profig Documentation

Release 0.5.1

Miguel Turner

Nov 01, 2019

Contents

profig is a straightforward configuration library for Python. Its objective is to make the most common tasks of configuration handling as simple as possible.

CHAPTER 1

Contents

1.1 Overview

profig is a straightforward configuration library for Python.

1.1.1 Motivation

Why another configuration library? The simple answer is that none of the available options give me everything I want, with an API that I enjoy using. This library provides a lot of powerful functionality, but never at the cost of simplicity.

1.1.2 Features

- Automatic value conversion.
- Section nesting.
- Dict-like access.
- Single-file module with no dependencies.
- Extensible input/output formats.
- Built-in support for INI files and the Windows registry.
- Preserves ordering and comments of INI files.
- Full Unicode support.
- Supports Python 2.7+ and 3.2+.

1.1.3 Installation

profig installs using *easy_install* or *pip*:

```
$ pip install profig
```

1.1.4 Example

Basic usage is cake. Let's assume our config file looks like this:

```
[server]
host = 192.168.1.1
port = 9090
```

First, we specify the defaults and types to expect:

```
>>> cfg = profig.Config('server.cfg')
>>> cfg.init('server.host', 'localhost')
>>> cfg.init('server.port', 8080)
```

Then, we sync our current state with the state of the config file:

```
>>> cfg.sync()
```

As expected, we can access the updated values without undue effort, either directly:

```
>>> cfg['server.host']
'192.168.1.1'
```

Or by section. Notice that the type of the *port* option is preserved:

```
>>> server_cfg = cfg.section('server')
>>> server_cfg['port']
9090
```

1.1.5 Resources

- Documentation
- PyPI
- Repository

1.2 Guide

1.2.1 Sections

A *Config* object can be used directly without any initialization:

```
import profig
cfg = profig.Config()
cfg['server.host'] = '8.8.8.8'
cfg['server.port'] = 8181
```

Configuration keys are `.` delimited strings where each name refers to a section. In the example above, *server* is a section name, and *host* and *port* are sections which are assigned values.

For a quick look at the hierarchical structure of the options, you can easily get the dict representation:

```
>>> cfg.as_dict()
{'server': {'host': '8.8.8.8', 'port': 8181}}
```

Section can also be organized and accessed in hierarchies:

```
>>> cfg['server.port']
8181
>>> cfg.section('server').as_dict()
{'host': 'localhost', 'port': 8181}
```

1.2.2 Initialization

One of the most useful features of the `profig` library is the ability to initialize the options that are expected to be set on a *`Config`* object.

If you were to sync the following config file without any initialization:

```
[server]
host = 127.0.0.1
port = 8080
```

The port number would be interpreted as a string, and you would have to convert it manually. You can avoid doing this each time you access the value by using the *`init()`* method:

```
>>> cfg.init('server.host', '127.0.0.1')
>>> cfg.init('server.port', 8080, int)
```

The second argument to *`init()`* specifies a default value for the option. The third argument is optional and is used by an internal *`Coercer`* to automatically (de)serialize any value that is set for the option. If the third argument is not provided, the type of the default value will be used to select the correct coercer.

1.2.3 Strict Mode

Strict mode can be enabled by passing *strict=True* to the *`Config`* constructor:

```
>>> cfg = profig.Config(strict=True)
# or
>>> cfg.strict = True
```

By default, *`Config`* objects can be assigned a new value simply by setting the value on a key, just as with the standard *dict*. When strict mode is enabled, however, assignments are only allowed for keys that have already been initialized using *`init()`*.

Strict mode prevents typos in the names of configuration options. In addition, any sync performed while in strict mode will clear old or deprecated options from the output file.

1.2.4 Automatic Typing (Coercion)

Automatic typing is referred to as "coercion" within the `profig` library.

Functions that adapt (object -> string) or convert (string -> object) values are referred to as "coercers".

- Values are "adapted" into strings.
- Strings are "converted" into values.

