Cement Framework

Release 2.11.1

Data Folk Labs, LLC

Aug 18, 2017

Contents

Warning: This documentation is for the development version of Cement 2.11.x. For production please use, and reference the current stable version of Cement 2.8.x until this version is officially released as stable/2.12.x.

Cement is an advanced CLI Application Framework for Python. Its goal is to introduce a standard, and feature-full platform for both simple and complex command line applications as well as support rapid development needs without sacrificing quality. Cement is flexible, and it's use cases span from the simplicity of a micro-framework to the complexity of a mega-framework. Whether it's a single file script, or a multi-tier application, Cement is the foundation you've been looking for.

The first commit to Git was on Dec 4, 2009. Since then, the framework has seen several iterations in design, and has continued to grow and improve since it's inception. Cement is the most stable, and complete framework for command line and backend application development. Core features include (but are not limited to):

- Core pieces of the framework are customizable via handlers/interfaces
- Extension handler interface to easily extend framework functionality
- Config handler supports parsing multiple config files into one config
- Argument handler parses command line arguments and merges with config
- Log handler supports console and file logging
- Plugin handler provides an interface to easily extend your application
- Hook support adds a bit of magic to apps and also ties into framework
- Handler system connects implementation classes with Interfaces
- Output handler interface renders return dictionaries to console
- Cache handler interface adds caching support for improved performance
- Controller handler supports sub-commands, and nested controllers
- Zero external dependencies* of the core library
- 100% test coverage using `nose` and `coverage`
- 100% PEP8 and style compliant using `flake8`
- Extensive Sphinx documentation
- Tested on Python 2.6, 2.7, 3.3, 3.4, 3.5

Note that argparse is required as an external dependency for Python < 2.7 and < 3.2. Additionally, some optional extensions that are shipped with the mainline Cement sources do require external dependencies. It is the responsibility of the application developer to include these dependencies along with their application if they intend to use any optional extensions that have external dependencies, as Cement explicitly does not include them.

CHAPTER 1

Getting More Information

- DOCS: http://builtoncement.com/2.11/
- CODE: http://github.com/datafolklabs/cement/
- PYPI: http://pypi.python.org/pypi/cement/
- SITE: http://builtoncement.com/
- T-CI: http://travis-ci.org/datafolklabs/cement
- HELP: cement@librelist.org - #cement - gitter.im/datafolklabs/cement

CHAPTER 2

Mailing List

Sign up for the Cement Framework mailing list to recieve updates regarding new releases, important features, and other related news. This not an open email thread, but rather an extremely minimal, low noise announcement only list. You can unsubscribe at any time.

CHAPTER 3

Documentation

ChangeLog

All bugs/feature details can be found at:

> https://github.com/datafolklabs/cement/issues/XXXXX

Where XXXXX is the 'Issue #' referenced below. Additionally, this change log is available online at:

> http://builtoncement.com/2.8/changes/

See the *Upgrading* section for more information related to any incompatible changes, and how to update your application to fix them. Also check out the *What's New* section for details on new features.

2.11.1 - DEVELOPMENT (will be released as dev/2.11.2 or stable/2.12.0)

This is a branch off of the 2.10.x stable code base. Maintenance releases for 2.10.x will happen under the stable/2.10.x git branch, while forward feature development will happen as 2.11.x under the git master branch.

Bugs:

- Issue #397 - `[ext.logging]` Removes deprecated `warn` from ILog validator, in-favor of `warning`
- Issue #401 - `[ext.daemon]` Can't get user in daemon extension
- Issue #415 - `[core]` FrameworkError when reusing CementApp object

Features:

- None

Refactoring:

- Issue #378 - `[tests]` Refactor dto comply with Flake8
- Issue #418 - `[ext.daemon]` Ability to daemonize without `--daemon`

Incompatible:

- None

Deprecation:

- None

2.10.2 - Thu July 14, 2016

Bumping version due to issue with uploading to PyPi.

2.10.0 - Thu July 14, 2016

Bugs:

- Issue #363 - CementTestCase does not delete temporary files/directories
- Issue #346 - AttributeError: 'module' object has no attribute 'SIGHUP' on Windows
- Issue #352 - `CementApp.extend()` breaks on `CementApp.reload()`
- Issue #366 - Output handler override options dissappear
- Issue #385 - JsonOutputHandler/YamlOutputHandler/MustacheOutputHandler Do not pass keyword args down to backend render functions
- Issue #393 - `CementApp.Meta.hooks` ignored for hooks defined by extensions

Features:

- Issue #350 - Support for plugin directories
- Issue #370 - Handlebars templating support
- PR #371 - Jinja2 templating support
- Issue #373 - Switch over to using Flake8 for PEP8 and style compliance
- PR #375 - Redis cache handler support
- Issue #379 - Support for alternative config file extensions
- Issue #380 - Support for Cython/Compiled Plugins
- Issue #389 - ConfigObj support for Python 3
- Issue #394 - Watchdog extension for cross-platform filesystem event monitoring
- Issue #395 - Ability to pass metadata keyword arguments to handlers via `CementApp.Meta.meta_defaults`.

Refactoring;

- Issue #386 - Partially deprecated use of `imp` in favor of `importlib` on Python >= 3.1
- Issue #390 - ArgparseArgumentHandler should store unknown arguments

Incompatible:

- None

Deprecation:

- Issue #365 - Deprecated `LoggingLogHandler.warn()`
- Issue #372 - Deprecated Explicit Python 3.2 Support

- Issue #376 - Deprecated `cement.core.interface.list()`

2.8.0 - Wed Feb 24, 2016

Bugs:

- Issue #310 - Resolved issue where console/file logs had duplicate entries under certain circumstances.
- Issue #314 - Resolved inconsistent behavor in `colorlog` extension.
- Issue #316 - Running Nose tests with `-s` option causes traceback
- Issue #317 - Calling `CementApp.run()` should return results from `Controller._dispatch()`
- Issue #320 - Partials not rendering with Mustache templates
- Issue #325 - Wrap Utility Doesn't Support Unicode
- Issue #328 - Post Run Hook Not Executing
- Issue #331 - KeyError: 'USER' When Running Nose Tests
- Issue #338 - Support non-ascii encoding for email subject and body

Features:

- Issue #205 - Added new `ArgparseController` and `expose` decorator in `ext_argparse` to eventually replace `CementBaseController`.
- Issue #299 - Added Argcomplete Framework Extension
- Issue #322 - Added Tabulate Framework Extension (tabulatized output)
- Issue #336 - Added Support for `CementApp.reload()` (SIGHUP)
- Issue #337 - Added `app.run_forever()` alternative run method.
- Issue #342 - Added Alarm/Timeout Support
- Issue #343 - Memcached Extension/Pylibmc Now Supports Python 3

Refactoring:

- Issue #311 - Refactor Hooks/Handlers into CementApp
- Issue #319 - Use `os.devnull` instead of internal `NullOut` hack.

Incompatible:

- Issue #311 - Hook `signal` now requires `app` argument.
- Issue #313 - Default `Cement.Meta.exit_on_close` to `False`. Calls to `sys.exit()` should be explicit by the app developer, and not implied by the framework.
- Issue #332 - Output Handler Interface Must Support Keyword Arguments

2.6.0 - Thu May 14, 2015

Bugs:

- Issue #294 - Added work-around for scenario where an app wants to support arbitrary positional argument with a value of `default`. By default, this will attempt to explicitly call the `default` command rather than using `default` as the argument. This fix adds `CementBaseController.Meta.default_func` allowing the developer to override the name of the default function that is called if no matching sub-command is passed.

Features:

- Issue #197 - Added support for colorized logging.
- Issue #281 - Added support for Python *with* statement.
- Issue #282 - Added support to define/register hooks and handlers via `CementApp.Meta`.
- Issue #290 - Added ability to disable Cement framework logging via `CementApp.Meta.framework_logging = False`.
- Issue #297 - Added **experimental** support for reloading configurations anytime config files and/or plugin config files are modified. Optional extension *ext_reload_config*.

Incompatible:

- Issue #308 - No longer require explicit `CementApp.Meta.base_controller` if a controller with the label of `base` is registered. This is potentially backward in-compatible in that previously `CementBaseController.Meta.label` defaulted to `base`. It now defaults to `None`, which makes more sense but will break for any controllers that have not explicitly set a `label` of `base`.

2.4.0 - Wed Sep 17, 2014

Bugs:

- Issue #211 - LoggingLogHandler namespace causes issues (regression)
- Issue #235 - Duplicate Config Section when plugin already loaded
- Issue #246 - Plugin extension does not disable after already being enabled

Features:

- Issue #119 - Added cement.utils.shell.Prompt to quickly gather user input in several different ways.
- Issue #182 - Added a mail interface with basic implementations of 'dummy' (just prints message to console) and 'smtp'
- Issue #229 - Ability to override handlers via command line options. Also related: Issue #225.
- Issue #248 - Added documentation for BASH Auto-Completion example.
- Issue #249 - Allow utils.shell.exec_cmd* to accept additional. parameters, passing all *args* and *kwargs* down to subprocess.Popen. Allows features such as changing the current working directory, and setting a timeout value, etc (everything that Popen supports).
- Issue #257 - CementBaseController._dispatch() should return result of controller function.
- Issue #259 - Add yaml and yaml_configobj config handlers.
- Issue #262 - Add json and json_configobj config handlers.
- Issue #267 - Add support for multiple *plugin_dirs* and *plugin_config_dirs*
- Issue #269 - Allow app.close() to accept an exit code, and exit with that code.
- Issue #270 - Add support for multiple template_dirs
- Issue #274 - Add cement.core.interface.list function
- Issue #275 - Added *examples/* directory with working examples based on Examples section of the documentation.

Incompatible:

- Issue #227 - Standardize handler config section naming conventions. All handler config sections are now labeled after `interface.handler` (i.e. `output.json`, or `mail.sendgrid`, etc).
- Issue #229 - Handler override options deprecate the use of `--json`, and `--yaml` output handler options.
- Issue #260 - Extensions/Plugins/Bootstrap modules must accept *app* argument in *load()* function. See Upgrading doc for more information.

2.2.0 - Wed Jan 29, 2014

Bugs:

- Issue #211 - LoggingLogHandler namespace causes issues
- Issue #215 - Epilog not printed on –help

Features:

- Issue #209 - Added app.debug property to allow developers to know if *–debug* was passed at command line of via the config
- Issue #219 - Merged ext.memcached into mainline
- Issue #222 - Merged ext.configobj into mainline
- Issue #223 - Merged ext.genshi into mainline
- Issue #224 - Merged ext.yaml into mainline

Incompatible:

- Issue #202 - Deprecated namespace packaging for cement and cement.ext. Resolves issues with certain IDE's and other situations where the lack of a proper `__init__.py` causes issues. This change means that external extensions can no longer share the `cement.ext` module namespace, and must have it's own unique module path.

Misc:

- Official Git repo relocated to: http://github.com/datafolklabs/cement

2.1.4 - Tue Oct 29, 2013

Bugs:

- Issue #191 - KeyError when using minimal logger with –debug
- Issue #199 - Moved post_argument_parsing hook down, after arguments_override_config logic happens.
- Issue #204 - utils.misc.wrap should handle None type
- Issue #208 - LoggingLogHandler does not honor Meta.config_section

Features:

- Issue #190 - Merged daemon extension into core
- Issue #194 - Added pre_argument_parsing/post_argument_parsing hooks
- Issue #196 - Added utils.misc.wrap
- Issue #200 - Merged ext.mustache into mainline.
- Issue #203 - Added support for external template directory
- Issue #207 - Added support for alternative 'display' name for stacked controllers

Incompatible:

- Issue #163 - LoggingLogHandler.Meta.clear_loggers was changed from a boolean option, to a list. Additionally, LoggingLogHandler.clear_loggers() now requires a *namespace* argument. ILog interface no longer defines *clear_loggers()* as a required function (though ILog interfaces are welcome to implement one if necessary).
- Issue #173 - Deprecated 'has_key()' from configparser extension
- Issue #201 - Add Deprecation Warning for CementApp.get_last_rendered()

2.1.2 - Thu Nov 1st, 2012

This is a branch off of the 2.0.x stable code base. Maintenance releases for 2.0.x will happen under the stable/2.0.x git branch, while forward feature development will happen here under as 2.1.x under the git master branch.

Bugs:

- Issue #162 - Unable to set log 'level' by config
- Issue #167 - Non-stacked controllers not listed in –help
- Issue #169 - Fixed tests.utils.shell_tests timeout issue
- Issue #171 - No handlers could be found for logger
- Issue #183 - os.environ['HOME'] does not exist on Windows

Features:

- Issue #161 - Ability to override *usage*
- Issue #164 - Store previously rendered data as app._last_rendered, and retrievable by app.get_last_rendered().
- Issue #165 - Allow utils.fs.backup() to support a suffix kwarg
- Issue #166 - Ability to set the 'app' for CementTestCase.make_app()
- Issue #170 - Added support for *nested* and *embedded* controllers.

Misc:

- Issue #172 - 100% PEP8 Compliant
- Issue #160 - Refactor CementApp._resolve_handler() as handler.resolve()

Deprecation Notices:

- Issue #173 - ConfigParserConfigHandler.has_key() is now deprecated, and will be removed in future versions. Please use *if key in app.config.keys(section)* instead.

Incompatible Changes:

- Issue #141 - Removed shortcuts from CementBaseController (such as log, pargs, etc). Use *self.app.<shortcut>* instead.
- Issue #167 - Listed above, in order to fix this issue as well as restrict future issues we implemented a hard requirement that all base controllers have the label 'base'. This should not be a major change for anyone using Cement 2.0.x as it is a simple 1 line change in any one application. As all documentation encourages the use of the label 'base' it should not be a wide spread incompatibility.
- Issue #179 - CementBaseController *hidden*, *visible*, and *exposed* have been removed, and replaced by other private means of managing the dispatch of commands.
- CementBaseController no longer implements a *default* command.

- Issue #177 - Renamed several cement.core.backend pieces including: *handlers* -> *__handlers__*, *hooks* -> *__hooks__*, *SAVED_STDOUT* -> *__saved_stdout__*, and *SAVED_STDERR* -> *__saved_stderr__*.
- Issue #178 - Moved *backend.defaults()* to *utils.misc.init_defaults()*, and *backend.minimal_logger()* to *utils.misc.minimal_logger()*

2.0.0 - Fri Aug 03, 2012

This is the initial stable release of the 2.0.x branch. Future releases of this branch will include bug/security fixes and minor feature updates. Forward moving feature development will continue out of the 2.1.x branch.

Bugs:

- Issue #143 - Incorrect doc regarding logging. The LoggingLogHandler now supports an optional 'namespace' argument allowing the developer to override the log prefix.
- Issue #150 - foundation.CementApp.Meta.argv now defaults to None, but is overridden in __init__() with sys.argv if no other argv is passed as meta.

Features:

- Issue #138 - Added 'shell' argument to utils.shell.exec_cmd() and utils.shell.exec_cmd2().
- Issue #140 - Included ~/.myapp/config in default config search
- Issue #144 - Added a note on Contributing, as well as a CONTRIBUTORS file.
- Issue #152 - Added 'argument_formatter' to ControllerBaseClass.Meta.
- Issue #153 - Added spawn_process() and spawn_thread() to utils.shell.

Incompatible Changes:

- **Issue #100 - Interfaces audit.**
  - ILog.level() redefined as ILog.get_level()
  - IPlugin.loaded_plugins attribute redefined as IPlugin.get_loaded_plugins()
  - IPlugin.enable_plugins attribute redefined as IPlugin.get_enabled_plugins()
  - IPlugin.disabled_plugins attribute redefined as IPlugin.get_disabled_plugins()
- Issue #145 - The IArgument interface no longer specifies that the *parsed_args* be maintained by the handler implementation. This means that *app.args.parsed_args* is no longer available, however *app.pargs* functions just the same as it points to *app._parsed_args*.
- **Issue #148 - The CementExtensionHandler.loaded_extensions property is** now a callable at CementExtensionHandler.get_loaded_extensions().
- **Issue #151 - Removed all previously deprecated code including:**
  - Removed CementApp.Meta.defaults, and CementBaseHandler.Meta.defaults. Now use *config_defaults* instead.
  - Removed cement.foundation.lay_cement(). Now use CementApp directly.
  - Removed cement.handler.enabled(). Now use *handler.registered()* instead.
- Issue #154 - Validate for handler config_defaults and config_section by default. Only incompatible if not previously sub-classing from handler.CementBaseHandler.
- Issue #157 - CementRuntimeError renamed as FrameworkError
- Issue #158 - CementSignalError renamed as CaughtSignal

- Issue #159 - CementInterfaceError renamed as InterfaceError

Misc:

- Issue #155 - Removed unused CementArgumentError, and CementConfigError. These types of exceptions should be defined at the application level.

1.9.14 - Sun Jul 16, 2012

Bugs:

- Issue #127 - Inherited positional arguments listed multiple times

Feature Enhancements:

- Issue #131 - Controller aliases
- Issue #126 - Add a 'bootstrap' importer to CementApp
- Added cement.utils.test.CementTestCase for improved testing.

Incompatible Changes:

- **Issue #129 - Simplify extensions/plugins/hooks/etc.**
  - Hooks renamed from 'cement_xxx_hook' to just 'xxx'. For example, the 'cement_pre_setup_hook' is now simply 'pre_setup'. Additionally, the 'cement_on_close_hook' is now broken out into 'pre_close', and 'post_close'.
  - The cement.core.hooks.register decorator was replaced with a simple function of the same name. New usage is: register('hook_name', hook_func).
  - Plugins, extensions, and the application bootstrap now attempt to call a 'load()' function, meaning library code and loading code can live in the same file (i.e. hooks won't be registered just because you imported an extension to sub-class a handler, etc).
- cement.utils.test_helper moved to cement.utils.test.
- By default, command line arguments no longer override config settings. This is now configurable by the CementApp.Meta.arguments_override_config boolean. Related: Issue #136.

1.9.12 - Thu Jul 05, 2012

Bugs:

- Fixed version mis-hap in setup.py

1.9.10 - Wed Jul 04, 2012

Feature Enhancements:

- Issue #118 - Added utils.fs.backup() to safely backup files/dirs.

Misc:

- Issue #111 - Use relative imports (makes cement more portable as it can be included and distributed with 3rd party sources).
- Issue #120 - Use standard json rather than relying on jsonpickle

Incompatible Changes:

- core.util.abspath moved to utils.fs.abspath
- core.util.is_true moved to utils.misc.is_true
- Namespace reverted from 'cement2' back to 'cement'.
- The following extensions have been removed from the cement source tree, and are now available externally (see: http://github.com/cement): daemon, memcached, configobj, yaml, genshi.

1.9.8 - Thu May 3, 2012

Feature Enhancements:

- Issue #95 - Add a 'config_section' Meta default for all handlers. Required to parse config options for a handler.
- Issue #97 - Add a standard cache handler interface.
- Issue #105 - Add 'meta_override' and 'core_meta_override' list to CementApp().Meta. Also resolves Issue #104.
- Issue #108 - Add CementApp.extend() functionality.
- Issue #109 - Add cement.ext.memcached extension

Incompatible Changes:

- Issue #103 - plugin_bootstrap_module renamed as plugin_bootstrap.
- Issue #106 - Deprecate Meta.defaults in favor of Meta.config_defaults
- Issue #107 - Make the app name the default config section, not [base]

1.9.6 - Wed Apr 18, 2012

Bug Fixes:

- Issue #89 - Secondary controllers display under other controllers
- Issue #90 - Logging to file doesn't expand '~'
- Issue #91 - Logging to file doesn't create basedir

Feature Enhancements:

- Issue #88 - Add cement.ext.genshi extension, provides Genshi Text Templating Launguage support.
- Issue #93 - Add epilog support for CementBaseController.

