading and writing.

  Changing this value affects whether single line values will be quoted or not when writing.

- 'create_empty': `False`

  If this value is `True` and the file specified by `infile` doesn't exist, ConfigObj will create an empty file. This can be a useful test that the filename makes sense: an impossible filename will cause an error.

  Altering this value after initial parsing has no effect.

- 'file_error': `False`

  If this value is `True` and the file specified by `infile` doesn't exist, ConfigObj will raise an `IOError`. This error will be raised whenever an attempt to load the `infile` occurs, either in the constructor or using the reload method.

- 'interpolation': `True`

  Whether string interpolation is switched on or not. It is on (`True`) by default.

  You can set this attribute to change whether string interpolation is done when values are fetched. See the *String Interpolation* section for more details.

  New in ConfigObj 4.7.0: Interpolation will also be done in list values.

- 'configspec': `None`

  If you want to use the validation system, you supply a configspec. This is effectively a type of config file that specifies a check for each member. This check can be used to do type conversion as well as check that the value is within your required parameters.

  You provide a configspec in the same way as you do the initial file: a filename, or list of lines, etc. See the *validation* section for full details on how to use the system.

  When parsed, every section has a `configspec` with a dictionary of configspec checks for *that section*.

- 'stringify': `True`

  If you use the validation scheme, it can do type checking *and* conversion for you. This means you may want to set members to integers, or other non-string values.

  If 'stringify' is set to `True` (default) then non-string values will be converted to strings when you write the config file. The *validation* process converts values from strings to the required type.

  If 'stringify' is set to `False`, attempting to set a member to a non-string value[7] will raise a `TypeError` (no type conversion is done by validation).

- 'indent_type': `'    '`

[7] Other than lists of strings.

Indentation is not significant; it can however be present in the input and output config. Any combination of tabs and spaces may be used: the string will be repeated for each level of indentation. Typical values are: `''` (no indentation), `'  '` (indentation with four spaces, the default), `'\t'` (indentation with one tab).

If this option is not specified, and the ConfigObj is initialised with a dictionary, the indentation used in the output is the default one, that is, four spaces.

If this option is not specified, and the ConfigObj is initialised with a list of lines or a file, the indentation used in the first indented line is selected and used in all output lines. If no input line is indented, no output line will be either.

If this option *is* specified, the option value is used in the output config, overriding the type of indentation in the input config (if any).

- 'encoding': `None`

  By default **ConfigObj** does not decode the file/strings you pass it into Unicode[8]. If you want your config file as Unicode (keys and members) you need to provide an encoding to decode the file with. This encoding will also be used to encode the config file when writing.

  You can change the encoding attribute at any time.

  Any characters in your strings that can't be encoded with the specified encoding will raise a `UnicodeEncodeError`.

  Note: `UTF16` encoded files will automatically be detected and decoded, even if `encoding` is `None`.

  This is because it is a 16-bit encoding, and ConfigObj will mangle it (split characters on byte boundaries) if it parses it without decoding.

- 'default_encoding': `None`

  When using the `write` method, **ConfigObj** uses the `encoding` attribute to encode the Unicode strings. If any members (or keys) have been set as byte strings instead of Unicode, these must first be decoded to Unicode before outputting in the specified encoding.

  `default_encoding`, if specified, is the encoding used to decode byte strings in the **ConfigObj** before writing. If this is `None`, then the Python default encoding (`sys.defaultencoding` - usually ASCII) is used.

  For most Western European users, a value of `latin-1` is sensible.

  `default_encoding` is *only* used if an `encoding` is specified.

  Any characters in byte-strings that can't be decoded using the `default_encoding` will raise a `UnicodeDecodeError`.

- 'unrepr': `False`

  The `unrepr` option reads and writes files in a different mode. This allows you to store and retrieve the basic Python data-types using config files.

  This uses Python syntax for lists and quoting. See *unrepr mode* for the full details.

- 'write_empty_values': `False`

  If `write_empty_values` is `True`, empty strings are written as empty values. See *Empty Values* for more details.

[8] The exception is if it detects a `UTF16` encoded file which it must decode before parsing.

- '_inspec': `False`

  Used internally by ConfigObj when parsing configspec files. If you are creating a ConfigObj instance from a configspec file you must pass True for this argument as well as `list_values=False`.

1.5.1 Methods

The ConfigObj is a subclass of an object called `Section`, which is itself a subclass of `dict`, the builtin dictionary type. This means it also has **all** the normal dictionary methods.

In addition, the following *Section Methods* may be useful :

- 'restore_default'
- 'restore_defaults'
- 'walk'
- 'merge'
- 'dict'
- 'as_bool'
- 'as_float'
- 'as_int'
- 'as_list'

Read about *Sections* for details of all the methods.

Hint: The *merge* method of sections is a recursive update.

You can use this to merge sections, or even whole ConfigObjs, into each other.

You would typically use this to create a default ConfigObj and then merge in user settings. This way users only need to specify values that are different from the default. You can use configspecs and validation to achieve the same thing of course.

The public methods available on ConfigObj are :

- 'write'
- 'validate'
- 'reset'
- 'reload'

write

```
write(file_object=None)
```

This method writes the current ConfigObj and takes a single, optional argument[9].

If you pass in a file like object to the `write` method, the config file will be written to this. (The only method of this object that is used is its `write` method, so a `StringIO` instance, or any other file like object will work.)

[9] The method signature shows that this method takes two arguments. The second is the section to be written. This is because the `write` method is called recursively.

Otherwise, the behaviour of this method depends on the `filename` attribute of the ConfigObj.

`filename` ConfigObj will write the configuration to the file specified.

`None` `write` returns a list of lines. (Not `'\n'` terminated)

First the 'initial_comment' is written, then the config file, followed by the 'final_comment'. Comment lines and inline comments are written with each key/value.

validate

```
validate(validator, preserve_errors=False, copy=False)
```

```
# filename is the config file
# filename2 is the configspec
# (which could also be hardcoded into your program)
config = ConfigObj(filename, configspec=filename2)
#
from configobj.validate import Validator
val = Validator()
test = config.validate(val)
if test == True:
    print 'Succeeded.'
```

The validate method uses the :validate: module to do the validation.

This method validates the ConfigObj against the configspec. By doing type conversion as well it can abstract away the config file altogether and present the config *data* to your application (in the types it expects it to be).

If the `configspec` attribute of the ConfigObj is `None`, it raises a `ValueError`.

If the *stringify* attribute is set, this process will convert values to the type defined in the configspec.

The validate method uses checks specified in the configspec and defined in the `Validator` object. It is very easy to extend.

The configspec looks like the config file, but instead of the value, you specify the check (and any default value). See the *validation* section for details.

Hint: The system of configspecs can seem confusing at first, but is actually quite simple and powerful. The best guide to them is this article on ConfigObj:

- An Introduction to ConfigObj

The `copy` parameter fills in missing values from the configspec (default values), *without* marking the values as defaults. It also causes comments to be copied from the configspec into the config file. This allows you to use a configspec to create default config files. (Normally default values aren't written out by the `write` method.)

As of ConfigObj 4.3.0 you can also pass in a ConfigObj instance as your configspec. This is especially useful if you need to specify the encoding of your configspec file. When you read your configspec file, you *must* specify `list_values=False`. If you need to support hashes inside the configspec values then you must also pass in `_inspec=True`. This is because configspec files actually use a different syntax to config files and inline comment support must be switched off to correctly read configspec files with hashes in the values.

```
from configobj import ConfigObj
configspec = ConfigObj(configspecfilename, encoding='UTF8',
                       list_values=False, _inspec=True)
config = ConfigObj(filename, configspec=configspec)
```

Return Value

By default, the validate method either returns `True` (everything passed) or a dictionary of `True` / `False` representing pass/fail. The dictionary follows the structure of the ConfigObj.

If a whole section passes then it is replaced with the value `True`. If a whole section fails, then it is replaced with the value `False`.

If a value is missing, and there is no default in the check, then the check automatically fails.

The `validate` method takes an optional keyword argument `preserve_errors`. If you set this to `True`, instead of getting `False` for failed checks you get the actual error object from the **validate** module. This usually contains useful information about why the check failed.

See the *flatten_errors* function for how to turn your results dictionary into a useful list of error messages.

Even if `preserve_errors` is `True`, missing keys or sections will still be represented by a `False` in the results dictionary.

Mentioning Default Values

In the check in your configspec, you can specify a default to be used - by using the `default` keyword. E.g.

```
key1 = integer(0, 30, default=15)
key2 = integer(default=15)
key3 = boolean(default=True)
key4 = option('Hello', 'Goodbye', 'Not Today', default='Not Today')
```

If the configspec check supplies a default and the value is missing in the config, then the default will be set in your ConfigObj. (It is still passed to the `Validator` so that type conversion can be done: this means the default value must still pass the check.)

ConfigObj keeps a record of which values come from defaults, using the `defaults` attribute of *sections*. Any key in this list isn't written out by the `write` method. If a key is set from outside (even to the same value) then it is removed from the `defaults` list.

There is additionally a special case default value of `None`. If you set the default value to `None` and the value is missing, the value will always be set to `None`. As the other checks don't return `None` (unless you implement your own that do), you can tell that this value came from a default value (and was missing from the config file). It allows an easy way of implementing optional values. Simply check (and ignore) members that are set to `None`.

Note: If *stringify* is `False` then `default=None` returns `''` instead of `None`. This is because setting a value to a non-string raises an error if stringify is unset.

The default value can be a list. See *List Values* for the way to do this.

Writing invalid default values is a *guaranteed* way of confusing your users. Default values **must** pass the check.

Mentioning Repeated Sections and Values

In the configspec it is possible to cause *every* sub-section in a section to be validated using the same configspec. You do this with a section in the configspec called `__many__`. Every sub-section in that section has the `__many__` configspec applied to it (without you having to explicitly name them in advance).

Your `__many__` section can have nested subsections, which can also include `__many__` type sections.

You can also specify that all values should be validated using the same configspec, by having a member with the name `__many__`. If you want to use repeated values along with repeated sections then you can call one of them `____many____` (triple underscores).

Sections with repeated sections or values can also have specifically named sub-sections or values. The `__many__` configspec will only be used to validate entries that don't have an explicit configspec.

See *Repeated Sections* for examples.

Mentioning SimpleVal

If you just want to check if all members are present, then you can use the `SimpleVal` object that comes with ConfigObj. It only fails members if they are missing.

