LayeredConfig Documentation

Release 0.1.0

Staffan Malmgren

November 22, 2014

Contents

LayeredConfig compiles configuration from files, environment variables, command line arguments, hard-coded default values, or other backends, and makes it available to your code in a simple way.

CHAPTER 1

Introduction

LayeredConfig compiles configuration from files, environment variables, command line arguments, hard-coded default values, or other backends, and makes it available to your code in a simple way.

```
from layeredconfig import (LayeredConfig, Defaults, INIFile,
                           Environment, Commandline)

# This represents four different way of specifying the value of the
# configuration option "hello":

# 1. hard-coded defaults
defaults = {"hello": "is it me you're looking for?"}

# 2. INI configuration file
with open("myapp.ini", "w") as fp:
    fp.write("""
[DEFAULT]
hello = kitty
""")

# 3. enironment variables
import os
os.environ['MYAPP_HELLO'] = 'goodbye'

# 4.command-line arguments
import sys
sys.argv = ['./myapp.py', '--hello=world']

# Create a config object that gets settings from these four
# sources.
config = LayeredConfig(Defaults(defaults),
                       INIFile("myapp.ini"),
                       Environment(prefix="MYAPP_"),
                       Commandline())

# Prints "Hello world!", i.e the value provided by command-line
# arguments. Latter sources take precedence over earlier sources.
print("Hello %s!" % config.hello)
```

- A flexible system makes it possible to specify the sources of configuration information, including which source takes precedence (*Precedence*). Implementations of common sources are included (*Available sources*) and an API for writing new ones exists (*Implenting custom ConfigSource classes*)
- Configuration can include subsections (ie. `config.downloading.refresh`, *Subsections*) and if a sub-

section does not contain a requested setting, it can optionally be fetched from the main configuration (if `config.module.retry` is missing, `config.retry` can be used instead, *Cascading*).

- Configuration settings can be changed by your code (i.e. to update a "lastmodified" setting or similar), and changes can be persisted (saved) to the backend of your choice (*Modification and persistance*)
- Configuration settings are typed (ie. if a setting should contain a date, it's made available to your code as a `date` object, not a `str`). If settings are fetched from backends that do not themselves provide typed data (ie. environment variables, which by themselves are strings only), a system for type coercion makes it possible to specify how data should be converted (*Typing*).

CHAPTER 2

Usage

To use LayeredConfig in a project:

```
from __future__ import print_function
from layeredconfig import LayeredConfig
```

Also, import any Configuration sources you want to use. It's common to have one source for code defaults, one configuration file (INI file in this example), one using environment variables as source, and one using command lines:

```
from layeredconfig import Defaults, INIFile, Environment, Commandline
```

Each configuration source must be initialized in some way. The `Defaults` source takes a `dict`, possibly nested:

```
from datetime import date, datetime
mydefaults = Defaults({'home': '/tmp/myapp',
                       'name': 'MyApp',
                       'dostuff': False,
                       'times': 4,
                       'duedate': date(2014, 10, 30),
                       'things': ['Huey', 'Dewey', 'Louie'],
                       'submodule': {
                           'retry': False,
                           'lastrun': datetime(2014, 10, 30, 16, 40, 22)
                           }
                       })
```

A configuration source such as `INIFile` takes the name of a file. In this example, we use a INI-style file.

```
myinifile = INIFile("myapp.ini")
```

Note: LayeredConfig uses the `configparser` module, which requires that each setting is placed within a section. By default, top-level settings are placed within the `[__root__]` section.

In this example, we assume that there is a file called `myapp.ini` within the current directory with the following contents:

```
[__root__]
home = /usr/home/staffan/.myapp

[submodule]
retry = True
lastrun = 2014-10-31 16:40:22
```

The `Environment` source uses environment variables as settings. Since the entire environment is not suitable to use as a configuration, use of this source requires that a `prefix` is given. Only environment variables starting with this

prefix are used. Furthermore, since the name of environment variable typically uses uppercase, they are by default lowercased by this source. This means that, in this example, the value of the environmentvariable `MYAPP_HOME` will be available as the configuration setting `home`.

```
env = {'MYAPP_HOME': 'C:\\Progra~1\\MyApp',
       'MYAPP_SUBMODULE_RETRY': 'True'}
myenv = Environment(env, prefix="MYAPP_")
```

