pop-config

Release 6.11

Sep 15, 2021

Contents:

`pop-config` is a core component of POP and a key part of POP's ability to "app merge" applications.

The word "config" has a lot of different meanings depending on context, so it's important that we clarify the capabilities of `pop-config`. In the world of POP, think of "config" as referring to *command-line arguments for a program*, *plus a bunch of other cool functionality related to command-line arguments*. It might be best to pretend `pop-config` is actually named `pop-args`. This would be a more accurate name and better convey its functionality.

The reason why the name `pop-config` is used may be that `pop-config` has a lot of very useful functionality, and taken as a whole, can often be used to manage the entire configuration of your application. For example, `pop-config` allows you to:

- Define command-line arguments for your application.
- Optionally set these command-line arguments via environment variables.
- Optionally source these command-line arguments from a file.
- Merge in plugins which augment the command-line arguments of your application.

The important thing to keep in mind is that all this functionality is *argument-centric*, and yes, many applications will find their configuration needs fully satisfied by `pop-config`.

But it's really `pop-args` :) Just keep that in the back of your head. It will help.

CHAPTER 1

Introduction: conf.py File

`pop-config` uses a `conf.py` file to define its functionality. The `conf.py` file should be located in the main directory of your POP project. So let's assume the directory for your source code is `my-project`, and your POP project inside it is called `my_project` (as created by `pop-seed` or `pop-create`). In this case, your `conf.py` file will be located at `my-project/my_project/conf.py`, and directories for POP subs for your project will appear *next* to the `conf.py` (in the same directory.)

One `conf.py` exists per POP project. `conf.py` can contain four Python dictionaries: `CONFIG`, `CLI_CONFIG`, `SUBCOMMANDS` and `DYNE`. Each dictionary serves a specific purpose. Between them you can define how the command-line arguments are presented, all configuration defaults, help documentation, etc. Here are the purposes of each dictionary:

CHAPTER 2

Dictionaries Overview

`DYNE`:

> The `DYNE` dictionary is used to allow your POP project to define dynamic names. Dynamic names are plugin subsystems that are shared across multiple projects and dynamically discovered. This allows you to, for example, have one "super-command" which can find a bunch of plugins that were installed by multiple different Python projects. All the plugins are organized under a dyne name. Each project maps the dyne name to a path inside its source code. The plugins in this directory are made available to other projects when the dyne is added via a call to `hub.pop.sub.add(dyne_name="foo")`. Then `hub.foo.plugin_name_1`, `hub.foo.another_plugin` will be available on the hub. You can also introspect on the plugins available via `for plugin in hub.foo:`, for example.

`CLI_CONFIG`:

> `CLI_CONFIG` is a dictionary that defines command-line arguments for your application. The command line arguments defined here will be accessible at `hub.OPT.pop_project_name.foo` or `hub.OPT["pop_project"]["foo"]` (assuming an option of `--foo`), for example. Each POP project has a single namespace for command-line options. The keys and values used in `CLI_CONFIG` will be very familiar if you have used the *argparse* module in Python.
>
> Please see the *CLI_CONFIG Dictionary* section for more details on how to use `CLI_CONFIG`.

`CONFIG`:

> `CONFIG` is a dictionary that defines "configuration" for your application, which are settings, but ones that are not available on the command-line. Configuration defined in `CONFIG`, while not settable on the command-line, is still accessible via `hub.OPT`.
>
> Where are `CONFIG` settings sourced from, if not from the command-line? Typically, they are simply default values, potentially overridable via a `pop-config` YAML configuration file.
>
> Please see the *CONFIG Dictionary* section for more details on how to use `CONFIG`.

`SUBCOMMANDS`:

> Think of `SUBCOMMANDS` as a companion to `CLI_CONFIG`. `SUBCOMMANDS` allows you to define higher-level actions on the command-line, each with their own separate arguments. For example, you may have `mycmd list` as well as `mycmd commit`. The subcommand is specified as just a literal string.