Write a configspec that has all the members you want to check for, but set every section to `''`.

```
val = SimpleVal()
test = config.validate(val)
if test is True:
    print 'Succeeded.'
```

Mentioning copy Mode

As discussed in *Mentioning Default Values*, you can use a configspec to supply default values. These are marked in the ConfigObj instance as defaults, and *not* written out by the `write` mode. This means that your users only need to supply values that are different from the defaults.

This can be inconvenient if you *do* want to write out the default values, for example to write out a default config file.

If you set `copy=True` when you call validate, then no values are marked as defaults. In addition, all comments from the configspec are copied into your ConfigObj instance. You can then call `write` to create your config file.

There is a limitation with this. In order to allow *String Interpolation* to work within configspecs, `DEFAULT` sections are not processed by validation; even in copy mode.

reload

If a ConfigObj instance was loaded from the filesystem, then this method will reload it. It will also reuse any configspec you supplied at instantiation (including reloading it from the filesystem if you passed it in as a filename).

If the ConfigObj does not have a filename attribute pointing to a file, then a `ReloadError` will be raised.

reset

This method takes no arguments and doesn't return anything. It restores a ConfigObj instance to a freshly created state.

1.5.2 Attributes

A ConfigObj has the following attributes :

- indent_type
- interpolation

- stringify
- BOM
- initial_comment
- final_comment
- list_values
- encoding
- default_encoding
- unrepr
- write_empty_values
- newlines

Note: This doesn't include *comments*, *inline_comments*, *defaults*, or *configspec*. These are actually attributes of *Sections*.

It also has the following attributes as a result of parsing. They correspond to options when the ConfigObj was created, but changing them has no effect.

- raise_errors
- create_empty
- file_error

interpolation

ConfigObj can perform string interpolation in a *similar* way to `ConfigParser`. See the *String Interpolation* section for full details.

If `interpolation` is set to `False`, then interpolation is *not* done when you fetch values.

stringify

If this attribute is set (`True`) then the *validate* method changes the values in the ConfigObj. These are turned back into strings when *write* is called.

If stringify is unset (`False`) then attempting to set a value to a non string (or a list of strings) will raise a `TypeError`.

BOM

If the initial config file *started* with the UTF8 Unicode signature (known slightly incorrectly as the BOM - Byte Order Mark), or the UTF16 BOM, then this attribute is set to `True`. Otherwise it is `False`.

If it is set to `True` when `write` is called then, if `encoding` is set to `None` *or* to `utf_8` (and variants) a UTF BOM will be written.

For UTF16 encodings, a BOM is *always* written.

initial_comment

This is a list of lines. If the ConfigObj is created from an existing file, it will contain any lines of comments before the start of the members.

If you create a new ConfigObj, this will be an empty list.

The write method puts these lines before it starts writing out the members.

final_comment

This is a list of lines. If the ConfigObj is created from an existing file, it will contain any lines of comments after the last member.

If you create a new ConfigObj, this will be an empty list.

The `write` method puts these lines after it finishes writing out the members.

list_values

This attribute is `True` or `False`. If set to `False` then values are not parsed for list values. In addition single line values are not unquoted.

This allows you to do your own parsing of values. It exists primarily to support the reading of the *configspec* - but has other use cases.

For example you could use the `LineParser` from the listquote module to read values for nested lists.

Single line values aren't quoted when writing - but multiline values are handled as normal.

> **Caution:** Because values aren't quoted, leading or trailing whitespace can be lost. This behaviour was changed in version 4.0.1. Prior to this, single line values might have been quoted; even with `list_values=False`. This means that files written by earlier versions of ConfigObj *could* now be incompatible and need the quotes removing by hand.

encoding

This is the encoding used to encode the output, when you call `write`. It must be a valid encoding recognised by Python.

If this value is `None` then no encoding is done when `write` is called.

default_encoding

If encoding is set, any byte-strings in your ConfigObj instance (keys or members) will first be decoded to Unicode using the encoding specified by the `default_encoding` attribute. This ensures that the output is in the encoding specified.

If this value is `None` then `sys.defaultencoding` is used instead.

unrepr

Another boolean value. If this is set, then `repr(value)` is used to write values. This writes values in a slightly different way to the normal ConfigObj file syntax.

This preserves basic Python data-types when read back in. See *unrepr mode* for more details.

write_empty_values

Also boolean. If set, values that are an empty string (`''`) are written as empty values. See *Empty Values* for more details.

newlines

When a config file is read, ConfigObj records the type of newline separators in the file and uses this separator when writing. It defaults to `None`, and ConfigObj uses the system default (`os.linesep`) if write is called without newlines having been set.

1.6 The Config File Format

You saw an example config file in the *Config Files* section. Here is a fuller specification of the config files used and created by ConfigObj.

The basic pattern for keywords is:

```
# comment line
# comment line
keyword = value # inline comment
```

Both keyword and value can optionally be surrounded in quotes. The equals sign is the only valid divider.

Values can have comments on the lines above them, and an inline comment after them. This, of course, is optional. See the *comments* section for details.

If a keyword or value starts or ends with whitespace, or contains a quote mark or comma, then it should be surrounded by quotes. Quotes are not necessary if whitespace is surrounded by non-whitespace.

Values can also be lists. Lists are comma separated. You indicate a single member list by a trailing comma, unless you have a config spec that uses `force_list`, which implies an automatic conversion of scalar values to a single-element list. An empty list is shown by a single comma:

```
keyword1 = value1, value2, value3
keyword2 = value1, # a single member list
keyword3 = , # an empty list
```

Values that contain line breaks (multi-line values) can be surrounded by triple quotes. These can also be used if a value contains both types of quotes. List members cannot be surrounded by triple quotes:

```
keyword1 = ''' A multi line value
on several
lines'''     # with a comment
keyword2 = '''I won't be "afraid".'''
#
keyword3 = """ A multi line value
on several
```

```
lines"""      # with a comment
keyword4 = """I won't be "afraid"."""
```

Warning: There is no way of safely quoting values that contain both types of triple quotes.

A line that starts with a '#', possibly preceded by whitespace, is a comment.

New sections are indicated by a section marker line. That is the section name in square brackets. Whitespace around the section name is ignored. The name can be quoted with single or double quotes. The marker can have comments before it and an inline comment after it:

```
# The First Section
[ section name 1 ] # first section
keyword1 = value1

# The Second Section
[ "section name 2" ] # second section
keyword2 = value2
```

Any subsections (sections that are *inside* the current section) are designated by repeating the square brackets before and after the section name. The number of square brackets represents the nesting level of the sub-section. Square brackets may be separated by whitespace; such whitespace, however, will not be present in the output config written by the `write` method.

Indentation is not significant, but can be preserved. See the description of the `indent_type` option, in the *ConfigObj specifications* chapter, for the details.

A *NestingError* will be raised if the number of the opening and the closing brackets in a section marker is not the same, or if a sub-section's nesting level is greater than the nesting level of it parent plus one.

In the outer section, single values can only appear before any sub-section. Otherwise they will belong to the sub-section immediately before them:

```
# initial comment
keyword1 = value1
keyword2 = value2

[section 1]
keyword1 = value1
keyword2 = value2

    [[sub-section]]
    # this is in section 1
    keyword1 = value1
    keyword2 = value2

        [[[nested section]]]
        # this is in sub section
        keyword1 = value1
        keyword2 = value2

    [[sub-section2]]
    # this is in section 1 again
    keyword1 = value1
    keyword2 = value2
```

```
[[sub-section3]]
# this is also in section 1, indentation is misleading here
keyword1 = value1
keyword2 = value2

# final comment
```

When parsed, the above config file produces the following data structure:

```
ConfigObj({
    'keyword1': 'value1',
    'keyword2': 'value2',
    'section 1': {
        'keyword1': 'value1',
        'keyword2': 'value2',
        'sub-section': {
            'keyword1': 'value1',
            'keyword2': 'value2',
            'nested section': {
                'keyword1': 'value1',
                'keyword2': 'value2',
            },
        },
        'sub-section2': {
            'keyword1': 'value1',
            'keyword2': 'value2',
        },
        'sub-section3': {
            'keyword1': 'value1',
            'keyword2': 'value2',
        },
    },
})
```

Sections are ordered: note how the structure of the resulting ConfigObj is in the same order as the original file.

Note: In ConfigObj 4.3.0 *empty values* became valid syntax. They are read as the empty string. There is also an option/attribute (`write_empty_values`) to allow the writing of these.

This is mainly to support 'legacy' config files, written from other applications. This is documented under *Empty Values*.

unrepr mode introduces *another* syntax variation, used for storing basic Python datatypes in config files.

1.7 Sections

Every section in a ConfigObj has certain properties. The ConfigObj itself also has these properties, because it too is a section (sometimes called the *root section*).

`Section` is a subclass of the standard new-class dictionary, therefore it has **all** the methods of a normal dictionary. This means you can `update` and `clear` sections.

Note: You create a new section by assigning a member to be a dictionary.

The new `Section` is created *from* the dictionary, but isn't the same thing as the dictionary. (So references to the dictionary you use to create the section *aren't* references to the new section).

Note the following.

```
config = ConfigObj()
vals = {'key1': 'value 1',
        'key2': 'value 2'
       }
config['vals'] = vals
config['vals'] == vals
True
config['vals'] is vals
False
```

If you now change `vals`, the changes won't be reflected in `config['vals']`.

A section is ordered, following its `scalars` and `sections` attributes documented below. This means that the following dictionary attributes return their results in order.

- '__iter__'

  More commonly known as `for member in section:`.

- '__repr__' and '__str__'

  Any time you print or display the ConfigObj.

- 'items'
- 'iteritems'
- 'iterkeys'
- 'itervalues'
- 'keys'
- 'popitem'
- 'values'

1.7.1 Section Attributes

- main

  A reference to the main ConfigObj.

- parent

  A reference to the 'parent' section, the section that this section is a member of.

  On the ConfigObj this attribute is a reference to itself. You can use this to walk up the sections, stopping when `section.parent is section`.

- depth

  The nesting level of the current section.

  If you create a new ConfigObj and add sections, 1 will be added to the depth level between sections.

- defaults

This attribute is a list of scalars that came from default values. Values that came from defaults aren't written out by the `write` method. Setting any of these values in the section removes them from the defaults list.

- default_values

  This attribute is a dictionary mapping keys to the default values for the keys. By default it is an empty dictionary and is populated when you validate the ConfigObj.

- scalars, sections

  These attributes are normal lists, representing the order that members, single values and subsections appear in the section. The order will either be the order of the original config file, *or* the order that you added members.