Refactoring:

- Issue #87 - Refactor Meta handling

Incompatible Changes:

- Issue #96 - Move 'setup()' functions to '_setup()'
- Moved CementBaseController.dispatch() to _dispatch()
- Moved CementBaseController.usage_text to _usage_text()
- Moved CementBaseController.help_text to _help_text()
- backend.defaults() no longer accepts an app name as an argument.
- foundation.lay_cement() is deprecated. Use foundation.CementApp() directly.

- No longer pass anything but 'app' object to handlers _setup() functions.
- handler.enabled() is deprecated. Use handler.registered().

1.9.4 - Wed Dec 21, 2011

Bug Fixes:

- Issue #73 - Hooks broken in Python 3
- Issue #81 - Controller defaults should be processed before base controller.setup()

Feature Enhancements:

- Issue #65 - Added 'daemon' extension. Process is daemonized by passing the '–daemon' option. Handles switching the run user/group, as well as managing a pid file.
- Issue #72 - Added new framework hooks.
- Issue #76 - Added app.close() functionality including a cement_on_close_hook() allowing plugins/extensions/etc to be able to cleanup on application exit.
- Issue #77 - Added default signal handler for SIGINT/SIGTERM as well as the cement_signal_hook which is called when any catch_signals are encountered.
- Issue #78 - Added cement_pre_render_hook, and cement_post_render_hook allowing developers to control the data that is rendered to console.
- Issue #84 - Ability to run all tests from utils/run_tests.sh

Incompatible Changes:

- Issue #72 - The framework hooks 'cement_add_args_hook' and 'cement_validate_config' were removed in favor of the new pre/post setup and run hooks.
- Issue #82 - Change 'meta' classes to Python-proper 'Meta', and interfaces to use 'IMeta'. Old functionality will be completely removed before Cement stable release.

1.9.2 - Wed Nov 02, 2011

This is an initial beta release of Cement, and therefore no bugs or features are listed. Future releases will detail all changes.

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Data Folk Labs, LLC. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Contributors

The following people have contributed to Cement, either by way of source code, documentation, or testing.

- BJ Dierkes (derks) - Creator, Primary Maintainer
- Kyle Rockman (rocktavious)
- Tomasz Czyż (spinus)
- Ildar Akhmetgaleev (akhilman)
- Nicolas Brisac (zacbri)
- Subhash Bhushan (subhashb)

Upgrading

This section outlines any information and changes that might need to be made in order to update your application built on previous versions of Cement.

Upgrading from 2.8.x to 2.9.x

Cement 2.9 introduces a few incompatible changes from the previous 2.8 stable release, as noted in the *ChangeLog*.

Deprecated: cement.core.interface.list()

This function should no longer be used in favor of `CementApp.handler.list_types()`. It will continue to work throughout Cement 2.x, however is not compatible if `CementApp.Meta.use_backend_globals == False`.

Related:

- Issue #366
- Issue #376

Upgrading from 2.6.x to 2.8.x

Cement 2.8 introduced a few incompatible changes from the previous 2.6 stable release, as noted in the *ChangeLog*.

TypeError: my_signal_hook() takes exactly 2 arguments (3 given)

In Cement 2.6, functions registered to the `signal` hook were only expected/required to accept the `signum` and `frame` arguments, however `signal` hook functions must now also accept the `app` object as an argument as well.

After upgrading to Cement 2.8, you might receive something similar to the following exception:

```
TypeError: my_signal_hook() takes exactly 2 arguments (3 given)
```

The fix is to simply prefix any `signal` hook functions with an `app` argument.

For example:

```
def my_signal_hook(signum, frame):
    pass
```

Would need to be:

```
def my_signal_hook(app, signum, frame):
    pass
```

Related:

- Issue #311

TypeError: render() got an unexpected keyword argument

In Cement 2.6, output handlers were not required to accept `**kwargs`, however this is now required to allow applications to mix different types of output handlers together that might support different features/usage.

After upgrading to Cement 2.8, you might receive something similar to the following exception:

```
TypeError: render() got an unexpected keyword argument
```

This would most likely be the case because you have created your own custom output handler, or are using a third-party output handler that has not been updated to support Cement 2.8 yet. The fix is to simply add `**kwargs` to the end of the *render()* method.

For example:

```
def render(self, data):
    pass
```

Would need to be:

```
def render(self, data, **kwargs):
    pass
```

CementApp.Meta.exit_on_close Defaults to False

In Cement 2.6, the feature to call `sys.exit()` when `app.close()` is called was implemented, however defaulting it to `True` is not the ideal behavior. The default is now `False`, making it the developers option to explicitly enable it.

To revert the change, and default `exit_on_close` to `True`, simply set it in `CementApp.Meta.exit_on_close`:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        exit_on_close = True
```

Upgrading from 2.4.x to 2.6.x

Cement 2.6 introduced a few incompatible changes from the previous 2.4 stable release, as noted in the *ChangeLog*.

InterfaceError: Invalid handler ... missing '_meta.label'.

Prior to Cement 2.5.2, `CementBaseController.Meta.label` defaulted to `base`. The new default is `None`, causing the potential for breakage of a controller that did not explicity set the `label` meta option.

You can resolve this error by explicity setting the `label` meta option:

```
class MyBaseController(CementBaseController):
    class Meta:
        label = 'base'
```

Upgrading from 2.2.x to 2.4.x

Cement 2.4 introduced a few incompatible changes from the previous 2.2 stable release, as noted in the *ChangeLog*.

Related:

- Issue #308

CementApp.render() Prints Output Without Calling print()

Before Cement 2.3.2 the `app.render()` function did not actually print anything, therefore you would have to call `print app.render()`. This now defaults to writing output to `sys.stdout`, but can be modified for the older behavior by passing `out=None` when calling it:

```
app.render(data, out=None)
```

Additionally, you can also now write directly to a file:

```
myfile = open('/path/to/myfile', 'w')
app.render(data, out=myfile)
myfile.close()
```

error: unrecognized arguments: –json/–yaml

After upgrading to Cement > 2.3.2 you might encounter the error:

```
error: unrecognized arguments: --json
```

Or similar errors like:

```
error: unrecognized arguments: --yaml
```

This is due to a design change, and a new feature allowing the end user to optionally override handlers via command line. Rather than having a unique option for every type of output handler, you now have one option that allows overriding the defined output handler by passing it the handler label.

Note that only handlers that have `overridable = True` in their meta-data will be valid options.

To resolve this issue, you simply need to pass `-o json` or `-o yaml` at command line to override the default output handler.

Related:

- Issue #229

NoSectionError: No section: 'log'

After upgrading to Cement > 2.3.2 you might encounter the error:

```
NoSectionError: No section: 'log'
```

In previous versions of Cement < 2.3.2, the default logging configuration section in the config file was `[log]`. This has been changed to `[log.logging]` in order to be consistent with all other handler configuration sections.

Another issue you might encounter due to the above change is that log related configuration settings read from a configuration file would no longer work. The necessary change to resolve this issue is to change all references of `log` in relation to the log configuration section, to `log.logging`.

Related:

- Issue #227

TypeError: load() takes no arguments (1 given)

After upgrading to Cement > 2.3.2 you might encounter the error:

```
TypeError: load() takes no arguments (1 given)
```

Previous versions of Cement < 2.3.2 did not require an *app* argument to be passed to the *load()* functions of extensions/plugins/bootstrap modules. In Cement > 2.3.2 all extension/plugins/bootstrap modules must accept a single argument named *app* which is the application object in its current state when *load()* is called.

To resolve this issue simply modify all relevant *load()* functions to accept the *app* argument. For example:

```
def load():
    pass
```

To:

```
def load(app):
    pass
```

Upgrading from 2.0.x to 2.2.x

Cement 2.2 introduced a few incompatible changes from the previous 2.0 stable release, as noted in the *Changelog*.

ImportError: cannot import name version

When attempting to install Cement > 2.1 on a system that already has an older version of Cement < 2.1 you will likely run into this error:

```
ImportError: cannot import name version
```

Currently we do not have a way to resolve this programatically in Cement. The resolution is to remove the older version of Cement < 2.1, and then re-install the newer version.

Related:

- Issue #237

FrameworkError: Duplicate Arguments/Commands

After upgrading, you might encounter one or both of the following errors related to application controllers:

```
cement.core.exc.FrameworkError: Duplicate command named 'mycommand' found
in controller '<__main__.MySecondController object at 0x10669ab50>'
```

```
cement.core.exc.FrameworkError: argument -f/--foo: conflicting option
string(s): -f, --foo
```

This is likely due to a change in how application controllers are configured. By default, all controllers are of type *embedded*, meaning that their arguments and commands are added to the parent controller. To resolve this issue you can change the *stacked_type* to *nested*, meaning that the stacked controller will be an additional sub-command under the parent (nesting a new level commands/arguments).

For example:

```
class MyStackedController(CementBaseController):
    class Meta:
        label = 'my_stacked_controller'
        stacked_on = 'base'
        stacked_type = 'nested'
```

Related:

- Issue #234

Discontinued use of Setuptools Namespace Packages

Previous versions of Cement utilitized Setuptools namespace packages in order to allow external libraries (such as optional framework extensions) to use the `cement.ext` namespace. Meaning that an extension packaged separately could use the namespace `cement.ext.ext_myextension` and be imported from the `cement.ext` namespace as if it were shipped with the mainline sources directly. This indirectly caused issues with certain IDE's due to the fact that namespace packages do not install a proper `__init__.py` and are handled differently by Setuptools.

With the move to merging optional extenions into mainline sources, we no longer require the use of Setuptools namespace packages. That said, if a developer had created their own extension using the `cement.ext` namespace, that extension would no longer work or worse may confusing Python into attempting to load `cement.ext` from the extension and not Cement causing even bigger problems.

To resolve this issue, simply change the extension module to anything other than `cement.ext`, such as `myapp.ext`.

Related:

- Issue #202

LoggingLogHandler Changes

The `clear_loggers` meta option is now a `list`, rather than a `boolean`. Therefore, rather than telling LoggingLogHandler to 'clear all previously defined loggers', you are telling it to 'clear only these previously defined loggers' in the list.

If your application utilizied the `LoggingLogHandler.Meta.clear_loggers` option, you would simply need to change it from a `boolean` to a list of loggers such as `['myapp', 'some_other_logging_namespace']`.

Related:

- Issue #163

ConfigParserConfigHandler Changes

The `ConfigParserConfigHandler.has_key()` function has been removed. To update your application for these changes, you would look for all code similar to the following:

```
if myapp.config.has_key('mysection', 'mykey'):
    # ...
```

And modify it to something similar to:

```
if 'mykey' in myapp.config.keys('mysection'):
    # ...
```

Related:

- Issue #173

CementApp Changes

The `CementApp.get_last_rendered()` function has been deprected. Developers should now use the `CementApp.last_rendered` property instead. To update your application for these changes, you would look for all code similar to:

```
CementApp.get_last_rendered()
```

And modify it to something similar to:

```
CementApp.last_rendered
```

Related:

- Issue #201 - Add Deprecation Warning for CementApp.get_last_rendered()

CementBaseController Changes

All short-cuts such as `log`, `pargs`, etc have been removed from CementBaseController due to the fact that these class members could clash if the developer added a command/function of the same name. To update your application for these changes, in any classes that subclass from `CementBaseController`, you might need to modify references to `self.log`, `self.pargs`, etc to `self.app.log`, `self.app.pargs`, etc.

Additionally, if you wish to re-implement these or other shortcuts, you can do so by overriding `_setup()` in your controller code, and add something similar to the following:

```
def _setup(self, *args, **kw):
    res = super(MyClass, self)._setup(*args, **kw)
    self.log = self.app.log
    self.pargs = self.app.pargs
    # etc

    return res
```

An additional change to `CementBaseController` is that the application's `base` controller attached to `YourApp.Meta.base_controller` now must have a label of `base`. Previously, the base controller could have any label however this is now a hard requirement. To update your application for these changes, simply change the label of your base controller to `base`.

Finally, the `CementBaseController` used to have members called `hidden`, `visible`, and `exposed` which were each a list of controller functions used for handling dispatch of commands, and how they are displayed in `--help`. These members no longer exist.

These members were never documented, and is very unlikely that anybody has ever used them directly. Updating your application for these changes would be outside the scope of this document.

Related:

- Issue #141
- Issue #167
- Issue #179

Backend Changes

Several backend pieces have been moved or renamed. For example `cement.core.backend.handlers` is now `cement.core.backend.__handlers__`, etc. The same goes for `cement.core.backend.SAVED_STDOUT` which is now `cement.core.backend.__saved_stdout__`. These are undocumented, and used specifically by Cement. It is unlikely that anyone has used these members directly, and updating your application for these changes is outside the scope of this document. See `cement.core.backend` to assess what, if any, change you may need to change in your code to compensate for these changes.

The `cement.core.backend.defaults()` function has moved to `cement.utils.misc.init_defaults()`. It's usage is exactly the same.

The `cement.core.backend.minimal_logger()` function has moved to `cement.utils.misc.minimal_logger`. It's usage is also the same.

Related:

- Issue #177
- Issue #178

What's New

New Features in Cement 2.9

Support for Multiple File Plugin Directories

Prior to Cement 2.9, application plugins were only supported as single files such as `myplugin.py`. Plugins can now be a single file, or full python modules like `myplugin/__init__.py`.

An example plugin might look like:

```
myplugin/
    __init__.py
    controllers.py
    templates/
        cmd1.mustache
        cmd2.mustache
        cmd3.mustache
```

The only thing required in a plugin is that it supply a `load()` function either in a `myplugin.py` or `myplugin/__init__.py`. The rest is up to the developer.

See *Application Plugins* for more information.

Related:

- Issue #350

Cross Platform Filesystem Event Monitoring via Watchdog

Applications can now monitor, and react to, filesystem events with a very easy wrapper around the Watchdog library. The extension makes it possible to add a list of directories to monitor, and link them with the class to handle any events while automating the proper setup, and teardown of the backend observer.

The Watchdog Extension will make it possible in future releases to properly handle reloading a running application any time configuration files are modified (partially implemented by the *reload_config* extension that has limitations and does not support reloading the app). Another common use case is the ability to reload a long running process any time source files are modified which will be useful for development when working on daemon-like apps so that the developer doesn't need to stop/start everytime changes are made.

See the *Watchdog Extension* for more information.

Related:

- Issue #326
- Issue #394

Ability To Pass Meta Defaults From CementApp.Meta Down To Handlers

Cement handlers are often referenced by their label, and not passed as pre-instantiated objects which requires the framework to instantiate them dynamically with no keyword arguments.

For example:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['json']
```

In the above, Cement will load the `json` extension, which registers `JsonOutputHandler`. When it comes time to recall that handler, it is looked up as `output.json` where `output` is the handler type (interface) and `json` is the handler label. The class is then instantiated without any arguments or keyword arguments before use. If a developer needed to override any meta options in `JsonOutputHandler.Meta` they would **previously** have had to sub-class it. Consider the following example, where we sub-class `JsonOutputHandler` in order to override `JsonOutputHandler.Meta.json_module`:

```
from cement.core.foundation import CementApp
from cement.ext.ext_json import JsonOutputHandler

class MyJsonOutputHandler(JsonOutputHandler):
    class Meta:
        json_module = 'ujson'

def override_json_output_handler(app):
    app.handler.register(MyJsonOutputHandler, force=True)

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['json']
        hooks = [
            ('post_setup', override_json_output_handler)
        ]
```

If there were anything else in the `JsonOutputHandler` that the developer needed to subclass, this would be fine. However the purpose of the above is soley to override `JsonOutputHandler.Meta.json_module`, which is tedious.

As of Cement 2.9, the above can be accomplished more-easily by the following by way of `CementApp.Meta.meta_defaults` (similar to how `config_defaults` are handled:

```
from cement.core.foundation import CementApp
from cement.utils.misc import init_defaults

META = init_defaults('output.json')
META['output.json']['json_module'] = 'ujson'

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['json']
        output_handler = 'json'
        meta_defaults = META
```

When `JsonOutputHandler` is instantiated, the defaults from `META['output.json']` will be passed as `**kwargs` (overriding builtin meta options).

Related:

- Issue #395

Additional Extensions

- *Jinja2* - Provides template based output handling using the Jinja2 templating language
- *Redis* - Provides caching support using Redis backend
- *Watchdog* - Provides cross-platform filesystem event monitoring using the Watchdog library.

- *Handlebars* - Provides template based output handling using the Handlebars templating language

New Features in Cement 2.8

ArgparseController

Work has finally begun, and is mostly complete on the refactoring of `CementBaseController`. The new *`cement.ext.ext_argparse.ArgparseController`* introduces the following improvements:

- Cleaner, and more direct use of `Argparse`
- Does not hijack `Argparse` usage in any way.
- Provides an accessible `sub-parser` for every nested controller, allowing the developer direct access to perform more advanced actions (argument grouping, mutually exclusive groups, etc).
- Provides the ability to define arguments at both the controller level, as well as the sub-command level (i.e. `myapp controller sub-command {options}`).
- Supports argument handling throughout the entire CLI chain (i.e. `myapp {options} controller {options} sub-command {options}`)

The `ArgparseController` will become the default in Cement 3, however `CementBaseController` will remain the default in Cement 2.x. Developers are encouraged to begin porting to `ArgparseController` as soon possible, as `CementBaseController` will be removed in Cement 3 completely.

Related:

- Issue #205

Extensions

- *Argcomplete* - Provides the ability to magically perform BASH autocompletion by simply loading the `argcomplete` extension. (Requires `ArgparseArgumentHandler` and `ArgparseController` to function).
- *Tabulate* - Provides tabularized output familiar to users of MySQL, PGSQL, Etc.
- *Alarm* - Provides quick access to setting an application alarm to easily handling timing out long running operations.
- *Memcached* - Now supported on Python 3.

Misc Enhancements

- Cement now supports the ability to reload runtime within the current process via `app.reload()`. This will enable future refactoring of the `ext_reload_config` extension that is intended to handle reloading runtime after configuration files are modified. This affectively adds `SIGHUP` support.

New Features in Cement 2.6

Extensions

- *Reload Config* - Provides the ability to automatically reload `app.config` any time configuration files are modified.

- *ColorLog* - Provides colorized logging to console (based on standard logging module).

Python With Statement Support

Using the `with` statement makes setting up, running, and closing Cement apps easier and cleaner. The following is the recommended way of creating, and running Cement apps:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    app.run()
```

Or a more complete example:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'

with MyApp() as app:
    try:
        app.run()
    except Exception as e:
        # handle all your exceptions... this is just an example
        print('Caught Exception: %s' % e)
```

When the `with` statement is initialized, the `app` object is created, and then right away `app.setup()` is called before entering the block. When the `with` block is exited `app.close()` is also called. This offers a much cleaner approach, while still ensuring that the essential pieces are run appropriately. If you require more control over how/when `app.setup()` and `app.close()` are run, you can still do this the old way:

```
from cement.core.foundation import CementApp

app = CementApp('myapp')
app.setup()
app.run()
app.close()
```

But doesn't that just feel clunky?

