cfgparse — python configuration file parser module

Dan Gass (dan.gass@gmail.com)

April 30, 2005

Version 1.2

Download Source and Documentation: https://sourceforge.net/projects/cfgparse

Requires Python 2.3 -or- Python 2.2 and textwrap module from 2.3

`cfgparse` is a more convenient, flexible, and powerful module for parsing configuration files than the standard library `ConfigParser` module. `cfgparse` uses a more declarative style modelled after the popular `optparse` standard library module.

`cfgparse` can optionally cooperate with the `optparse` module to provide coordination between command line and configuration file options. In addition, the cooperation can be used to allow the user to control features of the parser from the command line.

If you like this module and want to see it in the standard Python distribution, please take the time and add your comments to the Python Configuration File Parser Shootout wiki: http://www.python.org/moin/ConfigParserShootout.

Standard Features

- Simple `ini` style configuration syntax
- Type checking with error handling and help messages
- Help summary modelled after that in `optparse`
- Round trip - read, modify, write configuration files with comment retention
- Cooperates with `optparse` for configuration file options that should be overridden by command line options

Advanced Features

- Supports heirarchically organized option settings
  - User may store multiple option settings in a arbitrarily deep keyed dictionary.
  - Application uses a key list to walk into the dictionary to obtain a setting.
  - User controls key list with setting in configuration file.
  - Supports adding keys to the list through a command line option or from environment variables.
- Supports allowing user control of configuration files used.
  - Environment variables may be used to allow user to specify a default configuration file.

  - Command line options to specify configuration file supported.
  - Configuration files may include other configuration files where where sections are read in parallel.
  - Configuration files may be nested heirarchically by including configuration files from within a section or subsection.
- Configuration files may alternatively be written in Python.
  - full power and flexibility of Python available for creation of option settings
  - allows options settings to be real Python objects
  - this feature is NOT enabled by default
- May be extended to support syntax such as XML.

For example:

```
# file: intro.ini
retries = 10
```

And script:

```
# file: intro.py
import cfgparse
c = cfgparse.ConfigParser()
c.add_option(’retries’, type=’int’)
c.add_file(’intro.ini’)
opts = c.parse()
print ’Number of retries:’,opts.retries
```

Results in:

```
$ python intro.py
Number of retries: 10
```

1 Public interface summary

The following classes, methods, and objects are intended to be used directly it is unlikely the interface will change. The module also contains other publicly available classes, methods, and objects. It is intended that these will be available for advanced users for use in subclassing the configuration parser. Since this module is relatively new it is not recommended that these are used at this time since implementation details of this module may change.

- class `ConfigParser`
  - `__init__`([*description*][,*allow_py*] [,*formatter*][,*exception*]) ¡2.1¿
  - `add_file`([*cfgfile*][,*content*][,*type*] [,*keys*][,*parent*]) ¡2.2¿
  - `add_env_file`(*var*) ¡2.2¿
  - `add_option`(*name*[,*help*][, *type*] [, *choices*][, *dest*][, *metavar*][, *default*] [, *check*][, *keys*]) ¡2.3¿
  - `add_option_group`(*title* [,*description*]) ¡2.4¿
  - `parse`([*optparser*][,*args*]) ¡2.5¿
  - `add_optparse_help_option`(*option_group*[,*switches*] [,*dest*][,*help*]) ¡3.1¿
  - `add_optparse_keys_option`(*option_group*[,*switches*] [,*dest*][,*help*]) ¡3.3¿
  - `add_optparse_files_option`(*option_group*[,*switches*] [,*dest*][,*help*]) ¡3.4¿
  - `print_help`([*file*]) ¡8.1¿
  - `add_note`(*note*) ¡8.2¿
- class `ConfigFileIni` ¡2.2¿
  - `get_filename`()
  - `set_option`(*name*,*value*[,*keys*] [,*help*]) ¡4.2¿
  - `write`(*outfile*) ¡4.3¿
- class `ConfigFilePy` ¡2.2¿
  - `get_filename`()
- class `Option` ¡2.3¿
  - `get`([keys][,errors]) ¡2.5¿
  - `set`(*value*[,*cfgfile*][,*keys*]) ¡4.1¿
  - `add_note`(*note*) ¡8.2¿
- class `OptionGroup` ¡2.4¿
  - `add_option`(*name*[,*help*][, *type*] [, *choices*][, *dest*][, *metavar*][, *default*] [, *check*][, *keys*]) ¡2.3¿
- class `IndentedHelpFormatter` ¡2.1¿
- class `TitledHelpFormatter` ¡2.1¿
- `SUPPRESS_HELP` ¡2.3.4¿

2 Basics

2.1 Creating the parser

class `ConfigParser`

This class can be used to parse options from user configuration files. A single instance of this class is typically created. This single instance may be used to parse multiple configuration files and obtain multiple configuration options.

The constructor for this class is:

`ConfigParser`([*description*][, *allow_py*][, *formatter*][, *exception*])

description is an optional string keyword argument and controls the introductory text placed above configuration option help text. See ”Option Help” (section 2.3.4).

allow_py is an optional boolean keyword argument and when set to `True`, allows Python based configuraton files to be read (executed). The default is `False`. Enabling this feature poses a potential security hole for your application.

formatter is an optional keyword argument and controls the configuration option help text style. Set to either the `IndentedHelpFormatter` or `TitledHelpFormatter` class in the `cfgparse` module (or a subclass of either).

exception is an optional keyword argument and controls the how user errors are handled by the parser. If set to `False` or omitted, errors are written to `sys.stderr` and `sys.exit()` is called. If set to `True`, the `ConfigParserUserError` exception is raised. Otherwise set this argument to the custom exception class that should be raised.

In many applications the defaults for constructing an instance of `ConfigParser` are sufficient.