Defining Types

Type information can be assigned by initializing a key with a default value:

```
>>> cfg.init('server.port', 8080)
>>> cfg.sync()
>>> port = cfg['server.port']
>>> port
9090
>>> type(port)
<class 'int'>
```

When a type cannot be easily inferred, a specific type can by specified as the third argument to *init()*:

```
>>> cfg.init('editor.open_files', [], 'path_list')
```

In this case we are using a custom coercer type that has been registered to handle lists of paths. Now, supposing we have the following in a config file:

```
[editor]
open_files = profig.py:test_profig.py
```

We can the sync the config file, and access the stored value:

```
>>> cfg.sync()
>>> cfg['editor.open_files']
['profig.py', 'test_profig.py']
```

We can look at how the value is stored in the config file by using the section directly:

```
>>> sect = cfg.section('editor.open_files')
>>> sect.value(convert=False)
'profig.py:test_profig.py'
```

Note: When using coercion, it is important to take into account that, by default, no validation of the values is performed. It is, however, simple to have a coercer validate as well. See *Using Coercers as Validators* for details.

Adding Coercers

Functions that define how an object is coerced (adapted or converted) can be defined using the *Coercer* object that is accessible as an attribute of the root *Config* object.

Defining a new coercer (or overriding an existing one) is as simple as passing two functions to *register()*. For example, a simple coercer for a *bool* type could be defined like this:

```
>>> cfg.coercer.register(bool, lambda x: str(int(x)), lambda x: bool(int(x)))
```

The first argument to *register()* represents a type value that will be used to determine the correct coercer to use. A class object, when available, is most convenient, because it allows using a call to *type(obj)* to determine which coercer to use. However, any value can be used for the registration, including, e.g. strings or tuples.

The second argument specifies how to *adapt* a value to a string, while the third argument specifies host to *convert* a string to a value.

Using Coercers as Validators

By default, a coercer will only raise exceptions if there is a fundamental incompatibility in the values it is trying to coerce. What this means is that even if you register a key with a type of *int*, no restriction is placed on the value that can actually be set for the key. This can lead to unexpected errors:

```
>>> cfg.init('default.value', 1) # sets the type of the key to 'int'
>>> cfg['default.value'] = 4.5   # a float value can be set

# sync the float value to the config file
>>> buf = io.BytesIO()
>>> cfg.sync(buf)
>>> buf.getvalue()
'[default]\nvalue: 4.5\n'

# here, we change the float value
>>> buf = io.BytesIO('[default]\nvalue: 5.6\n')
>>> cfg.sync(buf)
>>> cfg['default.value'] # this now raises an exception
Traceback (most recent call last):
...
ConvertError: invalid literal for int() with base 10: '5.6'
```

This behavior can be changed by defining or overriding coercers in order to, for example, raise exceptions for unexpected ranges of inputs or other restrictions that should be in place for a given configuration value.

1.2.5 Synchronization

A *sync* is a combination of the read and write operations, executed in the following order:

1. Read in the changed values from all sources, except the values that have changed on the config object since the last time it was synced.
2. Write any values changed on the config object back to the primary source.

This is done using the *sync()* method:

```
>>> cfg.sync('app.cfg')
```

After a call to *sync()*, a new file with the following contents will be created in the current working directory:

```
[server]
host = 127.0.0.1
port = 8080
```

1.2.6 Sources

The sources a config object uses when it syncs can also be set in the *Config* constructor:

```
>>> cfg = profig.Config('app.cfg')
```

Sources can be either paths or file objects. Multiple sources can be provided:

```
>>> cfg = profig.Config('<appdir>/app.cfg', '<userdir>/app.cfg', '<sysdir>/app.cfg')
```

If more than one *source* is provided then a sync will update the config object with values from each of the sources in the order given. It will then write the values changed on the config object back to the first source.

Once sources have been provided, they will be used for any future syncs.

```
>>> cfg.sync()
```

1.2.7 Formats

A *Format* subclass defines how a configuration should be read/written from/to a source.

profig provides *Format* subclasses for both INI formatted files, and the Windows registry.

INI

INIFormat Syncs configuration with a file based on the standard INI format.

Because *INIFormat* is the default, an INI file can be sourced as follows:

```
>>> cfg = profig.Config('~/.config/black_knight.cfg')
```

The INI file format has no strict standard, however profig tries not to stray far from the most commonly supported features. In particular, it should read any INI file that the standard library's configparser can read. If it does not, please file an issue so that the discrepency can be fixed.