Related:

- Issue #281

Defining and Registering Hooks and Handlers from CementApp.Meta

Another improvement that lends itself nicely to code-cleanliness is the ability to define and register hooks and handlers from within `CementApp.Meta`. An example using application controllers and a simple `pre_run` hook looks like:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

def my_example_hook(app):
    pass

class BaseController(CementBaseController):
```

```
    class Meta:
        label = 'base'


class SecondController(CementBaseController):
    class Meta:
        label = 'second'


class MyApp(CementApp):
    class Meta:
        label = 'myapp'

        hooks = [
            ('pre_run', my_example_hook),
        ]

        handlers = [
            BaseController,
            SecondController,
        ]
```

Related:

- Issue #282

Projects Built on Cement™

The following is an incomplete lists of notable projects that are Built on Cement™:

- Elex (GitHub)
- Amazon Elastic Beanstalk CLI (PYPI)
- Easy Engine (GitHub)
- SentientHome
- Pubkey
- HCE Project

If you are building a project on the Cement Framework and would like to see your company or project listed here please create an issue and/or pull request on GitHub.

Frequenty Asked Questions

How do I Manually Print the Help Text for My Application

The common question is how to print the help text that you see when you pass `--help` to your application, but manually from within your code. This is a feature of ArgParse and is as simple as calling:

```
app.args.print_help()
```

Note that, this obviously will not work if using a different argument handler that implements the `IArgument` interface rather than the default `ArgparseArgumentHandler`.

API Reference

Cement Core Modules

cement.core.arg

Cement core argument module.

class cement.core.arg.**CementArgumentHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Argument Handlers should sub-class from.

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

interface
The interface that this class implements.

alias of *IArgument*

label = None
The string identifier of the handler implementation.

class cement.core.arg.**IArgument**
Bases: *cement.core.interface.Interface*

This class defines the Argument Handler Interface. Classes that implement this handler must provide the methods and attributes defined below. Implementations do *not* subclass from interfaces.

Example:

```
from cement.core import interface, arg

class MyArgumentHandler(arg.CementArgumentHandler):
    class Meta:
        interface = arg.IArgument
        label = 'my_argument_handler'
```

class IMeta
Interface meta-data options.

label = 'argument'
The string identifier of the interface.

validator(*klass*, *obj*)
Interface validator function.

IArgument.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object

Returns None

IArgument.**add_argument**(**args*, ***kw*)
Add arguments for parsing. This should be -o/–option or positional. Note that the interface defines the following parameters so that at the very least, external extensions can guarantee that they can properly add command line arguments when necessary. The implementation itself should, and will provide and support many more options than those listed here. That said, the implementation must support the following:

Parameters

- **args** – List of option arguments. Generally something like ['-h', '–help'].
- **dest** – The destination name (var). Default: arg[0]'s string.
- **help** – The help text for –help output (for that argument).
- **action** – Must support: ['store', 'store_true', 'store_false', 'store_const']
- **choices** – A list of valid values that can be passed to an option whose action is `store`.
- **const** – The value stored if action == 'store_const'.
- **default** – The default value.

Returns `None`

`IArgument.`**parse**(*arg_list*)
Parse the argument list (i.e. sys.argv). Can return any object as long as it's members contain those of the added arguments. For example, if adding a '-v/–version' option that stores to the dest of 'version', then the member must be callable as 'Object().version'.

Parameters **arg_list** – A list of command line arguments.

Returns Callable object

`cement.core.arg.`**argument_validator**(*klass*, *obj*)
Validates a handler implementation against the IArgument interface.

cement.core.backend

Cement core backend module.

cement.core.cache

Cement core cache module.

class `cement.core.cache.`**CementCacheHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Cache Handlers should sub-class from.

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

interface
The interface that this handler class implements.

alias of *ICache*

label = None
String identifier of this handler implementation.

class `cement.core.cache.`**ICache**
Bases: *cement.core.interface.Interface*

This class defines the Cache Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import cache

class MyCacheHandler(object):
    class Meta:
        interface = cache.ICache
        label = 'my_cache_handler'
    ...
```

class IMeta
Interface meta-data.

label = 'cache'
The label (or type identifier) of the interface.

validator(*klass*, *obj*)
Interface validator function.

ICache.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

Returns None

ICache.**delete**(*key*)
Deletes a key/value from the cache.

Parameters **key** – The key in the cache to delete.

Returns True if the key is successfully deleted, False otherwise.

Return type boolean

ICache.**get**(*key*, *fallback=None*)
Get the value for a key in the cache. If the key does not exist or the key/value in cache is expired, this functions must return 'fallback' (which in turn must default to None).

Parameters

- **key** – The key of the value stored in cache
- **fallback** – Optional value that is returned if the cache is expired or the key does not exist. Default: None

Returns Unknown (whatever the value is in cache, or the *fallback*)

ICache.**purge**()
Clears all data from the cache.

ICache.**set**(*key*, *value*, *time=None*)
Set the key/value in the cache for a set amount of *time*.

Parameters

- **key** – The key of the value to store in cache.
- **value** – The value of that key to store in cache.
- **time** (int (seconds) or None) – A one-off expire time. If no time is given, then a default value is used (determined by the implementation).

Returns None

cement.core.cache.**cache_validator**(*klass*, *obj*)
Validates a handler implementation against the ICache interface.

cement.core.config

Cement core config module.

class cement.core.config.**CementConfigHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Config Handlers should sub-class from.

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

interface
The interface that this handler implements.

alias of *IConfig*

label = None
The string identifier of the implementation.

CementConfigHandler.**_parse_file**(*file_path*)
Parse a configuration file at *file_path* and store it. This function must be provided by the handler implementation (that is sub-classing this).

Parameters **file_path** – The file system path to the configuration file.

Returns True if file was read properly, False otherwise

Return type boolean

CementConfigHandler.**parse_file**(*file_path*)
Ensure we are using the absolute/expanded path to *file_path*, and then call *_parse_file* to parse config file settings from it, overwriting existing config settings. If the file does not exist, returns False.

Developers sub-classing from here should generally override *_parse_file* which handles just the parsing of the file and leaving this function to wrap any checks/logging/etc.

Parameters **file_path** – The file system path to the configuration file.

Returns boolean

class cement.core.config.**IConfig**
Bases: *cement.core.interface.Interface*

This class defines the Config Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

All implementations must provide sane 'default' functionality when instantiated with no arguments. Meaning, it can and should accept optional parameters that alter how it functions, but can not require any parameters. When the framework first initializes handlers it does not pass anything too them, though a handler can be instantiated first (with or without parameters) and then passed to 'CementApp()' already instantiated.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import config

class MyConfigHandler(config.CementConfigHandler):
    class Meta:
```

```
        interface = config.IConfig
        label = 'my_config_handler'
    ...
```

class IMeta
Interface meta-data.

label = 'config'
The string identifier of the interface.

validator(*klass*, *obj*)
The validator function.

IConfig.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

Returns None

IConfig.**add_section**(*section*)
Add a new section if it doesn't exist.

Parameters **section** – The [section] label to create.

Returns None

IConfig.**get**(*section*, *key*)
Return a configuration value based on [section][key]. The return value type is unknown.

Parameters

- **section** – The [section] of the configuration to pull key value from.
- **key** – The configuration key to get the value from.

Returns The value of the *key* in *section*.

Return type Unknown

IConfig.**get_section_dict**(*section*)
Return a dict of configuration parameters for [section].

Parameters **section** – The config [section] to generate a dict from (using that section keys).

Returns A dictionary of the config section.

Return type dict

IConfig.**get_sections**()
Return a list of configuration sections. These are designated by a [block] label in a config file.

Returns A list of config sections.

Return type list

IConfig.**has_section**(*section*)
Returns whether or not the section exists.

Parameters **section** – The section to test for.

Returns boolean

IConfig.**keys**(*section*)
Return a list of configuration keys from *section*.

Parameters **section** – The config [section] to pull keys from.

Returns A list of keys in *section*.

Return type list

IConfig.**merge**(*dict_obj*, *override=True*)
Merges a dict object into the configuration.

Parameters

- **dict_obj** – The dictionary to merge into the config
- **override** – Boolean. Whether to override existing values. Default: True

Returns None

IConfig.**parse_file**(*file_path*)
Parse config file settings from file_path. Returns True if the file existed, and was parsed successfully. Returns False otherwise.

Parameters **file_path** – The path to the config file to parse.

Returns True if the file was parsed, False otherwise.

Return type boolean

IConfig.**set**(*section*, *key*, *value*)
Set a configuration value based at [section][key].

Parameters

- **section** – The [section] of the configuration to pull key value from.
- **key** – The configuration key to set the value at.
- **value** – The value to set.

Returns None

cement.core.config.**config_validator**(*klass*, *obj*)
Validates a handler implementation against the IConfig interface.

cement.core.controller

Cement core controller module.

class cement.core.controller.**CementBaseController**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

This is an implementation of the *IControllerHandler* interface, but as a base class that application controllers *should* subclass from. Registering it directly as a handler is useless.

NOTE: This handler **requires** that the applications 'arg_handler' be argparse. If using an alternative argument handler you will need to write your own controller base class.

NOTE: This the initial default implementation of CementBaseController. In the future it will be replaced by CementBaseController2, therefore using CementBaseController2 is recommended for new development.

Usage:

```
from cement.core.controller import CementBaseController

class MyAppBaseController(CementBaseController):
    class Meta:
```

```
        label = 'base'
        description = 'MyApp is awesome'
        config_defaults = dict()
        arguments = []
        epilog = "This is the text at the bottom of --help."
        # ...

class MyStackedController(CementBaseController):
    class Meta:
        label = 'second_controller'
        aliases = ['sec', 'secondary']
        stacked_on = 'base'
        stacked_type = 'embedded'
        # ...
```

class Meta

Controller meta-data (can be passed as keyword arguments to the parent class).

aliases = []

A list of aliases for the controller. Will be treated like command/function aliases for non-stacked controllers. For example: `myapp <controller_label> --help` is the same as `myapp <controller_alias> --help`.

aliases_only = False

When set to True, the controller label will not be displayed at command line, only the aliases will. Effectively, aliases[0] will appear as the label. This feature is useful for the situation Where you might want two controllers to have the same label when stacked on top of separate controllers. For example, 'myapp users list' and 'myapp servers list' where 'list' is a stacked controller, not a function.

argument_formatter

The argument formatter class to use to display –help output.

alias of `RawDescriptionHelpFormatter`

arguments = []

Arguments to pass to the argument_handler. The format is a list of tuples whos items are a (list, dict). Meaning:

```
[ ( ['-f', '--foo'], dict(dest='foo', help='foo option') ), ]
```

This is equivelant to manually adding each argument to the argument parser as in the following example:

```
parser.add_argument(['-f', '--foo'], help='foo option', dest='foo')
```

config_defaults = {}

Configuration defaults (type: dict) that are merged into the applications config object for the config_section mentioned above.

config_section = None

A config [section] to merge config_defaults into. Cement will default to controller.<label> if None is set.

default_func = 'default'

Function to call if no sub-command is passed. Note that this can **not** start with an _ due to backward compatibility restraints in how Cement discovers and maps commands.

description = None

The description shown at the top of '–help'. Default: None

epilog = None
The text that is displayed at the bottom when '–help' is passed.

hide = False
Whether or not to hide the controller entirely.

interface
The interface this class implements.

alias of *IController*

label = None
The string identifier for the controller.

stacked_on = 'base'
A label of another controller to 'stack' commands/arguments on top of.

stacked_type = 'embedded'
Whether to *embed* commands and arguments within the parent controller or to simply *nest* the controller under the parent controller (making it a sub-sub-command). Must be one of *['embedded', 'nested']* only if *stacked_on* is not *None*.

usage = None
The text that is displayed at the top when '–help' is passed. Although the default is *None*, Cement will set this to a generic usage based on the *prog*, *controller* name, etc if nothing else is passed.

CementBaseController.**_dispatch**()
Takes the remaining arguments from self.app.argv and parses for a command to dispatch, and if so... dispatches it.

CementBaseController.**_help_text**
Returns the help text displayed when '–help' is passed.

CementBaseController.**_setup**(*app_obj*)
See *IController._setup()*.

CementBaseController.**_usage_text**
Returns the usage text displayed when `--help` is passed.

class cement.core.controller.**IController**
Bases: *cement.core.interface.Interface*

This class defines the Controller Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import controller

class MyBaseController(controller.CementBaseController):
    class Meta:
        interface = controller.IController
        ...
```

class IMeta
Interface meta-data.

label = 'controller'
The string identifier of the interface.

validator(*klass*, *obj*)
The interface validator function.

IController.**_dispatch**()
Reads the application object's data to dispatch a command from this controller. For example, reading self.app.pargs to determine what command was passed, and then executing that command function.

Note that Cement does *not* parse arguments when calling _dispatch() on a controller, as it expects the controller to handle parsing arguments (I.e. self.app.args.parse()).

Returns Returns the result of the executed controller function, or `None` if no controller function is called.

IController.**_setup**(*app_obj*)
The _setup function is after application initialization and after it is determined that this controller was requested via command line arguments. Meaning, a controllers _setup() function is only called right before it's _dispatch() function is called to execute a command. Must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

Returns `None`

cement.core.controller.**controller_validator**(*klass*, *obj*)
Validates a handler implementation against the IController interface.

class cement.core.controller.**expose**(*help=''*, *hide=False*, *aliases=[]*, *aliases_only=False*)
Bases: `object`

Used to expose controller functions to be listed as commands, and to decorate the function with Meta data for the argument parser.

Parameters

- **help** (*str*) – Help text to display for that command.
- **hide** (*boolean*) – Whether the command should be visible.
- **aliases** (`list`) – Aliases to this command.
- **aliases_only** – Whether to only display the aliases (not the label). This is useful for situations where you have obscure function names which you do not want displayed. Effecively, if there are aliases and *aliases_only* is True, then aliases[0] will appear as the actual command/function label.

Usage:

```
from cement.core.controller import CementBaseController, expose

class MyAppBaseController(CementBaseController):
    class Meta:
        label = 'base'

    @expose(hide=True, aliases=['run'])
    def default(self):
        print("In MyAppBaseController.default()")

    @expose()
    def my_command(self):
        print("In MyAppBaseController.my_command()")
```

cement.core.exc

Cement core exceptions module.

exception cement.core.exc.**CaughtSignal**(*signum*, *frame*)
Bases: *cement.core.exc.FrameworkError*

Raised when a defined signal is caught. For more information regarding signals, reference the signal library.

Parameters

- **signum** – The signal number.
- **frame** – The signal frame.

exception cement.core.exc.**FrameworkError**(*msg*)
Bases: exceptions.Exception

General framework (non-application) related errors.

Parameters **msg** – The error message.

exception cement.core.exc.**InterfaceError**(*msg*)
Bases: *cement.core.exc.FrameworkError*

Interface related errors.

cement.core.extension

Cement core extensions module.

class cement.core.extension.**IExtension**
Bases: *cement.core.interface.Interface*

This class defines the Extension Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import extension

class MyExtensionHandler(object):
    class Meta:
        interface = extension.IExtension
        label = 'my_extension_handler'
    ...
```

class IMeta
Interface meta-data.

label = 'extension'
The string identifier of the interface.

validator(*klass*, *obj*)
The interface validator function.

IExtension.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

Returns None

IExtension.**load_extension**(*ext_module*)
Load an extension whose module is 'ext_module'. For example, 'cement.ext.ext_configobj'.

Parameters **ext_module** (str) – The name of the extension to load.

IExtension.**load_extensions**(*ext_list*)
Load all extensions from ext_list.

Parameters **ext_list** (list) – A list of extension modules to load. For example: ['cement.ext.ext_configobj', 'cement.ext.ext_logging']

cement.core.extension.**extension_validator**(*klass*, *obj*)
Validates an handler implementation against the IExtension interface.

cement.core.foundation

Cement core foundation module.

class cement.core.foundation.**CementApp**(*label=None*, ***kw*)
Bases: *cement.core.meta.MetaMixin*

The primary class to build applications from.

Usage:

The following is the simplest CementApp:

```
from cement.core.foundation import CementApp

with CementApp('helloworld') as app:
    app.run()
```

Alternatively, the above could be written as:

```
from cement.core.foundation import CementApp

app = foundation.CementApp('helloworld')
app.setup()
app.run()
app.close()
```

A more advanced example looks like:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

class MyController(CementBaseController):
    class Meta:
        label = 'base'
        arguments = [
            ( ['-f', '--foo'], dict(help='Notorious foo option') ),
            ]
        config_defaults = dict(
            debug=False,
            some_config_param='some_value',
            )

    @expose(help='This is the default command', hide=True)
```

```
    def default(self):
        print('Hello World')

class MyApp(CementApp):
    class Meta:
        label = 'helloworld'
        extensions = ['daemon','json',]
        base_controller = MyController

with MyApp() as app:
    app.run()
```

class Meta

Application meta-data (can also be passed as keyword arguments to the parent class).

alternative_module_mapping = {}

EXPERIMENTAL FEATURE: This is an experimental feature added in Cement 2.9.x and may or may not be removed in future versions of Cement.

Dictionary of alternative, **drop-in** replacement modules to use selectively throughout the application, framework, or extensions. Developers can optionally use the `CementApp.__import__()` method to import simple modules, and if that module exists in this mapping it will import the alternative library in it's place.

This is a low-level feature, and may not produce the results you are expecting. It's purpose is to allow the developer to replace specific modules at a high level. Example: For an application wanting to use `ujson` in place of `json`, the developer could set the following:

```
alternative_module_mapping = {
    'json' : 'ujson',
}
```

In the app, you would then load `json` as:

```
_json = app.__import__('json')
_json.dumps(data)
```

Obviously, the replacement module **must be** a drop-in replace and function the same.

argument_handler = 'argparse'

A handler class that implements the IArgument interface.

arguments_override_config = False

A boolean to toggle whether command line arguments should override configuration values if the argument name matches the config key. I.e. –foo=bar would override config['myapp']['foo'].

This is different from `override_arguments` in that if `arguments_override_config` is `True`, then all arguments will override (you don't have to list them all).

argv = None

A list of arguments to use for parsing command line arguments and options.

Note: Though Meta.argv defaults to None, Cement will set this to `list(sys.argv[1:])` if no argv is set in Meta during setup().

base_controller = None

This is the base application controller. If a controller is set, runtime operations are passed to the controller for command dispatch and argument parsing when CementApp.run() is called.

Note that cement will automatically set the *base_controller* to a registered controller whose label is 'base' (only if *base_controller* is not currently set).

bootstrap = None
A bootstrapping module to load after app creation, and before app.setup() is called. This is useful for larger applications that need to offload their bootstrapping code such as registering hooks/handlers/etc to another file.

This must be a dotted python module path. I.e. 'myapp.bootstrap' (myapp/bootstrap.py). Cement will then import the module, and if the module has a 'load()' function, that will also be called. Essentially, this is the same as an extension or plugin, but as a facility for the application itself to bootstrap 'hardcoded' application code. It is also called before plugins are loaded.

cache_handler = None
A handler class that implements the ICache interface.

catch_signals = [15, 2, 1]
List of signals to catch, and raise exc.CaughtSignal for. Can be set to None to disable signal handling.

config_defaults = None
Default configuration dictionary. Must be of type 'dict'.

config_dirs = None
List of config directories to search config files.

For each direcory cement will load all files that ends with `.conf`.

```
['/etc/<app_label>/<app_label>/conf.d',
 '~/.<app_label>/conf.d']
```

Directories and files inside are loaded in order, and have precedence in order. Therefore, the last configuration loaded has precedence (and overwrites settings loaded from previous configuration files).

These configuration will be overriden by configuration from `CementApp.Meta.config_files`.

Note that `.conf` is the default config file extension, defined by `CementApp.Meta.config_extension`.

config_extension = '.conf'
Extension used to identify application and plugin configuration files.

config_files = None
List of config files to parse.

Note: Though Meta.config_section defaults to None, Cement will set this to a default list based on Meta.label (or in other words, the name of the application). This will equate to:

```
['/etc/<app_label>/<app_label>.conf',
 '~/.<app_label>.conf',
 '~/.<app_label>/config']
```

Files are loaded in order, and have precedence in order. Therefore, the last configuration loaded has precedence (and overwrites settings loaded from previous configuration files).

Note that `.conf` is the default config file extension, defined by `CementApp.Meta.config_extension`.

config_handler = 'configparser'
A handler class that implements the IConfig interface.

config_section = None
The base configuration section for the application.

