ConfigParser Learning Notes

Thursday, April 10, 2014 09:36

Reference URL: http://pymotw.com/2/ConfigParser/

1. **Configuration File Format**
   The file format used by ConfigParser is similar to the format used by older versions of Microsoft Windows. It consists of one or more named sections, each of which can contain individual options with names and values.

   Config file sections are identified by looking for lines starting with [and ending with]. The value between the square brackets is the section name, and can contain any characters except square brackets.

   Options are listed one per line within a section. The line starts with the name of the option, which is separated from the value by a colon (:) or equal sign (=). Whitespace around the separator is ignored when the file is parsed.

   A sample configuration file with section “bug_tracker” and three options would look like:

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET
```

2. **Reading Configuration Files**
   The most common use for a configuration file is to have a user or system administrator edit the file with a regular text editor to set application behavior defaults, and then have the application read the file, parse it, and act based on its contents. Use the read() method of SafeConfigParser to read the configuration file.

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('simple.ini')

print parser.get('bug_tracker', 'url')
```

   This program reads the simple.ini file from the previous section and prints the value of the url option from the bug_tracker section.

```
$ python ConfigParser_read.py

http://localhost:8080/bugs/
```

   The read() method also accepts a list of filenames. Each name in turn is scanned, and if the file exists it is opened and read.

```
from ConfigParser import SafeConfigParser
import glob

parser = SafeConfigParser()

candidates = ['does_not_exist.ini', 'also-does-not-exist.ini', 'simple.ini', 'multisection.ini']

found = parser.read(candidates)

missing = set(candidates) - set(found)

print 'Found config files:', sorted(found)
print 'Missing files     :', sorted(missing)
```

   read() returns a list containing the names of the files successfully loaded, so the program can discover which configuration files are missing and decide whether to ignore them.

```
$ python ConfigParser_read_many.py

Found config files: ['multisection.ini', 'simple.ini']
Missing files     : ['also-does-not-exist.ini', 'does_not_exist.ini']
```

3. **Unicode Configuration Data**
   Configuration files containing Unicode data should be opened using the codecs module to set the proper encoding value.

   Changing the password value of the original input to contain Unicode characters and saving the results in UTF-8 encoding gives:

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = ßéç®é†
```

   The codecs file handle can be passed to readfp(), which uses the readline() method of its argument to get lines from the file and parse them.

```
from ConfigParser import SafeConfigParser
import codecs

parser = SafeConfigParser()

# Open the file with the correct encoding
with codecs.open('unicode.ini', 'r', encoding='utf-8') as f:
    parser.readfp(f)

password = parser.get('bug_tracker', 'password')

print 'Password:', password.encode('utf-8')
print 'Type    :', type(password)
print 'repr()  :', repr(password)
```

   The value returned by get() is a unicode object, so in order to print it safely it must be re-encoded as UTF-8.

```
$ python ConfigParser_unicode.py

Password: ßéç®é†
Type    : <type 'unicode'>
repr()  : u'\xdf\xe9\xe7\xae\xe9\u2020'
```

4. **Accessing Configuration Settings**

SafeConfigParser includes methods for examining the structure of the parsed configuration, including listing the sections and options, and getting their values. This configuration file includes two sections for separate web services:

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET

[wiki]
url = http://localhost:8080/wiki/
username = dhellmann
password = SECRET
```

And this sample program exercies some of the methods for looking at the configuration data, including sections(), options(), and items().

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

for section_name in parser.sections():
    print 'Section:', section_name
    print '  Options:', parser.options(section_name)
    for name, value in parser.items(section_name):
        print '  %s = %s' % (name, value)
    print
```

Both sections() and options() return lists of strings, while items() returns a list of tuples containing the name-value pairs.

```
$ python ConfigParser_structure.py

Section: bug_tracker
  Options: ['url', 'username', 'password']
  url = http://localhost:8080/bugs/
  username = dhellmann
  password = SECRET

Section: wiki
  Options: ['url', 'username', 'password']
  url = http://localhost:8080/wiki/
  username = dhellmann
  password = SECRET
```

5. **Testing whether values are present**

To test if a section exists, use has_section(), passing the section name.

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

for candidate in [ 'wiki', 'bug_tracker', 'dvcs' ]:
    print '%-12s: %s' % (candidate, parser.has_section(candidate))
```

Testing if a section exists before calling get() avoids exceptions for missing data.

```
$ python ConfigParser_has_section.py

wiki       : True
bug_tracker : True
dvcs       : False
```

Use has_option() to test if an option exists within a section.

