Thanks for helping out!

Feature requests, bug reports and PRs are always welcome!

This file provides a few pointers on how to contribute to dotdrop

and where to find information. For any question, feel free to open an issue.

For PR adding new features, I'd be very thankful if you could add either

a unittest testing the added feature or a bash script test, thanks!

# Code base

Dotdrop's code base is located in the [dotdrop directory](/dotdrop).

Here's an overview of the different files and their role:

\* \*\*action.py\*\*: represent the actions and transformations

\* \*\*cfg\_yaml.py\*\*: the lower level config parser

\* \*\*cfg\_aggregator.py\*\*: the higher level config parser

\* \*\*comparator.py\*\*: the class handling the comparison for `compare`

\* \*\*dictparser.py\*\*: abstract class for parsing dictionaries

\* \*\*dotdrop.py\*\*: the entry point and where the different cli commands are executed

\* \*\*dotfile.py\*\*: represent a dotfile

\* \*\*installer.py\*\*: the class handling the installation of dotfile for `install`

\* \*\*jhelpers.py\*\*: list of methods available in templates with jinja2

\* \*\*linktypes.py\*\*: enum for the three types of linking (none, symlink, children)

\* \*\*logger.py\*\*: the custom logger

\* \*\*options.py\*\*: the class embedding all the different options across dotdrop

\* \*\*profile.py\*\*: represent a profile

\* \*\*settings.py\*\*: represent the config settings

\* \*\*templategen.py\*\*: the jinja2 templating class

\* \*\*updater.py\*\*: the class handling the update of dotfiles for `update`

\* \*\*utils.py\*\*: some useful methods

## Config parsing

The configuration file (yaml) is parsed in two layers:

\* the lower layer in `cfg\_yaml.py`

\* the higher layer in `cfg\_aggregator.py`

Only the higher layer is accessible to other classes of dotdrop.

The lower layer part is only taking care of basic types and

does the following:

\* normalize all config entries

\* resolve paths (dotfiles src, dotpath, etc)

\* refactor actions to a common format

\* etc

\* import any data from external files (configs, variables, etc)

\* apply variable substitutions

\* complete any data if needed (add the "profile" variable, etc)

\* execute intrepreted variables through the shell

\* write new entries (dotfile, profile) into the dictionary and save it to a file

\* fix any deprecated entries (link\_by\_default, etc)

\* clear empty entries

In the end it makes sure the dictionary (or parts of it) accessed

by the higher layer is clean and normalized.

The higher layer will transform the dictionary parsed by the lower layer

into objects (profiles, dotfiles, actions, etc).

The higher layer has no notion of inclusion (profile included for example) or

file importing (import actions, etc) or even interpreted variables

(it only sees variables that have already been interpreted).

It does the following:

\* transform dictionaries into objects

\* patch list of keys with its corresponding object (for example dotfile's actions)

\* provide getters for every other classes of dotdrop needing to access elements

Note that any change to the yaml dictionary (adding a new profile or a new dotfile for

example) won't be \*seen\* by the higher layer until the config is reloaded. Consider the

`dirty` flag as a sign the file needs to be written and its representation in higher

levels in not accurate anymore.

## Variables resolution

How variables are resolved (pass through jinja2's

templating function) in the config file.

\* resolve `include` (the below merge is temporary just to resolve the `includes`)

\* `variables` and `dynvariables` are first merged and recursively resolved

\* `dynvariables` are executed

\* they are all merged and `include` paths are resolved

(allows to use something like `include {{@@ os @@}}.variables.yaml`)

\* `variables` and profile's `variables` are merged

\* `dynvariables` and profile's `dynvariables` are merged

\* `dynvariables` are executed

\* they are all merged into the final \*local\* `variables`

These are then used to resolve different elements in the config file:

see [this](https://github.com/deadc0de6/dotdrop/wiki/config-variables#config-available-variables)

Then additional variables (`import\_variables` and `import\_configs`) are

then merged and take precedence over local variables.

Note:

\* `dynvariables` > `variables`

\* profile `(dyn)variables` > any other `(dyn)variables`

\* profile `(dyn)variables` > profile's included `(dyn)variables`

\* imported `variables`/`dynvariables` > `(dyn)variables`

\* actions/transformations using variables are resolved at runtime

(when action/transformation is executed) and not when loading the config

# Testing

Dotdrop is tested with the use of the [tests.sh](/tests.sh) script.

\* test for PEP8 compliance with `pycodestyle` and `pyflakes`

\* run the unittest available in [tests directory](/tests)

\* run the bash script tests in [tests-ng directory](tests-ng)

## testing with unittest

All unittests are available in [tests directory](/tests)

and use [python unittest](https://docs.python.org/3/library/unittest.html).

The file [helpers.py](/tests/helpers.py) provides different helper methods

for the tests.

## testing with bash scripts

The bash scripts are available in [tests-ng directory](tests-ng).

These test entire workflows by simulating the use of dotdrop from end to end

for different use-cases (usually described in their filename).

Each script starts with the same boiler plate code that you can paste at the

start of your new test (see the head of the file down to `# this is the test`).

# Documentation

Most of dotdrop documentation is hosted in [its wiki](https://github.com/deadc0de6/dotdrop/wiki)