clac Documentation

Wesley Van Melle

Jan 25, 2019

Contents:

CHAPTER 1

What is CLAC?

CLAC is a multi-level configuration management library for use by applications which require multi-source configuration, such as multiple config files (which could have priority overriding based on location), environment variables, command-line parameters, and default configuration shipped with the application itself.

The theory behind the tool is that the each of the configuration sources is a layer, and each layer has a priority against the other layers, e.g. command-line options should override competing environment variables. The layers are collected into a single object, and a single function call will search for the proper value in each layer, returning the highest priority value.

1.1 Getting Started

First we install the library:

```
pip install clac
```

Next, we need to prepare each of our layers:

```
from clac import CLAC, DictLayer, DictStructure, EnvLayer

# Get the dicts for each of the layers

# rc-style files are usually flat keys, which could have dots in their names.
# Google .pylintrc or .coveragerc for examples of rc-style config files.
rcfile = {
    'lingua': 'franca',
    'salt.pepper': 'oregano',
    'foo': 'bar',
}
# Create the layer with a recognizable name
rc_layer = DictLayer('rcfile', rcfile)

# We can import toml or json directly into nested python dictionaries.
```

(continues on next page)

(continued from previous page)

```
# For this example we will assume that the following dict was imported
# earlier.
toml_dict = {
    'foo': 'baz',
    'spam': {
        'ham': 'eggs',
    },
    'salt': {
        'pepper': 'cayenne',
    },
}
# This time, we need to split the names by the dots, since these dicts have
# a nested-dict structure.
toml_layer = DictLayer('tomlfile', toml_dict, dot_strategy=DictStructure.Split)

# For environment variables, we can instatiate a layer directly
env_layer = EnvLayer('env')

# Now we can create the container for all of the layers
config = CLAC(rc_layer, toml_layer, env_layer)

# We could also build the object incrementally, separate from instatiation.
# Adding the rcfile layer first means that values from that rcfile values
# will override competing values from the toml layer.
config = CLAC()
config.add_layers(rc_layer)
# We can still add multiple layers at once.
config.add_layers(toml_layer, env_layer)
```

Now, we can check the various config entries.

```
# 'foo' is found in the rcfile first, so we get 'bar' instaed of 'baz'
assert config['foo'] == 'bar'
# But we can double-check the toml file if we want, using the name of the
# layer we want.
assert config.get('foo', layer_name='tomlfile') == 'baz'

# 'spam' is not found in the rcfile, so the CLAC checks the tomlfile
assert config['spam'] == {'ham': 'eggs'}

# Be careful when searching on partial keys, since flat DictLayers cannot
# support partial matching:

# 'salt' would be a partial match for rcfile, so toml file value is used
assert config['salt'] == {'pepper': 'cayenne'}
# 'salt.pepper' has an exact match in the rcfile, so we get that value
# before the toml file.
assert config['salt.pepper'] == 'oregano

# get() supports defaults and post-retrieval processing, but not at the same
# time.  Default values are *not* processed with the callback.

# Here, we set the callback to str, but still get an int, because the value
# was not found, and the callback was never executed.  This return the
# default value as-is.
assert config.get('missing', default=123, callback=str) == 123
```

1.2 API Reference

class clac.**CLAC**(**layers*)
Configuration container/manager.

`__init__()` parameters are the same as *add_layers()*.

add_layers(**layers*)
Adds layers to the lookup set. Called by `__init__()`

Parameters **layers** – A FIFO list of ConfigLayer instances. The first layer in this list will be the first layer queried.

build_lri() → Set[Tuple[str, Any]]
Returns the Layer Resolution Index (LRI)

The LRI is a `set` of 2-tuples which contain the first layer that a key can be found in, and the key itself.

get(*key: str*, *default: Any = None*, *layer_name: str = None*, *callback: Callable = None*) → Any
Gets values from config layers according to `key`.

Returns `default` if value is not found or LookupError is raised.

