Confloader Documentation

Release 1.1a1

Outernet Inc

July 06, 2016

Contents

1 Source code **3**

2 Documentation **5**
- 2.1 Writing .ini files . . . 5
- 2.2 Working with configuration files . . . 8
- 2.3 API documentation . . . 9

3 Indices and tables **13**

Python Module Index **15**

Confloader was developed to make handling configuration in .ini format easier. While Python standard library offers the framework for creating an .ini parser that is fine-tuned to your needs, Confloader developers found that rewriting the parser each time was tedious, and the basic tools provided by the standard library insufficient for repeated use.

While Confloader may be as flexible as the `ConfigParser` suite from the standard library, it has a growing number of features to cover majority of scenarios that applications may encounter, and offers facilities of combining and managing collections of configuration file fragments.

CHAPTER 1

Source code

Confloader source code can be found on GitHub and is released under BSD license. See the `LICENSE` file in the source tree for more information.

CHAPTER 2

Documentation

2.1 Writing .ini files

The .ini format that is used by Confloader is more or less the same as the one used by Python's standard library ConfigParser module.

2.1.1 Sections

The configuration file consists of sections which start with the section header in `[name]` format. Section naming is arbitrary and completely up to the user, but there are two special sections, `[global]` and `[config]` which have special handling in Confloader.

2.1.2 Options

The configuration options are specified using `key=value` format. Leading whitespace, whitespace around the equals sign, and whitespace around the value are ignored. A simple value may look like this:

```
foo = bar
```

Values can span multiple lines. Unlike `ConfigParser`, multi-line values have special meaning to Confloader, which we will discuss later. At it's simplest, multiline value may look like this:

```
foo = Long value that
      spans multiple lines.
```

Note that leading whitespace in the second line is completely ignored.

2.1.3 Data types

Values that *appear* to be of some type will be coerced to that type. Currently, coercion is supported for the following types:

- integer
- float
- byte size
- boolean
- null/None

- list

Numeric values

If the value is strictly numeric, it will be treated as an integer or a float. Presence of the decimal dot determines the actual type used. For instance:

```
foo = 12   # becomes int(12)
bar = 1.2  # becomes float(1.2)
```

Negative numbers are also supported with a − prefix. There cannot be any whitespace between the prefix and the digits, however.

Byte size values

Byte size values are similar to numeric values but they have 'KB', 'MB', or 'GB' suffix. The suffix may be separated from the digits by a blank, and is case-insensitive (e.g., 'KB' is the same as 'kb' and same as 'Kb').

These values translate to integers in bytes, where the prefixes are not metric but powers of 1024 as per JEDEC. Here is an example:

```
foo = 2MB  # becomes int(2 * 1024 * 1024) == int(2097152)
```

Boolean and null values

Boolean and null values are words with special meaning. These words are:

- yes (`True`)
- no (`False`)
- true (`True`)
- false (`False`)
- null (`None`)
- none (`None`)

These words are case-insensitive, so 'Yes' is the same as 'yes', and 'NULL' is the same as 'nuLL'.

Here are a few examples:

```
foo = yes
bar = False
baz = none
```

Lists

Lists are a special form of multi-line values. Lists are specified by starting the value with a newline and listing list items one item per line. For example:

```
foo =
    foo
    bar
    baz
```

The above value will be translated to a list of strings: `['foo', 'bar', 'baz']`.

All other types except multiline values and lists themselves can be used in lists. This inclues integers, floats, booleans, and bytes.

2.1.4 Referencing other configuration files

Configuration files can be made modular by cross-referencing other configuration file fragments. This is done by two list keys in a special `[config]` section. Here is an example:

```
[config]

defaults =
    networking.ini
    visuals.ini

include =
    /etc/myapp.d/networking.ini
    /etc/myapp.d/visuals.ini
    /etc/myapp.d/overrides.ini
```

The above example references two configuration files as defaults, and three files as includes. The primary difference between defaults and includes is in how they affect the configuration file in which they appear. Defaults serve as a base, which teh current configuration file overrides, while include override the current configuration.

The paths are evaluated relative to the configuration files. In the above example, the default configuration files are all assumed to reside in the same location as the configuation file in which they are referenced. Absolute paths are unaffected by this.

Paths may include glob patterns supported by Python's `glob` module. The above example for the `include` key can be rewritten as:

```
include =
    /etc/myapp.d/*.ini
```

All of the paths referenced by the `[config]` section are optional, in the sense that missing paths will not cause failure.