Note: Though Meta.config_section defaults to None, Cement will set this to the value of Meta.label (or in other words, the name of the application).

core_extensions = ['cement.ext.ext_dummy', 'cement.ext.ext_smtp', 'cement.ext.ext_plugin', 'cement.ext.ext_co
List of Cement core extensions. These are generally required by Cement and should only be modified if you know what you're doing. Use 'extensions' to add to this list, rather than overriding core extensions. That said if you want to prune down your application, you can remove core extensions if they are not necessary (for example if using your own log handler extension you likely don't want/need LoggingLogHandler to be registered).

core_handler_override_options = {'output': (['-o'], {'help': 'output handler'})}
Similar to `CementApp.Meta.handler_override_options` but these are the core defaults required by Cement. This dictionary can be overridden by `CementApp.Meta.handler_override_options` (when they are merged together).

core_meta_override = ['debug', 'plugin_config_dir', 'plugin_dir', 'ignore_deprecation_warnings', 'template_di
List of meta options that can/will be overridden by config options of the '[base]' config section (where [base] is the base configuration section of the application which is determined by Meta.config_section but defaults to Meta.label). These overrides are required by the framework to function properly and should not be used by end user (developers) unless you really know what you're doing. To add your own extended meta overrides please use 'meta_override'.

debug = False
Used internally, and should not be used by developers. This is set to *True* if *–debug* is passed at command line.

define_handlers = []
List of interfaces classes to define handlers. Must be a list of uninstantiated interface classes.

I.e. `['MyCustomInterface', 'SomeOtherInterface']`

define_hooks = []
List of hook definitions (label). Will be passed to `self.hook.define(<hook_label>)`. Must be a list of strings.

I.e. `['my_custom_hook', 'some_other_hook']`

exit_on_close = False
Whether or not to call `sys.exit()` when `close()` is called. The default is `False`, however if `True` then the app will call `sys.exit(X)` where `X` is `self.exit_code`.

extension_handler = 'cement'
A handler class that implements the IExtension interface.

extensions = []
List of additional framework extensions to load.

framework_logging = True
Whether or not to enable Cement framework logging. This is separate from the application log, and is generally used for debugging issues with the framework and/or extensions primarily in development.

This option is overridden by the environment variable *CEMENT_FRAMEWORK_LOGGING*. Therefore, if in production you do not want the Cement framework log enabled, you can set this option to `False` but override it in your environment by doing something like `export CEMENT_FRAMEWORK_LOGGING=1` in your shell whenever you need it enabled.

handler_override_options = {}
Dictionary of handler override options that will be added to the argument parser, and allow the end-user to override handlers. Useful for interfaces that have multiple uses within the same application (for example: Output Handler (json, yaml, etc) or maybe a Cloud Provider Handler (rackspace, digitalocean, amazon, etc).

This dictionary will merge with `CementApp.Meta.core_handler_override_options` but this one has precedence.

Dictionary Format:

```
<interface_name> = (option_arguments, help_text)
```

See `CementApp.Meta.core_handler_override_options` for an example of what this should look like.

Note, if set to `None` then no options will be defined, and the `CementApp.Meta.core_meta_override_options` will be ignore (not recommended as some extensions rely on this feature).

handlers = []
List of handler classes to register. Will be passed to `handler.register(<handler_class>)`. Must be a list of uninstantiated handler classes.

I.e. `[MyCustomHandler, SomeOtherHandler]`

hooks = []
List of hooks to register when the app is created. Will be passed to `self.hook.register(<hook_label>, <hook_func>)`. Must be a list of tuples in the form of `(<hook_label>, <hook_func>)`.

I.e. `[('post_argument_parsing', my_hook_func)]`.

ignore_deprecation_warnings = False
Disable deprecation warnings from being logged by Cement.

label = None
The name of the application. This should be the common name as you would see and use at the command line. For example 'helloworld', or 'my-awesome-app'.

log_handler = 'logging'
A handler class that implements the ILog interface.

mail_handler = 'dummy'
A handler class that implements the IMail interface.

meta_defaults = {}
Default metadata dictionary used to pass high level options from the application down to handlers at the point they are registered by the framework **if the handler has not already been instantiated**.

For example, if requiring the `json` extension, you might want to override `JsonOutputHandler.Meta.json_module` with `ujson` by doing the following

```
from cement.core.foundation import CementApp
from cement.utils.misc import init_defaults

META = init_defaults('output.json')
META['output.json']['json_module'] = 'ujson'

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['json']
        meta_defaults = META
```

meta_override = []
List of meta options that can/will be overridden by config options of the '[base]' config section (where

[base] is the base configuration section of the application which is determined by Meta.config_section but defaults to Meta.label).

output_handler = 'dummy'
A handler class that implements the IOutput interface.

override_arguments = ['debug']
List of arguments that override their configuration counter-part. For example, if `--debug` is passed (and it's config value is `debug`) then the `debug` key of all configuration sections will be overridden by the value of the command line option (`True` in this example).

This is different from `arguments_override_config` in that this is a selective list of specific arguments to override the config with (and not all arguments that match the config). This list will take affect whether `arguments_override_config` is `True` or `False`.

plugin_bootstrap = None
A python package (dotted import path) where plugin code can be loaded from. This is generally something like `myapp.plugins` where a plugin file would live at `myapp/plugins/myplugin.py`. This provides a facility for applications that have builtin plugins that ship with the applications source code and live in the same Python module.

Note: Though the meta default is `None`, Cement will set this to `<app_label>.plugins` if not set.

plugin_config_dir = None
A directory path where plugin config files can be found. Files must end in `.conf` (or the extension defined by `CementApp.Meta.config_extension`) or they will be ignored. By default, this setting is also overridden by the `[<app_label>] -> plugin_config_dir` config setting parsed in any of the application configuration files.

If set, this item will be **appended** to `CementApp.Meta.plugin_config_dirs` so that it's settings will have presedence over other configuration files.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `plugin_config_dir` without completely trumping the hard-coded list of default `plugin_config_dirs` defined by the app/developer.

plugin_config_dirs = None
A list of directory paths where plugin config files can be found. Files must end in `.conf` (or the extension defined by `CementApp.Meta.config_extension`) or they will be ignored.

Note: Though `CementApp.Meta.plugin_config_dirs` is `None`, Cement will set this to a default list based on `CementApp.Meta.label`. This will equate to:

```
['/etc/<app_label>/plugins.d', '~/.<app_label>/plugin.d']
```

Files are loaded in order, and have precedence in that order. Therefore, the last configuration loaded has precedence (and overwrites settings loaded from previous configuration files).

plugin_dir = None
A directory path where plugin code (modules) can be loaded from. By default, this setting is also overridden by the `[<app_label>] -> plugin_dir` config setting parsed in any of the application configuration files.

If set, this item will be **prepended** to `Meta.plugin_dirs` so that a users defined `plugin_dir` has precedence over others.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `plugin_dir` without completely trumping the hard-coded list of default `plugin_dirs` defined by the app/developer.

plugin_dirs = None
A list of directory paths where plugin code (modules) can be loaded from.

Note: Though `CementApp.Meta.plugin_dirs` is None, Cement will set this to a default list based on `CementApp.Meta.label` if not set. This will equate to:

```
['~/.<app_label>/plugins', '/usr/lib/<app_label>/plugins']
```

Modules are attempted to be loaded in order, and will stop loading once a plugin is successfully loaded from a directory. Therefore this is the oposite of configuration file loading, in that here the first has precedence.

plugin_handler = 'cement'
A handler class that implements the IPlugin interface.

plugins = []
A list of plugins to load. This is generally considered bad practice since plugins should be dynamically enabled/disabled via a plugin config file.

signal_handler (*signum*, *frame*)
A function that is called to handle any caught signals.

template_dir = None
A directory path where template files can be loaded from. By default, this setting is also overridden by the `[<app_label>] -> template_dir` config setting parsed in any of the application configuration files .

If set, this item will be **prepended** to `CementApp.Meta.template_dirs` (giving it precedence over other `template_dirs`.

template_dirs = None
A list of directory paths where template files can be loaded from.

Note: Though `CementApp.Meta.template_dirs` defaults to `None`, Cement will set this to a default list based on `CementApp.Meta.label`. This will equate to:

```
['~/.<app_label>/templates', '/usr/lib/<app_label>/templates']
```

Templates are attempted to be loaded in order, and will stop loading once a template is successfully loaded from a directory.

template_module = None
A python package (dotted import path) where template files can be loaded from. This is generally something like `myapp.templates` where a plugin file would live at `myapp/templates/mytemplate.txt`. Templates are first loaded from `CementApp.Meta.template_dirs`, and and secondly from `CementApp.Meta.template_module`. The `template_dirs` setting has presedence.

use_backend_globals = True
This is a backward compatibility feature. Cement 2.x.x relies on several global variables hidden in `cement.core.backend` used for things like storing hooks and handlers. Future versions of Cement will no longer use this mechanism, however in order to maintain backward compatibility this is still the default. By disabling this feature allows multiple instances of CementApp to be created from within the same runtime space without clobbering eachothers hooks/handers/etc.

Be warned that use of third-party extensions might break as they were built using backend globals, and probably have no idea this feature has changed or exists.

`CementApp._lay_cement()`
Initialize the framework.

CementApp.**add_arg**(**args*, ***kw*)
A shortcut for self.args.add_argument.

CementApp.**add_template_dir**(*path*)
Append a directory path to the list of template directories to parse for templates.

Parameters **path** – Directory path that contains template files.

Usage:

```
app.add_template_dir('/path/to/my/templates')
```

CementApp.**argv**
The arguments list that will be used when self.run() is called.

CementApp.**catch_signal**(*signum*)
Add `signum` to the list of signals to catch and handle by Cement.

Parameters **signum** – The signal number to catch. See Python `signal` library.

CementApp.**close**(*code=None*)
Close the application. This runs the `pre_close` and `post_close` hooks allowing plugins/extensions/etc to cleanup at the end of program execution.

Parameters **code** – An exit code to exit with (`int`), if `None` is passed then exit with whatever `self.exit_code` is currently set to. Note: `sys.exit()` will only be called if `CementApp.Meta.exit_on_close==True`.

CementApp.**debug**
Returns boolean based on whether `--debug` was passed at command line or set via the application's configuration file.

Returns boolean

CementApp.**extend**(*member_name*, *member_object*)
Extend the CementApp() object with additional functions/classes such as 'app.my_custom_function()', etc. It provides an interface for extensions to provide functionality that travel along with the application object.

Parameters

- **member_name** (`str`) – The name to attach the object to.
- **member_object** – The function or class object to attach to CementApp().

Raises cement.core.exc.FrameworkError

CementApp.**get_last_rendered**()
DEPRECATION WARNING: This function is deprecated as of Cement 2.1.3 in favor of the *self.last_rendered* property, and will be removed in future versions of Cement.

Return the (data, output_text) tuple of the last time self.render() was called.

Returns tuple (data, output_text)

CementApp.**last_rendered**
Return the (data, output_text) tuple of the last time self.render() was called.

Returns tuple (data, output_text)

CementApp.**pargs**
Returns the *parsed_args* object as returned by self.args.parse().

CementApp.**reload**()
This function is useful for reloading a running applications, for example to reload configuration settings, etc.

Returns None

CementApp.**remove_template_dir**(*path*)
Remove a directory path from the list of template directories to parse for templates.

Parameters **path** – Directory path that contains template files.

Usage:

```
app.remove_template_dir('/path/to/my/templates')
```

CementApp.**render**(*data*, *template=None*, *out=<open file '<stdout>', mode 'w'>*, ***kw*)
This is a simple wrapper around self.output.render() which simply returns an empty string if no self.output handler is defined.

Parameters

- **data** – The data dictionary to render.
- **template** – The template to render to. Default: None (some output handlers do not use templates).
- **out** – A file like object (sys.stdout, or actual file). Set to None is no output is desired (just render and return). Default: sys.stdout

CementApp.**run**()
This function wraps everything together (after self._setup() is called) to run the application.

Returns Returns the result of the executed controller function if a base controller is set and a controller function is called, otherwise None if no controller dispatched or no controller function was called.

CementApp.**run_forever**(*interval=1*, *tb=True*)
This function wraps run() with an endless while loop. If any exception is encountered it will be logged and then the application will be reloaded.

Parameters

- **interval** – The number of seconds to sleep before reloading the the appliction.
- **tb** – Whether or not to print traceback if exception occurs.

Returns It should never return.

CementApp.**setup**()
This function wraps all '_setup' actons in one call. It is called before self.run(), allowing the application to be _setup but not executed (possibly letting the developer perform other actions before full execution.).

All handlers should be instantiated and callable after setup is complete.

CementApp.**validate_config**()
Validate application config settings.

Usage:

```
import os
from cement.core import foundation

class MyApp(foundation.CementApp):
    class Meta:
```

```
        label = 'myapp'

    def validate_config(self):
        super(MyApp, self).validate_config()

        # test that the log file directory exist, if not create it
        logdir = os.path.dirname(self.config.get('log', 'file'))

        if not os.path.exists(logdir):
            os.makedirs(logdir)
```

cement.core.foundation.**add_handler_override_options**(*app*)
This is a post_setup hook that adds the handler override options to the argument parser

Parameters **app** – The application object.

cement.core.foundation.**cement_signal_handler**(*signum*, *frame*)
Catch a signal, run the 'signal' hook, and then raise an exception allowing the app to handle logic elsewhere.

Parameters

- **signum** – The signal number
- **frame** – The signal frame.

Raises cement.core.exc.CaughtSignal

cement.core.foundation.**handler_override**(*app*)
This is a post_argument_parsing hook that overrides a configured handler if defined in CementApp.Meta.handler_override_options and the option is passed at command line with a valid handler label.

Parameters **app** – The application object.

cement.core.handler

Cement core handler module.

class cement.core.handler.**CementBaseHandler**(***kw*)
Bases: *cement.core.meta.MetaMixin*

Base handler class that all Cement Handlers should subclass from.

class Meta
Handler meta-data (can also be passed as keyword arguments to the parent class).

config_defaults = None
A config dictionary that is merged into the applications config in the [<config_section>] block. These are defaults and do not override any existing defaults under that section.

config_section = None
A config [section] to merge config_defaults with.

Note: Though Meta.config_section defaults to None, Cement will set this to the value of <interface_label>.<handler_label> if no section is set by the user/developer.

interface = None
The interface that this class implements.

label = None
The string identifier of this handler.

overridable = False
Whether or not handler can be overridden by `CementApp.Meta.handler_override_options`. Will be listed as an available choice to override the specific handler (i.e. `CementApp.Meta.output_handler`, etc).

`CementBaseHandler._setup`(*app_obj*)
The _setup function is called during application initialization and must `setup` the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

Returns None

class `cement.core.handler.HandlerManager`(*use_backend_globals=False*)
Bases: `object`

Manages the handler system to define, get, resolve, etc handlers with the Cement Framework.

Parameters **use_backend_globals** – Whether to use backend globals (backward compatibility and deprecated).

define(*interface*)
Define a handler based on the provided interface. Defines a handler type based on `<interface>.IMeta.label`.

Parameters **interface** – The interface class that defines the interface to be implemented by handlers.

Raises *cement.core.exc.InterfaceError*

Raises *cement.core.exc.FrameworkError*

Usage:

```
app.handler.define(IDatabaseHandler)
```

defined(*handler_type*)
Test whether `handler_type` is defined.

Parameters **handler_type** – The name or `handler_type` of the handler (I.e. `log`, `config`, `output`, etc).

Returns True if the handler type is defined, False otherwise.

Return type `boolean`

Usage:

```
app.handler.defined('log')
```

get(*handler_type*, *handler_label*, **args*)
Get a handler object.

Parameters

- **handler_type** (`str`) – The type of handler (i.e. `output`)
- **handler_label** (`str`) – The label of the handler (i.e. `json`)
- **fallback** – A fallback value to return if handler_label doesn't exist.

Returns An uninstantiated handler object

Raises *cement.core.exc.FrameworkError*

Usage:

```
output = app.handler.get('output', 'json')
output.render(dict(foo='bar'))
```

list(*handler_type*)
Return a list of handlers for a given `handler_type`.

Parameters **handler_type** – The type of handler (i.e. `output`)

Returns List of handlers that match `hander_type`.

Return type `list`

Raises *cement.core.exc.FrameworkError*

Usage:

```
app.handler.list('log')
```

list_types()
Return a list of handler types (interface labels).

Returns List of handlers types (interface labels).

Return type `list`

Raises *cement.core.exc.FrameworkError*

Usage:

```
app.handler.list_types()
```

register(*handler_obj*, *force=False*)
Register a handler object to a handler. If the same object is already registered then no exception is raised, however if a different object attempts to be registered to the same name a `FrameworkError` is raised.

Parameters

- **handler_obj** – The uninstantiated handler object to register.
- **force** – Whether to allow replacement if an existing handler of the same `label` is already registered.

Raises *cement.core.exc.InterfaceError*

Raises *cement.core.exc.FrameworkError*

Usage:

```
class MyDatabaseHandler(object):
    class Meta:
        interface = IDatabase
        label = 'mysql'

    def connect(self):
        # ...

app.handler.register(MyDatabaseHandler)
```

registered(*handler_type*, *handler_label*)
Check if a handler is registered.

Parameters

- **handler_type** – The type of handler (interface label)
- **handler_label** – The label of the handler

Returns True if the handler is registered, False otherwise

Return type boolean

Usage:

```
app.handler.registered('log', 'colorlog')
```

resolve(*handler_type*, *handler_def*, ***kwargs*)

Resolves the actual handler, as it can be either a string identifying the handler to load from self.__handlers__, or it can be an instantiated or non-instantiated handler class.

Parameters

- **handler_type** – The type of handler (aka the interface label)
- **handler_def** (*str, uninstantiated object, or instantiated object*) – The handler as defined in CementApp.Meta.
- **raise_error** (*boolean*) – Whether or not to raise an exception if unable to resolve the handler.

Keywork meta_defaults Optional meta-data dictionary used as defaults to pass when instantiating uninstantiated handlers. See `CementApp.Meta.meta_defaults`.

Returns The instantiated handler object.

Usage:

```
# via label (str)
log = app.handler.resolve('log', 'colorlog')

# via uninstantiated handler class
log = app.handler.resolve('log', ColorLogHanddler)

# via instantiated handler instance
log = app.handler.resolve('log', ColorLogHandler())
```

cement.core.handler.**define**(*interface*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.define()` instead.

—

Define a handler based on the provided interface. Defines a handler type based on <interface>.IMeta.label.

Parameters **interface** – The interface class that defines the interface to be implemented by handlers.

Raises cement.core.exc.InterfaceError

Raises cement.core.exc.FrameworkError

Usage:

```
from cement.core import handler

handler.define(IDatabaseHandler)
```

cement.core.handler.**defined**(*handler_type*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.defined()` instead.

—

Test whether a handler type is defined.

Parameters **handler_type** – The name or 'type' of the handler (I.e. 'logging').

Returns True if the handler type is defined, False otherwise.

Return type `boolean`

cement.core.handler.**get**(*handler_type*, *handler_label*, **args*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.get()` instead.

—

Get a handler object.

Required Arguments:

Parameters

- **handler_type** (*str*) – The type of handler (i.e. 'output')
- **handler_label** (*str*) – The label of the handler (i.e. 'json')
- **fallback** – A fallback value to return if handler_label doesn't exist.

Returns An uninstantiated handler object

Raises cement.core.exc.FrameworkError

Usage:

from cement.core import handler output = handler.get('output', 'json') output.render(dict(foo='bar'))

cement.core.handler.**list**(*handler_type*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.list()` instead.

—

Return a list of handlers for a given type.

Parameters **handler_type** – The type of handler (i.e. 'output')

Returns List of handlers that match *type*.

Return type `list`

Raises cement.core.exc.FrameworkError

cement.core.handler.**register**(*handler_obj*, *force=False*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.register()` instead.

—

Register a handler object to a handler. If the same object is already registered then no exception is raised, however if a different object attempts to be registered to the same name a FrameworkError is raised.

Parameters

- **handler_obj** – The uninstantiated handler object to register.

- **force** – Whether to allow replacement if an existing handler of the same `label` is already registered.

Raises cement.core.exc.InterfaceError

Raises cement.core.exc.FrameworkError

Usage:

```
from cement.core import handler

class MyDatabaseHandler(object):
    class Meta:
        interface = IDatabase
        label = 'mysql'

    def connect(self):
    ...

handler.register(MyDatabaseHandler)
```

cement.core.handler.**registered**(*handler_type*, *handler_label*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.registered()` instead.