If `layer_name` is specified, the method will perform all of the same actions, but only against the layer with the specified name. Must be a str or None. Defaults to None.

If `callback` is specified, the method will pass the retrieved value to the callback, and return the result. If `callback` is None, the original result is returned as-is.

> **Warning:** If the value is not found, `callback` will not be executed on the default value. Applications should provide complete and prepared default values to the method.

Raises **MissingLayer** – if `layer_name` is specified but no layer with that name has been added.

Returns The value retrieved from the config layers, or `default` if no entry was found.

Note: If an exception is desired instead of a default value, `__getitem__` syntax (`clac_instance[key]`) should be used instead. This will raise a `NoConfigKey` exception. However, the `__getitem__` syntax does not support additional arguments. This means that only *get()* will support defaults and coercion, and only `__getitem__` will support exception bubbling.

---

insert_layers(**layers*, *raise_on_replace=True*)
Inserts layers into the start of the lookup.

If any layer name already exists, it will be inserted in the new position instead of retaining its old position. This can be used to reorder the priority of any or all of the layers. If a layer name is duplicated in the provided `layers` parameter, then the first value is taken, and the others are silently ignored.

If any layer name conflicts are detected while moving old layers into the rebuilt lookup, the objects are compared for identity. If the identities match, the new position of the layer is not overwritten with the layer's previous position. If the identities do not match, `LayerOverwriteError` is raised, and the operation is cancelled, having no effect on the original lookup. This check is ignored if `raise_on_replace` is False (default is True).

> **Warning:** This function will rebuild the internal lookup, which can be expensive if there are a large number of entries. However, having a large number of configuration sources is an unusual use case, and should not be considered a major performance impact which needs optimization.

layers
A copy of the internal layer lookup.

names
Returns a set of all unique config keys from all layers.

remove_layer(*name: str*, *error_ok: bool = True*)
Remove layer `name` from the manager.

Parameters

- **name** – The name of the layer to be removed.
- **error_ok** – bool specificying whether to ignore errors. Defaults to True. Will check to make sure layer is missing.

Raises `MissingLayer` if name is not found in lookup and `error_ok` is False

resolve(*key: str*) → Tuple[str, Any]
Returns that name of the layer accessed, and the value retrieved.

Parameters **key** – The key to search for.

Raises **NoConfigKey** – `key` not in any layer.

Returns 2-tuple: (layer, value)

setdefault(*key: str*, *default: Any = None*) → Any
Call `BaseConfigLayer.set_default()` on the first mutable layer.

class clac.**BaseConfigLayer**(*name: str*, *mutable: bool = False*)
Abstract Base class for ConfigLayer Implementation

This class cannot be used directly, and will raise `TypeError` if instantiated. Rather, it is meant to be subclassed to perform its own application-specific configuration handling. For example, a subclass named `ConfLayer` might be created to read UNIX-style configuration files.

Important: Because this class is based off of the stdlib collections.abc.Mapping abstract class, there are abstract methods not defined in this class which must still be defined by subclasses.

Parameters

- **name** – The qualified name of the config layer instance. Will be used to look up the specified layer by the CLAC. This name should not be changed after instantiation.
- **mutable** – `bool` representing whether the layer allows mutation. Readable using the *`mutable()`* property.

assert_mutable()
Raises `ImmutableLayer` if layer is not mutable.

get(*key: str*, *default=None*) → Any
Gets the value for the specified `key`

Returns `default` if `key` is not found.

mutable
Whether the layer instance is mutable or not.

This value is checked before all internal `set`-style calls. Any method which overrides `__setitem__()` or *`setdefault()`* must manually perform the check by calling `self.assert_mutable()` with no arguments.

name
The name of the layer.

The *`CLAC`* instance will use this name as a lookup key for all layer-specific operations. This will have no effect on non-specific `get` and `set` -type calls.

names
Returns the full list of keys in the Layer

setdefault(*key: str*, *default: Any = None*) → Any
If `key` is in the lookup, return its value.