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

for section in [ 'wiki', 'none' ]:
    print '%s section exists: %s' % (section, parser.has_section(section))
    for candidate in [ 'username', 'password', 'url', 'description' ]:
        print '%s.%-12s  : %s' % (section, candidate, parser.has_option(section, candidate))
    print
```

If the section does not exist, has_option() returns False.

```
$ python ConfigParser_has_option.py

wiki section exists: True
wiki.username     : True
wiki.password     : True
wiki.url          : True
wiki.description  : False
```

```
none section exists: False
none.username      : False
none.password      : False
none.url           : False
none.description   : False
```

6. **Value Types**

All section and option names are treated as strings, but option values can be strings, integers, floating point numbers, or booleans. There are a range of possible boolean values that are converted true or false. This example file includes one of each:

```
[ints]
positive = 1
negative = -5

[floats]
positive = 0.2
negative = -3.14

[booleans]
number_true = 1
number_false = 0
yn_true = yes
yn_false = no
tf_true = true
tf_false = false
onoff_true = on
onoff_false = false
```

SafeConfigParser does not make any attempt to understand the option type. The application is expected to use the correct method to fetch the value as the desired type. get() always returns a string. Use getint() for integers, getfloat() for floating point numbers, and getboolean() for boolean values.

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('types.ini')

print 'Integers:'
for name in parser.options('ints'):
    string_value = parser.get('ints', name)
    value = parser.getint('ints', name)
    print '  %-12s : %-7r -> %d' % (name, string_value, value)

print '\nFloats:'
for name in parser.options('floats'):
    string_value = parser.get('floats', name)
    value = parser.getfloat('floats', name)
    print '  %-12s : %-7r -> %0.2f' % (name, string_value, value)

print '\nBooleans:'
for name in parser.options('booleans'):
    string_value = parser.get('booleans', name)
    value = parser.getboolean('booleans', name)
    print '  %-12s : %-7r -> %s' % (name, string_value, value)
```

Running this program with the example input produces:

```
$ python ConfigParser_value_types.py

Integers:
  positive     : '1'     -> 1
  negative     : '-5'    -> -5

Floats:
  positive     : '0.2'   -> 0.20
  negative     : '-3.14' -> -3.14

Booleans:
  number_true  : '1'     -> True
  number_false : '0'     -> False
  yn_true      : 'yes'   -> True
  yn_false     : 'no'    -> False
  tf_true      : 'true'  -> True
  tf_false     : 'false' -> False
  onoff_true   : 'on'    -> True
  onoff_false  : 'false' -> False
```

7. **Options as Flags**

Usually the parser requires an explicit value for each option, but with the SafeConfigParser parameter allow_no_value set to True an option can appear by itself on a line in the input file, and be used as a flag.

```
import ConfigParser

# Requre values
try:
    parser = ConfigParser.SafeConfigParser()
    parser.read('allow_no_value.ini')
except ConfigParser.ParsingError, err:
    print 'Could not parse:', err

# Allow stand-alone option names
```

```
print '\nTrying again with allow_no_value=True'
parser = ConfigParser.SafeConfigParser(allow_no_value=True)
parser.read('allow_no_value.ini')
for flag in [ 'turn_feature_on', 'turn_other_feature_on' ]:
    print
    print flag
    exists = parser.has_option('flags', flag)
    print '  has_option:', exists
    if exists:
        print '       get:', parser.get('flags', flag)
```

When an option has no explicit value, has_option() reports that the option exists and get() returns None.

```
$ python ConfigParser_allow_no_value.py

Could not parse: File contains parsing errors: allow_no_value.ini
       [line  2]: 'turn_feature_on\n'

Trying again with allow_no_value=True

turn_feature_on
  has_option: True
       get: None

turn_other_feature_on
  has_option: False
```

8. **Modifying Settings**

While SafeConfigParser is primarily intended to be configured by reading settings from files, settings can also be populated by calling add_section() to create a new section, and set() to add or change an option.

```
import ConfigParser

parser = ConfigParser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://localhost:8080/bugs')
parser.set('bug_tracker', 'username', 'dhellmann')
parser.set('bug_tracker', 'password', 'secret')

for section in parser.sections():
    print section
    for name, value in parser.items(section):
        print '  %s = %r' % (name, value)
```

All options must be set as strings, even if they will be retrieved as integer, float, or boolean values.

```
$ python ConfigParser_populate.py

bug_tracker
  url = 'http://localhost:8080/bugs'
  username = 'dhellmann'
  password = 'secret'
```

Sections and options can be removed from a SafeConfigParser with remove_section() and remove_option().