  The order of members in this lists is the order that `write` creates in the config file. The `scalars` list is output before the `sections` list.

  Adding or removing members also alters these lists. You can manipulate the lists directly to alter the order of members.

  Warning: If you alter the `scalars`, `sections`, or `defaults` attributes so that they no longer reflect the contents of the section, you will break your ConfigObj.

  See also the `rename` method.

- comments

  This is a dictionary of comments associated with each member. Each entry is a list of lines. These lines are written out before the member.

- inline_comments

  This is *another* dictionary of comments associated with each member. Each entry is a string that is put inline with the member.

- configspec

  The configspec attribute is a dictionary mapping scalars to *checks*. A check defines the expected type and possibly the allowed values for a member.

  The configspec has the same format as a config file, but instead of values it has a specification for the value (which may include a default value). The *validate* method uses it to check the config file makes sense. If a configspec is passed in when the ConfigObj is created, then it is parsed and broken up to become the `configspec` attribute of each section.

  If you didn't pass in a configspec, this attribute will be `None` on the root section (the main ConfigObj).

  You can set the configspec attribute directly on a section.

  See the *validation* section for full details of how to write configspecs.

- extra_values

  By default an empty list. After *validation* this is populated with any members of the section that don't appear in the configspec (i.e. they are additional values). Rather than accessing this directly it may be more convenient to get all the extra values in a config file using the *get_extra_values* function.

  New in ConfigObj 4.7.0.

1.7.2 Section Methods

- **dict**

  This method takes no arguments. It returns a deep copy of the section as a dictionary. All subsections will also be dictionaries, and list values will be copies, rather than references to the original[10].

- **rename**

  `rename(oldkey, newkey)`

  This method renames a key, without affecting its position in the sequence.

- **merge**

  `merge(indict)`

  This method is a *recursive update* method. It allows you to merge two config files together.

  You would typically use this to create a default ConfigObj and then merge in user settings. This way users only need to specify values that are different from the default.

  For example :

```
# def_cfg contains your default config settings
# user_cfg contains the user settings
cfg = ConfigObj(def_cfg)
usr = ConfigObj(user_cfg)
#
cfg.merge(usr)

"""
cfg now contains a combination of the default settings and the user
settings.

The user settings will have overwritten any of the default ones.
"""
```

- **walk**

  This method can be used to transform values and names. See *walking a section* for examples and explanation.

- **as_bool**

  `as_bool(key)`

  Returns `True` if the key contains a string that represents `True`, or is the `True` object.

  Returns `False` if the key contains a string that represents `False`, or is the `False` object.

  Raises a `ValueError` if the key contains anything else.

  Strings that represent `True` are (not case sensitive):

```
true, yes, on, 1
```

  Strings that represent `False` are:

```
false, no, off, 0
```

[10] The dict method doesn't actually use the deepcopy mechanism. This means if you add nested lists (etc) to your ConfigObj, then the dictionary returned by dict may contain some references. For all *normal* ConfigObjs it will return a deepcopy.

- **as_int**

  `as_int(key)`

  This returns the value contained in the specified key as an integer.

  It raises a `ValueError` if the conversion can't be done.

- **as_float**

  `as_float(key)`

  This returns the value contained in the specified key as a float.

  It raises a `ValueError` if the conversion can't be done.

- **as_list**

  `as_list(key)`

  This returns the value contained in the specified key as a list.

  If it isn't a list it will be wrapped as a list so that you can guarantee the returned value will be a list.

- **restore_default**

  `restore_default(key)`

  Restore (and return) the default value for the specified key.

  This method will only work for a ConfigObj that was created with a configspec and has been validated.

  If there is no default value for this key, `KeyError` is raised.

- **restore_defaults**

  `restore_defaults()`

  Recursively restore default values to all members that have them.

  This method will only work for a ConfigObj that was created with a configspec and has been validated.

  It doesn't delete or modify entries without default values.

1.7.3 Walking a Section

Note: The walk method allows you to call a function on every member/name.

```
walk(function, raise_errors=True,
     call_on_sections=False, **keywargs)
```

`walk` is a method of the `Section` object. This means it is also a method of ConfigObj.

It walks through every member and calls a function on the keyword and value. It walks recursively through subsections.

It returns a dictionary of all the computed values.

If the function raises an exception, the default is to propagate the error, and stop. If `raise_errors=False` then it sets the return value for that keyword to `False` instead, and continues. This is similar to the way *validation* works.

Your function receives the arguments `(section, key)`. The current value is then `section[key]`[11]. Any unrecognised keyword arguments you pass to walk, are passed on to the function.

Normally `walk` just recurses into subsections. If you are transforming (or checking) names as well as values, then you want to be able to change the names of sections. In this case set `call_on_sections` to `True`. Now, on encountering a sub-section, *first* the function is called for the *whole* sub-section, and *then* it recurses into it's members. This means your function must be able to handle receiving dictionaries as well as strings and lists.

If you are using the return value from `walk` *and* `call_on_sections`, note that walk discards the return value when it calls your function.

Caution: You can use `walk` to transform the names of members of a section but you mustn't add or delete members.

1.7.4 Examples

You can use this for transforming all values in your ConfigObj. For example you might like the nested lists from ConfigObj 3. This was provided by the listquote module. You could switch off the parsing for list values (`list_values=False`) and use listquote to parse every value.

Another thing you might want to do is use the Python escape codes in your values. You might be *used* to using `\n` for line feed and `\t` for tab. Obviously we'd need to decode strings that come from the config file (using the escape codes). Before writing out we'll need to put the escape codes back in encode.

As an example we'll write a function to use with walk, that encodes or decodes values using the `string-escape` codec.

The function has to take each value and set the new value. As a bonus we'll create one function that will do decode *or* encode depending on a keyword argument.

We don't want to work with section names, we're only transforming values, so we can leave `call_on_sections` as `False`. This means the two datatypes we have to handle are strings and lists, we can ignore everything else. (We'll treat tuples as lists as well).

We're not using the return values, so it doesn't need to return anything, just change the values if appropriate.

```
def string_escape(section, key, encode=False):
    """
    A function to encode or decode using the 'string-escape' codec.
    To be passed to the walk method of a ConfigObj.
    By default it decodes.
    To encode, pass in the keyword argument ``encode=True``.
    """
    val = section[key]
    # is it a type we can work with
    # NOTE: for platforms where Python > 2.2
    # you can use basestring instead of (str, unicode)
    if not isinstance(val, (str, unicode, list, tuple)):
        # no !
        return
    elif isinstance(val, (str, unicode)):
        # it's a string !
        if not encode:
            section[key] = val.decode('string-escape')
```

[11] Passing `(section, key)` rather than `(value, key)` allows you to change the value by setting `section[key] = newval`. It also gives you access to the *rename* method of the section.

```
        else:
            section[key] = val.encode('string-escape')
    else:
        # it must be a list or tuple!
        # we'll be lazy and create a new list
        newval = []
        # we'll check every member of the list
        for entry in val:
            if isinstance(entry, (str, unicode)):
                if not encode:
                    newval.append(entry.decode('string-escape'))
                else:
                   newval.append(entry.encode('string-escape'))
            else:
                newval.append(entry)
        # done !
        section[key] =  newval

# assume we have a ConfigObj called ``config``
#
# To decode
config.walk(string_escape)
#
# To encode.
# Because ``walk`` doesn't recognise the ``encode`` argument
# it passes it to our function.
config.walk(string_escape, encode=True)
```

Here's a simple example of using `walk` to transform names and values. One usecase of this would be to create a *standard* config file with placeholders for section and keynames. You can then use walk to create new config files and change values and member names :

```
# We use 'XXXX' as a placeholder
config = '''
XXXXkey1 = XXXXvalue1
XXXXkey2 = XXXXvalue2
XXXXkey3 = XXXXvalue3
[XXXXsection1]
XXXXkey1 = XXXXvalue1
XXXXkey2 = XXXXvalue2
XXXXkey3 = XXXXvalue3
[XXXXsection2]
XXXXkey1 = XXXXvalue1
XXXXkey2 = XXXXvalue2
XXXXkey3 = XXXXvalue3
    [[XXXXsection1]]
    XXXXkey1 = XXXXvalue1
    XXXXkey2 = XXXXvalue2
    XXXXkey3 = XXXXvalue3
'''.splitlines()
cfg = ConfigObj(config)
#
def transform(section, key):
    val = section[key]
    newkey = key.replace('XXXX', 'CLIENT1')
    section.rename(key, newkey)
    if isinstance(val, (tuple, list, dict)):
        pass
```

```
    else:
        val = val.replace('XXXX', 'CLIENT1')
        section[newkey] = val
#
cfg.walk(transform, call_on_sections=True)
print cfg
ConfigObj({'CLIENT1key1': 'CLIENT1value1', 'CLIENT1key2': 'CLIENT1value2',
'CLIENT1key3': 'CLIENT1value3',
'CLIENT1section1': {'CLIENT1key1': 'CLIENT1value1',
    'CLIENT1key2': 'CLIENT1value2', 'CLIENT1key3': 'CLIENT1value3'},
'CLIENT1section2': {'CLIENT1key1': 'CLIENT1value1',
    'CLIENT1key2': 'CLIENT1value2', 'CLIENT1key3': 'CLIENT1value3',
    'CLIENT1section1': {'CLIENT1key1': 'CLIENT1value1',
        'CLIENT1key2': 'CLIENT1value2', 'CLIENT1key3': 'CLIENT1value3'}}})
```

1.8 Exceptions

There are several places where ConfigObj may raise exceptions (other than because of bugs).

1. **If a configspec filename you pass in doesn't exist, or a config file** filename doesn't exist *and* `file_error=True`, an `IOError` will be raised.
2. **If you try to set a non-string key, or a non string value when** `stringify=False`, a `TypeError` will be raised.
3. A badly built config file will cause parsing errors.
4. A parsing error can also occur when reading a configspec.
5. **In string interpolation you can specify a value that doesn't exist, or** create circular references (recursion).

Number 5 (which is actually two different types of exceptions) is documented in *String Interpolation*.

This section is about errors raised during parsing.

The base error class is `ConfigObjError`. This is a subclass of `SyntaxError`, so you can trap for `SyntaxError` without needing to directly import any of the ConfigObj exceptions.

The following other exceptions are defined (all deriving from `ConfigObjError`) :

- `NestingError`

  This error indicates either a mismatch in the brackets in a section marker, or an excessive level of nesting.

- `ParseError`

  This error indicates that a line is badly written. It is neither a valid `key = value` line, nor a valid section marker line, nor a comment line.