The only deviation profig makes is to support the fact that it is possible to set a value on any *ConfigSection*, including those at the top-level, which become section headers when output to an INI file. This is represented in the INI files as values set on the header:

```
[server] = true
host = localhost
```

Windows Registry

RegistryFormat Syncs configuration with the Windows registry.

To use the *RegistryFormat*, you have to specify which format to use and specify a path relative to *HKEY_CURRENT_USER* (configurable as a class attribute):

```
>>> cfg = profig.Config(r'Software\BlackKnight', format='registry')
```

RegistryFormat is, of course, only available on Windows machines.

Support for additional formats can be added easily by subclassing *Format*.

Defining a new Format

To add support for a new format you must subclass *Format()* and override the, *read()* and *write()* methods.

read() should assign values to `config`, which is a local instance of the *Config* object. A *context* value can optionally be returned which will be passed to *write()* during a sync. It can be used, for example, to track comments and ordering of the source. *None* can be returned if no context is needed.

write() accepts any context returned from a call to *read()* and should read the values to write from `config`.

1.2.8 Unicode

`profig` fully supports unicode. The encoding to use for all encoding/decoding operations can be set in the *Config* constructor:

```
>>> cfg = profig.Config(encoding='utf-8')
```

The default is to use the system's preferred encoding.

Keys are handled in a special way. Both byte-string keys and unicode keys are considered equivalent, but are stored internally as unicode keys. If a byte-string key is used, it will be decoded using the configured encoding.

Values are coerced into a unicode representation before being output to a config file. Note that this means that by specifically storing a byte-string as a value, `profig` will interpret the value as binary data. The specifics of how binary data is stored depends on the format being used. The default format (*INIFormat*) operates on binary files, therefore binary data is written directly to its sources.

So, if we consider the following examples using a Python 2 interpreter, where *str* objects are byte-strings:

```
>>> cfg['default.a'] = '\x00asdf'
```

Will be stored to a config file as binary data:

```
[default]
a = \x00asdf
```

If this is not the desired behavior, there are other options available when calling *init()*. First, we can explicitely use unicode values:

```
>>> cfg.init('a', u'asdf')
```

This ensures that only data matching the set encoding will be accepted.

If we expect the data to be binary data, but don't want to store it directly, we can use one of the available encodings: 'hex', and 'base64':

```
>>> cfg.init('a', 'asdf', 'hex')
```

Or third, we can register different coercers for byte-strings:

```
>>> cfg.coercer.register(bytes, compress, decompress)
```

1.3 API

1.3.1 Config

The primary API classes.

class profig.**Config**(**sources*, ***kwargs*)
The root configuration object.

Any number of sources can be set using *sources*. These are the sources that will be using when calling `sync()`.

The format of the sources can be set using *format*. This can be the registered name of a format, such as "ini", or a `Format` class or instance.

An encoding can be set using *encoding*. If *encoding* is not specified the encoding used is platform dependent: locale.getpreferredencoding(False).

Strict mode can be enabled by setting *strict* to *True*. In strict mode, accessing keys that have not been initialized will raise an `InvalidSectionError`.

The dict class used internally can be set using *dict_type*. By default an *OrderedDict* is used.

A `Coercer` can be set using *coercer*. If no coercer is passed in, a default will be created. If *None* is passed in, no coercer will be set and values will be read from and written to sources directly.

This is a subclass of `ConfigSection`.

classmethod known_formats()
Returns the formats registered with this class.

set_format(*format*)
Sets the format to use when processing sources.

format can be the registered name of a format, such as "ini", or a `Format` class or instance.

format
The `Format` to use to process sources.

class profig.**ConfigSection**(*name*, *parent*)
Represents a group of configuration options.

This class is not meant to be instantiated directly.

adapt(*encode=True*)
value -> str

as_dict(*flat=False*, *dict_type=None*)
Returns the configuration's keys and values as a dictionary.

If *flat* is *True*, returns a single-depth dict with `.` delimited keys.

If *dict_type* is not *None*, it should be the mapping class to use for the result. Otherwise, the *dict_type* set by `__init__()` will be used.

convert(*string*, *decode=True*)
str -> value

default()
Get the section's default value.

get(*key*, *default=None*)
If *key* exists, returns the value. Otherwise, returns *default*.

If *default* is not given, it defaults to *None*, so that this method never raises an exception.

init(*key*, *default*, *type=None*, *comment=None*)
Initializes *key* to the given *default* value.

