LayeredConfig Documentation

Release 0.3.4.dev1

Staffan Malmgren

Nov 11, 2019

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.

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:

```python
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("""
[__root__]
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. Implementations of many common sources are included and there is a API for writing new ones.

- Included configuration sources for INI files, YAML files, JSON file, PList files, etcd stores (read-write), Python source files, hard-coded defaults, command line options, environment variables (read-only).

- Configuration can include subsections (ie. `config.downloading.refresh`) and if a subsection 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).

- 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.

- Configuration settings are typed (ie. if a setting should contain a date, it's made available to your code as a `datetime.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.

Usage

To use LayeredConfig in a project:

```python
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:

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

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

```python
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.

```python
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, as well as etcd stores. 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: strings 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 if the `cascade` option is set.

```
cfg = LayeredConfig(mydefaults, myinifile, myenv, mycmdline, cascade=True)
subcfg = cfg.submodule
print(subcfg.home)   # prints '/opt/myapp', from Commandline source root section
```

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)
```

Using LayeredConfig with `argparse`

The standard module for handling command line arguments in python is `argparse`. This module handles much of the same things as LayeredConfig does (eg. defining the default values and types of arguments and making them easily accessed), but it isn't able to read parameter values from other sources such as INI files or environment variables.

LayeredConfig integrates with argparse through the *Commandline* config source. If you have existing code to set up an `argparse.ArgumentParser` object, you can re-use that with LayeredConfig.

```python
import sys
import argparse
from datetime import date, datetime
from layeredconfig import LayeredConfig, Defaults, INIFile, Commandline, UNIT_SEP
```

After this setup, you might want to create any number of config sources. In this example we use a *Defaults* object, mostly used for specifying the type of different arguments.

```python
defaults = Defaults({'home': str,
                     'name': 'MyApp',
                     'dostuff': bool,
                     'times': int,
                     'duedate': date,
                     'things': list,
                     'submodule': {'retry': bool,
                                   'lastrun': datetime
                                   }
                     })
```

And also an *INIFile* that is used to store actual values for most parameters.

```python
with open("myapp.ini", "w") as fp:
    fp.write("""[__root__]
home = /tmp/myapp
dostuff = False
times = 4
duedate = 2014-10-30
```

```
things = Huey, Dewey, Louie

[submodule]
retry = False
lastrun = 2014-10-30 16:40:22
""")
inifile = INIFile("myapp.ini")
```

Next up, we create an instance of `argparse.ArgumentParser` in the normal way. Note that in this example, we specify the types of some of the parameters, since this is representative of how ArgumentParser normally is used. But you can also omit this information (the `action` and `type` parameters to `add_argument()`) as long as you provide information through a Defaults config source object.

Note: we don't add arguments for `--duedate` or `--submodule-lastrun` to show that LayeredConfig can define these arguments based on other sources. Also note that defaults values are automatically fetched from either defaults or inifile.

```
parser = argparse.ArgumentParser("This is a simple program")
parser.add_argument("--home", help="The home directory of the app")
parser.add_argument('--dostuff', action="store_true", help="Do some work")
parser.add_argument("-t", "--times", type=int, help="Number of times to do it")
parser.add_argument('--things', action="append", help="Extra things to crunch")
parser.add_argument('--retry', action="store_true", help="Try again")
parser.add_argument("file", metavar="FILE", help="The filename to process")
```

Now, instead of calling `parse_args()`, you can pass this initialized parser object as a named parameter when creating a `Commandline` source, and use this to create a `LayeredConfig` object.

Note that you can use short parameters if you want, as long as you define long parameters (that map to your other parameter names) as well

```
sys.argv = ['./myapp.py', '--home=/opt/myapp', '-t=2', '--dostuff', 'file.txt']
cfg = LayeredConfig(defaults,
                    inifile,
                    Commandline(parser=parser))
print("Starting %s in %s for %r times (doing work: %s)" % (cfg.name,
                                                            cfg.home,
                                                            cfg.times,
                                                            cfg.dostuff))
# should print "Starting MyApp in /opt/myapp for 2 times (doing work: True)"
```

The standard feature of argparse to create a help text if the `-h` parameter is given still exists. Note that it will also display parameters such as *–name'*, which was defined in the `Defaults` object, not in the parser object.