—

Check if a handler is registered.

Parameters

- **handler_type** – The type of handler (interface label)
- **handler_label** – The label of the handler

Returns True if the handler is registered, False otherwise

Return type `boolean`

cement.core.handler.**resolve**(*handler_type*, *handler_def*, *raise_error=True*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.handler.resolve()` instead.

—

Resolves the actual handler, as it can be either a string identifying the handler to load from back-end.__handlers__, or it can be an instantiated or non-instantiated handler class.

Parameters

- **handler_type** – The type of handler (aka the interface label)
- **hander_def** – The handler as defined in CementApp.Meta.
- **raise_error** (*boolean*) – Whether or not to raise an exception if unable to resolve the handler.

Returns The instantiated handler object.

cement.core.hook

Cement core hooks module.

class cement.core.hook.**HookManager**(*use_backend_globals=False*)
Bases: object

Manages the hook system to define, get, run, etc hooks within the the Cement Framework and applications Built on Cement (tm).

Parameters **use_backend_globals** – Whether to use backend globals (backward compatibility and deprecated).

define(*name*)
Define a hook namespace that the application and plugins can register hooks in.

Parameters **name** – The name of the hook, stored as hooks['name']

Raises cement.core.exc.FrameworkError

Usage:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    app.hook.define('my_hook_name')
```

defined(*hook_name*)
Test whether a hook name is defined.

Parameters **hook_name** – The name of the hook. I.e. my_hook_does_awesome_things.

Returns True if the hook is defined, False otherwise.

Return type boolean

Usage:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    app.hook.defined('some_hook_name'):
        # do something about it
        pass
```

register(*name*, *func*, *weight=0*)
Register a function to a hook. The function will be called, in order of weight, when the hook is run.

Parameters

- **name** – The name of the hook to register too. I.e. pre_setup, post_run, etc.
- **func** – The function to register to the hook. This is an *un-instantiated*, non-instance method, simple function.
- **weight** (int) – The weight in which to order the hook function.

Usage:

```
from cement.core.foundation import CementApp

def my_hook_func(app):
    # do something with app?
    return True

with CementApp('myapp') as app:
```

```
    app.hook.define('my_hook_name')
    app.hook.register('my_hook_name', my_hook_func)
```

run(*name*, **args*, ***kwargs*)

Run all defined hooks in the namespace. Yields the result of each hook function run.

Parameters

- **name** – The name of the hook function.
- **args** – Additional arguments to be passed to the hook functions.
- **kwargs** – Additional keyword arguments to be passed to the hook functions.

Raises FrameworkError

Usage:

```
from cement.core.foundation import CementApp

def my_hook_func(app):
    # do something with app?
    return True

with CementApp('myapp') as app:
    app.hook.define('my_hook_name')
    app.hook.register('my_hook_name', my_hook_func)
    for res in app.hook.run('my_hook_name', self):
        # do something with the result?
        pass
```

cement.core.hook.**define**(*name*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.hook.define()` instead.

—

Define a hook namespace that plugins can register hooks in.

Parameters **name** – The name of the hook, stored as hooks['name']

Raises cement.core.exc.FrameworkError

Usage:

```
from cement.core import hook

hook.define('myhookname_hook')
```

cement.core.hook.**defined**(*hook_name*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.hook.defined()` instead.

—

Test whether a hook name is defined.

Parameters **hook_name** – The name of the hook. I.e. `my_hook_does_awesome_things`.

Returns True if the hook is defined, False otherwise.

Return type `boolean`

cement.core.hook.**register**(*name*, *func*, *weight=0*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.hook.register()` instead.

—

Register a function to a hook. The function will be called, in order of weight, when the hook is run.

Parameters

- **name** – The name of the hook to register too. I.e. `pre_setup`, `post_run`, etc.
- **func** – The function to register to the hook. This is an *un-instantiated*, non-instance method, simple function.
- **weight** (`int`) – The weight in which to order the hook function.

Usage:

```
from cement.core import hook

def my_hook(*args, **kwargs):
    # do something here
    res = 'Something to return'
    return res

hook.register('post_setup', my_hook)
```

cement.core.hook.**run**(*name*, **args*, ***kwargs*)

DEPRECATION WARNING: This function is deprecated as of Cement 2.7.x and will be removed in future versions of Cement. Use `CementApp.hook.run()` instead.

—

Run all defined hooks in the namespace. Yields the result of each hook function run.

Parameters

- **name** – The name of the hook function.
- **args** – Additional arguments to be passed to the hook functions.
- **kwargs** – Additional keyword arguments to be passed to the hook functions.

Raises FrameworkError

Usage:

```
from cement.core import hook

for result in hook.run('hook_name'):
    # do something with result from each hook function
    ...
```

cement.core.interface

Cement core interface module.

class cement.core.interface.**Attribute**(*description*)

Bases: `object`

An interface attribute definition.

Parameters **description** – The description of the attribute.

class cement.core.interface.**Interface**
Bases: object

An interface definition class. All Interfaces should subclass from here. Note that this is not an implementation and should never be used directly.

cement.core.interface.**list**()
DEPRECATION WARNING: This function is deprecated as of Cement 2.9 in favor of the *CementApp.handler.list_types()* function, and will be removed in future versions of Cement.

Return a list of defined interfaces (handler types).

Returns List of defined interfaces

Return type list

cement.core.interface.**validate**(*interface, obj, members=[], meta=['interface', 'label', 'config_defaults', 'config_section']*)
A wrapper to validate interfaces.

Parameters

- **interface** – The interface class to validate against
- **obj** – The object to validate.
- **members** – The object members that must exist.
- **meta** – The meta object members that must exist.

Raises cement.core.exc.InterfaceError

cement.core.log

Cement core log module.

class cement.core.log.**CementLogHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Log Handlers should sub-class from.

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

interface
The interface that this class implements.

alias of *ILog*

label = None
The string identifier of this handler.

class cement.core.log.**ILog**
Bases: *cement.core.interface.Interface*

This class defines the Log Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import log

class MyLogHandler(object):
    class Meta:
        interface = log.ILog
        label = 'my_log_handler'
    ...
```

class IMeta
Interface meta-data.

label = 'log'
The string identifier of the interface.

validator(*klass*, *obj*)
The interface validator function.

ILog.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

ILog.**debug**(*msg*)
Log to the 'DEBUG' facility.

Parameters **msg** – The message to log.

ILog.**error**(*msg*)
Log to the 'ERROR' facility.

Parameters **msg** – The message to log.

ILog.**fatal**(*msg*)
Log to the 'FATAL' facility.

Parameters **msg** – The message to log.

ILog.**get_level**()
Return a string representation of the log level.

ILog.**info**(*msg*)
Log to the 'INFO' facility.

Parameters **msg** – The message to log.

ILog.**set_level**()

Set the log level. Must except atleast one of: `['INFO', 'WARNING', 'ERROR', 'DEBUG', or 'FATAL']`.

ILog.**warning**(*msg*)
Log to the 'WARNING' facility.

Parameters **msg** – The message to log.

cement.core.log.**log_validator**(*klass*, *obj*)
Validates an handler implementation against the ILog interface.

cement.core.mail

Cement core mail module.

class cement.core.mail.**CementMailHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Mail Handlers should sub-class from.

Configuration Options

This handler supports the following configuration options under a

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

config_defaults = {'from_addr': 'noreply@example.com', 'cc': [], 'subject_prefix': '', 'bcc': [], 'to': [], 'subjec
Configuration default values

interface
The interface that this handler class implements.

alias of *IMail*

label = None
String identifier of this handler implementation.

class cement.core.mail.**IMail**
Bases: *cement.core.interface.Interface*

This class defines the Mail Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Configuration

Implementations much support the following configuration settings:

- **to** - Default `to` addresses (list, or comma separated depending on the ConfigHandler in use)
- **from_addr** - Default `from_addr` address
- **cc** - Default `cc` addresses (list, or comma separated depending on the ConfigHandler in use)
- **bcc** - Default `bcc` addresses (list, or comma separated depending on the ConfigHandler in use)
- **subject** - Default `subject`
- **subject_prefix** - Additional string to prepend to the `subject`

Usage

```
from cement.core import mail

class MyMailHandler(object):
    class Meta:
        interface = mail.IMail
        label = 'my_mail_handler'
    ...
```

class IMeta
Interface meta-data.

label = 'mail'
The label (or type identifier) of the interface.

validator(*klass*, *obj*)
Interface validator function.

IMail.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

Returns None

IMail.**send**(*body*, ***kwargs*)
Send a mail message. Keyword arguments override configuration defaults (cc, bcc, etc).

Parameters

- **body** (multiline string) – The message body to send
- **to** (list) – List of recipients (generally email addresses)
- **from_addr** (str) – Address (generally email) of the sender
- **cc** (list) – List of CC Recipients
- **bcc** (list) – List of BCC Recipients
- **subject** (str) – Message subject line

Returns Boolean (True if message is sent successfully, False otherwise)

Usage

```
# Using all configuration defaults
app.send('This is my message body')

# Overriding configuration defaults
app.send('My message body'
    to=['john@example.com'],
    from_addr='me@example.com',
    cc=['jane@example.com', 'rita@example.com'],
    subject='This is my subject',
    )
```

cement.core.mail.**mail_validator**(*klass*, *obj*)
Validates a handler implementation against the IMail interface.

cement.core.meta

Cement core meta functionality.

class cement.core.meta.**Meta**(***kwargs*)
Bases: object

Model that acts as a container class for a meta attributes for a larger class. It stuffs any kwarg it gets in it's init as an attribute of itself.

class cement.core.meta.**MetaMixin**(**args*, ***kwargs*)
Bases: object

Mixin that provides the Meta class support to add settings to instances of slumber objects. Meta settings cannot start with a _.

cement.core.output

Cement core output module.

class cement.core.output.**CementOutputHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Output Handlers should sub-class from.

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

interface
The interface that this class implements.

alias of *IOutput*

label = None
The string identifier of this handler.

class cement.core.output.**IOutput**
Bases: *cement.core.interface.Interface*

This class defines the Output Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import output

class MyOutputHandler(object):
    class Meta:
        interface = output.IOutput
        label = 'my_output_handler'
    ...
```

class IMeta
Interface meta-data.

label = 'output'
The string identifier of the interface.

validator(*klass*, *obj*)
The interface validator function.

IOutput.**_setup**(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

IOutput.**render**(*data_dict*, **args*, ***kwargs*)
Render the data_dict into output in some fashion. This function must access both `*args` and `**kwargs` to allow an application to mix output handlers that support different features.

Parameters **data_dict** – The dictionary whose data we need to render into output.

Returns string or unicode string or None

class cement.core.output.**TemplateOutputHandler**(**args*, ***kw*)
Bases: *cement.core.output.CementOutputHandler*

Base class for template base output handlers.

load_template(*template_path*)
Loads a template file first from `self.app._meta.template_dirs` and secondly from `self.app._meta.template_module`. The `template_dirs` have presedence.

Parameters **template_path** – The secondary path of the template **after** either `template_module` or `template_dirs` prefix (set via `CementApp.Meta`)

Returns The content of the template (str)

Raises FrameworkError if the template does not exist in either the `template_module` or `template_dirs`.

load_template_with_location(*template_path*)
Loads a template file first from `self.app._meta.template_dirs` and secondly from `self.app._meta.template_module`. The `template_dirs` have presedence.

Parameters **template_path** – The secondary path of the template **after** either `template_module` or `template_dirs` prefix (set via `CementApp.Meta`)

Returns A tuple that includes the content of the template (str), the type of template (str which is one of: `directory`, or `module`), and the `path` (str) of the directory or module)

Raises FrameworkError if the template does not exist in either the `template_module` or `template_dirs`.

`cement.core.output.`**output_validator**(*klass*, *obj*)
Validates an handler implementation against the IOutput interface.

cement.core.plugin

Cement core plugins module.

class `cement.core.plugin.`**CementPluginHandler**(**args*, ***kw*)
Bases: *cement.core.handler.CementBaseHandler*

Base class that all Plugin Handlers should sub-class from.

class Meta
Handler meta-data (can be passed as keyword arguments to the parent class).

interface
The interface that this class implements.

alias of *IPlugin*

label = None
The string identifier of this handler.

class `cement.core.plugin.`**IPlugin**
Bases: *cement.core.interface.Interface*

This class defines the Plugin Handler Interface. Classes that implement this handler must provide the methods and attributes defined below.

Implementations do *not* subclass from interfaces.

Usage:

```
from cement.core import plugin

class MyPluginHandler(object):
```

```
    class Meta:
        interface = plugin.IPlugin
        label = 'my_plugin_handler'
    ...
```

_setup(*app_obj*)
The _setup function is called during application initialization and must 'setup' the handler object making it ready for the framework or the application to make further calls to it.

Parameters **app_obj** – The application object.

get_disabled_plugins()
Returns a list of plugins that are disabled in the config.

get_enabled_plugins()
Returns a list of plugins that are enabled in the config.

get_loaded_plugins()
Returns a list of plugins that have been loaded.

load_plugin(*plugin_name*)
Load a plugin whose name is 'plugin_name'.

Parameters **plugin_name** – The name of the plugin to load.

load_plugins(*plugin_list*)
Load all plugins from plugin_list.

Parameters **plugin_list** – A list of plugin names to load.

cement.core.plugin.**plugin_validator**(*klass*, *obj*)
Validates an handler implementation against the IPlugin interface.

Cement Utility Modules

cement.utils.fs

Common File System Utilities.

cement.utils.fs.**abspath**(*path*)
Return an absolute path, while also expanding the '~' user directory shortcut.

Parameters **path** – The original path to expand.

Return type str

cement.utils.fs.**backup**(*path*, *suffix='.bak'*)
Rename a file or directory safely without overwriting an existing backup of the same name.

Parameters

- **path** – The path to the file or directory to make a backup of.
- **suffix** – The suffix to rename files with.

Returns The new path of backed up file/directory

Return type str

cement.utils.shell

Common Shell Utilities.

class cement.utils.shell.**Prompt**(*text=None*, **args*, ***kw*)

Bases: *cement.core.meta.MetaMixin*

A wrapper around *raw_input* or *input* (py3) whose purpose is to limit the redundent tasks of gather usr input. Can be used in several ways depending on the use case (simple input, options, and numbered selection).

Parameters **text** – The text displayed at the input prompt.

Usage:

Simple prompt to halt operations and wait for user to hit enter:

```
p = shell.Prompt("Press Enter To Continue", default='ENTER')
```

```
$ python myapp.py
Press Enter To Continue

$
```

Provide a numbered list for longer selections:

```
p = Prompt("Where do you live?",
        options=[
            'San Antonio, TX',
            'Austin, TX',
            'Dallas, TX',
            'Houston, TX',
            ],
        numbered = True,
        )
```

```
Where do you live?

1: San Antonio, TX
2: Austin, TX
3: Dallas, TX
4: Houston, TX

Enter the number for your selection:
```

Create a more complex prompt, and process the input from the user:

```
class MyPrompt(Prompt):
    class Meta:
        text = "Do you agree to the terms?"
        options = ['Yes', 'no', 'maybe-so']
        options_separator = '|'
        default = 'no'
        clear = True
        max_attempts = 99

    def process_input(self):
        if self.input.lower() == 'yes':
            # do something crazy
            pass
```

```
        else:
            # don't do anything... maybe exit?
            print("User doesn't agree! I'm outa here")
            sys.exit(1)

MyPrompt()
```

```
$ python myapp.py
[TERMINAL CLEAR]

Do you agree to the terms? [Yes|no|maybe-so] no
User doesn't agree! I'm outa here

$ echo $?

$ 1
```

class Meta

Optional meta-data (can also be passed as keyword arguments to the parent class).

auto = True

Whether or not to automatically prompt() the user once the class is instantiated.

case_insensitive = True

Whether to treat user input as case insensitive (only used to compare user input with available options).

clear = False

Whether or not to clear the terminal when prompting the user.

clear_command = 'clear'

Command to issue when clearing the terminal.

default = None

A default value to use if the user doesn't provide any input

max_attempts = 10

Max attempts to get proper input from the user before giving up.

max_attempts_exception = True

Raise an exception when max_attempts is hit? If not, Prompt passes the input through as *None*.

numbered = False

Display options in a numbered list, where the user can enter a number. Useful for long selections.

options = None

Options to provide to the user. If set, the input must match one of the items in the options selection.

options_separator = ','

Separator to use within the option selection (non-numbered)

selection_text = 'Enter the number for your selection:'

The text to display along with the numbered selection for user input.

Prompt.**process_input**()

Does not do anything. Is intended to be used in a sub-class to handle user input after it is prompted.

Prompt.**prompt**()

Prompt the user, and store their input as *self.input*.

cement.utils.shell.**exec_cmd**(*cmd_args*, **args*, ***kw*)

Execute a shell call using Subprocess. All additional **args* and ***kwargs* are passed directly to subprocess.Popen. See Subprocess for more information on the features of *Popen()*.

Parameters

- **cmd_args** (*list.*) – List of command line arguments.
- **args** – Additional arguments are passed to Popen().
- **kwargs** – Additional keyword arguments are passed to Popen().

Returns The (stdout, stderror, return_code) of the command.

Return type tuple

Usage:

```
from cement.utils import shell

stdout, stderr, exitcode = shell.exec_cmd(['echo', 'helloworld'])
```

cement.utils.shell.**exec_cmd2**(*cmd_args*, **args*, ***kw*)

Similar to exec_cmd, however does not capture stdout, stderr (therefore allowing it to print to console). All additional **args* and ***kwargs* are passed directly to subprocess.Popen. See Subprocess for more information on the features of *Popen()*.

Parameters

- **cmd_args** (*list.*) – List of command line arguments.
- **args** – Additional arguments are passed to Popen().
- **kwargs** – Additional keyword arguments are passed to Popen().

Returns The integer return code of the command.

Return type int

Usage:

```
from cement.utils import shell

exitcode = shell.exec_cmd2(['echo', 'helloworld'])
```

cement.utils.shell.**spawn_process**(*target*, *start=True*, *join=False*, **args*, ***kwargs*)

A quick wrapper around multiprocessing.Process(). By default the start() function will be called before the spawned process object is returned. See MultiProcessing for more information on the features of *Process()*.

Parameters

- **target** – The target function to execute in the sub-process.
- **start** – Call start() on the process before returning the process object.
- **join** – Call join() on the process before returning the process object. Only called if start=True.
- **args** – Additional arguments are passed to Process().
- **kwargs** – Additional keyword arguments are passed to Process().

Returns The process object returned by Process().

Usage:

```
from cement.utils import shell

def add(a, b):
    print(a + b)

p = shell.spawn_process(add, args=(12, 27))
p.join()
```

cement.utils.shell.**spawn_thread**(*target*, *start=True*, *join=False*, **args*, ***kwargs*)

A quick wrapper around threading.Thread(). By default the start() function will be called before the spawned thread object is returned See Threading for more information on the features of *Thread()*.

Parameters

- **target** – The target function to execute in the thread.
- **start** – Call start() on the thread before returning the thread object.
- **join** – Call join() on the thread before returning the thread object. Only called if start=True.
- **args** – Additional arguments are passed to Thread().
- **kwargs** – Additional keyword arguments are passed to Thread().

Returns The thread object returned by Thread().

Usage:

```
from cement.utils import shell

def add(a, b):
    print(a + b)

t = shell.spawn_thread(add, args=(12, 27))
t.join()
```

cement.utils.misc

Misc utilities.

cement.utils.misc.**init_defaults**(**sections*)

Returns a standard dictionary object to use for application defaults. If sections are given, it will create a nested dict for each section name.

Parameters **sections** – Section keys to create nested dictionaries for.

Returns Dictionary of nested dictionaries (sections)

Return type dict

```
from cement.core import foundation
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'section2', 'section3')
defaults['myapp']['debug'] = False
defaults['section2']['foo'] = 'bar
defaults['section3']['foo2'] = 'bar2'

app = foundation.CementApp('myapp', config_defaults=defaults)
```

cement.utils.misc.**is_true**(*item*)
Given a value, determine if it is one of [True, 'True', 'true', 1, '1'].