For example:

```
# file: construct.ini
retries = 10
```

```
# file: construct.py
import cfgparse
c = cfgparse.ConfigParser()
c.add_option('retries', type='int')
c.add_file('construct.ini')
opts = c.parse()
print 'Number of retries:',opts.retries
```

Results in:

```
$ python construct.py
Number of retries: 10
```

2.2 Adding Files

The `add_file()` method of `ConfigParser` is used to add configuration files to the parser. It returns an object which may be used for modifying option settings in the file and writing the changed file contents.

`add_file`([*cfgfile*][,*content*][, *type*][, *underkeys*])

cfgfile is an optional keyword argument and is used to pass a file name string or a file stream. This argument defaults to `None`. See table below for details.

content is an optional keyword argument and is used to pass a file contents string or a file stream. This argument defaults to `None`. See table below for details.

type is an optional keyword argument used to control the parser used to read the configuration file. Argument may be set to either `'ini'`, `'py'`, or `None` (default). When set to `None`, file name extension is used to determine type. `'py'` causes the file to be read (executed) as Python code otherwise the `'ini'` syntax is assumed.

keys is an optional keyword argument. When omitted, `None` or an empty list, options read from this file will be read and stored using the sections as specified in the configuration file. Any keys passed in are used to extend the section names in the configuration file when the file is read and stored. Note, file is only read and parsed if keys used to obtain option settings contain all the keys in this list. See "Keys" (section 6) for more details.

The following table summarizes the legal combinations *cfgfile* and *content* arguments and the resulting **file name** and **file contents** utilized.

cfgfile	***content***	**file name**	**file contents**
filename	`None`	*cfgfile*	file is opened and read
stream	`None`	stream.name	stream is read
filename	stream	*cfgfile*	stream is read
`None`	stream	'stream'	stream is read
filename	string	*cfgfile*	*content*
`None`	string	'heredoc'	*content*

The `add_env_file()` method of `ConfigParser` is used to read a configuration file specified by an environment variable and returns an object which may be used for modifying option settings in the file and writing the changed file contents. For `ini` configuration files the returned object is an instance of `ConfigFileIni`. If the environment variable does not exist, `None` will be returned.

`add_env_file`(*var*[,*keys*])

var is a required positional argument and is the name of the environment variable that contains the configuration filename to add. *keys* is an optional keyword argument. When omitted, `None` or an empty list, options read from this file will be read and stored using the sections as specified in the configuration file. Any keys passed in are used to extend the section names in the configuration file when the file is read and stored. Note, file is only read and parsed if keys used to obtain option settings contain all the keys in this list. See "Keys" (section 6) for more details.

2.3 Adding Options

The `add_option()` method of `ConfigParser` is used to add configuration options to the parser. This is the same concept as adding options with the `optparse` module and shares many of the same arguments (please bring any inconsistencies to the attention of the author). Defining the options in the configuration parser serves the following purposes:

- automatic type and value checking
- default values can be defined
- help can be automatically generated in a consistent format

Options must be added to the parser before the `parse()` method is called. Options may be added before or after files are added to the parser. The following is the `add_option()` method prototype:

`add_option`(*name* [, *help*] [, *type*] [, *choices*] [, *dest*] [, *metavar*] [, *default*] [, *check*] [, *keys*])

name is a positional string argument and is the exact name of the configuration option as it is to appear in the configuration file.

help is an optional string keyword argument and controls the help text associated with this option displayed when configuration help is written. Defaults to `None` which displays no additional help text beyond the option name and metavar. This may be set to `cfgparse.SUPPRESS_HELP` to completely eliminate the option from the help text. See ”Option help” (section 2.3.4) details. If using `optparse` and `cfgparse` in cooperation, *help* may be omitted here and will automatically be picked up from the `optparse` option. See ”Option cooperation” (section 3.2) for more details.

type is an optional string keyword argument and describes the type which the configuration option is to be converted into. See ”Option type” (section 2.3.2) for details. If using `optparse` and `cfgparse` in cooperation, *type* may be omitted here and will automatically be picked up from the `optparse` option. See ”Option cooperation” (section 3.2) for more details.

choices is an optional list keyword argument and is used to pass in the possible choices when *type* is set to `'choice'`. See ”Option type” (section 2.3.2) for details. If using `optparse` and `cfgparse` in cooperation, *choices* may be omitted here and will automatically be picked up from the `optparse` option. See ”Option cooperation” (section 3.2) for more details.

dest is an optional string keyword argument and controls the attribute name that the value of this option will be stored in when the `parse()` method creates an options object. This must be a unique value for every added option. Default is `None` which will cause the *name* argument to be used as the destination attribute. See ”Name and destination” (section 2.3.1) for more details. If using `optparse` and `cfgparse` in cooperation, *dest* must exactly match between the two options. See ”Option cooperation” (section 3.2) for more details.

metavar is an optional string keyword argument and is used control the help text associated with this option. Specifically, this text string is used directly after the `(`’=’`)` sign in the option=VALUE. By default the *dest* argument in all upper case is used. If using `optparse` and `cfgparse` in cooperation, *metavar* may be omitted here and will automatically be picked up from the `optparse` option. See ”Option cooperation” (section 3.2) for more details.

default is an optional keyword argument and is used control the configuration option default value when the configuration option cannot be found. Omitting this option will cause an exception to be raised when the option cannot be found. If using `optparse` and `cfgparse` in cooperation, *default* should NOT be set when adding the `optparse` option. See ”Option cooperation” (section 3.2) for more details.

check is an optional keyword argument and is used to pass in a function to validate (and possibly convert) the configuration option. Function interface requirements are defined in ”Option check” (section 2.3.3).

keys is an optional argument and is used to pass in the section keys to obtain the option. Typically this is set to the section name where the setting is expected to be (the [DEFAULT] section will also be searched as a last resort). Default is `None` which will cause the parser only obtain the option setting from the [DEFAULT] section. See ”Keys” (section 6) for more details.