Finally, the `Commandline` processes the contents of sys.argv and uses any parameter starting with `--` as a setting, such as `--home=/Users/staffan/Library/MyApp`. Arguments that do not match this (such as positional arguments or short options like `-f`) are made available through the `rest` property, to be used with eg. `argparse`.

```
mycmdline = Commandline(['-f', '--home=/opt/myapp', '--times=2', '--dostuff'])
rest = mycmdline.rest
```

Now that we have our config sources all set up, we can create the actual configuration object:

```
cfg = LayeredConfig(mydefaults,
                    myinifile,
                    myenv,
                    mycmdline)
```

And we use the attributes on the config object to access the settings:

```
print("%s starting, home in %s" % (cfg.name, cfg.home))
```

2.1 Precedence

Since several sources may contain a setting, A simple precedence system determines which setting is actually used. In the above example, the printed string is `"MyApp starting, home in /opt/myapp"`. This is because while `name` was specified only by the mydefaults source, `home` was specified by source with higher predecence (`mycmdline`). The order of sources passed to LayeredConfig determines predecence, with the last source having the highest predecence.

2.2 Config sources

Apart from the sources used above, there are classes for settings stored in JSON files, YAML files and PList files. Each source can to varying extent be configured with different parameters. See *Available sources* for further details.

You can also use a single source class multiple times, for example to have one system-wide config file together with a user config file, where settings in the latter override the former.

It's possible to write your own `ConfigSource`-based class to read (and possibly write) from any concievable kind of source.

2.3 Typing

The values retrieved can have many different types – not just strings.

```
delay = date.today() - cfg.duedate  # date
if cfg.dostuff: # bool
    for i in range(cfg.times):  # int
        print(", ".join(cfg.things))  # list
```

If a particular source doesn't contain intrinsic typing information, other sources can be used to find out what type a particular setting should be. LayeredConfig converts the data automatically.

Note: string are always `str` objects, (`unicode` in python 2). They are never `bytes` objects (`str` in python 2)

2.4 Subsections

It's possible to divide up settings and group them in subsections.

```
subcfg = cfg.submodule
if subcfg.retry:
    print(subcfg.lastrun.isoformat())
```

2.5 Cascading

If a particular setting is not available in a subsection, LayeredConfig can optionally look for the same setting in parent sections.

```
cfg = LayeredConfig(mydefaults, myinifile, myenv, mycmdline, cascade=True)
subcfg = cfg.submodule
print(subcfg.home)
```

2.6 Modification and persistance

It's possible to change a setting in a config object. It's also possible to write out the changed settings to a config source (ie. configuration files) by calling `write()`

```
subcfg.lastrun = datetime.now()
LayeredConfig.write(cfg)
```

CHAPTER 3

API reference

class layeredconfig.**LayeredConfig**(**sources*, ***kwargs*)

Creates a config object from one or more sources and provides unified access to a nested set of configuration parameters. The source of these parameters a config file (using .ini-file style syntax), command line parameters, and default settings embedded in code. Command line parameters override configuration file parameters, which in turn override default settings in code (hence **Layered** Config).

Configuration parameters are accessed as regular object attributes, not dict-style key/value pairs. Configuration parameter names should therefore be regular python identifiers, and preferrably avoid upper-case and "_" as well (i.e. only consist of the characters a-z and 0-9)

Configuration parameter values can be typed (strings, integers, booleans, dates, lists...). Even though some sources lack typing information (eg in INI files, command-line parameters and enviroment variables, everything is a string), LayeredConfig will attempt to find typing information in other sources and convert data.

Parameters

- ***sources** – Initialized ConfigSource-derived objects
- **cascade** (*bool*) – If an attempt to get a non-existing parameter on a sub (nested) configuration object should attempt to get the parameter on the parent config object. `False` by default,
- **writable** (*bool*) – Whether configuration values should be mutable. `True` by default. This does not affect `set()`.

static write(*config*)

Commits any pending modifications, ie save a configuration file if it has been marked "dirty" as a result of an normal assignment. The modifications are written to the first writable source in this config object.

Note: This is a static method, ie not a method on any object instance. This is because all attribute access on a LayeredConfig object is meant to retrieve configuration settings.