If *type* is not provided, the type of the default value will be used.

If a value is already set for the section at *key*, it will be coerced to *type*.

If a *comment* is provided, it may be written out to the config file in a manner consistent with the active `Format`.

read(**sources*, ***kwargs*)
Reads config values.

If *sources* are provided, read only from those sources. Otherwise, write to the sources in `sources`. A format for *sources* can be set using *format*.

reset(*recurse=True*, *clean=True*)
Resets this section to it's default value, leaving it in the same state as after a call to `ConfigSection.init()`.

If *recurse* is *True*, does the same to all the section's children. If *clean* is *True*, also clears the dirty flag on all sections.

section(*key*, *create=None*)
Returns a section object for *key*.

create will default to *False* when in strict mode. Otherwise it defaults to *True*.

If there is no existing section for *key*, and *create* is *False*, an `InvalidSectionError` is thrown.

sections(*recurse=False*, *only_valid=False*)
Returns the sections that are children to this section.

If *recurse* is *True*, returns grandchildren as well. If *only_valid* is *True*, returns only valid sections.

set_default(*value*)
Set the section's default value.

set_value(*value*)
Set the section's value.

sync(**sources*, ***kwargs*)
Reads from sources and writes any changes back to the first source.

If *sources* are provided, syncs only those sources. Otherwise, syncs the sources in `sources`.

format can be used to override the format used to read/write from the sources.

value()
Get the section's value.

write(*source=None*, *format=None*)
Writes config values.

If *source* is provided, write only to that source. Otherwise, write to the first source in `sources`. A format for *source* can be set using *format*. *format* is otherwise ignored.

dirty
True if this section's value has changed since the last write. Read-only.

has_children
True if this section has child sections. Read-only.

is_default
True if this section has a default value and its current value is equal to the default value. Read-only.

key
The section's key. Read-only.

name
The section's name. Read-only.

parent
The section's parent or *None*. Read-only.

root
Returns the root *ConfigSection* object. Read-only.

type
The type used for coercing the value for this section. Read only.

valid
True if this section has a valid value. Read-only.

1.3.2 Formats

Formats define de/serialization specifics.

class profig.**Format**

close(*file*)

flush(*file*)

open(*cfg*, *source*, *mode='r'*, *binary=True*)
Returns a file object.

If *source* is a file object, returns *source*. *mode* can be 'r' or 'w'. If *mode* is 'w', The file object will be truncated. If *binary* is *True*, the file will be opened in binary mode ('rb' or 'wb').

read(*cfg*, *file*)
Reads *file* to update *cfg*. Must be implemented in a subclass.

write(*cfg*, *file*, *values=None*)
Writes *cfg* to file. Must be implemented in a subclass.

error_mode
Specifies how the format should react to errors raised when processing a source.

Must be one of the following:

- ignore - Ignore all errors completely.
- warning - Log a warning for any errors.
- exception - Raise an exception for any error.

Only 'exception' will cause the format to stop processing a source.

error_modes = frozenset({'exception', 'warning', 'ignore'})
The supported error modes.

name = None
A convenient name for the format.

class profig.**INIFormat**
Implements reading/writing configurations from/to INI formatted files.

A header will be written for each section in the root config object.

class profig.**RegistryFormat**
Implements reading/writing configurations from/to the Windows registry.

Sections with children will be created as keys. Those without children will be created as values.

Keys will be created relative to `RegistryFormat.base_key`, which defaults to *HKEY_CURRENT_USER*.

1.3.3 Coercer

A Coercer handles the conversion of values to/from a human-readable string representation.

class profig.**Coercer**(*register_defaults=True*, *register_qt=None*)
The coercer class, with which adapters and converters can be registered.

adapt(*value*, *type=None*)
Adapt a *value* from the given *type* (type to string). If *type* is not provided the type of the value will be used.

convert(*value*, *type*)
Convert a *value* to the given *type* (string to type).

register(*type*, *adapter*, *converter*)
Register an adapter and converter for the given type.

register_adapter(*type*, *adapter*)
Register an adapter (type to string) for the given type.

register_choice(*type*, *choices*)
Registers an adapter and converter for a choice of values. Values passed into *adapt()* or *convert()* for *type* will have to be one of the choices. *choices* must be a dict that maps converted->adapted representations.

register_converter(*type*, *converter*)
Register a converter (string to type) for the given type.