2.3.1 Name and Destination

The *name* string argument of the `add_option()` method is used to specify the name of the option setting to be obtained from the configuration file. *name* is required and is case sensitive, the option name in the configuration file must exactly match the *name* argument.

The `parse()` method returns an object with attributes set to the option settings specified using the `add_option()` method. The *dest* string argument is used to control the name of the attribute. If *dest* argument is not present, the *name* is used as the attribute name. Each option added must have a unique destination attribute name.

For example:

```
# file: name_dest.ini
mail_server =  192.168.0.0
proxy_server = 192.168.0.100
```

And script:

```
# file: name_dest.py
import cfgparse
c = cfgparse.ConfigParser()
c.add_file('name_dest.ini')
c.add_option('mail_server')
c.add_option('proxy_server', dest='proxy')
opts = c.parse()
print 'Mail Server IP Address =',opts.mail_server
print 'Proxy Server IP Address =',opts.proxy
```

Results in:

```
$ python name_dest.py
Mail Server IP Address = 192.168.0.0
Proxy Server IP Address = 192.168.0.100
```

2.3.2 Option type

The *type* string argument of the `add_option()` method is used to specify the convert the option setting obtained from a configuration file into the desired type. If the configuration option setting cannot be converted to the desired type appropriate help text will be made available (either an exception is raised or `sys.exit()` is called dependent on `exception` argument when instantiating `ConfigParser`). The following table shows the legal values:

value	**result**
`None`	no conversion (default)*
`'choice'`	verifies option setting is a valid choice**
`'complex'`	converts to complex number
`'float'`	converts to floating point number
`'int'`	converts to an integer
`'long'`	converts to a long integer
`'string'`	converts to a string

Notes:

* When parsed, `ini` style configuration files automatically return strings as the option setting and no conversion is necessary. Python based configuration files return objects. Omitting the *type* argument (or setting *type* to *None*) allows the option setting object to remain as is.

** When *type* is set to `'choice'`, the *choices* argument must also be present and must be a list of strings of valid choices.

For example:

```
# file: type.ini
int_option = 10
float_option = 1.5
choice_option = APPLE
```

And script:

```
# file: type.py
import cfgparse
c = cfgparse.ConfigParser()
c.add_file('type.ini')
c.add_option('int_option', type='int')
c.add_option('float_option', type='float')
c.add_option('choice_option', type='choice', choices=['APPLE','ORANGE'])
opts = c.parse()
print opts.int_option*2
print opts.float_option*3
print opts.choice_option
```

Results in:

```
$ python type.py
20
4.5
APPLE
```

2.3.3 Option check

The *check* argument of the `add_option()` method is used to provide a function to check the option setting. In addition the function may also do any further conversions, but it is recommended that the *type* argument be used when possible. The *check* function must accept a single argument, the option setting. The function must return a tuple containing the option setting and an error message. If no error, the error message should be set to `None`.

For example:

```
# file: check.ini
timeout  = 10  # seconds
timeout2 = 101 # invalid
```

And script:

```
# file: check.py

def in_range(value):
    error = None
    if (value <= 0) or (value >= 100):
        error = "’%d’ not valid.  Must be between 0 and 100 seconds." % value
    value = value * 1000 # convert to milliseconds
    return value,error

import cfgparse
c = cfgparse.ConfigParser()
c.add_file(’check.ini’)
c.add_option(’timeout’, type=’int’, check=in_range)
opts = c.parse()
print "Valid timeout:",opts.timeout

c.add_option(’timeout2’, type=’int’, check=in_range)
opts = c.parse()
```

Results in:

```
$ python check.py
Valid timeout: 10000
ERROR: Configuration File Parser

Option: timeout2
File: [CWD]/check.ini
Section: [DEFAULT]
Line: 3
’101’ not valid.  Must be between 0 and 100 seconds.
```

2.3.4 Option Help

The *help* and *metavar* arguments of the `add_option()` method are used to control the configuration option help text automatically generated by the `print_help()` method.

For example:

```
# file: help.py
import cfgparse
c = cfgparse.ConfigParser(description=’Description of the option ’
    ’configuration.’)
c.add_option(’retries’, type=’int’, help=’Maximum number of retries.’)
c.add_option(’timeout’, type=’int’, metavar=’#SEC’,
             help=’Seconds between retries.’)
c.print_help()
```

Results in:

```
$ python help.py
Description of the option configuration.

Configuration file options:
  retries=RETRIES  Maximum number of retries.
  timeout=#SEC     Seconds between retries.
```

metavar controls the setting representation on the right side of the option/setting pair in the help text. If *metavar* is not specified or set to `None` the destination attribute name (in upper case) is used. The destination attribute name is controlled by the *name* and *dest* arguments.

2.3.5 Sections

Configuration files may be organized in sections with each section containing option/value pairs. The *keys* argument of the `add_option()` method can be used to select which section to obtain the option setting from.

For example:

```
# file: sections.ini
[DEFAULT]
# all devices use the same driver type
driver = ethernet

[DEV0]
# path to device #0
path = 192.168.0.0

[DEV1]
# path to device #1
path = 192.168.0.1
```

And script:

```
# file: sections.py
import cfgparse
c = cfgparse.ConfigParser()
c.add_option(’driver’, dest=’driver0’, keys=’DEV0’)
c.add_option(’driver’, dest=’driver1’, keys=’DEV1’)
c.add_option(’path’, dest=’path0’, keys=’DEV0’)
c.add_option(’path’, dest=’path1’, keys=’DEV1’)
c.add_file(’sections.ini’)
opts = c.parse()
print "DEV0:",opts.driver0,opts.path0
print "DEV1:",opts.driver1,opts.path1
```

Results in:

```
$ python sections.py
DEV0: ethernet 192.168.0.0
DEV1: ethernet 192.168.0.1
```

The `[DEFAULT]` section is special. If an option setting cannot be found in a section specified by *keys*, the option is obtained from the `[DEFAULT]` section. This section is also utilized if *keys* is not specified. Note, use of `[DEFAULT]` is optional. If omitted, option/setting pairs specified before any section declarations are considered part of the `[DEFAULT]` section. Although not advisable, multiple sections of the same name may exist and are treated as one section without error.