If `key` is not in the lookup, insert key with a value of `default` and return `default`. `default` defaults to `None`.

class clac.**DictLayer**(*name: str*, *config_dict: Optional[dict] = None*, *mutable: bool = False*, *dot_strategy: clac.core.DictStructure = <DictStructure.Flat: 2>*)
A config layer based on the python `dict` type.

name Behaves the same as all other layers. See *`BaseConfigLayer`* for more details.

config_dict An optional mapping object which contains the initial state of the configuration layer.

mutable A boolean representing whether the layer should allow mutation.

dot_strategy One of the variants of the *`DictStructure`* enum. Defaults to `DictStructure.Flat`.

`DictStructure.Split` will assume a nested dict structure. Keys will be split by the `.` character.

`DictStructure.Flat` will assume a flat dict structure. Keys are not modified before querying the underlying dict.

This example illustrates both usages

```
split = {
    'a': {
        'b': {
            'c': 'd'
        }
    }
}

flat = {'a.b.c': 'd')

layer1 = DictLayer('name', split, dot_strategy=DictStructure.Split)
assert layer1['a.b.c'] == 'd'

layer2 = DictLayer('name', flat, dot_strategy=DictStructure.Flat)
assert layer2['a.b.c'] == 'd'
```

names
Returns a strategy-aware set of valid keys

class clac.**DictStructure**
Enum for *`DictLayer`* structuring strategy.

This enum class exposes two variants:

- `Split`
- `Flat`

See *DictLayer* for more info.

1.3 Setting up a Development Environment

This document explains how to set up a devlopment environment for the CLAC project.

1.3.1 A note about Makefiles

CLAC includes a Makefile to simplify the devlopment process. This allows commands to be run in a virtual environment without having to actually activate the environment in your own shell. To support this on Windows (which is the real outlier) and other systems at the same time, the WORKON_HOME environment variable must be set, and must NOT contain backslashes. This means that on my windows platforms, the virtualenv home directory must be set to `D:/dev/venv` instead of `D:\dev\venv`.

1.3.2 Cloning the project

To clone the project, please follow the steps below (after reading the whole section to understand exactly what impact it will have):

- Ensure you are cloning the correct project (i.e. cloning your fork instead of the project itself)
- `git clone https://github.com/<yourusername>/clac`
- `make test`

The `make test` command will create a new virtual environment using the current python interpreter as a base. The environment will be stored in `$WORKON_HOME/clac`, and the full path to the executable will be written to a file named `.venv` at the root of the project.

1.3.3 Confirming your changes

In order to prevent destabilization of the project, all changes must pass CI checks without fail. Changes will not be accepted until this occurs. There are four checks which occur as of writing this:

- Unit testing through pytest - `make test`
- Code style/linting with pylint - `make lint`
- Static type analysis using mypy - `make lint` *must pass the pylint check first for make to work.*
- Building documentation powered by sphinx - `make docs`

If any one of these fails, your pull request will be rejected. If rejected, you must submit a patch to the pull request which resolves the issue.

Additionally, if your change is an enhancement or bugfix to the codebase, it must be accompanied by passing, coverage-proven unit tests. Changes without unit tests will not be accepted, but if you are unsure of how to unit test your changes, submit anyway and the project owner will help you to write effective unit tests for your change.

1.3.4 Working without make

If you cannot use make, or wish to setup your own environment in a different way, please follow the guide as-is, but referencing this section for alternative executions to mimic make's functionality.

This section will be written with the assumption that you understand how to configure your desired environment and use it properly, and is being provided as a convenience. Additionally, all of these commands should be executed from the project root.

`pip install` sections only need to be run once per environment.

make install `pip install -e .`

make install-dev `pip install -e .[fulldev]`

make test `pip install -e .[test,cov] && pytest --cov clac tests && python -m coverage html`

make docs `pip install -e .[docs] && sphinx-build -M html docs/source docs/build`

make lint `pip install -e .[lint] && pylint clac && mypy clac`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index