2.1.5 Extending lists

Lists can be extended between two configuration files. This is best described through an example:

```
# default.ini
[foo]

bar =
   1
   2
   3

# master.ini
[config]

defaults =
    default.ini

[foo]
```

```
+bar =
    4
    5
    6
```

By prefixing a key with a plus sign (+), the `bar` list in `master.ini` will be used to extend the `bar` list in `default.ini`. The resulting value will be `[1, 2, 3, 4, 5, 6]`.

This also applies to extensions defined in an include, which do not replace the original keys found in the configuration file in which it is referenced, but extends it instead.

When Confloads encounters an extend key, but there is nothing to extend, it will simply create an empty list and extend it. For example, if the `default.ini` in the above example did not contain any `bar` key, the result would be a list that contains only the elements from `master.ini`'s `bar` list: `[4, 5, 6]`.

2.2 Working with configuration files

This section gives you a quick overview of Confloader library usage.

2.2.1 Loading configuration files

Configuration files can be loaded from files or file descrptors and objects that support file-descriptor-like API (e.g., `StringIO`). To load a configuration file, you can use the `from_file()` method on the `confloader.ConfDict` class:

```
from confloader import ConfDict

conf = ConfDict.from_file('config.ini')
```

If the configuration file is blank, missing, contains no section, or has options that are dangling outside sections, or otherwise malformed, you will get a `ConfigurationError` exception. This exception is available as an attribute on the `ConfDict` class as convenience:

```
try:
    conf = ConfDict.from_file('nonexistent.ini')
except ConfDict.ConfigError:
    print('Oh noes!')
```

Application may specify its own defaults when loading configuration files. This is done by using the `defaults` argument which must be a dictionary:

```
conf = ConfDict.from_file('config.ini', defaults={
    'myoption1': 12,
    'myoption2': no
})
```

If, for some reason, you don't like type conversions, you can omit type conversion by passing the `skip_clean` flag:

```
conf = ConfDict.from_file('config.ini', skip_clean=True)
```

List extension can be suppressed by using `noextend` parameter:

```
conf = ConfDict.from_file('config.ini', noextend=True)
```

2.2.2 Adding options from configuration files at runtime

The confiuration object, once instantiated, can be further manipulated by calling the `import_from_file` method on the `ConfigDict` objects. For example:

```
conf = ConfDict.from_file('config.ini')
conf.import_from_file('fragment.ini')
```

This method has two modes. The first mode is the include mode, which overwrites existing options using the options from the specified file. The other mode is the defaults mode which only fills in the blank while leaving existing options intact. The defaults mode is enabled by supplying `as_defaults=True` argument.

By default, calling `import_from_file` on a non-existent configuration file will raise the `ConfigError` exception. This exception can be suppressed by passing the `ignore_missing=True` argument.

2.2.3 Accessing options

Options are accessed via keys that are a combination of the section name and option name.

```
[foo]

bar = 1
```

The `bar` option from the above example is accessed as `config['foo.bar']`.

There is a special section named `[global]`. Options that appear in this section are unprefixed.

```
[global]

foo = yes
```

The `foo` option from the above example is acessed as `conf['foo']`.

2.3 API documentation

class `confloader.`**`ConfDict`**(**args*, ***kwargs*)

Dictionary subclass that is used to hold the parsed configuration options.

`ConfDict` is instantiated the same way as dicts. For this reason, the paths to configuation files and similar are not passed to the constructor. Instead, you should use the *`from_file()`* classmethod.

Because this class is a dictionary, you can use the standard `dict` API to access and modify the keys. There is a minor difference when accessing key values, though. When using the subscript notation, *`ConfigurationFormatError`* is raised instead of `KeyError` when the key is missing.

exception `ConfigurationError`

Raised when application is not configured correctly.

exception `ConfDict.`**`ConfigurationFormatError`**(*keyerr*)

Raised when configuration file is malformed.

`ConfDict.`**`configure`**(*path*, *skip_clean=False*, *noextend=False*)

Configure the *`ConfDict`* instance for processing.

The `path` is a path to the configuration file. `skip_clean` parameter is a boolean flag that suppresses type conversion during parsing. `noextend` flag suppresses list extension.

classmethod ConfDict.**from_file**(*path*, *skip_clean=False*, *noextend=False*, *defaults={}*)
Load the values from the specified file. The skip_clean flag is used to suppress type conversion. noextend flag suppresses list extension.