```
sys.argv = ['./myapp.py', '-h']
cfg = LayeredConfig(defaults,
                    inifile,
                    Commandline(parser=parser))
```

> **Warning:** Using a bespoke `argparse.ArgumentParser` together with subsections is a bit more complicated. If you want to do that, you will need to setup each argument to the ArgumenParser object by explicitly naming the internal name for the attribute as specifid by the *dest* parameter, and separating the subsections with the special `layeredconfig.UNIT_SEP` delimiter, eg:
>
> ```
> parser.add_argument("--submodule-retry", help="Whether to retry the submodule",
>                     dest="submodule"+UNIT_SEP+"retry")
> ```
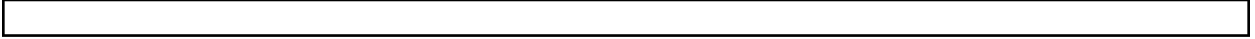
Embedding configuration in python files

In many cases, it's desirable to let the end user specify configuration in the same langauge as the rest of the system (Django and Sphinx are examples of frameworks that works this way). LayeredConfig provides the source *PyFile* that lets the user create configuration using normal python code.

If you create a file like `conf.py` with the following contents:

```python
from __future__ import unicode_literals

import os
import multiprocessing
from datetime import datetime, date

home = os.getcwd()
name = 'My App'
dostuff = name.istitle()
duedate = date.today()
submodule = Subsection()
submodule.retry = True
```

Note: The class `Subsection` will automatically be imported into `conf.py` and is used to create new subsections. Parameters in subsections are created as normal attributes on the subsection object.

And load it, together with a *Defaults* source like in previous examples:

```python
from layeredconfig import LayeredConfig, PyFile, Defaults
from datetime import date, datetime

conf = LayeredConfig(Defaults({'home': '/tmp/myapp',
                               'name': 'MyApp',
                               'dostuff': False,
                               'times': 4,
                               'duedate': date(2014, 10, 30),
```

(continues on next page)

```
                        'things': ['Huey', 'Dewey', 'Louie'],
                        'submodule': {
                            'retry': False,
                            'lastrun': datetime(2014, 10, 30, 16, 40, 22)
                        }
                    }),
                PyFile("conf.py"))
```

The configuration object will act the same as in previous examples, ie. values that are specified in `conf.py` be used, and values specified in the Defaults object only used if `conf.py` doesn't specify them.

Note: The *PyFile* source is read-only, so it should not be used when it's desirable to be able to save changed configuration parameters to a file. Use *PyFile* or one of the other *File sources in these cases.

It's also possible to keep system defaults in a separate python file, load these with one *PyFile* instance, and then let the user override parts using a separate *PyFile* instance. Functionally, this is not very different than loading system defaults using a *Defaults* source, but it might be preferable in some cases. As an example, if the file `defaults.py` contains the following code:

```python
from datetime import date, datetime

home = '/tmp/myapp'
name = 'MyApp'
dostuff = False
times = 4
duedate = date(2014, 10, 30),
things = ['Huey', 'Dewey', 'Louie']
submodule = Subsection()
submodule.retry = False
submodule.lastrun = datetime(2014, 10, 30, 16, 40, 22)
```

And a LayeredConfig object is initialized in the following way, then the resulting configuration object works identically to the above:

```python
conf = LayeredConfig(PyFile("defaults.py"),
                     PyFile("conf.py"))
```

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.

static dump(*config*)

Returns the entire content of the config object in a way that can be easily examined, compared or dumped to a string or file.

> **Parameters config** – The configuration object to dump
>
> **Return type** dict

static datetimeconvert(*value*)

Convert the string *value* to a `datetime` object. *value* is assumed to be on the form "YYYY-MM-DD HH:MM:SS" (optionally ending with fractions of a second).

static dateconvert(*value*)

Convert the string *value* to a `date` object. *value* is assumed to be on the form "YYYY-MM-DD".

static boolconvert(*value*)

Convert the string *value* to a boolean. `"True"` is converted to `True` and `"False"` is converted to `False`.

Note: If value is neither "True" nor "False", it's returned unchanged.

Available sources

6.1 Hardcoded defaults

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.

6.2 Environment variables

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 pro-
>   vided, 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)
>
> - **sectionsep** ([`str`](#)) – An alternate section separator instead of –.

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.

set(*key*, *val*)

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).

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`.

6.3 Command-line parameters

class layeredconfig.**Commandline**(*commandline=None*, *parser=None*, *sectionsep='-'*, *add_help=True*, ***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"`.

If an initialized ArgumentParser object is provided, the defined parameters in that object is used for supporting short form options (eg. `'-f'` instead of `'--force'`), typing information and help text. The standards argparse feature of printing a helpful message when the '-h' option is given is retained.

Parameters

- **commandline** (*list*) – Command line arguments, in list form like `sys.argv`. If not provided, uses the real `sys.argv`.

- **parser** (*argparse.ArgumentParser*) – An initialized/configured argparse object

- **sectionsep** (*str*) – An alternate section separator instead of –.

- **add_help** (*bool*) – Same as for ArgumentParser()

rest = []

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

setup(*config*)

Perform some post-initialization setup. This method will be called by the LayeredConfig constructor after its internal initialization is finished, with itself as argument. Sources may access all properties of the config object in order to eg. find out which parameters have been defined.

The sources will be called in the same order as they were provided to the LayeredConfig constructior, ie. lowest precedence first.

> **Parameters config** (*layeredconfig.LayeredConfig*) – The initialized config object that this source is a part of

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.

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).

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).

6.4 INI files

class layeredconfig.**INIFile**(*inifilename=None*, *rootsection='__root__'*, *sectionsep='.'*, *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.
> - **sectionsep** (`str`) – separator to use in section names to separate nested subsections. 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: Nested subsections is possible, but since the INI format does not natively support nesting, this is accomplished through specially-formatted section names, eg the config value mymodule.mysection.example would be expressed in the ini file as:

```
[mymodule.mysection]
example = value
```

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`

---

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`.

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.

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.

6.5 JSON files

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.

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).

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.

6.6 YAML files

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.

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.

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.

6.7 PList files

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.

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).

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.

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.

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.

6.8 Python files

class `layeredconfig.`**PyFile**(*pyfilename=None*, ***kwargs*)

Loads configuration from a python source file. Any variables defined in that file will be interpreted as configuration keys. The class `Subsection` is automatically imported into the context when the file is executed, and represents a subsection of the configuration. Any attribute set on such an object is treated as a configuration parameter on that subsection.

Note: The python source file is loaded and interpreted once, when creating the PyFile object. If a value is set by eg. calling a function, that function will only be called at load time, not when accessing the parameter.

Parameters pyfile (*str*) – The name of a file containing valid python code.

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.

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).

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).

6.9 `etcd` stores

class `layeredconfig.`**EtcdStore**(*baseurl='http://127.0.0.1:2379/v2/'*, ***kwargs*)
Loads configuration from a etcd store.

> **Parameters** `baseurl` – The main endpoint of the etcd store

`etcd` has no concept of typed values, so all data from this source are returned as strings.

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.

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=None*, *value=None*)

> 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.

CHAPTER 7

Implementing 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).

setup (*config*)

Perform some post-initialization setup. This method will be called by the LayeredConfig constructor after its internal initialization is finished, with itself as argument. Sources may access all properties of the config object in order to eg. find out which parameters have been defined.

The sources will be called in the same order as they were provided to the LayeredConfig constructior, ie. lowest precedence first.

> **Parameters config** (`layeredconfig.LayeredConfig`) – The initialized config object that this source is a part of

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().

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.

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).

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.

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).

Index