Parameters **config** (*layeredconfig.LayeredConfig*) – The configuration object to save

static set(*config*, *key*, *value*, *sourceid='defaults'*)

Sets a value in this config object *without* marking any source dirty, and with exact control of exactly where to set the value. This is mostly useful for low-level trickery with config objects.

Parameters

- **config** – The configuration object to set values on
- **key** – The parameter name
- **value** – The new value

- **sourceid** – The identifier for the underlying source that the value should be set on.

static get(*config*, *key*, *default=None*)

Gets a value from the config object, or return a default value if the parameter does not exist, like `dict.get()` does.

CHAPTER 4

Available sources

class layeredconfig.**Defaults**(*defaults=None*, ***kwargs*)
This source is initialized with a dict.

Parameters **defaults** (*dict*) – A dict with configuration keys and values. If any values are dicts, these are turned into nested config objects.

class layeredconfig.**Environment**(*environ=None*, *prefix=None*, *lower=True*, *sectionsep='_'*, ***kwargs*)
Loads settings from environment variables. If `prefix` is set to `MYAPP_`, the value of the environment variable `MYAPP_HOME` will be available as the configuration setting `home`.

Parameters

- **environ** (*dict*) – Environment variables, in dict form like `os.environ`. If not provided, uses the real `os.environ`.
- **prefix** (*str*) – Since the entire environment is not suitable to use as a configuration, only variables starting with this prefix are used.
- **lower** (*True*) – If true, lowercase the name of environment variables (since these typically uses uppercase)

class layeredconfig.**Commandline**(*commandline=None*, *sectionsep='-'*, ***kwargs*)
Load configuration from command line options. Any long-style parameters are turned into configuration values, and parameters containing the section separator (by default `"-"`) are turned into nested config objects (i.e. `--module-parameter=foo` results in `self.module.parameter == "foo"`.

Parameters

- **commandline** (*list*) – Command line arguments, in list form like `sys.argv`. If not provided, uses the real `sys.argv`.
- **sectionsep** (*str*) – An alternate section separator instead of `-`.

rest = []
The remainder of the command line, containing all parameters that couldn't be turned into configuration settings.

class layeredconfig.**INIFile**(*inifilename=None*, *rootsection='__root__'*, *writable=True*, ***kwargs*)
Loads and optionally saves configuration files in INI format, as handled by `configparser`.

Parameters

- **inifile** (*str*) – The name of a ini-style configuration file. The file should have a top-level section, by default named `__root__`, whose keys are turned into top-level configuration parameters. Any other sections in this file are turned into nested config objects.

- **rootsection** (*str*) – An alternative name for the top-level section. See note below.
- **writable** (*bool*) – Whether changes to the LayeredConfig object that has this INIFile object amongst its sources should be saved in the INI file.

Note: Since this source uses `configparser`, and since that module handles sections named `[DEFAULT]` differently, this module will have a sort-of automatic cascading feature for subsections if `DEFAULT` is used as `rootsection`

---

class `layeredconfig.`**`JSONFile`**(*jsonfilename=None*, *writable=True*, ***kwargs*)

Loads and optionally saves configuration files in JSON format. Since JSON has some support for typed values (supports numbers, lists, bools, but not dates or datetimes), data from this source are sometimes typed, sometimes only available as strings.

Parameters

- **jsonfile** (*str*) – The name of a JSON file, whose root element should be a JSON object (python dict). Nested objects are turned into nested config objects.
- **writable** (*bool*) – Whether changes to the LayeredConfig object that has this JSONFile object amongst its sources should be saved in the JSON file.

class `layeredconfig.`**`YAMLFile`**(*yamlfilename=None*, *writable=True*, ***kwargs*)

Loads and optionally saves configuration files in YAML format. Since YAML (and the library implementing the support, PyYAML) has automatic support for typed values, data from this source are typed.

Parameters

- **yamlfile** (*str*) – The name of a YAML file. Nested sections are turned into nested config objects.
- **writable** (*bool*) – Whether changes to the LayeredConfig object that has this YAMLFile object amongst its sources should be saved in the YAML file.

class `layeredconfig.`**`PListFile`**(*plistfilename=None*, *writable=True*, ***kwargs*)

Loads and optionally saves configuration files in PList format. Since PList has some support for typed values (supports numbers, lists, bools, datetimes *but not dates*), data from this source are sometimes typed, sometimes only available as strings.