- `DuplicateError`

  The keyword or section specified already exists.

- `ConfigspecError`

  An error occurred whilst parsing a configspec.

- `UnreprError`

  An error occurred when parsing a value in *unrepr mode*.

- `ReloadError`

  `reload` was called on a ConfigObj instance that doesn't have a valid filename attribute.

When parsing a configspec, ConfigObj will stop on the first error it encounters. It will raise a `ConfigspecError`. This will have an `error` attribute, which is the actual error that was raised.

Behaviour when parsing a config file depends on the option `raise_errors`. If ConfigObj encounters an error while parsing a config file:

If `raise_errors=True` then ConfigObj will raise the appropriate error and parsing will stop.

If `raise_errors=False` (the default) then parsing will continue to the end and *all* errors will be collected.

If `raise_errors` is False and multiple errors are found a `ConfigObjError` is raised. The error raised has a `config` attribute, which is the parts of the ConfigObj that parsed successfully. It also has an attribute `errors`, which is a list of *all* the errors raised. Each entry in the list is an instance of the appropriate error type. Each one has the following attributes (useful for delivering a sensible error message to your user) :

- `line`: the original line that caused the error.
- `line_number`: its number in the config file.
- `message`: the error message that accompanied the error.

If only one error is found, then that error is re-raised. The error still has the `config` and `errors` attributes. This means that your error handling code can be the same whether one error is raised in parsing , or several.

It also means that in the most common case (a single error) a useful error message will be raised.

Unless you want to format the error message differently from the default, you should use `str(ex)` or better yet, use the exception in a format where the conversion is implicit. This uses the exception's `__str__()` method which in all likelyhood will output all the information you want to know.

Note: One wrongly written line could break the basic structure of your config file. This could cause every line after it to flag an error, so having a list of all the lines that caused errors may not be as useful as it sounds.

1.9 Validation

Hint: The system of configspecs can seem confusing at first, but is actually quite simple and powerful. The best reference is my article on ConfigObj:

- An Introduction to ConfigObj

Validation is done through a combination of the *configspec* and a `Validator` object. For this you need *validate.py*[12]. See *downloading* if you don't have a copy.

Validation can perform two different operations :

1. **Check that a value meets a specification. For example, check that a value** is an integer between one and six, or is a choice from a specific set of options.
2. **It can convert the value into the type required. For example, if one of** your values is a port number, validation will turn it into an integer for you.

[12] Minimum required version of *validate.py* 0.2.0 .

So validation can act as a transparent layer between the datatypes of your application configuration (boolean, integers, floats, etc) and the text format of your config file.

1.9.1 configspec

The `validate` method checks members against an entry in the configspec. Your configspec therefore resembles your config file, with a check for every member.

In order to perform validation you need a `Validator` object. This has several useful built-in check functions. You can also create your own custom functions and register them with your Validator object.

Each check is the name of one of these functions, including any parameters and keyword arguments. The configspecs look like function calls, and they map to function calls.

The basic datatypes that an un-extended Validator can test for are :

- boolean values (True and False)
- integers (including minimum and maximum values)
- floats (including min and max)
- strings (including min and max length)
- IP addresses (v4 only)

It can also handle lists of these types and restrict a value to being one from a set of options.

An example configspec is going to look something like:

```
port = integer(0, 100)
user = string(max=25)
mode = option('quiet', 'loud', 'silent')
```

You can specify default values, and also have the same configspec applied to several sections. This is called *repeated sections*.

For full details on writing configspecs, please refer to the *validate.py documentation*.

Important: Your configspec is read by ConfigObj in the same way as a config file.

That means you can do interpolation *within* your configspec.

In order to allow this, checks in the 'DEFAULT' section (of the root level of your configspec) are *not* used.

If you want to use a configspec *without* interpolation being done in it you can create your configspec manually and switch off interpolation:

```
from configobj import ConfigObj

configspec = ConfigObj(spec_filename, interpolation=False, list_values=False,
                       _inspec=True)
conf = ConfigObj(config_filename, configspec=configspec)
```

If you need to specify the encoding of your configspec, then you can pass in a ConfigObj instance as your configspec. When you read your configspec file, you *must* specify `list_values=False`. If you need to support hashes in configspec values then you must also pass in `_inspec=True`.

```
from configobj import ConfigObj
configspec = ConfigObj(configspecfilename, encoding='UTF8',
                       list_values=False, _inspec=True)
config = ConfigObj(filename, configspec=configspec)
```

1.9.2 Type Conversion

By default, validation does type conversion. This means that if you specify `integer` as the check, then calling *validate* will actually change the value to an integer (so long as the check succeeds).

It also means that when you call the *write* method, the value will be converted back into a string using the `str` function.

To switch this off, and leave values as strings after validation, you need to set the *stringify* attribute to `False`. If this is the case, attempting to set a value to a non-string will raise an error.

1.9.3 Default Values

You can set a default value in your check. If the value is missing from the config file then this value will be used instead. This means that your user only has to supply values that differ from the defaults.

If you *don't* supply a default then for a value to be missing is an error, and this will show in the *return value* from validate.

Additionally you can set the default to be `None`. This means the value will be set to `None` (the object) *whichever check is used*. (It will be set to `''` rather than `None` if *stringify* is `False`). You can use this to easily implement optional values in your config files.

```
port = integer(0, 100, default=80)
user = string(max=25, default=0)
mode = option('quiet', 'loud', 'silent', default='loud')
nick = string(default=None)
```

Note: Because the default goes through type conversion, it also has to pass the check.

Note that `default=None` is case sensitive.

List Values

It's possible that you will want to specify a list as a default value. To avoid confusing syntax with commas and quotes you use a list constructor to specify that keyword arguments are lists. This includes the `default` value. This makes checks look something like:

```
checkname(default=list('val1', 'val2', 'val3'))
```

This works with all keyword arguments, but is most useful for default values.

1.9.4 Repeated Sections

Repeated sections are a way of specifying a configspec for a section that should be applied to all unspecified subsections in the same section.

The easiest way of explaining this is to give an example. Suppose you have a config file that describes a dog. That dog has various attributes, but it can also have many fleas. You don't know in advance how many fleas there will be, or what they will be called, but you want each flea validated against the same configspec.

We can define a section called *fleas*. We want every flea in that section (every sub-section) to have the same configspec applied to it. We do this by defining a single section called `__many__`.

```
[dog]
name = string(default=Rover)
age = float(0, 99, default=0)

    [[fleas]]

        [[[__many__]]]
        bloodsucker = boolean(default=True)
        children = integer(default=10000)
        size = option(small, tiny, micro, default=tiny)
```

Every flea on our dog will now be validated using the `__many__` configspec.

`__many__` sections can have sub-sections, including their own `__many__` sub-sections. Defaults work in the normal way in repeated sections.

1.9.5 Repeated Values

As well as using `__many__` to validate unspecified sections you can use it to validate values. For example, to specify that all values in a section should be integers:

```
[section]
__many__ = integer
```

If you want to use repeated values alongside repeated sections you can call one `__many__` and the other `____many____` (with three underscores).

1.9.6 Copy Mode

Because you can specify default values in your configspec, you can use ConfigObj to write out default config files for your application.

However, normally values supplied from a default in a configspec are *not* written out by the `write` method.

To do this, you need to specify `copy=True` when you call validate. As well as not marking values as default, all the comments in the configspec file will be copied into your ConfigObj instance.

```
from configobj import ConfigObj
from configobj.validate import Validator
vdt = Validator()
config = ConfigObj(configspec='default.ini')
config.filename = 'new_default.ini'
config.validate(vdt, copy=True)
config.write()
```

If you need to support hashes in the configspec values then you must create it with `_inspec=True`. This has the side effect of switching off the parsing of inline comments, meaning that they won't be copied into the new config file. (ConfigObj syntax is slightly different from configspec syntax and the parser can't support both inline comments and hashes in configspec values.)

1.9.7 Validation and Interpolation

String interpolation and validation don't play well together. When validation changes type it sets the value. If the value uses interpolation, then the interpolation reference would normally be overwritten. Calling `write` would then use the absolute value and the interpolation reference would be lost.

As a compromise - if the value is unchanged by validation then it is not reset. This means strings that pass through validation unmodified will not be overwritten. If validation changes type - the value has to be overwritten, and any interpolation references are lost.

1.9.8 Extra Values

After validation the `extra_values` member of every section that is listed in the configspec will be populated with the names of members that are in the config file but not in the configspec.

If you are reporting configuration errors to your user this information can be useful, for example some missing entries may be due to misspelt entries that appear as extra values.

See the *get_extra_values* function

New in ConfigObj 4.7.0.

1.9.9 SimpleVal

You may not need a full validation process, but still want to check if all the expected values are present.

Provided as part of the ConfigObj module is the `SimpleVal` object. This has a dummy `test` method that always passes.

The only reason a test will fail is if the value is missing. The return value from `validate` will either be `True`, meaning all present, or a dictionary with `False` for all missing values/sections.

To use it, you still need to pass in a valid configspec when you create the ConfigObj, but just set all the values to `''`. Then create an instance of `SimpleVal` and pass it to the `validate` method.

As a trivial example if you had the following config file:

```
# config file for an application
port = 80
protocol = http
domain = voidspace
top_level_domain = org.uk
```

You would write the following configspec:

```
port = ''
protocol = ''
domain = ''
top_level_domain = ''
```

```
config = Configobj(filename, configspec=configspec)
val = SimpleVal()
test = config.validate(val)
if test == True:
    print 'All values present.'
elif test == False:
    print 'No values present!'
```

```
else:
    for entry in test:
        if test[entry] == False:
            print '"%s" missing.' % entry
```

1.10 Empty values

Many config files from other applications allow empty values. As of version 4.3.0, ConfigObj will read these as an empty string.

A new option/attribute has been added (`write_empty_values`) to allow ConfigObj to write empty strings as empty values.

```
from configobj import ConfigObj
cfg = '''
    key =
    key2 = # a comment
'''.splitlines()
config = ConfigObj(cfg)
print config
ConfigObj({'key': '', 'key2': ''})

config.write_empty_values = True
for line in config.write():
    print line

key =
key2 =      # a comment
```

1.11 unrepr mode

The `unrepr` option allows you to store and retrieve the basic Python data-types using config files. It has to use a slightly different syntax to normal ConfigObj files. Unsurprisingly it uses Python syntax.

This means that lists are different (they are surrounded by square brackets), and strings *must* be quoted.

The types that `unrepr` can work with are :

strings, lists tuples
None, True, False
dictionaries, integers, floats
longs and complex numbers

You can't store classes, types or instances.