Parameters **item** – The item to convert to a boolean.

Returns True if *item* is in `[True, 'True', 'true', 1, '1']`, False otherwise.

Return type boolean

cement.utils.misc.**minimal_logger**(*namespace*, *debug=False*)
Setup just enough for cement to be able to do debug logging. This is the logger used by the Cement framework, which is setup and accessed before the application is functional (and more importantly before the applications log handler is usable).

Parameters

- **namespace** – The logging namespace. This is generally '__name__' or anything you want.
- **debug** (*boolean*) – Toggle debug output. Default: False

Returns Logger object

```
from cement.utils.misc import minimal_logger
LOG = minimal_logger('cement')
LOG.debug('This is a debug message')
```

cement.utils.misc.**rando**(*salt=None*)
Generate a random MD5 hash for whatever purpose. Useful for testing or any other time that something random is required.

Parameters **salt** – Optional 'salt', if None then random() is used.

Returns Random MD5 hash (str).

cement.utils.misc.**random**() → x in the interval [0, 1).

cement.utils.misc.**wrap**(*text*, *width=77*, *indent=''*, *long_words=False*, *hyphens=False*)
Wrap text for cleaner output (this is a simple wrapper around *textwrap.TextWrapper* in the standard library).

Parameters

- **text** – The text to wrap
- **width** – The max width of a line before breaking
- **indent** – String to prefix subsequent lines after breaking
- **long_words** – Break on long words
- **hyphens** – Break on hyphens

Returns str(text)

cement.utils.test

Cement testing utilities.

class cement.utils.test.**CementTestCase**(**args*, ***kw*)
Bases: unittest.case.TestCase

A sub-class of unittest.TestCase.

app_class
The test class that is used by self.make_app to create an app.

alias of *TestApp*

eq(*a*, *b*, *msg=None*)
Shorthand for 'assert a == b, "%r != %r" % (a, b)'.

make_app(**args*, ***kw*)
Create a generic app using TestApp. Arguments and Keyword Arguments are passed to the app.

ok(*expr*, *msg=None*)
Shorthand for assert.

reset_backend()
Remove all registered hooks and handlers from the backend.

setUp()
Sets up self.app with a generic TestApp(). Also resets the backend hooks and handlers so that everytime an app is created it is setup clean each time.

tearDown()
Tears down the test environment (if necessary), removes any temporary files/directories, etc.

class cement.utils.test.**TestApp**(*label=None*, ***kw*)
Bases: *cement.core.foundation.CementApp*

Basic CementApp for generic testing.

class cement.utils.test.**raises**
Test must raise one of expected exceptions to pass.

Example use:

```
@raises(TypeError, ValueError)
def test_raises_type_error():
    raise TypeError("This test passes")

@raises(Exception)
def test_that_fails_by_passing():
    pass
```

If you want to test many assertions about exceptions in a single test, you may want to use *assert_raises* instead.

cement.utils.test.**ok**(*expr*, *msg=None*)
Shorthand for assert. Saves 3 whole characters!

cement.utils.test.**eq**(*a*, *b*, *msg=None*)
Shorthand for 'assert a == b, "%r != %r" % (a, b)

Cement Extension Modules

cement.ext.ext_alarm

The Alarm Extension provides easy access to setting an application alarm to handle timing out operations. See the Python Signal Library.

Requirements

- No external dependencies.
- Only available on Unix/Linux

Configuration

This extension does not honor any application configuration settings.

Usage

```
import time
from cement.core.foundation import CementApp
from cement.core.exc import CaughtSignal


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        exit_on_close = True
        extensions = ['alarm']


with MyApp() as app:
    try:
        app.run()
        app.alarm.set(3, "The operation timed out after 3 seconds!")

        # do something that takes time to operate
        time.sleep(5)

        app.alarm.stop()

    except CaughtSignal as e:
        print(e.msg)
        app.exit_code = 1
```

Looks like:

```
$ python myapp.py
ERROR: The operation timed out after 3 seconds!
Caught signal 14
```

class cement.ext.ext_alarm.**AlarmManager**(**args*, ***kw*)

Bases: object

Lets the developer easily set and stop an alarm. If the alarm exceeds the given time it will raise signal.SIGALRM.

set(*time*, *msg*)

Set the application alarm to time seconds. If the time is exceeded signal.SIGALRM is raised.

Parameters

- **time** – The time in seconds to set the alarm to.

- **msg** – The message to display if the alarm is triggered.

stop()
Stop the application alarm.

cement.ext.ext_argcomplete

The Argcomplete Extension provides the necessary hooks to utilize the Argcomplete Library, and perform auto-completion of command line arguments/options/sub-parsers/etc.

Requirements

- Argcomplete (`pip install argcomplete`)
- Argparse

This extension currently only works when using *cement.ext.ext_argparse.ArgparseArgumentHandler* (default) and *cement.ext.ext_argparse.ArgparseController* (new in Cement 2.8). It will not work with *cement.core.controller.CementBaseController*.

Configuration

This extension does not honor any application configuration settings.

Usage

myapp.py

```
#!/usr/bin/env python

from cement.core.foundation import CementApp
from cement.ext.ext_argparse import ArgparseController, expose


class BaseController(ArgparseController):
    class Meta:
        label = 'base'
        arguments = [
            (['-f', '--foo'], dict(help='base foo option', dest='foo'))
        ]

    @expose(hide=True)
    def default(self):
        print('Inside BaseController.default')

    @expose()
    def command1(self):
        print('Inside BaseController.command1')


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['argcomplete']
```

```
        handlers = [BaseController]

with MyApp() as app:
    app.run()
```

Note the #! line, which allows us to call our script directly (specifically for this example). The Argcomplete library requires the end-user to modify their environment in order to perform auto-completion. For this example, we are using a non-global option for demonstration purposes only. In the *real world* you will need to setup Argcomplete for your actual application entry-point name (i.e. `myapp` if installed as `/usr/bin/myapp`, etc).

```
$ eval "$(register-python-argcomplete myapp.py)"

$ ./myapp.py [tab][tab]

--debug                                 -h
-o                                      --help
--quiet                                 command1
                                        default
```

See the Argcomplete Documentation on how to properly integrate it's usage into your application deployment. This extension simply enables Argcomplete to do it's thing on application startup.

cement.ext.ext_argparse

The Argparse Extension provides argument handling based on `argparse.ArgumentParser`, and is the default argument handler used by *`cement.core.foundation.CementApp`*. In addition, this extension also provides *`ArgparseController`* that enables rapid development via application controllers based on Argparse.

Requirements

- Python 2.7+, Python 3+
- Some features of *`ArgparseController`* are only available in Python 3 including controller and function/command `aliases` (Python 3+) and controller default functions/command (Python 3.4+).

Configuration

This extension does not have any application configuration settings.

Usage

The following is an example application using both the *`ArgparseArgumentHandler`* and *`ArgparseController`*. Note that the default `arg_handler` is already set to `ArgparseArgumentHandler`` by `CementApp`.

```
from cement.core.foundation import CementApp
from cement.ext.ext_argparse import ArgparseController, expose


class BaseController(ArgparseController):
    class Meta:
        label = 'base'
```

```
        arguments = [
            (['--base-foo'], dict(help='base foo option')),
        ]

    @expose(hide=True)
    def default(self):
        # Note: Default commands are only available in Python 3.4+
        print('Inside BaseController.default')

        if self.app.pargs.base_foo:
            # do something with self.app.pargs.base_foo
            print('Base Foo > %s' % self.app.pargs.base_foo)

    @expose(
        arguments=[
            (['--command1-opt'],
             dict(help='option under command1', action='store_true'))
        ],
        aliases=['cmd1'],
        help='command1 is a sub-command under myapp base controller',
    )
    def command1(self):
        print('Inside BaseController.command1')

        if self.app.pargs.command1_opt:
            # do something with self.app.pargs.command1_opt
            pass


class EmbeddedController(ArgparseController):
    class Meta:
        label = 'embedded_controller'
        stacked_on = 'base'
        stacked_type = 'embedded'

    @expose(help="command2 embedded under base controller")
    def command2(self):
        print('Inside EmbeddedController.command2')


class NestedController(ArgparseController):
    class Meta:
        label = 'nested_controller'
        stacked_on = 'base'
        stacked_type = 'nested'
        arguments = [
            (['--nested-opt'],
             dict(help='option under nested-controller')),
        ]

    @expose(help="command3 under nested-controller")
    def command3(self):
        print('Inside NestedController.command3')


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
```

```
        handlers = [
            BaseController,
            EmbeddedController,
            NestedController
        ]


with MyApp() as app:
    app.run()
```

The above looks like:

```
$ python myapp.py --help
usage: myapp.py [-h] [--debug] [--quiet] [--base-foo BASE_FOO]
                {nested-controller,command1,cmd1,default,command2} ...

optional arguments:
  -h, --help            show this help message and exit
  --debug               toggle debug output
  --quiet               suppress all output
  --base-foo BASE_FOO   base foo option

sub-commands:
  {nested-controller,command1,cmd1,default,command2}
    nested-controller   nested-controller controller
    command1 (cmd1)     command1 is a sub-command under base controller
    command2            command2 embedded under base controller


$ python myapp.py --base-foo bar
Inside BaseController.default
Base Foo > bar


$ python myapp.py command1 --help
usage: myapp.py command1 [-h] [--command1-opt]

optional arguments:
  -h, --help      show this help message and exit
  --command1-opt  option under command1


$ python myapp.py command1
Inside BaseController.command1


$ python myapp.py command2
Inside EmbeddedController.command2


$ python myapp.py nested-controller --help
usage: myapp.py nested-controller [-h] [--nested-opt] {command3} ...

optional arguments:
  -h, --help            show this help message and exit
  --nested-opt          option under nested-controller

sub-commands:
```

```
  {command3}
    command3            command3 under nested-controller


$ python myapp.py nested-controller command3
Inside NestedController.command3
```

class cement.ext.ext_argparse.**ArgparseArgumentHandler**(**args*, ***kw*)

Bases: argparse.ArgumentParser, *cement.core.arg.CementArgumentHandler*

This class implements the *cement.core.arg.IArgument* interface, and sub-classes from argparse.ArgumentParser. Please reference the argparse documentation for full usage of the class.

Arguments and Keyword arguments are passed directly to ArgumentParser on initialization.

class Meta

Handler meta-data.

ignore_unknown_arguments = False

Whether or not to ignore any arguments passed that are not defined. Default behavoir by Argparse is to raise an "unknown argument" exception by Argparse.

This affectively triggers the difference between using parse_args and parse_known_args. Unknown arguments will be accessible as unknown_args.

interface

The interface that this class implements.

alias of IArgument

label = 'argparse'

The string identifier of the handler.

ArgparseArgumentHandler.**add_argument**(**args*, ***kw*)

Add an argument to the parser. Arguments and keyword arguments are passed directly to ArgumentParser.add_argument(). See the argparse.ArgumentParser documentation for help.

ArgparseArgumentHandler.**parse**(*arg_list*)

Parse a list of arguments, and return them as an object. Meaning an argument name of 'foo' will be stored as parsed_args.foo.

Parameters **arg_list** – A list of arguments (generally sys.argv) to be parsed.

Returns object whose members are the arguments parsed.

class cement.ext.ext_argparse.**ArgparseController**(**args*, ***kw*)

Bases: *cement.core.handler.CementBaseHandler*

This is an implementation of the *cement.core.controller.IController* interface, but as a base class that application controllers should subclass from. Registering it directly as a handler is useless.

NOTE: This handler **requires** that the applications arg_handler be argparse. If using an alternative argument handler you will need to write your own controller base class or modify this one.

NOTE: This is a re-implementation of *cement.core.controller.CementBaseController*. In the future, this class will eventually replace it as the default.

Usage:

```
from cement.ext.ext_argparse import ArgparseController

class Base(ArgparseController):
```

```
    class Meta:
        label = 'base'
        description = 'description at the top of --help'
        epilog = "the text at the bottom of --help."
        arguments = [
            (['-f', '--foo'], dict(help='my foo option', dest='foo')),
        ]

class Second(ArgparseController):
    class Meta:
        label = 'second'
        stacked_on = 'base'
        stacked_type = 'embedded'
        arguments = [
            (['--foo2'], dict(help='my foo2 option', dest='foo2')),
        ]
```

class Meta

Controller meta-data (can be passed as keyword arguments to the parent class).

aliases = []

A list of aliases for the controller/sub-parser. **Only available in Python > 3**.

arguments = []

Arguments to pass to the argument_handler. The format is a list of tuples whos items are a (list, dict). Meaning:

```
[ ( ['-f', '--foo'], dict(help='foo option', dest='foo') ), ]
```

This is equivelant to manually adding each argument to the argument parser as in the following example:

```
add_argument('-f', '--foo', help='foo option', dest='foo')
```

config_defaults = {}

Configuration defaults (type: dict) that are merged into the applications config object for the config_section mentioned above.

config_section = None

A config [section] to merge config_defaults into. Cement will default to controller.<label> if None is set.

default_func = 'default'

Function to call if no sub-command is passed. Note that this can **not** start with an _ due to backward compatibility restraints in how Cement discovers and maps commands.

Note: Currently, default function/sub-command only functions on Python > 3.4. Previous versions of Python/Argparse will throw the exception `error: too few arguments`.

description = None

Description for the sub-parser group in help output.

epilog = None

The text that is displayed at the bottom when `--help` is passed.

help = None

Text for the controller/sub-parser group in help output (for nested stacked controllers only).

hide = False

Whether or not to hide the controller entirely.

interface
alias of `IController`

label = 'base'
The string identifier for the controller.

parser_options = {}
Additional keyword arguments passed when `ArgumentParser.add_parser()` is called to create this controller sub-parser. WARNING: This could break things, use at your own risk. Useful if you need additional features from Argparse that is not built into the controller Meta-data.

stacked_on = 'base'
A label of another controller to 'stack' commands/arguments on top of.

stacked_type = 'embedded'
Whether to embed commands and arguments within the parent controller's namespace, or to nest this controller under the parent controller (making it a sub-command). Must be one of `['embedded', 'nested']`.

subparser_options = {}
Additional keyword arguments passed when `ArgumentParser.add_subparsers()` is called to create this controller namespace. WARNING: This could break things, use at your own risk. Useful if you need additional features from Argparse that is not built into the controller Meta-data.

title = 'sub-commands'
The title for the sub-parser group in help output.

usage = None
The text that is displayed at the top when `--help` is passed. Defaults to Argparse standard usage.

`ArgparseController._post_argument_parsing()`
Called on every controller just after arguments are parsed (assuming that the parser hasn't thrown an exception). Provides an alternative means of handling passed arguments. Note that, this function is called on every controller, regardless of what namespace and sub-command is eventually going to be called. Therefore, every controller can handle their arguments if the user passed them.

For example:

```
$ myapp --foo bar some-controller --foo2 bar2 some-command
```

In the above, the `base` controller (or a nested controller) would handle `--foo`, while `some-controller` would handle `foo2` before `some-command` is executed.

```
class Base(ArgparseController):
    class Meta:
        label = 'base'

        arguments = [
            (['-f', '--foo'],
             dict(help='my foo option', dest=foo)),
        ]

    def _post_argument_parsing(self):
        if self.app.pargs.foo:
            print('Got Foo Option Before Controller Dispatch')
```

Note that `self._parser` within a controller is that individual controllers `sub-parser`, and is not the root parser `app.args` (unless you are the `base` controller, in which case `self._parser` is synonymous with `app.args`).

ArgparseController.**_pre_argument_parsing**()
Called on every controller just before arguments are parsed. Provides an alternative means of adding arguments to the controller, giving more control than using `Meta.arguments`.

```
class Base(ArgparseController):
    class Meta:
        label = 'base'

    def _pre_argument_parsing(self):
        p = self._parser
        p.add_argument('-f', '--foo',
                       help='my foo option',
                       dest='foo')

    def _post_argument_parsing(self):
        if self.app.pargs.foo:
            print('Got Foo Option Before Controller Dispatch')
```

ArgparseController.**_setup**(*app*)
See *IController._setup()*.

class cement.ext.ext_argparse.**expose**(*hide=False*, *arguments=[]*, ***parser_options*)
Bases: object

Used to expose functions to be listed as sub-commands under the controller namespace. It also decorates the function with meta-data for the argument parser.

Parameters

- **hide** (boolean) – Whether the command should be visible.
- **arguments** – List of tuples that define arguments to add to this commands sub-parser.
- **parser_options** (dict) – Additional options to pass to Argparse.

Usage:

```
from cement.ext.ext_argparse import ArgparseController, expose

class Base(ArgparseController):
    class Meta:
        label = 'base'

    # Note: Default functions only work in Python > 3.4
    @expose(hide=True)
    def default(self):
        print("In Base.default()")

    @expose(
        help='this is the help message for my_command',
        aliases=['my_cmd'], # only available in Python 3+
        arguments=[
            (['-f', '--foo'],
             dict(help='foo option', action='store', dest='foo')),
        ]
    )
    def my_command(self):
        print("In Base.my_command()")
```

cement.ext.ext_colorlog

The ColorLog Extension provides logging based on the standard `logging` module and is a drop-in replacement for the default log handler *cement.ext.ext_logging.LoggingLogHandler*.

Requirements

- ColorLog (`pip install colorlog`)

Configuration

This handler honors all of the same configuration settings as the `LoggingLogHandler` including:

- level
- file
- to_console
- rotate
- max_bytes
- max_files

In addition, it also supports:

- colorize_file_log
- colorize_console_log

A sample config section (in any config file) might look like:

```
[log.colorlog]
file = /path/to/config/file
level = info
to_console = true
rotate = true
max_bytes = 512000
max_files = 4
colorize_file_log = false
colorize_console_log = true
```

Usage

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['colorlog']
        log_handler = 'colorlog'

with MyApp() as app:
    app.run()
    app.log.debug('This is my debug message')
    app.log.info('This is my info message')
```

```
    app.log.warning('This is my warning message')
    app.log.error('This is my error message')
    app.log.fatal('This is my critical message')
```

The colors can be customized by passing in a `colors` dictionary mapping overriding the `ColorLogHandler.Meta.colors` meta-data:

```
from cement.core.foundation import CementApp
from cement.ext.ext_colorlog import ColorLogHandler

COLORS = {
    'DEBUG':    'cyan',
    'INFO':     'green',
    'WARNING':  'yellow',
    'ERROR':    'red',
    'CRITICAL': 'red,bg_white',
}

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        log_handler = ColorLogHandler(colors=COLORS)
```

Or by sub-classing and creating your own custom class:

```
from cement.core.foundation import CementApp
from cement.ext.ext_colorlog import ColorLogHandler

class MyCustomLog(ColorLogHandler):
    class Meta:
        label = 'my_custom_log'
        colors = {
            'DEBUG':    'cyan',
            'INFO':     'green',
            'WARNING':  'yellow',
            'ERROR':    'red',
            'CRITICAL': 'red,bg_white',
        }

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        log_handler = MyCustomLog
```

class cement.ext.ext_colorlog.**ColorLogHandler**(**args*, ***kw*)

Bases: *cement.ext.ext_logging.LoggingLogHandler*

This class implements the *cement.core.log.ILog* interface. It is a sub-class of *cement.ext.ext_logging.LoggingLogHandler* which is based on the standard `logging` library, and adds colorized console output using the ColorLog library.

Note This extension has an external dependency on `colorlog`. You must include `colorlog` in your applications dependencies as Cement explicitly does **not** include external dependencies for optional extensions.

class Meta

Handler meta-data.

colors = {'DEBUG': 'cyan', 'INFO': 'green', 'WARNING': 'yellow', 'CRITICAL': 'red,bg_white', 'ERROR': 'red
Color mapping for each log level

config_defaults = {'to_console': True, 'max_files': 4, 'colorize_file_log': False, 'level': 'INFO', 'rotate': False,
Default configuration settings. Will be overridden by the same settings in any application configuration file under a `[log.colorlog]` block.

formatter_class
Formatter class to use for colorized logging

alias of `ColoredFormatter`

formatter_class_without_color
Formatter class to use for non-colorized logging (non-tty, file, etc)

alias of `Formatter`

label = 'colorlog'
The string identifier of the handler.

cement.ext.ext_configobj

The ConfigObj Extension provides configuration handling based on configobj. It is a drop-in replacement for the default config handler *cement.ext.ext_configparser.ConfigParserConfigHandler*.

One of the primary features of ConfigObj is that you can access the application configuration as a dictionary object.

Requirements

- ConfigObj (`pip install configobj`)

Configuration

This extension does not honor any application configuration settings.

Usage

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['configobj']
        config_handler = 'configobj'

with MyApp() as app:
    app.run()

    # get a config setting
    app.config['myapp']['foo']

    # set a config setting
    app.config['myapp']['foo'] = 'bar2'

    # etc.
```

class cement.ext.ext_configobj.**ConfigObjConfigHandler**(**args*, ***kw*)
Bases: *cement.core.config.CementConfigHandler*, ConfigObj

This class implements the *IConfig* interface, and sub-classes from configobj.ConfigObj, which is an external library and not included with Python. Please reference the ConfigObj documentation for full usage of the class.

