configparser — Work with Configuration Files

Purpose: Read/write configuration files similar to Windows INI files

Use the `configparser` module to manage user-editable configuration files for an application. The contents of the configuration files can be organized into groups and several option value types are supported, including integers, floating point values, and Booleans. Option values can be combined using Python formatting strings, to build longer values such as URLs from shorter values like host names and port numbers.

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.

Lines starting with semi-colon (`;`) or octothorpe (`#`) are treated as comments and not visible when accessing the contents of the configuration file programmatically.

This sample configuration file has a section named `bug_tracker` with three options, `url`, `username`, and `password`.

```
# This is a simple example with comments.
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
; You should not store passwords in plain text
; configuration files.
password = SECRET
```

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 `ConfigParser` to read the configuration file.

```
# configparser_read.py

from configparser import ConfigParser

parser = ConfigParser()
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.

```
$ python3 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.

```
# configparser_read_many.py

from configparser import ConfigParser
import glob
```

```
parser = ConfigParser()

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 or treat the condition as an error.

```
$ python3 configparser_read_many.py

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

Unicode Configuration Data

Configuration files containing Unicode data should be read using the proper encoding value. The following example file changes the password value of the original input to contain Unicode characters and is encoded using UTF-8.

```
# unicode.ini

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

The file is opened with the appropriate decoder, converting the UTF-8 data to native Unicode strings.

```
# configparser_unicode.py

from configparser import ConfigParser
import codecs

parser = ConfigParser()
# Open the file with the correct encoding
parser.read('unicode.ini', encoding='utf-8')

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 string, so in order to print it safely it must be re-encoded as UTF-8.

```
$ python3 configparser_unicode.py

Password: b'\xc3\x9f\xc3\xa9\xc3\xa7\xc2\xae\xc3\xa9\xe2\x80\xa0
'
Type    : <class 'str'>
repr()  : 'ßéç®é†'
```

Accessing Configuration Settings

`ConfigParser` 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 exercises some of the methods for looking at the configuration data, including `sections()`, `options()`, and `items()`.

```
# configparser_structure.py

from configparser import ConfigParser

parser = ConfigParser()
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('  {} = {}'.format(name, value))
    print()
```

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

```
$ python3 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
```

A `ConfigParser` also supports the same mapping API as *dict*, with the `ConfigParser` acting as one dictionary containing separate dictionaries for each section.

```
# configparser_structure_dict.py

from configparser import ConfigParser

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

for section_name in parser:
    print('Section:', section_name)
    section = parser[section_name]
    print('  Options:', list(section.keys()))
    for name in section:
        print('  {} = {}'.format(name, section[name]))
    print()
```

Using the mapping API to access the same configuration file produces the same output.

```
$ python3 configparser_structure_dict.py

Section: DEFAULT
  Options: []

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
```

Testing Whether Values Are Present

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

```python
# configparser_has_section.py

from configparser import ConfigParser

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

for candidate in ['wiki', 'bug_tracker', 'dvcs']:
    print('{:<12}: {}'.format(
        candidate, parser.has_section(candidate)))
```

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

```
$ python3 configparser_has_section.py

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

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

```python
# configparser_has_option.py

from configparser import ConfigParser

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

SECTIONS = ['wiki', 'none']
OPTIONS = ['username', 'password', 'url', 'description']

for section in SECTIONS:
    has_section = parser.has_section(section)
    print('{} section exists: {}'.format(section, has_section))
    for candidate in OPTIONS:
        has_option = parser.has_option(section, candidate)
        print('{}.{:<12}  : {}'.format(
            section, candidate, has_option))
    print()
```

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

```
$ python3 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
```

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. The following example file includes one of each.

```
# types.ini

[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
```

`ConfigParser` 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.

```
# configparser_value_types.py

from configparser import ConfigParser

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

print('Integers:')
for name in parser.options('ints'):
    string_value = parser.get('ints', name)
    value = parser.getint('ints', name)
    print('  {:<12} : {!r:<7} -> {}'.format(
        name, string_value, value))

print('\nFloats:')
for name in parser.options('floats'):
    string_value = parser.get('floats', name)
    value = parser.getfloat('floats', name)
    print('  {:<12} : {!r:<7} -> {:0.2f}'.format(
        name, string_value, value))

print('\nBooleans:')
for name in parser.options('booleans'):
    string_value = parser.get('booleans', name)
    value = parser.getboolean('booleans', name)
    print('  {:<12} : {!r:<7} -> {}'.format(
        name, string_value, value))
```

Running this program with the example input produces the following output.