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('multisection.ini')

print 'Read values:\n'
for section in parser.sections():
    print section
    for name, value in parser.items(section):
        print '  %s = %r' % (name, value)

parser.remove_option('bug_tracker', 'password')
parser.remove_section('wiki')

print '\nModified values:\n'
for section in parser.sections():
    print section
    for name, value in parser.items(section):
        print '  %s = %r' % (name, value)
```

Removing a section deletes any options it contains.

```
$ python ConfigParser_remove.py

Read values:

bug_tracker
  url = 'http://localhost:8080/bugs/'
  username = 'dhellmann'
  password = 'SECRET'
wiki
  url = 'http://localhost:8080/wiki/'
  username = 'dhellmann'
  password = 'SECRET'
```

```
Modified values:

bug_tracker
  url = 'http://localhost:8080/bugs/'
  username = 'dhellmann'
```

9. **Saving Configuration Files**
   Once a SafeConfigParser is populated with desired data, it can be saved to a file by calling the write() method. This makes it possible to provide a user interface for editing the configuration settings, without having to write any code to manage the file.

```
import ConfigParser
import sys

parser = ConfigParser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://localhost:8080/bugs')
parser.set('bug_tracker', 'username', 'dhellmann')
parser.set('bug_tracker', 'password', 'secret')

parser.write(sys.stdout)
```

   The write() method takes a file-like object as argument. It writes the data out in the INI format so it can be parsed again by SafeConfigParser.

```
$ python ConfigParser_write.py

[bug_tracker]
url = http://localhost:8080/bugs
username = dhellmann
password = secret
```

10. **Option Search Path**
    SafeConfigParser uses a multi-step search process when looking for an option.
    Before starting the option search, the section name is tested. If the section does not exist, and the name is not the special value DEFAULT, then NoSectionError is raised.
    a. If the option name appears in the vars dictionary passed to get(), the value from vars is returned.
    b. If the option name appears in the specified section, the value from that section is returned.
    c. If the option name appears in the DEFAULT section, that value is returned.
    d. If the option name appears in the defaults dictionary passed to the constructor, that value is returned.

    If the name is not found in any of those locations, NoOptionError is raised.

    The search path behavior can be demonstrated using this configuration file:

```
[DEFAULT]
file-only = value from DEFAULT section
init-and-file = value from DEFAULT section
from-section = value from DEFAULT section
from-vars = value from DEFAULT section

[sect]
section-only = value from section in file
from-section = value from section in file
from-vars = value from section in file
```

    and this test program:

```
import ConfigParser

# Define the names of the options
option_names =  [
   'from-default',
   'from-section', 'section-only',
   'file-only', 'init-only', 'init-and-file',
   'from-vars',
   ]

# Initialize the parser with some defaults
parser = ConfigParser.SafeConfigParser(
   defaults={'from-default':'value from defaults passed to init',
             'init-only':'value from defaults passed to init',
             'init-and-file':'value from defaults passed to init',
             'from-section':'value from defaults passed to init',
             'from-vars':'value from defaults passed to init',
             })

print 'Defaults before loading file:'
defaults = parser.defaults()
for name in option_names:
   if name in defaults:
      print '  %-15s = %r' % (name, defaults[name])

# Load the configuration file
parser.read('with-defaults.ini')

print '\nDefaults after loading file:'
defaults = parser.defaults()
for name in option_names:
   if name in defaults:
      print '  %-15s = %r' % (name, defaults[name])
```

```
# Define some local overrides
vars = {'from-vars':'value from vars'}

# Show the values of all of the options
print '\nOption lookup:'
for name in option_names:
    value = parser.get('sect', name, vars=vars)
    print '  %-15s = %r' % (name, value)

# Show error messages for options that do not exist
print '\nError cases:'
try:
    print 'No such option :', parser.get('sect', 'no-option')
except ConfigParser.NoOptionError, err:
    print str(err)

try:
    print 'No such section:', parser.get('no-sect', 'no-option')
except ConfigParser.NoSectionError, err:
    print str(err)
```

The output shows the origin for the value of each option, and illustrates the way defaults from different sources override existing values.