2.4 Groups

The `add_option_group()` method can be used to create groups of options. The purpose of grouping options is strictly for organizing the help text to make it more presentable to the user.

`add_option_group`(*title*[,*description*])

title is a required positional string argument and generally is a few words used to label the configuration option group.

description is an optional keyword argument and can be used to provide a more lengthy description of the configuration option group.

`add_option_group()` returns an instance of the `OptionGroup` class which has the same `add_option()` method as the parser.

For example:

```
import cfgparse
c = cfgparse.ConfigParser()
c.add_option('opt0', help='Help for opt0')

group = c.add_option_group('Group 1')
group.add_option('opt1', help='Help for opt1')

group = c.add_option_group('Group 2','Some long winded discussion about '
    'group 2 that will not fit all on a single line if that single line '
    'is not extremely wide.')
group.add_option('opt2', help='Help for opt2')

c.print_help()
```

Results in:

```
$ python groups.py
Configuration file options:
  opt0=OPT0    Help for opt0

  Group 1:
    opt1=OPT1  Help for opt1

  Group 2:
    Some long winded discussion about group 2 that will not fit all on a
    single line if that single line is not extremely wide.

    opt2=OPT2  Help for opt2
```

2.5 Parsing and Obtaining Options

Configuration file options settings may be obtained either all at once or one at a time. The `parse()` method may be used for obtaining all the option settings all at once and returns the options bundled in a single object as attributes. If errors are found, appropriate help text will be made available (either an exception is raised or `sys.exit()` is called dependent on `exception` argument when instantiating `ConfigParser`). This allows errors to be reported to the user up front.

`parse``(`[*optparser*][, *args*]`)`

optparse is an optional keyword argument and is used to pass in an instance of a command line option parser with which the configuration option parser is to cooperate with. Omitting this argument or setting to `None` avoids interfacing to a command line option parser. When used, the `parse()` method will return a tuple of the bundled options object and the command line arguments. The bundled options object will contain the options from both the command line parser and the configuration files. Presence of this option will also allow enable the use of other cooperation features documented in ”Command line cooperation” (section 3).

args is an optional keyword argument and are the arguments to be parsed by the command line option parser. Omitting this argument or setting to `None` causes arguments in `sys.argv` (from the command line) to be utilized.

An alternative method is to get the options as they are needed using the `get()` method of the object returned by the `add_option()` method.

`get``(`[*keys*][, *errors*]`)`

keys is an optional keyword argument and is used to pass in additional keys to use obtain the option setting. See ”Keys” (section 6) for more information.

errors is an optional keyword argument. If omitted or set to `None` any errors will cause appropriate help text will be made available (either an exception is raised or `sys.exit()` is called dependent on `exception` argument when instantiating `ConfigParser`). Otherwise pass a list and help text describing the error will appended into the list.

For example:

```
# file: parsing.ini
retries = 10
```

And script:

```
# file: parsing.py
import cfgparse
c = cfgparse.ConfigParser()
c.add_file(’parsing.ini’)

retries = c.add_option(’retries’, type=’int’)
print retries.get()

timeout = c.add_option(’timeout’, type=’int’)
print timeout.get()
```

Results in:

```
$ python parsing.py
10
ERROR: Configuration File Parser

Option: timeout
No valid default found.
keys=DEFAULT
```

3 Command line cooperation

The *optparser* and *args* arguments of the `parse()` method are utilized to enable cooperation between the configuation option parser and an instance of the `optparse.OptionParser` class.

When cooperation is enabled the options object created by the `parse()` method contains attributes for options from both the command line option parser and the configuration file parser. If the destination attribute name is the same for an option in both the command line and configuration file option parsers, the parsers will cooperate with one another as documented in ”Option cooperation” (section 3.2).

3.1 Help switch

The `add_optparse_help_option()` method is used to set up the cooperation between the command line and configuration file parsers to automatically generate configuration file help text using a command line switch.

`add_optparse_help_option`(*option_group*[, *switches*] [, *dest*][, *help*])

> *option_group* is a required positional argument and must be set to an instance of the `optparse` module’s `OptionParser` class or an option group of that class. The `OptionParser` instance must also be passed into the `parse()` method.
>
> *switches* is an optional keyword argument and is used to set the command line switches the user can use to invoke the configuration file help printout. *switches* must be a tuple and defaults to `(’--cfghelp’,)` when *switches* is omitted.
>
> *dest* is an optional string keyword argument and is used set the name of the destination attribute of the options object returned by the command line option parser. When omitted, *dest* defaults to `’cfgparse_help’`.
>
> *help* is an optional string keyword argument and is used to set the command line switch help string. When omitted a reasonable default help string is utilized.

For example:

```
# file: coop_help.py
import optparse, cfgparse
o = optparse.OptionParser()
c = cfgparse.ConfigParser()

c.add_optparse_help_option(o)
c.add_option(’retries’, type=’int’, help=’Maximum number of retries.’)
c.add_option(’timeout’, type=’int’, metavar=’#SEC’,
             help=’Seconds between retries.’)
(opts,args) = c.parse(o)

print "Should not get here if command line help switch present"
```

Results in:

```
$ python coop_help.py --help
usage: coop_help.py [options]

options:
  -h, --help  show this help message and exit
  --cfghelp   Show configuration file help and exit.

$ python coop_help.py --cfghelp
Configuration file options:
  retries=RETRIES  Maximum number of retries.
  timeout=#SEC     Seconds between retries.
```

3.2 Option cooperation

When the *optparser* argument of the `parse()` method is specified, the same option may be controlled by command line switches or configuration file settings. To enable the cooperation of an option, the destination attribute name must be the same for both the command line parser option and the configuration parser option.