```
$ python3 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
```

Custom type converters can be added by passing conversion functions in the `converters` argument to `ConfigParser`. Each converter receives a single input value, and should transform that value into the appropriate return type.

```python
# configparser_custom_types.py

from configparser import ConfigParser
import datetime


def parse_iso_datetime(s):
    print('parse_iso_datetime({!r})'.format(s))
    return datetime.datetime.strptime(s, '%Y-%m-%dT%H:%M:%S.%f')


parser = ConfigParser(
    converters={
        'datetime': parse_iso_datetime,
    }
)
parser.read('custom_types.ini')

string_value = parser['datetimes']['due_date']
value = parser.getdatetime('datetimes', 'due_date')
print('due_date : {!r} -> {!r}'.format(string_value, value))
```

Adding a converter causes `ConfigParser` to automatically create a retrieval method for that type, using the name of the type as specified in `converters`. In this example, the `'datetime'` converter causes a new `getdatetime()` method to be added.

```
$ python3 configparser_custom_types.py

parse_iso_datetime('2015-11-08T11:30:05.905898')
due_date : '2015-11-08T11:30:05.905898' -> datetime.datetime(201
5, 11, 8, 11, 30, 5, 905898)
```

It is also possible to add converter methods directly to a subclass of `ConfigParser`.

Options as Flags

Usually, the parser requires an explicit value for each option, but with the `ConfigParser` 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.

```python
# configparser_allow_no_value.py

import configparser

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

# Allow stand-alone option names
print('\nTrying again with allow_no_value=True')
parser = configparser.ConfigParser(allow_no_value=True)
parser.read('allow_no_value.ini')
for flag in ['turn_feature_on', 'turn_other_feature_on']:
    print('\n', 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.

```
$ python3 configparser_allow_no_value.py

Could not parse: Source 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
```

Multi-line Strings

String values can span multiple lines, if subsequent lines are indented.

```
[example]
message = This is a multi-line string.
  With two paragraphs.

  They are separated by a completely empty line.
```

Within the indented multi-line values, blank lines are treated as part of the value and preserved.

```
$ python3 configparser_multiline.py

This is a multi-line string.
With two paragraphs.

They are separated by a completely empty line.
```

Modifying Settings

While ConfigParser 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.

```
# configparser_populate.py

import configparser

parser = configparser.ConfigParser()

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('  {} = {!r}'.format(name, value))
```

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

```
$ python3 configparser_populate.py

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

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

```
# configparser_remove.py

from configparser import ConfigParser

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

print('Read values:\n')
for section in parser.sections():
    print(section)
    for name, value in parser.items(section):
        print('  {} = {!r}'.format(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('  {} = {!r}'.format(name, value))
```

Removing a section deletes any options it contains.

```
$ python3 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'
```

Saving Configuration Files

Once a `ConfigParser` 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.

```
# configparser_write.py

import configparser
import sys

parser = configparser.ConfigParser()

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 `ConfigParser`.

```
$ python3 configparser_write.py

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

Warning

Comments in the original configuration file are not preserved when reading, modifying, and re-writing a configuration file.

Option Search Path

`ConfigParser` 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.

1. If the option name appears in the `vars` dictionary passed to `get()`, the value from `vars` is returned.
2. If the option name appears in the specified section, the value from that section is returned.
3. If the option name appears in the `DEFAULT` section, that value is returned.
4. 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
```

This test program includes default settings for options not specified in the configuration file, and overrides some values that are defined in the file.

```
# configparser_defaults.py

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
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',
}
parser = configparser.ConfigParser(defaults=DEFAULTS)

print('Defaults before loading file:')
defaults = parser.defaults()
for name in option_names:
    if name in defaults:
```

```
    if name in defaults:
        print('  {:<15} = {!r}'.format(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('  {:<15} = {!r}'.format(name, defaults[name]))

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

# Show the values of all the options
print('\nOption lookup:')
for name in option_names:
    value = parser.get('sect', name, vars=vars)
    print('  {:<15} = {!r}'.format(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 as err:
    print(err)

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

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

```
$ python3 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 option 'no-option' in section: 'sect'
No section: 'no-sect'
```

Combining Values with Interpolation

`ConfigParser` 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. Options named within the value being fetched are replaced with their values in turn, until no more substitution is necessary.

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.