Arguments and keyword arguments are passed directly to ConfigObj on initialization.

class Meta
Handler meta-data.

interface
alias of IConfig

ConfigObjConfigHandler.**_parse_file**(*file_path*)
Parse a configuration file at *file_path* and store it.

Parameters **file_path** – The file system path to the configuration file.

Returns boolean (True if file was read properly, False otherwise)

ConfigObjConfigHandler.**add_section**(*section*)
Add a section to the configuration.

Parameters **section** – The configuration [section] to add.

ConfigObjConfigHandler.**get**(*section*, *key*)
Get a value for a given key under section.

Parameters

- **section** – The configuration [section].
- **key** – The configuration key under the section.

Returns unknown (the value of the key)

ConfigObjConfigHandler.**get_section_dict**(*section*)
Return a dict representation of a section.

Parameters **section** – The section of the configuration. I.e. [block_section]

Returns dict

ConfigObjConfigHandler.**get_sections**()
Return a list of [section] that exist in the configuration.

Returns list

ConfigObjConfigHandler.**has_section**(*section*)
Return True/False whether the configuration [section] exists.

Parameters **section** – The section to check for.

Returns bool

ConfigObjConfigHandler.**keys**(*section*)
Return a list of keys for a given section.

Parameters **section** – The configuration [section].

ConfigObjConfigHandler.**merge**(*dict_obj*, *override=True*)
Merge a dictionary into our config. If override is True then existing config values are overridden by those passed in.

Parameters

- **dict_obj** – A dictionary of configuration keys/values to merge into our existing config (self).
- **override** – Whether or not to override existing values in the config.

Returns None

ConfigObjConfigHandler.**set**(*section*, *key*, *value*)
Set a configuration key value under [section].

Parameters

- **section** – The configuration [section].
- **key** – The configuration key under the section.
- **value** – The value to set the key to.

Returns None

cement.ext.ext_configparser

The ConfigParser Extension provides configuration handling based on the standard `ConfigParser`, and is the default configuration handler used by Cement.

Requirements

- No external dependencies.

Configuration

This extension does not honor any application configuration settings.

Usage

```
from cement.core.foundation import CementApp

with CementApp() as app:
    app.run()

    # get a config setting
    app.config.get('myapp', 'foo')

    # set a config setting
    app.config.set('myapp', 'foo', 'bar2')

    # etc.
```

class cement.ext.ext_configparser.**ConfigParserConfigHandler**(**args*, ***kw*)
Bases: *cement.core.config.CementConfigHandler*, ConfigParser.RawConfigParser

This class is an implementation of the *IConfig* interface. It handles configuration file parsing and the like by sub-classing from the standard ConfigParser library. Please see the ConfigParser documentation for full usage of the class.

Additional arguments and keyword arguments are passed directly to RawConfigParser on initialization.

class **Meta**
Handler meta-data.

interface
The interface that this handler implements.

alias of `IConfig`

label = 'configparser'
The string identifier of this handler.

`ConfigParserConfigHandler.`**_parse_file**(*file_path*)
Parse a configuration file at *file_path* and store it.

Parameters **file_path** – The file system path to the configuration file.

Returns boolean (True if file was read properly, False otherwise)

`ConfigParserConfigHandler.`**add_section**(*section*)
Adds a block section to the config.

Parameters **section** – The section to add.

`ConfigParserConfigHandler.`**get_section_dict**(*section*)
Return a dict representation of a section.

Parameters **section** – The section of the configuration. I.e. [block_section]

Returns Dictionary reprisentation of the config section.

Return type `dict`

`ConfigParserConfigHandler.`**get_sections**()
Return a list of configuration sections or [blocks].

Returns List of sections.

Return type `list`

`ConfigParserConfigHandler.`**keys**(*section*)
Return a list of keys within 'section'.

Parameters **section** – The config section (I.e. [block_section]).

Returns List of keys in the *section*.

Return type `list`

`ConfigParserConfigHandler.`**merge**(*dict_obj*, *override=True*)
Merge a dictionary into our config. If override is True then existing config values are overridden by those passed in.

Parameters

- **dict_obj** – A dictionary of configuration keys/values to merge into our existing config (self).
- **override** – Whether or not to override existing values in the config.

cement.ext.ext_daemon

The Daemon Extension enables applications Built on Cement (tm) to easily perform standard daemonization functions.

Requirements

- Python 2.6+, Python 3+
- Available on Unix/Linux only

Features

- Configurable runtime user and group
- Adds the `--daemon` command line option
- Adds `app.daemonize()` function to trigger daemon functionality where necessary (either in a cement `pre_run` hook or an application controller sub-command, etc)
- Manages a pid file including cleanup on `app.close()`

Configuration

The daemon extension is configurable with the following settings under the [daemon] section.

- **user** - The user name to run the process as. Default: os.getlogin()
- **group** - The group name to run the process as. Default: The primary group of the 'user'.
- **dir** - The directory to run the process in. Default: /
- **pid_file** - The filesystem path to store the PID (Process ID) file. Default: None
- **umask** - The umask value to pass to os.umask(). Default: 0

Configurations can be passed as defaults to a CementApp:

```
from cement.core.foundation import CementApp
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'daemon')
defaults['daemon']['user'] = 'myuser'
defaults['daemon']['group'] = 'mygroup'
defaults['daemon']['dir'] = '/var/lib/myapp/'
defaults['daemon']['pid_file'] = '/var/run/myapp/myapp.pid'
defaults['daemon']['umask'] = 0

app = CementApp('myapp', config_defaults=defaults)
```

Application defaults are then overridden by configurations parsed via a `[demon]` config section in any of the applications configuration paths. An example configuration block would look like:

```
[daemon]
user = myuser
group = mygroup
dir = /var/lib/myapp/
pid_file = /var/run/myapp/myapp.pid
umask = 0
```

Usage

The following example shows how to add the daemon extension, as well as trigger daemon functionality before `app.run()` is called.

```
from time import sleep
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['daemon']

with MyApp() as app:
    app.daemonize()
    app.run()

    count = 0
    while True:
        count = count + 1
        print('Iteration: %s' % count)
        sleep(10)
```

An alternative to the above is to put the `daemonize()` call within a framework hook:

```
def make_daemon(app):
    app.daemonize()

def load(app):
    app.hook.register('pre_run', make_daemon)
```

Finally, some applications may prefer to only daemonize certain sub-commands rather than the entire parent application. For example:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose


class MyBaseController(CementBaseController):
    class Meta:
        label = 'base'

    @expose(help="run the daemon command.")
    def run_forever(self):
        from time import sleep
        self.app.daemonize()

        count = 0
        while True:
            count = count + 1
            print(count)
            sleep(10)

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = MyBaseController
        extensions = ['daemon']
```

```
with MyApp() as app:
    app.run()
```

By default, even after `app.daemonize()` is called... the application will continue to run in the foreground, but will still manage the pid and user/group switching. To detach a process and send it to the background you simply pass the `--daemon` option at command line.

```
$ python example.py --daemon

$ ps -x | grep example
37421 ??         0:00.01 python example2.py --daemon
37452 ttys000    0:00.00 grep example
```

Daemizing Without Commandline Option

Some use cases might require daemonizing the process without having to always pass the `--daemon` option, or where passing the option might be redundant. You can work around that programatically by simply overriding the `daemon` argument value in orcer to force daemonization even if `--daemon` wasn't passed.

```
app.pargs.daemon = True
app.daemonize()
```

Note that this would only work **after** arguments have been parsed (i.e. after `app.run()` is called).

class `cement.ext.ext_daemon.`**`Environment`**`(`***kw*`)`

Bases: `object`

This class provides a mechanism for altering the running processes environment.

Optional Arguments:

Parameters

- **`stdin`** – A file to read STDIN from. Default: `/dev/null`
- **`stdout`** – A file to write STDOUT to. Default: `/dev/null`
- **`stderr`** – A file to write STDERR to. Default: `/dev/null`
- **`dir`** – The directory to run the process in.
- **`pid_file`** – The filesystem path to where the PID (Process ID) should be written to. Default: None
- **`user`** – The user name to run the process as. Default: `os.getlogin()`
- **`group`** – The group name to run the process as. Default: The primary group of `os.getlogin()`.
- **`umask`** – The umask to pass to os.umask(). Default: `0`

`_write_pid_file``()`

Writes `os.getpid()` out to `self.pid_file`.

`daemonize``()`

Fork the current process into a daemon.

References:

UNIX Programming FAQ: 1.7 How do I get my program to act like a daemon? http://www.unixguide.net/unix/programming/1.7.shtml http://www.faqs.org/faqs/unix-faq/programmer/faq/

Advanced Programming in the Unix Environment

23. Richard Stevens, 1992, Addison-Wesley, ISBN 0-201-56317-7.

switch()
Switch the current process's user/group to `self.user`, and `self.group`. Change directory to `self.dir`, and write the current pid out to `self.pid_file`.

`cement.ext.ext_daemon.`**cleanup**(*app*)
After application run time, this hook just attempts to clean up the pid_file if one was set, and exists.

`cement.ext.ext_daemon.`**daemonize**()
This function switches the running user/group to that configured in `config['daemon']['user']` and `config['daemon']['group']`. The default user is `os.getlogin()` and the default group is that user's primary group. A pid_file and directory to run in is also passed to the environment.

It is important to note that with the daemon extension enabled, the environment will switch user/group/set pid/etc regardless of whether the `--daemon` option was passed at command line or not. However, the process will only 'daemonize' if the option is passed to do so. This allows the program to run exactly the same in forground or background.

`cement.ext.ext_daemon.`**extend_app**(*app*)
Adds the `--daemon` argument to the argument object, and sets the default `[daemon]` config section options.

cement.ext.ext_dummy

The Dummy Extension provides several 'placeholder' type handlers to either mock operations or provide local-only usage during development. A perfect example is the *DummyMailHandler* that can be use during development or staging to prevent real email messages from being sent externally.

Requirements

- No external dependencies

Configuration

- See each handler's documentation regarding what configurations they support.

Usage

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['dummy']
        output_handler = 'dummy'
        mail_handler = 'dummy'

with MyApp() as app:
    app.run()
```

class cement.ext.ext_dummy.**DummyMailHandler**(**args*, ***kw*)

Bases: *cement.core.mail.CementMailHandler*

This class implements the *cement.core.mail.IMail* interface, but is intended for use in development as no email is actually sent.

Usage

```
class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        mail_handler = 'dummy'

with MyApp() as app:
    app.run()

    app.mail.send('This is my fake message',
        subject='This is my subject',
        to=['john@example.com', 'rita@example.com'],
        from_addr='me@example.com',
        )
```

The above will print the following to console:

```
======================================================================
DUMMY MAIL MESSAGE
----------------------------------------------------------------------

To: john@example.com, rita@example.com
From: me@example.com
CC:
BCC:
Subject: This is my subject

---

This is my fake message

----------------------------------------------------------------------
```

Configuration

This handler supports the following configuration settings:

- **to** - Default `to` addresses (list, or comma separated depending on the ConfigHandler in use)
- **from_addr** - Default `from_addr` address
- **cc** - Default `cc` addresses (list, or comma separated depending on the ConfigHandler in use)
- **bcc** - Default `bcc` addresses (list, or comma separated depending on the ConfigHandler in use)
- **subject** - Default `subject`
- **subject_prefix** - Additional string to prepend to the `subject`

You can add these to any application configuration file under a `[mail.dummy]` section, for example:

~/.myapp.conf

```
[myapp]
```

```
# set the mail handler to use
mail_handler = dummy


[mail.dummy]

# default to addresses (comma separated list)
to = me@example.com

# default from address
from = someone_else@example.com

# default cc addresses (comma separated list)
cc = jane@example.com, rita@example.com

# default bcc addresses (comma separated list)
bcc = blackhole@example.com, someone_else@example.com

# default subject
subject = This is The Default Subject

# additional prefix to prepend to the subject
subject_prefix = MY PREFIX >
```

class Meta

Handler meta-data.

label = 'dummy'

Unique identifier for this handler

DummyMailHandler.**send**(*body*, ***kw*)

Mimic sending an email message, but really just print what would be sent to console. Keyword arguments override configuration defaults (cc, bcc, etc).

Parameters

- **body** (multiline string) – The message body to send
- **to** (list) – List of recipients (generally email addresses)
- **from_addr** (str) – Address (generally email) of the sender
- **cc** (list) – List of CC Recipients
- **bcc** (list) – List of BCC Recipients
- **subject** (str) – Message subject line

Returns Boolean (True if message is sent successfully, False otherwise)

Usage

```
# Using all configuration defaults
app.mail.send('This is my message body')

# Overriding configuration defaults
app.mail.send('My message body'
    to=['john@example.com'],
    from_addr='me@example.com',
    cc=['jane@example.com', 'rita@example.com'],
    subject='This is my subject',
    )
```

class cement.ext.ext_dummy.**DummyOutputHandler**(**args*, ***kw*)
Bases: *cement.core.output.CementOutputHandler*

This class is an internal implementation of the *cement.core.output.IOutput* interface. It does not take any parameters on initialization, and does not actually output anything.

class Meta
Handler meta-data

interface
The interface this class implements.

alias of IOutput

label = 'dummy'
The string identifier of this handler.

overridable = False
Whether or not to include dummy as an available to choice to override the output_handler via command line options.

DummyOutputHandler.**render**(*data_dict*, *template=None*, ***kw*)
This implementation does not actually render anything to output, but rather logs it to the debug facility.

Parameters

- **data_dict** – The data dictionary to render.
- **template** – The template parameter is not used by this implementation at all.

Returns None

cement.ext.ext_genshi

The Genshi Extension module provides output templating based on the Genshi Text Templating Language.

Requirements

- Genshi (pip install genshi)

Configuration

To **prepend** a directory to the template_dirs list defined by the application/developer, an end-user can add the configuration option template_dir to their application configuration file under the main config section:

```
[myapp]
template_dir = /path/to/my/templates
```

Usage

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
```

```
        label = 'myapp'
        extensions = ['genshi']
        output_handler = 'genshi'
        template_module = 'myapp.templates'
        template_dirs = [
            '~/.myapp/templates',
            '/usr/lib/myapp/templates',
            ]

with MyApp() as app:
    app.run()

    # create some data
    data = dict(foo='bar')

    # render the data to STDOUT (default) via a template
    app.render(data, 'my_template.genshi')
```

Note that the above `template_module` and `template_dirs` are the auto-defined defaults but are added here for clarity. From here, you would then put a Genshi template file in `myapp/templates/my_template.genshi` or `/usr/lib/myapp/templates/my_template.genshi`.

class `cement.ext.ext_genshi.`**`GenshiOutputHandler`**(**args*, ***kw*)

Bases: *cement.core.output.TemplateOutputHandler*

This class implements the *IOutput* interface. It provides text output from template and uses the Genshi Text Templating Language. Please see the developer documentation on *Output Handling*.

class `Meta`

Handler meta-data.

`interface`

alias of `IOutput`

`GenshiOutputHandler.`**`render`**(*data_dict*, ***kw*)

Take a data dictionary and render it using the given template file.

Required Arguments:

Parameters

- **`data_dict`** – The data dictionary to render.
- **`template`** – The path to the template, after the `template_module` or `template_dirs` prefix as defined in the application.

Returns str (the rendered template text)

Genshi Syntax Basics

Printing Variables

```
Hello ${user_name}
```

Where `user_name` is a variable returned from the controller. Will display:

```
Hello Johnny
```

Conditional Statements

```
{% if foo %}\\
Label: ${example.label}
{% end %}\\
```

Will only output `Label: <label>` if `foo == True`.

For Loops

```
{% for item in items %}\\
  - ${item}
{% end %}\\
```

Where `items` is a list returned from the controller. Will display:

```
- list item 1
- list item 2
- list item 3
```

Functions

```
{% def greeting(name) %}\\
Hello, ${name}!
{% end %}\\

${greeting('World')}\\
${greeting('Edward')}
```

Will output:

```
Hello, World!
Hello, Edward!
```

Formatted Columns

```
{# --------------------- 78 character baseline --------------------------- #}\\

label               ver       description
==================  ========  ================================================
{% for plugin in plugins %}\\
${"%-18s" % plugin['label']}  ${"%-8s" % plugin['version']}  ${"%-48s" % plugin[
↪'description']}
{% end %}
```

Output looks like:

```
$ helloworld list-plugins

label               ver       description
==================  ========  ================================================
example1            1.0.34    Example plugin v1.x for helloworld
example2            2.1       Example plugin v2.x for helloworld
```

cement.ext.ext_handlebars

cement.ext.ext_jinja2

The Jinja2 Extension module provides output templating based on the Jinja2 Templating Language.

Requirements

- Jinja2 (`pip install Jinja2`)

Configuration

To **prepend** a directory to the `template_dirs` list defined by the application/developer, an end-user can add the configuration option `template_dir` to their application configuration file under the main config section:

```
[myapp]
template_dir = /path/to/my/templates
```

Usage

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['jinja2']
        output_handler = 'jinja2'
        template_module = 'myapp.templates'
        template_dirs = [
            '~/.myapp/templates',
            '/usr/lib/myapp/templates',
            ]

with MyApp() as app:
    app.run()

    # create some data
    data = dict(foo='bar')

    # render the data to STDOUT (default) via a template
    app.render(data, 'my_template.jinja2')
```

Note that the above `template_module` and `template_dirs` are the auto-defined defaults but are added here for clarity. From here, you would then put a Jinja2 template file in `myapp/templates/my_template.jinja2` or `/usr/lib/myapp/templates/my_template.jinja2`.

class `cement.ext.ext_jinja2.`**`Jinja2OutputHandler`**(**args*, ***kw*)

Bases: *`cement.core.output.TemplateOutputHandler`*

This class implements the *IOutput* interface. It provides text output from template and uses the Jinja2 Templating Language. Please see the developer documentation on *Output Handling*.

class `Meta`

Handler meta-data.

`interface`

alias of `IOutput`

`Jinja2OutputHandler.`**`render`**(*data_dict*, *template=None*, ***kw*)

Take a data dictionary and render it using the given template file. Additional keyword arguments are ignored.

Required Arguments:

Parameters

- **data_dict** – The data dictionary to render.
- **template** – The path to the template, after the `template_module` or `template_dirs` prefix as defined in the application.

Returns str (the rendered template text)

cement.ext.ext_json

The JSON Extension adds the *JsonOutputHandler* to render output in pure JSON, as well as the *JsonConfigHandler* that allows applications to use JSON configuration files as a drop-in replacement of the default *cement.ext.ext_configparser.ConfigParserConfigHandler*.

Requirements

- No external dependencies.

Configuration

This extension does not support any configuration settings.