The cooperation is designed so that command line switches have priority over configuration file settings. **IMPORTANT:** when adding options to the command line parser using the `add_option()` method, omit the *default* argument. If it is not omitted the configuration file setting will never be used!

Many of the arguments of the configuration file parser `add_option()` method may be omitted if they were specified when the option was added to the command line parser. When the `parse()` method of the configuration file parser is invoked they will be copied from the command line parser options to the configuration file parser options. Sharing of arguments in the `add_option()` method is not bidirectional. The following is a list of arguments which may be shared:

- *help*
- *type*
- *choices*
- *metavar*

Option cooperation cross reference information is added to the help text associated with both the command line and configuration file parsers. In the case of the command line option help, the information states the existance of the configuration file option. In the case of the configuration file option help, the information states the existance of the associated command line option.

For example:

```
# file: coop_opt.ini
timeout = 10
```

And script:

```
import optparse, cfgparse
o = optparse.OptionParser()
c = cfgparse.ConfigParser()

c.add_optparse_help_option(o)
o.add_option('--timeout', type='int',
             help='Time between retries in seconds.')
c.add_option('timeout')
c.add_file('coop_opt.ini')

(opts,args) = c.parse(o)

print "timeout:",opts.timeout
```

Results in:

```
$ python coop_opt.py
timeout: 10

$ python coop_opt.py --timeout=5
timeout: 5

$ python coop_opt.py --help
usage: coop_opt.py [options]

options:
  -h, --help         show this help message and exit
  --cfghelp          Show configuration file help and exit.
  --timeout=TIMEOUT  Time between retries in seconds.  See also 'timeout'
                     option in configuration file help.

$ python coop_opt.py --cfghelp
Configuration file options:
  timeout=TIMEOUT  Time between retries in seconds.  See also '--timeout'
                   command line switch.
```

3.3 Keys switch

The `add_optparse_keys_option()` method is used to set up the cooperation between the command line and configuration file parsers to allow the user to specify a keys list from the command line to control the keys used to obtain option settings from the configuration file. For more information on how command line keys are used to obtain option settings see "Keys" (section 6).

`add_optparse_keys_option`(*option_group* [, *switches*] [, *dest*] [, *help*])

option_group is a required positional argument and must be set to an instance of the `optparse` module's `OptionParser` class or an option group of that class. The `OptionParser` instance must also be passed into the `parse()` method.

switches is an optional keyword argument and is used to set the command line switches the user can use to specify section keys to be used by the parser for finding option settings. *switches* must be a tuple and defaults to `('-k','--keys',)` when *switches* is omitted.

dest is an optional string keyword argument and is used set the name of the destination attribute of the options object returned by the command line option parser. When omitted, *dest* defaults to `'cfgparse_keys'`.

help is an optional string keyword argument and is used to set the command line switch help string. When omitted a reasonable default help string is utilized.

For example:

```
# file: coop_keys.ini
[DEV0]
# path to device #0
path = 192.168.0.0

[DEV1]
# path to device #1
path = 192.168.0.1
```

And script:

```
# file: coop_keys.py
import optparse
import cfgparse
o = optparse.OptionParser()
c = cfgparse.ConfigParser()
c.add_optparse_keys_option(o)
c.add_option('path')
c.add_file('coop_keys.ini')
(opts,args) = c.parse(o)
print "Path:",opts.path
```

Results in:

```
$ python coop_keys.py
ERROR: Configuration File Parser

Option: path
No valid default found.
keys=DEFAULT

$ python coop_keys.py --keys=DEV0
Path: 192.168.0.0
```

3.4 Files switch

The `add_optparse_files_option()` method is used to set up cooperation between the command line and configuration file parsers to automatically add configuration files to the configuration file parser that were specified using a command line switch (the files are added when the `parse()` method is called).

`add_optparse_files_option`(*option_group* [, *switches*] [, *dest*] [, *help*])

option_group is a required positional argument and must be set to an instance of the `optparse` module's `OptionParser` class or an option group of that class. The `OptionParser` instance must also be passed into the `parse()` method.

switches is an optional keyword argument and is used to set the command line switches the user can use to specify configuration files to be added to the parser. *switches* must be a tuple and defaults to `('--cfgfiles',)` when *switches* is omitted.

dest is an optional string keyword argument and is used set the name of the destination attribute of the options object returned by the command line option parser. When omitted, *dest* defaults to `'cfgparse_files'`.

help is an optional string keyword argument and is used to set the command line switch help string. When omitted a reasonable default help string is utilized.

For example:

```
# file: coop_files.ini
path = 192.168.0.0
```

And script:

```
# file: coop_files.py
import optparse
import cfgparse
o = optparse.OptionParser()
c = cfgparse.ConfigParser()
c.add_optparse_files_option(o)
c.add_option('path')
(opts,args) = c.parse(o)
print "Path:",opts.path
```

Results in:

```
$ python coop_files.py
ERROR: Configuration File Parser

Option: path
No valid default found.
keys=DEFAULT

$ python coop_files.py --cfgfiles=coop_files.ini
Path: 192.168.0.0
```

4 Round Trip

This module supports read/modify/write of `ini` style configuration files with comment retention. Two methods exist for modifying configuration file option settings. The first is the `set()` method of the object returned from the `ConfigParser` class `add_option()` method. The second is the `set_option()` method of the object returned from the `ConfigParser` class `add_file()` method. The `write()` method of the object returned from the `add_file()` and `set()` methods can be used to write the modified file contents. The following subsections provide more detail.

4.1 set method

The `set()` method of the object returned from the `add_option()` method of the `ConfigParser` class can be used to modify an option setting in a configuration file.

`set``(`*value*,[, *cfgfile*][, *keys*]`)`

value is a required positional argument and is the new option value.

cfgfile is an optional keyword argument. If omitted or set to `None` the configuration file will be found. Otherwise the object returned by the `add_file()` method may be passed in to modify or add a setting to a specific file.

keys is an optional string keyword argument. Use of this argument is dependent on the *cfgfile* argument and is discussed further below.