You may also specify default options using the defaults argument. This argument should be a dict. Values specified in this dict are overridden by the values present in the configuration file.

ConfDict.**get_option**(*section*, *name*, *default=None*)
Returns a single configuration option that matches the given section and option names. Optional default value can be specified using the default parameter, and this value is returned when the option is not found.

As with *get_section()* method, this method operates on the parsed configuration file rather than dictionary data.

ConfDict.**get_section**(*name*)
Returns an iterable containing options for a given section. This method does *not* return the dict values, but instead uses the underlying parser object to retrieve the values from the parsed configuration file.

ConfDict.**import_from_file**(*path*, *as_defaults=False*, *ignore_missing=False*)
Imports additional options from specified file. The as_default flag can be used to cause the options to only be imported if they are not already present. The ignore_missing suppresses the *ConfigurationError* exception when the specified file is missing.

ConfDict.**load**()
Parses and loads the configuration data. This method will trigger a sequence of operations:

- initialize the parser, and load and parse the configuration file
- check the configuration file
- perform preprocessing (check for references to other files)
- process the sections
- process any includes or extensions

Any problems with the referenced defaults and includes will propagate to this call.

Note: Using this method for reloading the configuration is not recommended. Instead, create a new instance using the *from_file()* method.

ConfDict.**sections**
Returns an iterable containing the names of sections. This method uses the underlying parser object and does not work with the dict values.

ConfDict.**setdefaults**(*other*)
This method is a counterpart of the update() method and works like setdefault(). The other argument is a dict or dict-like object, whose key-value pairs are added to the *ConfDict* object if the key does not exist already.

exception confloader.**ConfigurationError**
Raised when application is not configured correctly.

exception confloader.**ConfigurationFormatError**(*keyerr*)
Raised when configuration file is malformed.

confloader.**extend_key**(*d*, *key*, *val*)
Extends a dictionary key with a specified iterable. If the key does not exist, it is assigned a list before extending. If the key exists, but maps to a non-list value, the key value is convereted to a list before being extended.

confloader.**get_compound_key**(*section*, *key*)
Return the key that will be used to look up configuration options. Except for the global keys, the compoint key is in <section>.<option> format.

confloader.**get_config_path**(*default=None*)
Attempt to obtain and return a path to configuration file specified by --conf command line argument, and fall back on specified default path. Default value is None.

confloader.**make_list**(*val*)
If the value is not a list, it is converted to a list. Iterables like tuple and list itself are converted to lists, whereas strings, integers, and other values are converted to a list whose sole item is the original value.

confloader.**parse_key**(*section*, *key*)
Given section name and option name (key), return a compound key and a flag that is True if the option marks an extension.

confloader.**parse_size**(*size*)
Parses size with B, KB, MB, or GB suffix and returns in size bytes. The suffix is not metric but based on powers of 1024. The suffix is also case-insensitive.

confloader.**parse_value**(*val*)
Detect value type and coerce to appropriate Python type. The input must be a string and the value's type is derived based on it's formatting. The following types are supported:

- boolean ('yes', 'no', 'true', 'false', case-insensitive)
- None ('null', 'none', case-insensitive)
- integer (any number of digits, optionally prefixed with minus sign)
- float (digits with floating point, optionally prefix with minus sign)
- byte sizes (same as float, but with KB, MB, or GB suffix)
- lists (any value that sarts with a newline)

Other values are returned as is.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

C

confloader, 9

C

ConfDict (class in confloader), 9
ConfDict.ConfigurationError, 9
ConfDict.ConfigurationFormatError, 9
ConfigurationError, 10
ConfigurationFormatError, 10
configure() (confloader.ConfDict method), 9
confloader (module), 9

E

extend_key() (in module confloader), 10

F

from_file() (confloader.ConfDict class method), 9

G

get_compound_key() (in module confloader), 10
get_config_path() (in module confloader), 11
get_option() (confloader.ConfDict method), 10
get_section() (confloader.ConfDict method), 10

I

import_from_file() (confloader.ConfDict method), 10

L

load() (confloader.ConfDict method), 10

M

make_list() (in module confloader), 11

P

parse_key() (in module confloader), 11
parse_size() (in module confloader), 11
parse_value() (in module confloader), 11

S

sections (confloader.ConfDict attribute), 10
setdefaults() (confloader.ConfDict method), 10