Parameters

- **plistfile** (*str*) – The name of a PList file. Nested sections are turned into nested config objects.
- **writable** (*bool*) – Whether changes to the LayeredConfig object that has this PListFile object amongst its sources should be saved in the PList file.

CHAPTER 5

Implenting custom ConfigSource classes

If you want to get configuration settings from some other sources than the built-in sources, you should create a class that derives from `ConfigSource` and implement a few methods. If your chosen source can expose the settings as a (possibly nested) `dict`, it might be easier to derive from `DictSource` which already provide implementations of many methods.

class `layeredconfig.`**`ConfigSource`**(***kwargs*)

The constructor of the class should set up needed resources, such as opening and parsing a configuration file.

It is a good idea to keep whatever connection handles, data access objects, or other resources needed to retrieve the settings, as unprocessed as possible. The methods that actually need the data (`has()`, `get()`, `subsection()`, `subsections()` and possibly `typed()`) should use those resources directly instead of reading from cached locally stored copies.

The constructor must call the superclass' `__init__` method with all remaining keyword arguments, ie. `super(MySource, self).__init__(**kwargs)`.

`dirty` = **False**

For writable sources, whether any parameter value in this source has been changed so that a call to `save()` might be needed.

`identifier` = **None**

A string identifying this source, primarily used with `LayeredConfig.set()`.

`writable` = **False**

Whether or not this source can accept changed configuration settings and store them in the same place as the original setting came from.

`parent` = **None**

The parent of this source, if this represents a nested configuration source, or None

`source` = **None**

By convention, this should be your main connection handle, data access object, or other resource neededed to retrieve the settings.

`has`(*key*)

This method should return true if the parameter identified by `key` is present in this configuration source. It is up to each configuration source to define the semantics of what exactly "is present" means, but a guideline is that only real values should count as being present. If you only have some sort of placeholder or typing information for `key` this should probably not return True.

Note that it is possible that a configuration source would return True for `typed(some_key)` and at the same time return False for `has(some_key)`, if the source only carries typing information, not real values.

get(*key*)
Should return the actual value of the parameter identified by `key`. If `has(some_key)` returns True, `get(some_key)` should always succeed. If the configuration source does not include intrinsic typing information (ie. everything looks like a string) this method should return the string as-is, LayeredConfig is responsible for converting it to the correct type.

keys()

typed(*key*)
Should return True if this source contains typing information for `key`, ie information about which data type this parameter should be.

For sources where everything is stored as a string, this should generally return False (no way of distinguishing an actual string from a date formatted as a string).

subsections()
Should return a list (or other iterator) of subsection keys, ie names that represent subsections of this configuration source. Not all configuration sources need to support subsections. In that case, this should just return an empty list.

subsection(*key*)
Should return the subsection identified by `key`, in the form of a new object of the same class, but initialized differently. Exactly how will depend on the source, but as a general rule the same resource handle used as `self.source` should be passed to the new object. Often, the subsection key will need to be provided to the new object as well, so that `get()` and other methods can use it to look in the correct place.

As a general rule, the constructor should be called with a `parent` parameter set to `self`.

set(*key*, *value*)
Should set the parameter identified by `key` to the new value `value`.

This method should be prepared for any type of value, ie ints, lists, dates, bools... If the backend cannot handle the given type, it should convert to a str itself.

Note that this does not mean that the changes should be persisted in the backend data, only in the existing objects view of the data (only when `save()` is called, the changes should be persisted).

save()
Persist changed data to the backend. This generally means to update a loaded configuration file with all changed data, or similar.

This method will only ever be called if `writable` is True, and only if `dirty` has been set to True.

If your source is read-only, you don't have to implement this method.

typevalue(*key*, *value*)
Given a parameter identified by `key` and an untyped string, convert that string to the type that our version of key has.

class layeredconfig.**DictSource**(***kwargs*)
If your backend data is exposable as a python dict, you can subclass from this class to avoid implementing `has()`, `get()`, `keys()`, `subsection()` and `subsections()`. You only need to write `__init__()` (which should set `self.source` to that exposed dict), and possibly `typed()` and `save()`.