If *cfgfile* argument is omitted or set to `None` the *keys* argument will be used to first get the current option setting. The *keys* are used in the same way as they are in the `get()` method. If a setting cannot be found in any of the configuration files added, `OptionNotFound` will be raised. If the setting is found it will be modified and the originating (modified) file object will be returned. Note, if the option is defined in multiple locations, this method will only modify the one found.

If *cfgfile* argument is set, its `set_option()` method is called using *value*. The *name*, *help*, and *keys* arguments of the `add_option()` method that created the object `set()` was invoked on are passed on to `set_option()`. If *keys* of the `set()` method was not omitted those keys are used instead.

For example:

```
[DEFAULT]
# this section applies to all devices
timeout = 10 # in seconds
retries = 3

[DEV0]
# this section if for settings specific to device #0
retries = 5 # overrides default
```

And script:

```
import cfgparse,sys
c = cfgparse.ConfigParser()
f = c.add_file('set.ini')
r = c.add_option('retries',type='int',keys='DEV0')
t = c.add_option('timeout',type='int',keys='DEV0',
                 help='Time between retries in seconds.')
r.set(7)
t.set(20)
f.write(sys.stdout)
```

Results in:

```
$ python set1.py
[DEFAULT]
# this section applies to all devices
timeout = 20 # in seconds
retries = 3

[DEV0]
# this section if for settings specific to device #0
retries = 7 # overrides default
```

The `timeout` option in the `[DEFAULT]` section was modified. This is due to *cfgfile* not being specified. The option setting was located using the same methodology as the `get()` method and that option was modified. If this behavior is not desired, use the *cfgfile* argument.

For example:

```
import cfgparse,sys
c = cfgparse.ConfigParser()
f = c.add_file('set.ini')
r = c.add_option('retries',type='int',keys='DEV1',
                 help='Number of times to try again.')
t = c.add_option('timeout',type='int',keys='DEV0',
                 help='Time between retries in seconds.')
r.set(7,f)
t.set(20,f)
f.write(sys.stdout)
```

Results in:

```
$ python set2.py
[DEFAULT]
# this section applies to all devices
timeout = 10 # in seconds
retries = 3

[DEV0]
# this section if for settings specific to device #0
retries = 5 # overrides default

# Time between retries in seconds.
timeout = 20

[DEV1]

# Number of times to try again.
retries = 7
```

The above example also demonstrates the creation of new sections as necessary by modifying the `retries` option with the `DEV1` key.

4.2 set_option method

The `set_option()` method of the object returned from the `add_file()` method of the `ConfigParser` class can be used to modify an option setting in a specific configuration file. This method offers more direct control of which section the option to be set is located in. If the option is not in the section specified, the option will be added to that section. If multiple copies of the option exist in the desired section, all copies will be updated. This method does not return anything.

`set_option`(*name*, *value*[, *keys*][, *help*])

name is a required positional argument and is the name of the option to set the value of.

value is a required positional argument and is the new option value.

keys is an optional keyword argument. This key list identifies the configuration file section where option is located or to be added.

help is an optional string keyword argument. This string is placed ahead of an option setting if it is necessary to add the option to a section.

For example:

```
[DEFAULT]

# this section applies to all devices
timeout = 10 # in seconds
retries = 3


[DEV0]

# this section if for settings specific to device #0
retries = 5 # overrides default
```

And script:

```
import cfgparse,sys
c = cfgparse.ConfigParser()
f = c.add_file('set_option.ini')
f.set_option('retries',6)
f.set_option('retries',10,keys='DEV0')
f.set_option('retries',100,keys='DEV1',help='In new section')
f.write(sys.stdout)
```

Results in:

```
$ python set_option.py
[DEFAULT]

# this section applies to all devices
timeout = 10 # in seconds
retries = 6


[DEV0]

# this section if for settings specific to device #0
retries = 10 # overrides default


[DEV1]

# In new section
retries = 100
```

4.3 write method

A `write()` method can be used to write the reconstructed file contents with the new settings, options, and potentially sections. This write method is available in the file object that is returned by the `add_file()` method of the `ConfigParser` or the `set()` method of an option object.

`write`(*file*)

file is a required positional argument. If *file* is a file object, the reconstructed file contents will be written using the file object's `write()` method. Otherwise it is expected that *file* is a file name string and the file will be opened and written to.

The two previous sections show example uses of this method.

5 INI Syntax Summary

5.1 Comments

Comments in the configuration file should start with the '#' character and continue until the end of that line. Comments may be placed anywhere within the file including on same lines as section names or option settings.

Although not recommended, lines of text without an '=' sign are ignored and can be considered comments.

5.2 Option settings

Lines containing an '=' sign are interpretted to be option settings. The line is stripped of comments and split on the first '='. Both the option name and setting are stripped of leading and trailing white space.

Multiline settings may exist with the use of XML notation to encapsulate the setting. The `<b>` and `<v>` tags are available for this purpose. The `<b>` tag is short for "block" and is useful for most purposes. The `<v>` tag is short for "verbatim" and is the same as block except that text encapsulated is not subjected to the text substitution algorithms described in "Text Substitution" (section 5.5). The `<b>` and `<v>` tags may not be nested within themselves. `<v>` takes precedence over `<b>`.

Option names are case sensitive and may not contain the `[ ] =` characters nor use the ¡b¿ or ¡v¿ tags.

For example:

```
opt1 = simple # comment
opt2=simple # same

opt3 = <b> # not a comment (actually line 0 of setting)
Line 1 of a multi-line setting
Line 2 of a multi-line setting
</b> # comment can go here
```

5.3 Sections

A configuration file may contain one or more sections to store multiple settings of the same option. A section is started with a line that contains a section name enclosed in [ ] and all option settings that follow are part of that section until the next section is reached. If the file contains no section names all settings are deposited into the `[DEFAULT]` section. If options settings exist before the first section in a file, those option settings are a part of the `[DEFAULT]` section. Simple section names can be considered as keys into a dictionary.