1.4 Recipes

1.4.1 Setting Values from the Command-Line

It can be convenient to provide users with the ability to override config options using command-line switches. Here is an example of how that can be done using *argparse*:

```
cfg = profig.Config()
cfg.init('server.host', 'localhost')
cfg.init('server.port', 8080)

parser = argparse.ArgumentParser()
parser.add_argument('-O', dest='options', action='append',
    metavar='<key>:<value>', help='Overrides an option in the config file')

args = parser.parse_args(['-O', 'server.port:9090'])

# update option values
cfg.update(opt.split(':') for opt in args.options)

print(cfg['server.port']) # -> 9090
```

If you need to provide a list of available options to the user, you can simply iterate over the config object:

```
>>> for k in cfg:
...     print(k)
server.host
server.port
```

You can also restrict the options that can be set from the command-line to a specific section:

```
args = parser.parse_args(['-O', 'port:9090'])
cfg.section('server').update(opt.split(':') for opt in args.options)
```

1.4.2 Multiprocess Sychronization

One way to synchronize a config file across multiple processes is to use a lock file. This allows processes to make a modifications to a config file in a safe way and have that change be reflected across all other processes when they sync again.

Here is an example using the lockfile module:

```
>>> lock = lockfile.FileLock('.cfglock')
>>> with lock:
...     cfg.sync()
```

1.4.3 Serialization

Because *Config* objects are based on dicts, it is easy to read/write configs from a serialization format such as JSON or msgpack:

```
>>> import json
>>> s = json.dumps(cfg.as_dict())
>>> cfg.update(json.loads(s))
```

1.4.4 Format Strings

Config objects can be used directly in format strings in several ways:

```
>>> '{0[server.host]}:{0[server.port]}'.format(cfg)
localhost:8080
>>> '{c[server.host]}:{c[server.port]}'.format(c=cfg)
localhost:8080
>>> '{host}:{port}'.format(**c.section('server'))
localhost:8080
```

In my opinion, this largely resolves the use cases for the interpolation feature of the stdlib configparser.

1.5 Development

All contributions to `profig` are welcome. Begin by forking the central repository:

```
$ hg clone ssh://hg@bitbucket.org/dhagrow/profig
```

1.5.1 Coding Style

Officially, `profig` follows the guidelines outlined by PEP8.

1.5.2 Tests

All tests are in `tests.py`. They can be run with either Python 2 or Python 3:

```
$ python2 -m tests
.................................
----------------------------------------------------------------------
Ran 33 tests in 0.425s

OK
$ python3 -m tests
.................................
----------------------------------------------------------------------
Ran 33 tests in 0.409s

OK
```

1.6 Release History

0.5.1 (2019-10-24)

- removed additional built-in formats (to avoid unnecessary bloat)

0.5.0 (2019-09-26)

- bugfixes
- experimentation with built-in TOML, YAML, and MessagePack formats

0.4.1 (2015-01-22)

- coercer support for datetime objects
- "strict" mode fixes

0.4.0 (2014-10-27)

- added "strict" mode to support safer configurations
- Windows registry format supports all types
- bugfixes/simplifications/clarifications

0.3.3 (2014-07-11)

- bugfixes

0.3.2 (2014-06-30)

- added support for the Windows registry
- bugfixes

0.3.1 (2014-06-04)

- fixed release package

0.3.0 (2014-06-04)

- byte-string values are read/written directly from/to sources.
- added support for Python 3.2.
- bugfixes

0.2.9 (2014-05-27)

- bugfixes
- new syntax for sections with both children, and a value: [section] = value

0.2.8 (2014-04-11)

- INI format is now the default
- custom ProfigFormat has been removed
- can now get/set comments for keys
- comments and whitespace read from sources are preserved
- filtering of keys when syncing has been removed (temporarily?)

0.2.7 (2014-03-29)

- improved INI support
- bugfixes

0.2.6 (2014-03-26)

- full unicode support

0.2.5 (2014-03-21)

- fix broken Python 3 compatibility

0.2.4 (2014-03-21)

- added support for Python 2
- bugfixes

0.2.3 and earlier (2014-03-12)

- initial releases

Index

W