`unrepr` uses `repr(object)` to write out values, so it currently *doesn't* check that you are writing valid objects. If you attempt to read an unsupported value, ConfigObj will raise a `configobj.UnknownType` exception.

Values that are triple quoted cased. The triple quotes are removed *before* converting. This means that you can use triple quotes to write dictionaries over several lines in your config files. They won't be written like this though.

If you are writing config files by hand, for use with `unrepr`, you should be aware of the following differences from normal ConfigObj syntax :

List : `['A List', 'With', 'Strings']`

Strings : `"Must be quoted."`
Backslash : `"The backslash must be escaped \\"`

These all follow normal Python syntax.

In unrepr mode *inline comments* are not saved. This is because lines are parsed using the compiler package which discards comments.

1.12 String Interpolation

Note: String interpolation can slow down (slightly) the fetching of values from your config object. If you aren't using interpolation and it is performance critical then create your instance with `interpolation=False`.

ConfigObj allows string interpolation *similar* to the way `ConfigParser` or `string.Template` work. The value of the `interpolation` attribute determines which style of interpolation you want to use. Valid values are "ConfigParser" or "Template" (case-insensitive, so "configparser" and "template" will also work). For backwards compatibility reasons, the value `True` is also a valid value for the `interpolation` attribute, and will select `ConfigParser`-style interpolation. At some undetermined point in the future, that default *may* change to `Template`-style interpolation.

For `ConfigParser`-style interpolation, you specify a value to be substituted by including `%(name)s` in the value.

For `Template`-style interpolation, you specify a value to be substituted by including `${cl}name{cr}` in the value. Alternately, if 'name' is a valid Python identifier (i.e., is composed of nothing but alphanumeric characters, plus the underscore character), then the braces are optional and the value can be written as `$name`.

Note that `ConfigParser`-style interpolation and `Template`-style interpolation are mutually exclusive; you cannot have a configuration file that's a mix of one or the other. Pick one and stick to it. `Template`-style interpolation is simpler to read and write by hand, and is recommended if you don't have a particular reason to use `ConfigParser`-style.

Interpolation checks first the current section to see if `name` is the key to a value. ('name' is case sensitive).

If it doesn't find it, next it checks the 'DEFAULT' sub-section of the current section.

If it still doesn't find it, it moves on to check the parent section and the parent section's 'DEFAULT' subsection, and so on all the way up to the main section.

If the value specified isn't found in any of these locations, then a `MissingInterpolationOption` error is raised (a subclass of `ConfigObjError`).

If it is found then the returned value is also checked for substitutions. This allows you to make up compound values (for example directory paths) that use more than one default value. It also means it's possible to create circular references. If there are any circular references which would cause an infinite interpolation loop, an `InterpolationLoopError` is raised.

Both of these errors are subclasses of `InterpolationError`, which is a subclass of `ConfigObjError`.

String interpolation and validation don't play well together. This is because validation overwrites values - and so may erase the interpolation references. See *Validation and Interpolation*. (This can only happen if validation has to *change* the value).

New in ConfigObj 4.7.0: String interpolation is now done in members of list values.

1.12.1 String Interpolation and List Values

Since version 4.7 string interpolation is done on string members of list values. If interpolation changes any members of the list then what you get back is a *copy* of the list rather than the original list.

This makes fetching list values slightly slower when interpolation is on, it also means that if you mutate the list changes won't be reflected in the original list:

```
>>> c = ConfigObj()
>>> c['foo'] = 'boo'
>>> c['bar'] = ['%(foo)s']
>>> c['bar']
['boo']
>>> c['bar'].append('fish')
>>> c['bar']
['boo']
```

Instead of mutating the list you must create a new list and reassign it.

1.13 Comments

Any line that starts with a '#', possibly preceded by whitespace, is a comment.

If a config file starts with comments then these are preserved as the *initial_comment*.

If a config file ends with comments then these are preserved as the *final_comment*.

Every key or section marker may have lines of comments immediately above it. These are saved as the `comments` attribute of the section. Each member is a list of lines.

You can customize the line comment markers by changing the `COMMENT_MARKERS` class variable of `ConfigObj`, one way to do that is to inherit from it:

```
class ConfigObjPHP(ConfigObj):
    """Handle classic INI style comments to read 'php.ini'."""
    COMMENT_MARKERS = ['#', ';']
```

You can also have a comment inline with a value. These are saved as the `inline_comments` attribute of the section, with one entry per member of the section.

Subsections (section markers in the config file) can also have comments.

See *Section Attributes* for more on these attributes.

These comments are all written back out by the `write` method.

1.14 flatten_errors

```
flatten_errors(cfg, res)
```

Validation is a powerful way of checking that the values supplied by the user make sense.

The *validate* method returns a results dictionary that represents pass or fail for each value. This doesn't give you any information about *why* the check failed.

`flatten_errors` is an example function that turns a results dictionary into a flat list, that only contains values that *failed*.

`cfg` is the ConfigObj instance being checked, `res` is the results dictionary returned by `validate`.

It returns a list of keys that failed. Each member of the list is a tuple:

```
([list of sections...], key, result)
```

If `validate` was called with `preserve_errors=False` (the default) then `result` will always be `False`.

list of sections is a flattened list of sections that the key was found in.

If the section was missing then key will be `None`.

If the value (or section) was missing then `result` will be `False`.

If `validate` was called with `preserve_errors=True` and a value was present, but failed the check, then `result` will be the exception object returned. You can use this as a string that describes the failure.

For example :

> *The value "3" is of the wrong type.*

1.14.1 Example Usage

The output from `flatten_errors` is a list of tuples.

Here is an example of how you could present this information to the user.

```
vtor = validate.Validator()
# ini is your config file - cs is the configspec
cfg = ConfigObj(ini, configspec=cs)
res = cfg.validate(vtor, preserve_errors=True)
for entry in flatten_errors(cfg, res):
    # each entry is a tuple
    section_list, key, error = entry
    if key is not None:
       section_list.append(key)
    else:
        section_list.append('[missing section]')
    section_string = ', '.join(section_list)
    if error == False:
        error = 'Missing value or section.'
    print section_string, ' = ', error
```

1.15 get_extra_values

```
get_extra_values(conf)
```

New in ConfigObj 4.7.0.

Find all the values and sections not in the configspec from a validated ConfigObj.

`get_extra_values` returns a list of tuples where each tuple represents either an extra section, or an extra value.

The tuples contain two values, a tuple representing the section the value is in and the name of the extra values. For extra values in the top level section the first member will be an empty tuple. For values in the 'foo' section the first member will be `('foo',)`. For members in the 'bar' subsection of the 'foo' section the first member will be `('foo', 'bar')`.

Extra sections will only have one entry. Values and subsections inside an extra section aren't listed separately.

NOTE: If you call `get_extra_values` on a ConfigObj instance that hasn't been validated it will return an empty list.

1.15.1 Example Usage

The output from `get_extra_values` is a list of tuples.

Here is an example of how you could present this information to the user.

```
vtor = validate.Validator()
# ini is your config file - cs is the configspec
cfg = ConfigObj(ini, configspec=cs)
cfg.validate(vtor, preserve_errors=True)

for sections, name in get_extra_values(cfg):

    # this code gets the extra values themselves
    the_section = cfg
    for section in sections:
        the_section = the_section[section]

    # the_value may be a section or a value
    the_value = the_section[name]

    section_or_value = 'value'
    if isinstance(the_value, dict):
        # Sections are subclasses of dict
        section_or_value = 'section'

    section_string = ', '.join(sections) or "top level"
    print 'Extra entry in section: %s. Entry %r is a %s' % (section_string, name,␣
↪section_or_value)
```

1.16 CREDITS

ConfigObj version 4 and forward is written by (and copyright) Michael Foord, Nicola Larosa, Rob Dennis and Eli Courtwright.

Rob Dennis and Eli Courtwright added Python 2 and 3 compatibility in a single source starting with version 5, and have taken stewardship of ConfigObj moving forward.

Particularly thanks to Nicola Larosa for help on the config file spec, the validation system and the doctests.

validate.py was originally written by Michael Foord and Mark Andrews.

Thanks to many others for input, patches and bugfixes.