For example:

```
[DEFAULT]
opt1 = v_default

[section1]
opt1 = v_section1
```

Can be thought of as:

```
opt1 = { ’DEFAULT’ : ’v_default’,
         ’section1’ : ’v_section1’ }
```

5.4 Section keys

Section names may consist of one or more keys. Commas and periods in a section name are used to split the section name into its component keys. If a period or a comma is not to be treated as a key split point, encapsulate the section components with single or double quotes. If one component is encapsulated, all must be encapsulated. Keys within the section name can be considered as keys into a dictionary.

For example:

```
path = 192.168.0.99

[rack0.dev0]
path = 192.168.0.0

[rack0.dev1]
path = 192.168.0.1

[’lab1.rack0’,’dev0’]
path = 192.168.0.2

[’lab1.rack0’,’dev1’]
path = 192.168.0.3
```

Can be thought of as:

```
path = { ’DEFAULT’ : ’192.168.0.99’,
         ’rack0’ : {
            ’dev0’ : ’192.168.0.0’,
            ’dev1’ : ’192.168.0.1’ },
         ’lab1.rack0’ : {
             ’dev0’ : ’192.168.0.2’,
             ’dev1’ : ’192.168.0.3’ } }
```

But syntax is available to extend the section keys with the section name notation immediately following the option name so that the above example could have been written as:

```
path = 192.168.0.99

[rack0]
path[dev0] = 192.168.0.0
path[dev1] = 192.168.0.1

['lab1.rack0']
path[dev0] = 192.168.0.2
path[dev1] = 192.168.0.3
```

5.5 Text substitution

Text in an option setting may include text from other option settings. String substitutions are implemented using Python's mapped key string formatting style. The name of the option to use to substitute text is containted within the string formatting code.

For example:

```
path = %(base)s\site-packages
base = C:\Python24\lib
```

Is equivalent to:

```
path = C:\Python24\lib\site-packages
```

The text substitution option must exist in the same section or in the `[DEFAULT]` section.

For example, this works:

```
[DEFAULT]
base = C:\Python24\lib
[SECTION1]
path = %(base)s\site-packages
```

But this does not:

```
[SECTION0]
base = C:\Python24\lib
[SECTION1]
path = %(base)s\site-packages
```

Nesting is only allowed up to 10 levels deep in order to prevent endless loops.

5.6 Including Files

Configuration files may include other configuration files. The format is the same as an option/setting pair except a special `<include>` option name is used.

For example:

```
# file1.ini
[DEFAULT]
<include> = file2.ini
retries = 3
```

```
# file2.ini
[DEFAULT]
timeout = 10
```

Is equivalent to:

```
# file1.ini
[DEFAULT]
timeout = 10
retries = 3
```

Inclusion of configuration files from within a section is the same as if the settings in the included file were defined within that section. If the included file contains sections, those section names are extended.

For example:

```
# system.ini
[RACK0]
<include> = rack0.ini
```

```
# rack0.ini
[DEFAULT]
desc = ’main rack’
[DEV0]
path = 192.168.0.0
[DEV1]
path = 192.168.0.1
```

Is equivalent to:

```
# system.ini
[RACK0]
desc = ’main rack’
[RACK0.DEV0]
path = 192.168.0.0
[RACK0.DEV1]
path = 192.168.0.1
```

The extended section key syntax may also be used with the `<include>` special option. For example, using rack0.ini from the last example:

```
# system.ini
[DEFAULT]
<include>[RACK0] = rack0.ini
```

Is equivalent to:

```
# system.ini
[RACK0]
desc = ’main rack’
path[DEV0] = 192.168.0.0
path[DEV1] = 192.168.0.1
```

Configuration files included within sections are not necessarily read immediately (or ever). The included files are added to a pending list and are only read if all the section keys that it is associated with are in the active key list for a particular option being requested when the `parse()` or `get()` methods are called.

5.7 Default keys

The `[DEFAULT]` section may contain a special `<keys>` option/setting pair that can be used to provide a default list of section keys to use to obtain options. `<keys>` found in sections other than `[DEFAULT]` are ignored.

For example:

```
# system.ini
[DEFAULT]
<keys> = RACK0,DEV1
desc = ’default description’
[RACK0]
desc = ’main rack’
path[DEV0] = 192.168.0.0
path[DEV1] = 192.168.0.1
```

Is equivalent to:

```
# system.ini
[DEFAULT]
path = 192.168.0.1
desc = ’main rack’
[RACK0]
desc = ’main rack’
path[DEV0] = 192.168.0.0
path[DEV1] = 192.168.0.1
```

This provides a convenient mechanism for the user to quickly make default selections without needing to reorganize or copy sections of the configuration file. The `<keys>` keys list may contain many keys and need not be an exact section name. Order is important but in the example above it could have been reversed with no ill effect. See ”Keys” (section 6) for more information.

5.8 Environment keys

The `[DEFAULT]` section may contain a special `<keys_variable>` option/setting pair that can be used to specify an environment variable name to be used to provide a default list of section keys to use to obtain options. `<keys_variable>` found in sections other than `[DEFAULT]` are ignored.

For example, setting the keys environment variable from the command line:

```
Linux: $ export keys=RACK0,DEV1
DOS:   C:\> set keys=RACK0,DEV1
```

Makes:

```
# system.ini
[DEFAULT]
<keys_variable> = keys
desc = ’default description’
[RACK0]
desc = ’main rack’
path[DEV0] = 192.168.0.0
path[DEV1] = 192.168.0.1
```

Equivalent to:

```
# system.ini
[DEFAULT]
path = 192.168.0.1
desc = ’main rack’
[RACK0]
desc = ’main rack’
path[DEV0] = 192.168.0.0
path[DEV1] = 192.168.0.1
```

6 Keys

When configuration files are read, all option settings are merged together in a master dictionary that is heirarchically organized using the section keys as keys in the dictionary. The ”Sections” (section 5.3) and ”Section keys” (section 5.4) examples may help visualize this concept. When duplicate options exist at the same level, either in the same file or different files, the last one in wins.