Command-line arguments defined in `CLI_CONFIG` can be specified as being specific to a subcommand or can be made available to all subcommands.

Please see the *SUBCOMMANDS Dictionary* section for more details on subcommands.

CHAPTER 3

Steps for Using `pop-config`

To use `pop-config`, at the bare minimum you will want to create a `conf.py` for your project at `my-project/my_project/conf.py`. It's important to note that there is only one `conf.py` per POP project, and only one set of config per `conf.py`.

In `conf.py`, you will define `CLI_CONFIG` to specify all command-line options for your application. Here is an example of a `CLI_CONFIG` definition:

```
CLI_CONFIG = {
    "force": {"options": ["--force"], "action": "store_true", "default": False},
    "nopush": {"options": ["--nopush"], "action": "store_true", "default": False},
    "prod": {"options": ["--prod"], "action": "store_true", "default": False},
    "db": {"options": ["--db"], "action": "store_true", "default": False},
    "release": {"positional": True},
}
```

Then, somewhere in the startup code of your command, you will have something similar to the following Python code:

```
hub.pop.sub.add("my_project")
# or hub.pop.sub.add(dyne_name="my_project") if you are using a dynamic name
hub.pop.config.load(["my_project"], cli="my_project")
```

After this last command finishes, the config defined in `conf.py` as well as any user-specified arguments will be available on the `hub` at `hub.OPT["my_project"].option_name`. If you defined any subcommands via `SUBCOMMANDS`, you will be able to determine the subcommand specified by inspecting the `hub.SUBPARSER` variable, which is a string that specifies the subcommand. Any subcommand-specific options will be accessible at `hub.OPT["my_project"].option_name` – there is no special hierarchy for subcommand options – they are just mapped to the same place as regular options.

Once this initialization is done, then various parts of your application can look at `hub.OPT["my_project"]` and use the settings found to influence its behavior. Because `hub.OPT` is globally available to all POP functions, you do not need to pass around these options as arguments to functions and can read them from a central, consistent location on the hub.

CHAPTER 4

CONFIG Dictionary

The `CONFIG` dictionary is used to define settings in your application that may or may not also be settable on the command-line. This means that the bulk of your application's settings will be defined in the `CONFIG` dictionary, even if only to serve as a place to define their `default` setting and `help` string.

```
CONFIG = {
    "name": {
        "default": "frank",
        "help": "Enter the name to use",
    },
}
```

This simple example creates a config setting called `name` and sets sets the documentation for the configuration value and what the default value should be.

If we then wanted to allow `name` to be set from the command-line, we would add a companion entry to `CLI_CONFIG`, which might look like this:

```
CLI_CONFIG = {
    "name": {}
}
```

We could now set the value of `name` via the `--name foo` or `--name=foo` option.

It is also possible that we may have things in `CONFIG` that simply aren't settable via the command-line and are never intended to be. These options can be overridden using `pop-config` configuration files (see Using Configuration Files).

4.1 Basic Settings

Nearly every config setting needs to have 2 basic options, *default* and *help*. These are very self explanatory, *default* sets the default value of the option if no option is passed and *help* presents, not only the command line help, but is also the single source of documentation for the option.

Here is a simple example:

```
CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

This establishes the basic data for the setting and is all that is needed for settings in the *CONFIG* dictionary.

4.2 Destination

When the argument is named "test" it will appear on the option namespace as "test". This may not always be desirable. If the name of the option and where it needs to be stored differs, then use the *dest* option:

```
CONFIG = {
    "test": {
        "default": "Red",
        "dest": "cheese",
        "help": "What color to test",
    },
}
```

In this example the option will be stored under the name "cheese", accessible at `hub.OPT["my_project"].cheese`.

4.3 Location

Once the config system has been run, all configuration data will appear in the *hub.OPT* namespace. This means that in our above example, if the system in question is part of an app named *myapp*, then the option data will be present at *hub.OPT["myapp"]["test"]*. Because we use special dictionaries, it is also possible to access this value as `hub.OPT.myapp.test`. This works fine as long as there are no hyphens in your project name or option, in which case the index-based access method can be used.

CHAPTER 5

CLI_CONFIG Dictionary

The `CLI_CONFIG` dictionary is used to expose a setting on the command-line of your application so it can be changed. Any values set in `CLI_CONFIG` will inherit settings from `CONFIG`, so that if you have a `name` key in both dictionaries, the final settings for the command-line options will inherit the `CONFIG` settings, too.

All options that appear on the CLI need to be activated in the `CLI_CONFIG` but the basic configuration needs to be in the `CONFIG` dictionary.

`Pop-config` uses Python's venerable *argparse* under the hood to present and process the arguments. Pop-config will also, transparently, passes options from the dictionary into *argparse*, this makes Pop-config transparently compatible with new *argparse* options that are made available.

This document is intended, therefore, to present the most commonly used options, please see the *argparse* doc for more in depth data on available options.

If the command-line option is very simple it can be as simple as just doing this:

```
CLI_CONFIG = {
    "test": {},
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

This would present a command-line option `--test` which can be specified as `--test=red` or `--test blue`.

This document will cover all available options for the `CLI_CONFIG`, remember that the *default* and *help* values should always be in the `CONFIG` section. This is a POP best practice.

5.1 Options

By default the options presented on the command line are identical to the name of the value. So for the above example the presented option would be *–test*. If alternative options are desired, they can be easily added:

```
CLI_CONFIG = {
    "test": {
        "options": ["-t", "--testy-mc-tester", "-Q"],
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

5.2 Positional Arguments

Positional arguments are very common and can create a much more user friendly experience for users. Adding positional arguments are easy. Just use the *positional* argument:

```
CLI_CONFIG = {
    "test": {
        "positional": True,
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

It would now be possible to specify the argument as `mycmd foo` (without needing a `--test`).

When working with multiple positional arguments the *display_priority* flag can be used to control their order:

```
CLI_CONFIG = {
    "test": {
        "positional": True,
        "display_priority": 2,
    },
    "run": {
        "positional": True,
        "display_priority": 1,
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
```

(continues on next page)

(continued from previous page)

```
    },
    "run": {
        "default": "green",
        "help": "What color to run",
    },
}
```

In the above example the first argument will be *run* and the second will be *test*.

5.3 Accepting Environment Variables

Operating systems allow for configuration options to be passed in via specific means. In Unix based systems like Linux and MacOS, environment variables can be used. In Windows based systems the registry can be used. To allow for an os variable to be used just add the `os` option:

```
CLI_CONFIG = {
    "test": {
        "os": "MYAPP_TEST",
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

Now the flag can be set by setting the environment variable `MYAPP_TEST` to the desired configuration value.

5.4 Actions

Actions allow a command line argument to perform an action, or flip a switch.

The *action* option passes through to *argparse*, if the examples in this document do not make sense you can also check the *argparse* section on *action*.

A number of actions are supported by *argparse*. Arguable the most frequently used actions are *store_true* and *store_false*:

```
CLI_CONFIG = {
    "test": {
        "action": "store_true",
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

A few other useful actions are *append* and *count*. If *append* is used then every time the argument is used the option passed to the argument is appended to the final list. The *count* option allows for the number of times that the argument is passed to be counted up. This is useful for situations where you want to specify what the verbosity of the output should be, so that you can pass *-vvv* in a similar fashion to ssh.

5.5 Number of Arguments

The number of arguments that should be expected can also be set using the *nargs* option. This allows for a specific or fluid number of options to be passed into a single cli option.

The *nargs* option passes through to *argparse*, if the examples in this document do not make sense you can also check the *argsparse* section on *nargs*.