```
# configparser_interpolation.py

from configparser import ConfigParser

parser = ConfigParser()
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.

```
$ python3 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/
```

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.

```
[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
```

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

```
# configparser_interpolation_defaults.py

from configparser import ConfigParser

parser = ConfigParser()
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.

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

```
$ python3 configparser_interpolation_defaults.py

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

Substitution Errors

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

```
# configparser_interpolation_recursion.py

import configparser

parser = configparser.ConfigParser()

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

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

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

```
$ python3 configparser_interpolation_recursion.py

ERROR: Recursion limit exceeded in value substitution: option 'o
pt' in section 'sect' contains an interpolation key which cannot
 be substituted in 10 steps. Raw value: '%(opt)s'
```

Missing values result in an InterpolationMissingOptionError exception.

```
# configparser_interpolation_error.py

import configparser

parser = configparser.ConfigParser()

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 as err:
    print('ERROR:', err)
```

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

```
$ python3 configparser_interpolation_error.py

ERROR: Bad value substitution: option 'url' in section
'bug_tracker' contains an interpolation key 'server' which is
not a valid option name. Raw value:
'http://%(server)s:%(port)s/bugs'
```

Escaping Special Characters

Since % starts the interpolation instructions, a literal % in a value must be escaped as %%.

```
[escape]
value = a literal %% must be escaped
```

Reading the value does not require any special consideration.

```
# configparser_escape.py
```

```
from configparser import ConfigParser
import os

filename = 'escape.ini'
config = ConfigParser()
config.read([filename])

value = config.get('escape', 'value')

print(value)
```

When the value is read, the `%%` is converted to `%` automatically.

```
$ python3 configparser_escape.py

a literal % must be escaped
```

Extended Interpolation

`ConfigParser` supports alternate interpolation implementations Passing an object that supports the API defined by `Interpolation` to the `interpolation` parameter. For example, using `ExtendedInterpolation` instead of the default `BasicInterpolation` enables a different syntax using `${}` to indicate variables.

```
# configparser_extendedinterpolation.py

from configparser import ConfigParser, ExtendedInterpolation

parser = ConfigParser(interpolation=ExtendedInterpolation())
parser.read('extended_interpolation.ini')

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

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

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

Extended interpolation supports accessing values from other sections of the configuration file by prefixing the variable name with the section name and a colon (`:`).

```
[intranet]
server = localhost
port = 8080

[bug_tracker]
url = http://${intranet:server}:${intranet:port}/bugs/
username = dhellmann
password = SECRET
```

Referring to values in other sections of the file makes it possible to share a hierarchy of values, without placing all defaults in the `DEFAULTS` section.

```
$ python3 configparser_extendedinterpolation.py

Original value       : http://localhost:8080/bugs/
Altered port value   : http://localhost:9090/bugs/
Without interpolation: http://${intranet:server}:${intranet:port
}/bugs/
```

Disabling Interpolation

To disable interpolation, pass None instead of an `Interpolation` object.

```
# configparser_nointerpolation.py
```

```
from configparser import ConfigParser

parser = ConfigParser(interpolation=None)
parser.read('interpolation.ini')

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

This enables any syntax that might have been processed by the interpolation object to be safely ignored.

```
$ python3 configparser_nointerpolation.py

Without interpolation: %(protocol)s://%(server)s:%(port)s/bugs/
```

See also

- Standard library documentation for configparser
- ConfigObj – An advanced configuration file parser with support for features like content validation.
- Python 2 to 3 porting notes for configparser

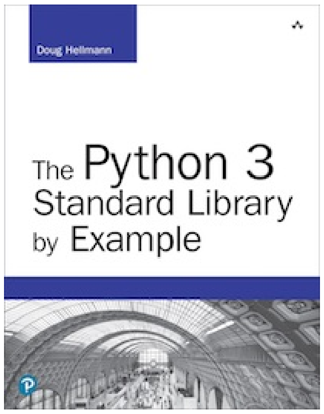
shlex — Parse Shell-style Syntaxes   logging — Report Status, Error, and Informational Messages

Quick Links

This page was last updated 2016-12-30.

Navigation

shlex — Parse Shell-style Syntaxes
logging — Report Status, Error, and Informational Messages



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?