When the `parse()` method is called or when an option `get()` method is called a list of keys is constructed to walk through the dictionary of settings. The key list is constructed in the following order with the keys with top priority listed first

- `get()` *keys* argument (see section 2.5 ”Parsing”)
- `add_option()` *keys* argument (see section 2.3 ”Adding options”)
- command line keys (see section 3 ”Command line cooperation”)
- environment keys (see section 5.8 ”Environment keys”)

- configuration file keys (see section 5.7 ”Default keys”)
- `’DEFAULT’` key

The algorithm for obtaining a setting from the dictionary of heirarchically organized settings is:

```
start at top of option dictionary
start at top of key list  <-----------------------
get next key <-------------------------------     |
    no keys left) DONE - no setting found    |    |
is key in dictionary?                        |    |
    no)  ------------------------------------     |
    yes) is setting a dictionary?                 |
        yes) proceed to next level  --------------
        no) DONE - setting found
```

This implementation allows the user to store multiple settings in the configuration file and provides great flexibility for controlling which settings get used.

7 Python configuration files

Python based configuration files may be used in place of or in combination with `ini` configuration files. Since this may introduce a security hole in your application, this feature must be enabled using the *allow_py* argument when creating an instance of the `ConfigParser`.

Use of Python to construct option settings allows the greatest flexibility but also requires a bit more sophisticated user as the configuration file is truly a Python script and is executed. One significant advantage of a Python based configuration file is that settings may be Python objects. For example in an application where files need to be located it may be tempting to ask the user to provide a string which contains a list of paths to scan. What happens if the user wants a set of text files that contains lists of files (such as Visual Studio project files) scanned instead? It would be better to provide a base class that is your best guess as to what most users need. Most users could then make the settting an instance of that class passing in the string of paths into the constructor. Advanced users can would be able to subclass it, maintaining the interface, but changing the implementation to their particular needs.

Another significant advantage is that users can construct settings on the fly with the full power and flexibility of Python. There are no special text substitution techniques like the `ini` syntax since the user can use Python to do it.

Default Keys

`KEYS` is used instead of `<keys>`. For example:

```
KEYS = ’RACK0,DEV0’
```

Ie equivalent to:

```
[DEFAULT]
<keys> = RACK0,DEV0
```

Environment keys

`KEYS_VARIABLE` is used instead of `<keys_variable>`. For example:

```
KEYS_VARIABLE = ’keys’
```

Ie equivalent to:

```
[DEFAULT]
<keys_variable> = keys
```

Includes

`CONFIG_FILES` is used instead of `<include>`. For example:

```
CONFIG_FILES = ’file1.ini’
```

Is equivalent to:

```
[DEFAULT]
<include> = file1.ini
```

And:

```
CONFIG_FILES = [’file2.ini’,’file3.py’]
```

Is equivalent to:

```
[DEFAULT]
<include> = file2.ini
<include> = file3.ini
```

And:

```
CONFIG_FILES = { ’DEFAULT’ : ’rack0.ini’,
                 ’RACK1’ : ’rack1.ini’,
                 ’RACK2’ : ’rack2.ini’ }
```

Is equivalent to:

```
[DEFAULT]
<include> = rack0.ini
[RACK1]
<include> = rack1.ini
[RACK2]
<include> = rack2.ini
```

Settings

After execution of the configuration file all remaining options found that do not begin with an underscore (_) are considered to be an option setting. To avoid polluting the settings dictionary the configuration files should either clean up after itself or name intermediate variables with an underscore. For example:

```
# import with leading underscore to avoid cleanup later
import os as _os

for x in range(10):
    pass # do something
del x # clean up so don’t need leading _
```

```
driver = ’ethernet’
path = { ’DEFAULT’ : ’192.168.0.99’,
         ’DEV0’    : ’192.168.0.0’,
         ’DEV1’    : ’192.168.0.1’ }
```

Ie equivalent to:

```
[DEFAULT]
driver = ’ethernet’
path = 192.168.0.99
[DEV0]
path = 192.168.0.0
[DEV1]
path = 192.168.0.1
```

8 misc

8.1 Printing help

The `print_help` method can automatically generate help information for all the added options just like the `optparse` command line parser `print_help` method.

`print_help``(`[*file*]`)`

file is an optional keyword argument. If omitted, the generated help will be written to `sys.stdout`. Otherwise *file* is a expected to be a file object and the generated help will be written using the file object’s `write()` method.

8.2 Help Notes

The `add_note` method of the `ConfigParser` may be used to append preformatted text to the help text.

`add_note``(`*note*`)`

note is a positional string argument and is the preformatted text to be appended to the end of the generated help text.

The `add_note` method of instances of an option object returned by the `add_option` method (same prototype) performs the same action but also causes the preformatted text to be printed if an error occurs when option is retrieved. Also the preformatted text is inserted as a comment above option settings added to configuration files.

8.3 History

The roots of this module began with a different configuration parser written by the same author (http://sourceforge.net/projects/config-py). This module contained some of the core design blending option settings from multiple files and the use of section keys for flexible option storage and retrieval. Since that time a need for coordination between the command line and configuration file option parsers caused an evolution of the interface to be modelled after the `optparse` module. Perhaps modelled is a bit weak of an acknowledgment because all of the help formatting code was borrowed almost verbatim from that module. The `ini` syntax was added so that the module would be a viable candidate in the ConfigParserShootout activity (http://www.python.org/moin/ConfigParserShootout) which hopes to produce a new configuration parser module for inclusion in the standard library of a future Python release.