5.5.1 Integer (1)

Specifying an integer defines the explicit number of options to require:

```
CLI_CONFIG = {
    "test": {
        "nargs": 3,
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

The above example will require that exactly 3 options are passed to *–test*.

5.5.2 Question Mark (?)

One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced.

5.5.3 Asterisk (*)

All command-line arguments present are gathered into a list.

```
CLI_CONFIG = {
    "test": {
        "nargs": "*",
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
```

(continues on next page)

(continued from previous page)

```
    },
}
```

5.5.4 Plus (+)

Just like '*', all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line argument present.

5.6 Type

The value type can be enforced with the *type* option. A type can be passed in that will be enforced, such as *int* or *str*.

```
CLI_CONFIG = {
    "test": {
        "type": int,
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

5.7 Render

Sometimes it is desirable to load up complex data structures from the command line. This can be done with the *render* option. The *render* option allows you to specify that the argument passed will be rendered using a data serialization medium such as json or yaml.

```
CLI_CONFIG = {
    "test": {
        "render": "yaml",
    },
}

CONFIG = {
    "test": {
        "default": "Red",
        "help": "What color to test",
    },
}
```

This cli could then look like this:

```
myapp --test "Food: true"
```

Then the resulting value would be: *{"Food": True}*

CHAPTER 6

SUBCOMMANDS Dictionary

Sometimes it is desirable to have subcommands. Subcommands allow your CLI to work in a way similar to the git cli, where you have multiple routines that all can be called from a single command.

This example shows how multiple subcommands can be defined and utilized.

```
CLI_CONFIG = {
    "name": {
        "subcommands": ["test", "apply"],
    },
    "weight": {},
    "power": {
        "subcommands": ["apply"],
    },
}
CONFIG = {
    "name": {
        "default": "frank",
        "help": "Enter the name to use",
    },
    "weight": {
        "default": "150",
        "help": "Enter how heavy it should be",
    },
    "power": {
        "default": "100",
        "help": "Enter how powerful it should be",
    },
}

SUBCOMMANDS = {
    "test": {
        "help": "Used to test",
        "desc": "When running in test mode, things will be tested",
    },
    "apply": {
```

(continues on next page)

(continued from previous page)

```
        "help": "Used to apply",
        "desc": "When running in apply mode, things will be applied",
    },
}
```

In this example we see that the option *name* will be available under the subcommands *test* and *apply*. The option *power* will be available only under the subcommand *apply* and the option *weight* is globally available.

6.1 Detecting the Subparser

When the subparser is used the desired subparser is set on the hub as the variable *hub.SUBPARSER*. This makes it easy to know what subparser is being used anywhere in your code:

```
def run(hub):
    if hub.SUBPARSER == "test":
        run_test()
    elif hub.SUBPARSER == "apply":
        run_apply()
```

6.1.1 Global Subcommand Options

Sometimes an option should be made available to all subcommands, including the root of the command. It is easy to do this! Just add the option *_global_* to the list of subcommands.

```
CLI_CONFIG = {
    "name": {
        "subcommands": ["_global_"],
    },
}
CONFIG = {
    "name": {
        "default": "frank",
        "help": "Enter the name to use",
}

SUBCOMMANDS = {
    "test": {
        "help": "Used to test",
        "desc": "When running in test mode, things will be tested",
    },
    "apply": {
        "help": "Used to apply",
        "desc": "When running in apply mode, things will be applied",
    },
}
```

In the above example, the *–name* option is made available to the root and all subcommands.

CHAPTER 7

Advanced Topics

7.1 Root Rewriting with *root_dir*

If you define a *root_dir* config option in your main application, the root rewriting system is enabled (but is not necessarily active) for your application and all its plugins.

The default value for *root_dir* should be "/".

Explicitly setting *root_dir=/* at run time guarantees that root rewriting will not be activated.

7.1.1 Activating Root Rewriting

Eligible config options are modified if *root_dir* is set either manually or automatically.