```
$ python ConfigParser_defaults.py

Defaults before loading file:
  from-default    = 'value from defaults passed to init'
  from-section    = 'value from defaults passed to init'
  init-only       = 'value from defaults passed to init'
  init-and-file   = 'value from defaults passed to init'
  from-vars       = 'value from defaults passed to init'

Defaults after loading file:
  from-default    = 'value from defaults passed to init'
  from-section    = 'value from DEFAULT section'
  file-only       = 'value from DEFAULT section'
  init-only       = 'value from defaults passed to init'
  init-and-file   = 'value from DEFAULT section'
  from-vars       = 'value from DEFAULT section'

Option lookup:
  from-default    = 'value from defaults passed to init'
  from-section    = 'value from section in file'
  section-only    = 'value from section in file'
  file-only       = 'value from DEFAULT section'
  init-only       = 'value from defaults passed to init'
  init-and-file   = 'value from DEFAULT section'
  from-vars       = 'value from vars'

Error cases:
No such option : No option 'no-option' in section: 'sect'
No such section: No section: 'no-sect'
```

11. **Combining Values with Interpolation**

SafeConfigParser provides a feature called interpolation that can be used to combine values together. Values containing standard Python format strings trigger the interpolation feature when they are retrieved with get(). Options named within the value being fetched are replaced with their values in turn, until no more substitution is necessary.

The URL examples from earlier in this section can be rewritten to use interpolation to make it easier to change only part of the value. For example, this configuration file separates the protocol, hostname, and port from the URL as separate options.

```
[bug_tracker]
protocol = http
server = localhost
port = 8080
url = %(protocol)s://%(server)s:%(port)s/bugs/
username = dhellmann
password = SECRET
```

Interpolation is performed by default each time get() is called. Pass a true value in the raw argument to retrieve the original value, without interpolation.

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('interpolation.ini')

print 'Original value      :', parser.get('bug_tracker', 'url')

parser.set('bug_tracker', 'port', '9090')
print 'Altered port value  :', parser.get('bug_tracker', 'url')

print 'Without interpolation:', parser.get('bug_tracker', 'url', raw=True)
```

Because the value is computed by get(), changing one of the settings being used by the url value changes the return value.

```
$ python ConfigParser_interpolation.py

Original value      : http://localhost:8080/bugs/
Altered port value  : http://localhost:9090/bugs/
Without interpolation: %(protocol)s://%(server)s:%(port)s/bugs/
```

12. **Using Defaults**

Values for interpolation do not need to appear in the same section as the original option. Defaults can be mixed with override values. Using this config file:

```
[DEFAULT]
url = %(protocol)s://%(server)s:%(port)s/bugs/
protocol = http
server = bugs.example.com
port = 80

[bug_tracker]
server = localhost
port = 8080
username = dhellmann
password = SECRET
```

The url value comes from the DEFAULT section, and the substitution starts by looking in bug_tracker and falling back to DEFAULT for pieces not found.

```
from ConfigParser import SafeConfigParser

parser = SafeConfigParser()
parser.read('interpolation_defaults.ini')

print 'URL:', parser.get('bug_tracker', 'url')
```

The hostname and port values come from the bug_tracker section, but the protocol comes from DEFAULT.

```
$ python ConfigParser_interpolation_defaults.py

URL: http://localhost:8080/bugs/
```

13. **Substitution Errors**

Substitution stops after MAX_INTERPOLATION_DEPTH steps to avoid problems due to recursive references.

```
import ConfigParser

parser = ConfigParser.SafeConfigParser()

parser.add_section('sect')
parser.set('sect', 'opt', '%(opt)s')

try:
   print parser.get('sect', 'opt')
except ConfigParser.InterpolationDepthError, err:
   print 'ERROR:', err
```

An InterpolationDepthError exception is raised if there are too many substitution steps.

```
$ python ConfigParser_interpolation_recursion.py

ERROR: Value interpolation too deeply recursive:
      section: [sect]
      option : opt
      rawval : %(opt)s
```

Missing values result in an InterpolationMissingOptionError exception.

```
import ConfigParser

parser = ConfigParser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://%(server)s:%(port)s/bugs')

try:
   print parser.get('bug_tracker', 'url')
except ConfigParser.InterpolationMissingOptionError, err:
   print 'ERROR:', err
```

Since no server value is defined, the url cannot be constructed.

```
$ python ConfigParser_interpolation_error.py

ERROR: Bad value substitution:
      section: [bug_tracker]
      option : url
      key    : server
      rawval : :%(port)s/bugs
```