1.17 LICENSE

ConfigObj, and related files, are licensed under the BSD license. This is a very unrestrictive license, but it comes with the usual disclaimer. This is free software: test it, break it, just don't blame us if it eats your data ! Of course if it does, let us know and we'll fix the problem so it doesn't happen to anyone else:

```
Copyright (C) 2005-2014:
(name) : (email)
Michael Foord: fuzzyman AT voidspace DOT org DOT uk
Nicola Larosa: nico AT tekNico DOT net
Rob Dennis: rdennis AT gmail DOT com
Eli Courtwright: eli AT courtwright DOT org

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

    * None of the authors names may be used to endorse or
      promote products derived from this software without
      specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

You should also be able to find a copy of this license at : BSD License

1.18 TODO

Better support for configuration from multiple files, including tracking *where* the original file came from and writing changes to the correct file.

Make `newline` a keyword argument (as well as an attribute) ?

`UTF16` encoded files, when returned as a list of lines, will have the BOM at the start of every line. Should this be removed from all but the first line ?

Option to set warning type for unicode decode ? (Defaults to strict).

A method to optionally remove uniform indentation from multiline values. (do as an example of using `walk` - along with string-escape)

Should the results dictionary from validate be an ordered dictionary if odict is available ?

Implement some of the sequence methods (which include slicing) from the newer `odict` ?

Preserve line numbers of values (and possibly the original text of each value).

1.19 ISSUES

Note: Please file any bug reports at the Github Page

There is currently no way to specify the encoding of a configspec file.

As a consequence of the changes to configspec handling in version 4.6.0, when you create a ConfigObj instance and provide a configspec, the configspec attribute is only set on the ConfigObj instance - it isn't set on the sections until you validate. You also can't set the configspec attribute to be a dictionary. This wasn't documented but did work previously.

In order to fix the problem with hashes in configspecs I had to turn off the parsing of inline comments in configspecs. This will only affect you if you are using `copy=True` when validating and expecting inline comments to be copied from the configspec into the ConfigObj instance (all other comments will be copied as usual).

If you *create* the configspec by passing in a ConfigObj instance (usual way is to pass in a filename or list of lines) then you should pass in `_inspec=True` to the constructor to allow hashes in values. This is the magic that switches off inline comment parsing.

When using `copy` mode for validation, it won't copy `DEFAULT` sections. This is so that you *can* use interpolation in configspec files. This is probably true even if interpolation is off in the configspec.

You can't have a keyword with the same name as a section (in the same section). They are both dictionary keys - so they would overlap.

ConfigObj doesn't quote and unquote values if `list_values=False`. This means that leading or trailing whitespace in values will be lost when writing. (Unless you manually quote).

Interpolation checks first the current section, then the 'DEFAULT' subsection of the current section, before moving on to the current section's parent and so on up the tree.

Does it matter that we don't support the ':' divider, which is supported by `ConfigParser` ?

String interpolation and validation don't play well together. When validation changes type it sets the value. This will correctly fetch the value using interpolation - but then overwrite the interpolation reference. If the value is unchanged by validation (it's a string) - but other types will be.

1.20 CHANGELOG

This is an abbreviated changelog showing the major releases up to version 4. From version 4 it lists all releases and changes.

1.20.1 2014/08/25 - Version 5.0.6

- BUGFIX: Did not correctly handle %-chars in invalid lines
- BUGFIX: unhelpful error message when nesting invalid

1.20.2 2014/04/28 - Version 5.0.5

- BUGFIX: error in writing out config files to disk with non-ascii characters

1.20.3 2014/04/11 - Version 5.0.4

- BUGFIX: correcting that the code path fixed in 5.0.3 didn't cover reading in config files

1.20.4 2014/04/04 - Version 5.0.3

- BUGFIX: not handling unicode encoding well, especially with respect to writing out files

1.20.5 2014/02/27 - Version 5.0.2

- Specific error message for installing version this version on Python versions older than 2.5
- Documentation corrections

1.20.6 2014/02/19 - Version 5.0.1

- BUGFIX: Fixed regression on python 2.x where passing an `encoding` parameter did not convert a bytestring config file (which is the most common) to unicode. Added unit tests for this and related cases
- BUGFIX: A particular error message would fail to display with a type error on python 2.6 only

1.20.7 2014/02/08 - Version 5.0.0

- Python 3 single-source compatibility at the cost of a more restrictive set of versions: 2.6, 2.7, 3.2, 3.3 (otherwise unchanged)
- New maintainers: Rob Dennis and Eli Courtwright
- New home on github

1.20.8 2010/02/27 - Version 4.7.2

- BUGFIX: Restore Python 2.3 compatibility
- BUGFIX: Members that were lists were being returned as copies due to interpolation introduced in 4.7. Lists are now only copies if interpolation changes a list member.
- BUGFIX: `pop` now does interpolation in list values as well.
- BUGFIX: where interpolation matches a section name rather than a value it is ignored instead of raising an exception on fetching the item.
- BUGFIX: values that use interpolation to reference members that don't exist can now be repr'd.
- BUGFIX: Fix to avoid writing '\r\r\n' on Windows when given a file opened in text write mode ('w').

See *String Interpolation and List Values* for information about the problem with lists and interpolation.

1.20.9 2010/02/06 - Version 4.7.1

- Fix bug in options deprecation warning added in 4.7.0

1.20.10 2010/01/09 - Version 4.7.0

- Minimum supported version of Python is now 2.3
- ~25% performance improvement thanks to Christian Heimes
- String interpolation now works in list value members

- After validation any additional entries not in the configspec are listed in the `extra_values` section member
- Addition of the `get_extra_values` function for finding all extra values in a validated ConfigObj instance
- Deprecated the use of the `options` dictionary in the ConfigObj constructor and added explicit keyword arguments instead. Use **options if you want to initialise a ConfigObj instance from a dictionary
- Constructing a ConfigObj from an existing ConfigObj instance now preserves the order of values and sections from the original instance in the new one
- BUGFIX: Checks that failed validation would not populate `default_values` and `restore_default_value()` wouldn't work for those entries
- BUGFIX: clear() now clears 'defaults'
- BUGFIX: empty values in list values were accidentally valid syntax. They now raise a `ParseError`. e.g. "value = 1, , 2"
- BUGFIX: Change to the result of a call to `validate` when `preserve_errors` is True. Previously sections where *all* values failed validation would return False for the section rather than preserving the errors. False will now only be returned for a section if it is missing
- Distribution includes version 1.0.1 of validate.py
- Removed __revision__ and __docformat__

1.20.11 2009/04/13 - Version 4.6.0

- Pickling of ConfigObj instances now supported (thanks to Christian Heimes)
- Hashes in confgspecs are now allowed (see note below)
- Replaced use of hasattr (which can swallow exceptions) with getattr
- __many__ in configspecs can refer to scalars (ordinary values) as well as sections
- You can use ___many___ (three underscores!) where you want to use __many__ as well
- You can now have normal sections inside configspec sections that use __many__
- You can now create an empty ConfigObj with a configspec, programmatically set values and then validate
- A section that was supplied as a value (or vice-versa) in the actual config file would cause an exception during validation (the config file is still broken of course, but it is now handled gracefully)
- Added `as_list` method
- Removed the deprecated `istrue`, `encode` and `decode` methods
- Running test_configobj.py now also runs the doctests in the configobj module

As a consequence of the changes to configspec handling, when you create a ConfigObj instance and provide a configspec, the configspec attribute is only set on the ConfigObj instance - it isn't set on the sections until you validate. You also can't set the configspec attribute to be a dictionary. This wasn't documented but did work previously.

In order to fix the problem with hashes in configspecs I had to turn off the parsing of inline comments in configspecs. This will only affect you if you are using `copy=True` when validating and expecting inline comments to be copied from the configspec into the ConfigObj instance (all other comments will be copied as usual).

If you *create* the configspec by passing in a ConfigObj instance (usual way is to pass in a filename or list of lines) then you should pass in `_inspec=True` to the constructor to allow hashes in values. This is the magic that switches off inline comment parsing.

1.20.12 2008/06/27 - Version 4.5.3

BUGFIX: fixed a problem with `copy=True` when validating with configspecs that use `__many__` sections.

1.20.13 2008/02/05 - Version 4.5.2

Distribution updated to include version 0.3.2 of *validate*. This means that `None` as a default value in configspecs works.

1.20.14 2008/02/05 - Version 4.5.1

Distribution updated to include version 0.3.1 of *validate*. This means that Unicode configspecs now work.

1.20.15 2008/02/05 - Version 4.5.0

ConfigObj will now guarantee that files will be written terminated with a newline.

ConfigObj will no longer attempt to import the `validate` module, until/unless you call `ConfigObj.validate` with `preserve_errors=True`. This makes it faster to import.

New methods `restore_default` and `restore_defaults`. `restore_default` resets an entry to its default value (and returns that value). `restore_defaults` resets all entries to their default value. It doesn't modify entries without a default value. You must have validated a ConfigObj (which populates the `default_values` dictionary) before calling these methods.

BUGFIX: Proper quoting of keys, values and list values that contain hashes (when writing). When `list_values=False`, values containing hashes are triple quoted.

Added the `reload` method. This reloads a ConfigObj from file. If the filename attribute is not set then a `ReloadError` (a new exception inheriting from `IOError`) is raised.

BUGFIX: Files are read in with 'rb' mode, so that native/non-native line endings work!

Minor efficiency improvement in `unrepr` mode.

Added missing docstrings for some overidden dictionary methods.

Added the `reset` method. This restores a ConfigObj to a freshly created state.

Removed old CHANGELOG file.

1.20.16 2007/02/04 - Version 4.4.0

Official release of 4.4.0

1.20.17 2006/12/17 - Version 4.3.3-alpha4

By Nicola Larosa

Allowed arbitrary indentation in the `indent_type` parameter, removed the `NUM_INDENT_SPACES` and `MAX_INTERPOL_DEPTH` (a leftover) constants, added indentation tests (including another docutils workaround, sigh), updated the documentation.

By Michael Foord

Made the import of `compiler` conditional so that `ConfigObj` can be used with IronPython.

1.20.18 2006/12/17 - Version 4.3.3-alpha3

By Nicola Larosa

Added a missing `self.` in the _handle_comment method and a related test, per Sourceforge bug #1523975.

1.20.19 2006/12/09 - Version 4.3.3-alpha2

By Nicola Larosa

Changed interpolation search strategy, based on this patch by Robin Munn: http://sourceforge.net/mailarchive/message.php?msg_id=17125993

1.20.20 2006/12/09 - Version 4.3.3-alpha1

By Nicola Larosa

Added Template-style interpolation, with tests, based on this patch by Robin Munn: http://sourceforge.net/mailarchive/message.php?msg_id=17125991 (awful archives, bad Sourceforge, bad).

1.20.21 2006/06/04 - Version 4.3.2

Changed error handling, if parsing finds a single error then that error will be re-raised. That error will still have an `errors` and a `config` attribute.

Fixed bug where '\n' terminated files could be truncated.

Bugfix in `unrepr` mode, it couldn't handle '#' in values. (Thanks to Philippe Normand for the report.)

As a consequence of this fix, ConfigObj doesn't now keep inline comments in `unrepr` mode. This is because the parser in the compiler package doesn't keep comments.

Error messages are now more useful. They tell you the number of parsing errors and the line number of the first error. (In the case of multiple errors.)

Line numbers in exceptions now start at 1, not 0.

Errors in `unrepr` mode are now handled the same way as in the normal mode. The errors stored will be an `UnreprError`.

1.20.22 2006/04/29 - Version 4.3.1

Added `validate.py` back into `configobj.zip`. (Thanks to Stewart Midwinter)

Updated to `validate.py` 0.2.2.

Preserve tuples when calling the `dict` method. (Thanks to Gustavo Niemeyer.)

Changed `__repr__` to return a string that contains `ConfigObj({ ... })`.

Change so that an options dictionary isn't modified by passing it to ConfigObj. (Thanks to Artarious.)

Added ability to handle negative integers in `unrepr`. (Thanks to Kevin Dangoor.)

1.20.23 2006/03/24 - Version 4.3.0

Moved the tests and the CHANGELOG (etc) into a separate file. This has reduced the size of `configobj.py` by about 40%.

Added the `unrepr` mode to reading and writing config files. Thanks to Kevin Dangoor for this suggestion.

Empty values are now valid syntax. They are read as an empty string `''`. (`key =`, or `key = # comment`.)

`validate` now honours the order of the configspec.

Added the `copy` mode to validate. Thanks to Louis Cordier for this suggestion.

Fixed bug where files written on windows could be given `'\r\r\n'` line terminators.

Fixed bug where last occurring comment line could be interpreted as the final comment if the last line isn't terminated.

Fixed bug where nested list values would be flattened when `write` is called. Now sub-lists have a string representation written instead.

Deprecated `encode` and `decode` methods instead.

You can now pass in a ConfigObj instance as a configspec (remember to read the configspec file using `list_values=False`).

Sorted footnotes in the docs.

1.20.24 2006/02/16 - Version 4.2.0

Removed `BOM_UTF8` from `__all__`.

The `BOM` attribute has become a boolean. (Defaults to `False`.) It is *only* `True` for the `UTF16/UTF8` encodings.

File like objects no longer need a `seek` attribute.

Full unicode support added. New options/attributes `encoding`, `default_encoding`.

ConfigObj no longer keeps a reference to file like objects. Instead the `write` method takes a file like object as an optional argument. (Which will be used in preference of the `filename` attribute if that exists as well.)

utf16 files decoded to unicode.

If `BOM` is `True`, but no encoding specified, then the utf8 BOM is written out at the start of the file. (It will normally only be `True` if the utf8 BOM was found when the file was read.)

Thanks to Aaron Bentley for help and testing on the unicode issues.

File paths are *not* converted to absolute paths, relative paths will remain relative as the `filename` attribute.

Fixed bug where `final_comment` wasn't returned if `write` is returning a list of lines.

Deprecated `istrue`, replaced it with `as_bool`.

Added `as_int` and `as_float`.

1.20.25 2005/12/14 - Version 4.1.0

Added `merge`, a recursive update.

Added `preserve_errors` to `validate` and the `flatten_errors` example function.

Thanks to Matthew Brett for suggestions and helping me iron out bugs.

Fixed bug where a config file is *all* comment, the comment will now be `initial_comment` rather than `final_comment`.

Validation no longer done on the 'DEFAULT' section (only in the root level). This allows interpolation in configspecs.

Also use the new list syntax in *validate* 0.2.1. (For configspecs).

1.20.26 2005/12/02 - Version 4.0.2

Fixed bug in `create_empty`. Thanks to Paul Jimenez for the report.

1.20.27 2005/11/05 - Version 4.0.1

Fixed bug in `Section.walk` when transforming names as well as values.

Added the `istrue` method. (Fetches the boolean equivalent of a string value).

Fixed `list_values=False` - they are now only quoted/unquoted if they are multiline values.

List values are written as `item, item` rather than `item,item`.

1.20.28 2005/10/17 - Version 4.0.0

ConfigObj 4.0.0 Final

Fixed bug in `setdefault`. When creating a new section with setdefault the reference returned would be to the dictionary passed in *not* to the new section. Bug fixed and behaviour documented.

Obscure typo/bug fixed in `write`. Wouldn't have affected anyone though.

1.20.29 2005/09/09 - Version 4.0.0 beta 5

Removed `PositionError`.

Allowed quotes around keys as documented.

Fixed bug with commas in comments. (matched as a list value)

1.20.30 2005/09/07 - Version 4.0.0 beta 4

Fixed bug in `__delitem__`. Deleting an item no longer deletes the `inline_comments` attribute.

Fixed bug in initialising ConfigObj from a ConfigObj.

Changed the mailing list address.

1.20.31 2005/08/28 - Version 4.0.0 beta 3

Interpolation is switched off before writing out files.

Fixed bug in handling `StringIO` instances. (Thanks to report from Gustavo Niemeyer.)

Moved the doctests from the `__init__` method to a separate function. (For the sake of IDE calltips).

1.20.32 2005/08/25 - Version 4.0.0 beta 2

Amendments to *validate.py*.

First public release.

1.20.33 2005/08/21 - Version 4.0.0 beta 1

Reads nested subsections to any depth.

Multiline values.

Simplified options and methods.

New list syntax.

Faster, smaller, and better parser.

Validation greatly improved. Includes:

- type conversion
- default values
- repeated sections

Improved error handling.

Plus lots of other improvements.

1.20.34 2004/05/24 - Version 3.0.0

Several incompatible changes: another major overhaul and change. (Lots of improvements though).

Added support for standard config files with sections. This has an entirely new interface: each section is a dictionary of values.

Changed the update method to be called writein: update clashes with a dict method.

Made various attributes keyword arguments, added several.

Configspecs and orderlists have changed a great deal.

Removed support for adding dictionaries: use update instead.

Now subclasses a new class called caselessDict. This should add various dictionary methods that could have caused errors before.

It also preserves the original casing of keywords when writing them back out.

Comments are also saved using a `caselessDict`.

Using a non-string key will now raise a `TypeError` rather than converting the key.

Added an exceptions keyword for *much* better handling of errors.

Made `creatempty=False` the default.

Now checks indict *and* any keyword args. Keyword args take precedence over indict.

`' ', ':', '=', ','` and `'\t'` are now all valid dividers where the keyword is unquoted.

ConfigObj now does no type checking against configspec when you set items.

delete and add methods removed (they were unnecessary).

Docs rewritten to include all this gumph and more; actually ConfigObj is *really* easy to use.

Support for stdout was removed.

A few new methods added.

Charmap is now incorporated into ConfigObj.

1.20.35 2004/03/14 - Version 2.0.0 beta

Re-written it to subclass dict. My first forays into inheritance and operator overloading.

The config object now behaves like a dictionary.

I've completely broken the interface, but I don't think anyone was really using it anyway.

This new version is much more 'classy'.

It will also read straight from/to a filename and completely parse a config file without you *having* to supply a config spec.

Uses listparse, so can handle nested list items as values.

No longer has getval and setval methods: use normal dictionary methods, or add and delete.

1.20.36 2004/01/29 - Version 1.0.5

Version 1.0.5 has a couple of bugfixes as well as a couple of useful additions over previous versions.

Since 1.0.0 the buildconfig function has been moved into this distribution, and the methods reset, verify, getval and setval have been added.

A couple of bugs have been fixed.

1.20.37 Origins

ConfigObj originated in a set of functions for reading config files in the atlantibots project. The original functions were written by Rob McNeur.

1.21 Footnotes

CHAPTER 2

Using the Validator class

Authors Michael Foord, Nicola Larosa, Rob Dennis, Eli Courtwright, Mark Andrews

Version Validate 2.0.0

Date 2014/02/08

Homepage Github Page

License BSD License

Support Mailing List

Validate Manual

2.1 Introduction

Validation is used to check that supplied values conform to a specification.

The value can be supplied as a string, e.g. from a config file. In this case the check will also *convert* the value to the required type. This allows you to add validation as a transparent layer to access data stored as strings. The validation checks that the data is correct *and* converts it to the expected type.

Checks are also strings, and are easy to write. One generic system can be used to validate information from different sources via a single consistent mechanism.

Checks look like function calls, and map to function calls. They can include parameters and keyword arguments. These arguments are passed to the relevant function by the `Validator` instance, along with the value being checked.

The syntax for checks also allows for specifying a default value. This default value can be `None`, no matter what the type of the check. This can be used to indicate that a value was missing, and so holds no useful value.

Functions either return a new value, or raise an exception. See *Validator Exceptions* for the low down on the exception classes that `validate.py` defines.

Some standard functions are provided, for basic data types; these come built into every validator. Additional checks are easy to write: they can be provided when the `Validator` is instantiated, or added afterwards.

Validate was primarily written to support ConfigObj, but is designed to be applicable to many other situations.

For support and bug reports please use the ConfigObj Github Page

2.2 Downloading

The current version is **2.0.0**, dated 8th February 2014.

You can obtain validate in the following ways :

2.2.1 Files

- validate.py from Github Page
- The latest development version can be obtained from the Github Page.

2.3 The standard functions

The standard functions come built-in to every `Validator` instance. They work with the following basic data types :

- integer
- float
- boolean
- string
- ip_addr

plus lists of these datatypes.

Adding additional checks is done through coding simple functions.

The full set of standard checks are :

'integer' matches integer values (including negative). Takes optional 'min' and 'max' arguments:

```
integer()
integer(3, 9)     # any value from 3 to 9
integer(min=0)  # any positive value
integer(max=9)
```

'float' matches float values Has the same parameters as the integer check.

'boolean'

matches boolean values: `True` or `False`. Acceptable string values for True are:

```
true, on, yes, 1
```

Acceptable string values for False are:

```
false, off, no, 0
```

Any other value raises an error.

'string' matches any string. Takes optional keyword args 'min' and 'max' to specify min and max length of string.

'ip_addr' matches an Internet Protocol address, v.4, represented by a dotted-quad string, i.e. '1.2.3.4'.

'list' matches any list. Takes optional keyword args 'min', and 'max' to specify min and max sizes of the list. The list checks always return a list.

'force_list' is the same as 'list', but if anything but a list or tuple is passed in, it will coerce it into a list containing that value. Useful to avoid confusion for users not accustomed to Python idioms and thus forget the trailing comma to turn a single value into a list.

'tuple' matches any list. This check returns a tuple rather than a list.

'int_list' Matches a list of integers. Takes the same arguments as list.

'float_list' Matches a list of floats. Takes the same arguments as list.

'bool_list' Matches a list of boolean values. Takes the same arguments as list.

'string_list' Matches a list of strings. Takes the same arguments as list.

'ip_addr_list' Matches a list of IP addresses. Takes the same arguments as list.

'mixed_list' Matches a list with different types in specific positions. List size must match the number of arguments.

Each position can be one of:

```
int, str, boolean, float, ip_addr
```

So to specify a list with two strings followed by two integers, you write the check as:

```
mixed_list(str, str, int, int)
```

'pass' matches everything: it never fails and the value is unchanged. It is also the default if no check is specified.

'option' matches any from a list of options. You specify this test with:

```
option('option 1', 'option 2', 'option 3')
```

The following code will work without you having to specifically add the functions yourself.