Setting *root_dir* via config file, environment variable or command line takes priority.

If *root_dir* is not set and the program is run as a non-root user, *root_dir* is automatically set to ".{program_name}" in the user's home directory.

7.1.2 Eligible Config Options

Config options for your program (including its plugins) may be rewritten if the following are all true:

- The config option name ends in _dir, _path or _file
- The config option has a default that is an absolute path
- The config option has a default that includes its pop project name as a path component

Assuming your project is named "foo":

```
CONFIG = {
    "root_dir": {"default": "/"},
    "a_dir": {"default": "/var/log/foo"},
```

(continues on next page)

(continued from previous page)

```
    "b_file": {"default": "/var/log/foo"},
    "c_path": {"default": "/var/log/foo/"},

    "w": {"default": "/var/log/foo"},  # doesn't end in _dir/_path/_file
    "x_dir": {"default": "/var/log/foobar"},  # foo isn't a path component
    "y_dir": {"default": "/var/log/"},  # project/plugin name not in path
    "z_dir": {"default": "path/to/foo"},  # not an absolute path
}
```

Variables a-c may be rewritten, w-z will never be automatically rewritten by the root system.

When rewritten, the new root_dir is prepended to variables. If you set *root_dir* to myroot, options will be set as if these were your defaults:

```
CONFIG = {
    "root_dir": {"default": "myroot"},
    "a_dir": {"default": "myroot/var/log/foo"},
    "b_file": {"default": "myroot/var/log/foo"},
    "c_path": {"default": "myroot/var/log/foo/"},
}
```

A explicitly set option via config file, environment variable or command line will override these rewritten values, if you set *root_dir=myroot* and *a_dir=/var/log/foo*, *a_dir* will be preserved as if this were your config block:

```
CONFIG = {
    "root_dir": {"default": "myroot"},
    "a_dir": {"default": "/var/log/foo"},
    "b_file": {"default": "myroot/var/log/foo"},
    "c_path": {"default": "myroot/var/log/foo/"},
}
```

7.1.3 Disabling Root Rewriting

If you are a developer and do not want to enable the roots system, do not provide a *root_dir* config option. If you want to use *root_dir* in your project without root rewriting, consider naming it *root* or *root_path*.

If you are a user and do not want default paths to be rewritten, explicitly set *root_dir* to "*/*", the default value.

Similarly, if you explicitly set any *_dir*, *_path* or *_file* config variable to the default value, it will not be rewritten.

CHAPTER 8

App Merging

8.1 Source

By default the *CLI_CONFIG* references the local *CONFIG* setting. The *source* option allows you to reference a documented configuration from a separate project configuration. This powerful option allows you to manage the arguments and flags in a namespace of an app that is being merged into this app. The benefit here is that the *CONFIG* values do not need to be rewritten and you maintain a single authoritative source of documentation.

When using *source* in the *CLI_CONFIG* the namespace that defined the option in the *CONFIG* dictionary will own the option. This makes it easy to have an application that uses its own config namespace be app merged into another application that can then transparently manage the configuration of the merged app.

Therefore, if we have 2 projects' conf.py files, one can reference the other. The source option references the project name. So if the first file is in project "test" and the second file is for project "other", an argument can reference the conf.py in project "other":

test .. code-block:: python

CLI_CONFIG = { "test": {}, "oranges": {

"source": "other",

},

}

CONFIG = {

"test": { "default": "Red", "help": "What color to test",

},

}

other

```
CLI_CONFIG = {
}

CONFIG = {
    "oranges": {
        "default": "Many",
        "help": "The amount of oranges to enjoy.",
    },
}
```

8.2 Dyne

A powerful option in the *CLI_CONFIG* is *dyne*. This uses vertical app merging to modify another application's cli options. This allows a vertical app merge repo to define cli arguments that will be made available when the plugins are installed to extend an external app.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search