```
from configobj.validate import Validator
#
vtor = Validator()
newval1 = vtor.check('integer', value1)
newval2 = vtor.check('boolean', value2)
# etc ...
```

Note: Of course, if these checks fail they raise exceptions. So you should wrap them in `try...except` blocks. Better still, use ConfigObj for a higher level interface.

2.4 Using Validator

Using `Validator` is very easy. It has one public attribute and one public method.

Shown below are the different steps in using `Validator`.

The only additional thing you need to know, is about *Writing check functions*.

2.4.1 Instantiate

```
from configobj.validate import Validator
vtor = Validator()
```

or even :

```
from configobj.validate import Validator
#
fdict = {
    'check_name1': function1,
    'check_name2': function2,
    'check_name3': function3,
}
#
vtor = Validator(fdict)
```

The second method adds a set of your functions as soon as your validator is created. They are stored in the `vtor.functions` dictionary. The 'key' you give them in this dictionary is the name you use in your checks (not the original function name).

Dictionary keys/functions you pass in can override the built-in ones if you want.

2.4.2 Adding functions

The code shown above, for adding functions on instantiation, has exactly the same effect as the following code :

```
from configobj.validate import Validator
#
vtor = Validator()
vtor.functions['check_name1'] = function1
vtor.functions['check_name2'] = function2
vtor.functions['check_name3'] = function3
```

`vtor.functions` is just a dictionary that maps names to functions, so we could also have called `vtor.functions.update(fdict)`.

2.4.3 Writing the check

As we've heard, the checks map to the names in the `functions` dictionary. You've got a full list of *The standard functions* and the arguments they take.

If you're using `Validator` from ConfigObj, then your checks will look like:

```
keyword = int_list(max=6)
```

but the check part will be identical .

2.4.4 The check method

If you're not using `Validator` from ConfigObj, then you'll need to call the `check` method yourself.

If the check fails then it will raise an exception, so you'll want to trap that. Here's the basic example :

```
from configobj.validate import Validator, ValidateError
#
vtor = Validator()
check = "integer(0, 9)"
value = 3
try:
    newvalue = vtor.check(check, value)
except ValidateError:
    print 'Check Failed.'
else:
    print 'Check passed.'
```

Caution: Although the value can be a string, if it represents a list it should already have been turned into a list of strings.

Default Values

Some values may not be available, and you may want to be able to specify a default as part of the check.

You do this by passing the keyword `missing=True` to the `check` method, as well as a `default=value` in the check. (Constructing these checks is done automatically by ConfigObj: you only need to know about the `default=value` part) :

```
check1 = 'integer(default=50)'
check2 = 'option("val 1", "val 2", "val 3", default="val 1")'

assert vtor.check(check1, '', missing=True) == 50
assert vtor.check(check2, '', missing=True) == "val 1"
```

If you pass in `missing=True` to the check method, then the actual value is ignored. If no default is specified in the check, a `ValidateMissingValue` exception is raised. If a default is specified then that is passed to the check instead.

If the check has `default=None` (case sensitive) then `vtor.check` will *always* return `None` (the object). This makes it easy to tell your program that this check contains no useful value when missing, i.e. the value is optional, and may be omitted without harm.

Note: As of version 0.3.0, if you specify `default='None'` (note the quote marks around `None`) then it will be interpreted as the string `'None'`.

List Values

It's possible that you would like your default value to be a list. It's even possible that you will write your own check functions - and would like to pass them keyword arguments as lists from within the check.

To avoid confusing syntax with commas and quotes you use a list constructor to specify that keyword arguments are lists. This includes the `default` value. This makes checks look something like:

```
checkname(default=list('val1', 'val2', 'val3'))
```

2.4.5 get_default_value

`Validator` instances have a `get_default_value` method. It takes a `check` string (the same string you would pass to the `check` method) and returns the default value, converted to the right type. If the check doesn't define a default value then this method raises a `KeyError`.

If the `check` has been seen before then it will have been parsed and cached already, so this method is not expensive to call (however the conversion is done each time).

2.5 Validator Exceptions

Note: If you only use Validator through ConfigObj, it traps these Exceptions for you. You will still need to know about them for writing your own check functions.

`vtor.check` indicates that the check has failed by raising an exception. The appropriate error should be raised in the check function.

The base error class is `ValidateError`. All errors (except for `VdtParamError`) raised are sub-classes of this.

If an unrecognised check is specified then `VdtUnknownCheckError` is raised.

There are also `VdtTypeError` and `VdtValueError`.

If incorrect parameters are passed to a check function then it will (or should) raise `VdtParamError`. As this indicates *programmer* error, rather than an error in the value, it is a subclass of `SyntaxError` instead of `ValidateError`.

Note: This means it *won't* be caught by ConfigObj - but propagated instead.

If the value supplied is the wrong type, then the check should raise `VdtTypeError`. e.g. the check requires the value to be an integer (or representation of an integer) and something else was supplied.

If the value supplied is the right type, but an unacceptable value, then the check should raise `VdtValueError`. e.g. the check requires the value to be an integer (or representation of an integer) less than ten and a higher value was supplied.

Both `VdtTypeError` and `VdtValueError` are initialised with the incorrect value. In other words you raise them like this :

```
raise VdtTypeError(value)
#
raise VdtValueError(value)
```

`VdtValueError` has the following subclasses, which should be raised if they are more appropriate.

- `VdtValueTooSmallError`
- `VdtValueTooBigError`
- `VdtValueTooShortError`
- `VdtValueTooLongError`

2.6 Writing check functions

Writing check functions is easy.

The check function will receive the value as its first argument, followed by any other parameters and keyword arguments.

If the check fails, it should raise a `VdtTypeError` or a `VdtValueError` (or an appropriate subclass).

All parameters and keyword arguments are *always* passed as strings. (Parsed from the check string).

The value might be a string (or list of strings) and need converting to the right type - alternatively it might already be a list of integers. Our function needs to be able to handle either.

If the check passes then it should return the value (possibly converted to the right type).

And that's it !

2.6.1 Example

Here is an example function that requires a list of integers. Each integer must be between 0 and 99.

It takes a single argument specifying the length of the list. (Which allows us to use the same check in more than one place). If the length can't be converted to an integer then we need to raise `VdtParamError`.

Next we check that the value is a list. Anything else should raise a `VdtTypeError`. The list should also have 'length' entries. If the list has more or less entries then we will need to raise a `VdtValueTooShortError` or a `VdtValueTooLongError`.

Then we need to check every entry in the list. Each entry should be an integer between 0 and 99, or a string representation of an integer between 0 and 99. Any other type is a `VdtTypeError`, any other value is a `VdtValueError` (either too big, or too small).

```
def special_list(value, length):
    """
    Check that the supplied value is a list of integers,
    with 'length' entries, and each entry between 0 and 99.
    """
    # length is supplied as a string
    # we need to convert it to an integer
    try:
        length = int(length)
    except ValueError:
        raise VdtParamError('length', length)
    #
    # Check the supplied value is a list
    if not isinstance(value, list):
        raise VdtTypeError(value)
    #
    # check the length of the list is correct
    if len(value) > length:
        raise VdtValueTooLongError(value)
    elif len(value) < length:
        raise VdtValueTooShortError(value)
    #
    # Next, check every member in the list
    # converting strings as necessary
    out = []
    for entry in value:
```

```
        if not isinstance(entry, (str, unicode, int)):
            # a value in the list
            # is neither an integer nor a string
            raise VdtTypeError(value)
        elif isinstance(entry, (str, unicode)):
            if not entry.isdigit():
                raise VdtTypeError(value)
            else:
                entry = int(entry)
        if entry < 0:
            raise VdtValueTooSmallError(value)
        elif entry > 99:
            raise VdtValueTooBigError(value)
        out.append(entry)
    #
    # if we got this far, all is well
    # return the new list
    return out
```

If you are only using validate from ConfigObj then the error type (*TooBig*, *TooSmall*, etc) is lost - so you may only want to raise `VdtValueError`.

Caution: If your function raises an exception that isn't a subclass of `ValidateError`, then ConfigObj won't trap it. This means validation will fail.

This is why our function starts by checking the type of the value. If we are passed the wrong type (e.g. an integer rather than a list) we get a `VdtTypeError` rather than bombing out when we try to iterate over the value.

If you are using validate in another circumstance you may want to create your own subclasses of `ValidateError` which convey more specific information.

2.7 Known Issues

The following parses and then blows up. The resulting error message is confusing:

```
checkname(default=list(1, 2, 3, 4)
```

This is because it parses as: `checkname(default="list(1", 2, 3, 4)`. That isn't actually unreasonable, but the error message won't help you work out what has happened.

2.8 TODO

- A regex check function ?
- A timestamp check function ? (Using the `parse` function from `DateUtil` perhaps).

2.9 ISSUES

Note: Please file any bug reports to the Github Page

If we could pull tuples out of arguments, it would be easier to specify arguments for 'mixed_lists'.

2.10 CHANGELOG

2.10.1 2014/02/08 - Version 2.0.0

- Python 3 single-source compatibility at the cost of a more restrictive set of versions: 2.6, 2.7, 3.2, 3.3 (otherwise unchanged)
- New maintainers: Rob Dennis and Eli Courtwright
- New home on github

2.10.2 2009/10/25 - Version 1.0.1

- BUGFIX: Fixed compatibility with Python 2.3.

2.10.3 2009/04/13 - Version 1.0.0

- BUGFIX: can now handle multiline strings.
- Addition of 'force_list' validation option.

As the API is stable and there are no known bugs or outstanding feature requests I am marking this 1.0.

2.10.4 2008/02/24 - Version 0.3.2

BUGFIX: Handling of None as default value fixed.

2.10.5 2008/02/05 - Version 0.3.1

BUGFIX: Unicode checks no longer broken.

2.10.6 2008/02/05 - Version 0.3.0

Improved performance with a parse cache.

New `get_default_value` method. Given a check it returns the default value (converted to the correct type) or raises a `KeyError` if the check doesn't specify a default.

Added 'tuple' check and corresponding 'is_tuple' function (which always returns a tuple).

BUGFIX: A quoted 'None' as a default value is no longer treated as None, but as the string 'None'.

BUGFIX: We weren't unquoting keyword arguments of length two, so an empty string didn't work as a default.

BUGFIX: Strings no longer pass the 'is_list' check. Additionally, the list checks always return lists.

A couple of documentation bug fixes.

Removed CHANGELOG from module.

2.10.7 2007/02/04 Version 0.2.3

Release of 0.2.3

2.10.8 2006/12/17 Version 0.2.3-alpha1

By Nicola Larosa

Fixed validate doc to talk of `boolean` instead of `bool`; changed the `is_bool` function to `is_boolean` (Sourceforge bug #1531525).

2.10.9 2006/04/29 Version 0.2.2

Addressed bug where a string would pass the `is_list` test. (Thanks to Konrad Wojas.)

2.10.10 2005/12/16 Version 0.2.1

Fixed bug so we can handle keyword argument values with commas.

We now use a list constructor for passing list values to keyword arguments (including `default`):

```
default=list("val", "val", "val")
```

Added the `_test` test.

Moved a function call outside a try... except block.

2.10.11 2005/08/18 Version 0.2.0

Updated by Michael Foord and Nicola Larosa

Does type conversion as well.

2.10.12 2005/02/01 Version 0.1.0

Initial version developed by Michael Foord and Mark Andrews.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search