Usage

myapp.conf

```
{
    "myapp": {
        "foo": "bar"
    }
}
```

myapp.py

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['json']
        config_handler = 'json'

        # you probably don't want this to be json by default.. but you can
        # output_handler = 'json'

with MyApp() as app:
    app.run()

    # create some data
    data = dict(foo=app.config.get('myapp', 'foo'))

    app.render(data)
```

In general, you likely would not set `output_handler` to `json`, but rather another type of output handler that display readable output to the end-user (i.e. Mustache, Genshi, or Tabulate). By default Cement adds the `-o` command line option to allow the end user to override the output handler. For example: passing `-o json` will override the default output handler and set it to `JsonOutputHandler`.

See `CementApp.Meta.handler_override_options`.

```
$ python myapp.py -o json
{"foo": "bar"}
```

What if I Want To Use UltraJson or Something Else?

It is possible to override the backend `json` library module to use, for example if you wanted to use UltraJson (`ujson`) or another **drop-in replacement** library. The recommended solution would be to override the `JsonOutputHandler` with you're own sub-classed version, and modify the `json_module` meta-data option.

```
from cement.ext.ext_json import JsonOutputHandler

class MyJsonHandler(JsonOutputHandler):
    class Meta:
        json_module = 'ujson'

# then, the class must be replaced via a 'post_setup' hook

def override_json(app):
    app.handler.register(MyJsonHandler, force=True)

app.hook.register('post_setup', override_json)
```

class cement.ext.ext_json.**JsonConfigHandler**(**args*, ***kw*)

Bases: *cement.ext.ext_configparser.ConfigParserConfigHandler*

This class implements the *IConfig* interface, and provides the same functionality of *ConfigParserConfigHandler* but with JSON configuration files.

class Meta

Handler meta-data.

json_module = 'json'

Backend JSON library module to use (*json*, *ujson*).

JsonConfigHandler.**_parse_file**(*file_path*)

Parse JSON configuration file settings from file_path, overwriting existing config settings. If the file does not exist, returns False.

Parameters **file_path** – The file system path to the JSON configuration file.

Returns boolean

class cement.ext.ext_json.**JsonOutputHandler**(**args*, ***kw*)

Bases: *cement.core.output.CementOutputHandler*

This class implements the *IOutput* interface. It provides JSON output from a data dictionary using the json module of the standard library. Please see the developer documentation on *Output Handling*.

This handler forces Cement to suppress console output until `app.render` is called (keeping the output pure JSON). If troubleshooting issues, you will need to pass the `--debug` option in order to unsuppress output and see what's happening.

class Meta
Handler meta-data

interface
The interface this class implements.

alias of `IOutput`

json_module = 'json'
Backend JSON library module to use (*json*, *ujson*)

label = 'json'
The string identifier of this handler.

overridable = True
Whether or not to include `json` as an available choice to override the `output_handler` via command line options.

`JsonOutputHandler.`**render**(*data_dict*, *template=None*, ***kw*)
Take a data dictionary and render it as Json output. Note that the template option is received here per the interface, however this handler just ignores it. Additional keyword arguments passed to `json.dumps()`.

Parameters

- **data_dict** – The data dictionary to render.
- **template** – This option is completely ignored.

Returns A JSON encoded string.

Return type `str`

`cement.ext.ext_json.`**suppress_output_after_render**(*app*, *out_text*)
This is a `post_render` hook that suppresses console output again after rendering, only if the `JsonOutputHandler` is triggered via command line.

Parameters **app** – The application object.

`cement.ext.ext_json.`**suppress_output_before_run**(*app*)
This is a `post_argument_parsing` hook that suppresses console output if the `JsonOutputHandler` is triggered via command line.

Parameters **app** – The application object.

`cement.ext.ext_json.`**unsuppress_output_before_render**(*app*, *data*)
This is a `pre_render` that unsuppresses console output if the `JsonOutputHandler` is triggered via command line so that the JSON is the only thing in the output.

Parameters **app** – The application object.

cement.ext.ext_json_configobj

The JSON ConfigObj Extension is a combination of the `JsonConfigHandler` and `ConfigObjConfigHandler` which allows the application to read JSON configuration files into a ConfigObj based configuration handler.

Requirements

- ConfigObj (`pip install configobj`)

Configuration

This extension does not support any configuration settings.

Usage

myapp.conf

```
{
    "myapp": {
        "foo": "bar"
    }
}
```

myapp.py

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['json_configobj']
        config_handler = 'json_configobj'

with MyApp() as app:
    app.run()

    # get config settings
    app.config['myapp']['foo']

    # set config settings
    app.config['myapp']['foo'] = 'bar2'

    # etc...
```

class cement.ext.ext_json_configobj.**JsonConfigObjConfigHandler**(**args*, ***kw*)

Bases: *cement.ext.ext_configobj.ConfigObjConfigHandler*

This class implements the *IConfig* interface, and provides the same functionality of *ConfigObjConfigHandler* but with JSON configuration files.

Note This extension has an external dependency on `ConfigObj`. You must include `configobj` in your application's dependencies as Cement explicitly does *not* include external dependencies for optional extensions.

class Meta

Handler meta-data.

json_module = 'json'

Backend JSON module to use (`json`, `ujson`, etc)

label = 'json_configobj'

The string identifier of this handler.

JsonConfigObjConfigHandler.**_parse_file**(*file_path*)

Parse JSON configuration file settings from `file_path`, overwriting existing config settings. If the file does not exist, returns `False`.

Parameters **file_path** – The file system path to the JSON configuration file.

Returns boolean

cement.ext.ext_logging

The Logging Extension provides log handling based on the standard `logging.Logger`, and is the default log handler used by Cement.

Requirements

- No external dependencies.

Configuration

This extension honors the following configuration settings from the config section `log.logging`:

- level
- file
- to_console
- rotate
- max_bytes
- max_files

A sample config section (in any config file) might look like:

```
[log.logging]
file = /path/to/config/file
level = info
to_console = true
rotate = true
max_bytes = 512000
max_files = 4
```

Usage

```
from cement.core.foundation import CementApp

with MyApp() as app:
    app.log.info("This is an info message")
    app.log.warning("This is an warning message")
    app.log.error("This is an error message")
    app.log.fatal("This is a fatal message")
    app.log.debug("This is a debug message")
```

class cement.ext.ext_logging.**LoggingLogHandler**(**args*, ***kw*)

Bases: *cement.core.log.CementLogHandler*

This class is an implementation of the *ILog* interface, and sets up the logging facility using the standard Python logging module.

class **Meta**
Handler meta-data.

clear_loggers = []
List of logger namespaces to clear. Useful when imported software also sets up logging and you end up with duplicate log entries.

Changes in Cement 2.1.3. Previous versions only supported *clear_loggers* as a boolean, but did fully support clearing non-app logging namespaces.

config_defaults = {'to_console': True, 'max_bytes': 512000, 'file': None, 'level': 'INFO', 'max_files': 4, 'rotate
The default configuration dictionary to populate the `log` section.

console_format = '%(levelname)s: %(message)s'
The logging format for the consoler logger.

debug_format = '%(asctime)s (%(levelname)s) %(namespace)s : %(message)s'
The logging format for both file and console if `debug==True`.

file_format = '%(asctime)s (%(levelname)s) %(namespace)s : %(message)s'
The logging format for the file logger.

formatter_class
Class to use as the formatter

alias of `Formatter`

interface
The interface that this class implements.

alias of `ILog`

label = 'logging'
The string identifier of this handler.

namespace = None
The logging namespace.

Note: Although Meta.namespace defaults to None, Cement will set this to the application label (CementApp.Meta.label) if not set during setup.

`LoggingLogHandler.`**_setup_console_log**`()`
Add a console log handler.

`LoggingLogHandler.`**_setup_file_log**`()`
Add a file log handler.

`LoggingLogHandler.`**clear_loggers**`(`*namespace*`)`
Clear any previously configured loggers for `namespace`.

`LoggingLogHandler.`**debug**`(`*msg*, *namespace=None*, ***kw*`)`
Log to the DEBUG facility.

Parameters

- **msg** – The message to log.
- **namespace** – A log prefix, generally the module `__name__` that the log is coming from. Will default to self._meta.namespace if None is passed. For debugging, it can be useful to set this to `__file__`, though `__name__` is much less verbose.
- **kw** – Keyword arguments are passed on to the backend logging system.

`LoggingLogHandler.`**error**`(`*msg*, *namespace=None*, ***kw*`)`
Log to the ERROR facility.

Parameters

- **msg** – The message to log.
- **namespace** – A log prefix, generally the module __name__ that the log is coming from. Will default to self._meta.namespace if None is passed.
- **kw** – Keyword arguments are passed on to the backend logging system.

LoggingLogHandler.**fatal**(*msg*, *namespace=None*, ***kw*)
Log to the FATAL (aka CRITICAL) facility.

Parameters

- **msg** – The message to log.
- **namespace** – A log prefix, generally the module __name__ that the log is coming from. Will default to self._meta.namespace if None is passed.
- **kw** – Keyword arguments are passed on to the backend logging system.

LoggingLogHandler.**get_level**()
Returns the current log level.

LoggingLogHandler.**info**(*msg*, *namespace=None*, ***kw*)
Log to the INFO facility.

Parameters

- **msg** – The message to log.
- **namespace** – A log prefix, generally the module __name__ that the log is coming from. Will default to self._meta.namespace if None is passed.
- **kw** – Keyword arguments are passed on to the backend logging system.

LoggingLogHandler.**set_level**(*level*)
Set the log level. Must be one of the log levels configured in self.levels which are `['INFO', 'WARNING', 'ERROR', 'DEBUG', 'FATAL']`.

Parameters **level** – The log level to set.

LoggingLogHandler.**warn**(*msg*, *namespace=None*, ***kw*)
DEPRECATION WARNING: This function is deprecated as of Cement 2.9.x in favor of the `LoggingLogHandler.warning()` function, and will be removed in future versions of Cement.

See: :ref:LoggingLogHandler.warning():

LoggingLogHandler.**warning**(*msg*, *namespace=None*, ***kw*)
Log to the WARNING facility.

Parameters

- **msg** – The message to log.
- **namespace** – A log prefix, generally the module __name__ that the log is coming from. Will default to self._meta.namespace if None is passed.
- **kw** – Keyword arguments are passed on to the backend logging system.

cement.ext.ext_memcached

The Memcached Extension provides application caching and key/value store support via Memcache.

Requirements

- **pylibmc (`pip install pylibmc`)**
  - Note: There are known issues installing `pylibmc` on OSX/Homebrew via PIP. This post might be helpful.

Configuration

This extension honors the following config settings under a `[cache.memcached]` section in any configuration file:

- **expire_time** - The default time in second to expire items in the cache. Default: 0 (does not expire).
- **hosts** - List of Memcached servers.

Configurations can be passed as defaults to a CementApp:

```
from cement.core.foundation import CementApp
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'cache.memcached')
defaults['cache.memcached']['expire_time'] = 0
defaults['cache.memcached']['hosts'] = ['127.0.0.1']

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        config_defaults = defaults
        extensions = ['memcached']
        cache_handler = 'memcached'
```

Additionally, an application configuration file might have a section like the following:

```
[myapp]

# set the cache handler to use
cache_handler = memcached


[cache.memcached]

# time in seconds that an item in the cache will expire
expire_time = 3600

# comma seperated list of memcached servers
hosts = 127.0.0.1, cache.example.com
```

Usage

```
from cement.core import foundation
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'memcached')
defaults['cache.memcached']['expire_time'] = 300 # seconds
defaults['cache.memcached']['hosts'] = ['127.0.0.1']
```

```
class MyApp(foundation.CementApp):
    class Meta:
        label = 'myapp'
        config_defaults = defaults
        extensions = ['memcached']
        cache_handler = 'memcached'

with MyApp() as app:
    # Run the app
    app.run()

    # Set a cached value
    app.cache.set('my_key', 'my value')

    # Get a cached value
    app.cache.get('my_key')

    # Delete a cached value
    app.cache.delete('my_key')

    # Delete the entire cache
    app.cache.purge()
```

class cement.ext.ext_memcached.**MemcachedCacheHandler**(**args*, ***kw*)

Bases: *cement.core.cache.CementCacheHandler*

This class implements the *ICache* interface. It provides a caching interface using the pylibmc library.

Note This extension has an external dependency on `pylibmc`. You must include `pylibmc` in your applications dependencies as Cement explicitly does *not* include external dependencies for optional extensions.

class Meta

Handler meta-data.

interface

alias of `ICache`

MemcachedCacheHandler.**_config**(*key*)

This is a simple wrapper, and is equivalent to: `self.app.config.get('cache.memcached', <key>)`.

Parameters **key** – The key to get a config value from the 'cache.memcached' config section.

Returns The value of the given key.

MemcachedCacheHandler.**_fix_hosts**()

Useful to fix up the hosts configuration (i.e. convert a comma-separated string into a list). This function does not return anything, however it is expected to set the *hosts* value of the `[cache.memcached]` section (which is what this extension reads for it's host configution).

Returns `None`

MemcachedCacheHandler.**delete**(*key*, ***kw*)

Delete an item from the cache for the given `key`. Any additional keyword arguments will be passed directly to the *pylibmc* delete function.

Parameters **key** – The key to delete from the cache.

Returns `None`

MemcachedCacheHandler.**get**(*key*, *fallback=None*, ***kw*)
Get a value from the cache. Any additional keyword arguments will be passed directly to *pylibmc* get function.

Parameters

- **key** – The key of the item in the cache to get.
- **fallback** – The value to return if the item is not found in the cache.

Returns The value of the item in the cache, or the *fallback* value.

MemcachedCacheHandler.**purge**(***kw*)
Purge the entire cache, all keys and values will be lost. Any additional keyword arguments will be passed directly to the pylibmc `flush_all()` function.

Returns `None`

MemcachedCacheHandler.**set**(*key*, *value*, *time=None*, ***kw*)
Set a value in the cache for the given `key`. Any additional keyword arguments will be passed directly to the *pylibmc* set function.

Parameters

- **key** – The key of the item in the cache to set.
- **value** – The value of the item to set.
- **time** – The expiration time (in seconds) to keep the item cached. Defaults to *expire_time* as defined in the applications configuration.

Returns `None`

cement.ext.ext_mustache

The Mustache Extension provides output templating based on the Mustache Templating Language.

Requirements

- pystache (`pip install pystache`)

Configuration

To **prepend** a directory to the `template_dirs` list defined by the application/developer, an end-user can add the configuration option `template_dir` to their application configuration file under the main config section:

```
[myapp]
template_dir = /path/to/my/templates
```

Usage

```
from cement.core import foundation

class MyApp(foundation.CementApp):
    class Meta:
        label = 'myapp'
```

```
        extensions = ['mustache']
        output_handler = 'mustache'
        template_module = 'myapp.templates'
        template_dirs = [
            '~/.myapp/templates',
            '/usr/lib/myapp/templates',
            ]
# ...
```

Note that the above `template_module` and `template_dirs` are the auto-defined defaults but are added here for clarity. From here, you would then put a Mustache template file in `myapp/templates/my_template.mustache` or `/usr/lib/myapp/templates/my_template.mustache` and then render a data dictionary with it:

```
app.render(some_data_dict, 'my_template.mustache')
```

Loading Partials

Mustache supports `partials`, or in other words template `includes`. These are also loaded by the output handler, but require a full file name. The partials will be loaded in the same way as the base templates

For example:

templates/base.mustache

```
Inside base.mustache
{{> partial.mustache}}
```

template/partial.mustache

```
Inside partial.mustache
```

Would output:

```
Inside base.mustache
Inside partial.mustache
```

class `cement.ext.ext_mustache.`**`MustacheOutputHandler`**(**args*, ***kw*)

Bases: *`cement.core.output.TemplateOutputHandler`*

This class implements the *IOutput* interface. It provides text output from template and uses the Mustache Templating Language. Please see the developer documentation on *Output Handling*.

Note This extension has an external dependency on `pystache`. You must include `pystache` in your applications dependencies as Cement explicitly does **not** include external dependencies for optional extensions.

class `Meta`

Handler meta-data.

`interface`

alias of `IOutput`

`overridable` = False

Whether or not to include `mustache` as an available to choice to override the `output_handler` via command line options.

MustacheOutputHandler.**render**(*data_dict*, *template=None*, ***kw*)
Take a data dictionary and render it using the given template file. Additional keyword arguments passed to `stache.render()`.

Required Arguments:

Parameters

- **data_dict** – The data dictionary to render.
- **template** – The path to the template, after the `template_module` or `template_dirs` prefix as defined in the application.

Returns str (the rendered template text)

cement.ext.ext_plugin

The Plugin Extension handles application plugin support, and is the default plugin handler used by Cement.

Requirements

- No external dependencies

Configuration

This extension does not directly honor any configuration settings.

Usage

For usage information see *Application Plugins*.

class cement.ext.ext_plugin.**CementPluginHandler**
Bases: *cement.core.plugin.CementPluginHandler*

This class is an internal implementation of the *IPlugin* interface. It does not take any parameters on initialization.

class Meta
Handler meta-data.

interface
The interface that this class implements.

alias of `IPlugin`

label = 'cement'
The string identifier for this class.

CementPluginHandler.**_load_plugin_from_bootstrap**(*plugin_name*, *base_package*)
Load a plugin from a python package. Returns True if no ImportError is encountered.

Parameters

- **plugin_name** (`str`) – The name of the plugin, also the name of the module to load from base_package. I.e. `myapp.bootstrap.myplugin`
- **base_package** (`str`) – The base python package to load the plugin module from. I.e.'myapp.bootstrap' or similar.

Returns True is the plugin was loaded, False otherwise

Raises ImportError

CementPluginHandler.**_load_plugin_from_dir**(*plugin_name*, *plugin_dir*)
Load a plugin from a directory path rather than a python package within sys.path. This would either be myplugin.py or myplugin/__init__.py within the given plugin_dir.

Parameters

- **plugin_name** – The name of the plugin.
- **plugin_dir** – The filesystem directory path where the plugin exists.

CementPluginHandler.**get_disabled_plugins**()
List of disabled plugins

CementPluginHandler.**get_enabled_plugins**()
List of plugins that are enabled (not necessary loaded yet).

CementPluginHandler.**get_loaded_plugins**()
List of plugins that have been loaded.

CementPluginHandler.**load_plugin**(*plugin_name*)
Load a plugin whose name is plugin_name. First attempt to load from a plugin directory (plugin_dir), secondly attempt to load from a bootstrap module (plugin_bootstrap) determined by CementApp.Meta.plugin_bootstrap.

Upon successful loading of a plugin, the plugin name is appended to the self._loaded_plugins list.

Parameters **plugin_name** (str) – The name of the plugin to load.

Raises *cement.core.exc.FrameworkError*

CementPluginHandler.**load_plugins**(*plugin_list*)
Load a list of plugins. Each plugin name is passed to self.load_plugin().

Parameters **plugin_list** – A list of plugin names to load.

cement.ext.ext_redis

cement.ext.ext_reload_config

WARNING: THIS EXTENSION IS EXPERIMENTAL

Experimental extension may (and probably will) change at any time. Please do not rely on these features until they are more fully vetted.

The Reload Config Framework Extension enables applications Built on Cement (tm) to easily reload configuration settings any time configuration files are modified without stopping/restarting the process.

Requirements

- Python 2.6+, Python 3+
- Python Modules: pyinotify
- Linux (Kernel 2.6.13+)

Features

- Application configuration files (`CementApp.Meta.config_files`) are reloaded if modified.
- Application plugin configuration files (Anything found in (`CementApp.Meta.plugin_config_dirs`) are reloaded if modified.
- The framework calls `CementApp.config.parse_file()` on any watched files once the kernel has signaled a modification.
- New configurations settings are accessible via `CementApp.config` nearly immediately once the kernel (inotify) picks up the change.
- Provides a `pre_reload_config` and `post_reload_config` hook so that applications can tie into the event and perform operations any time a configuration file is modified.
- Asynchronously monitors configuration files for changes via inotify. Long running processes are not blocked by the operations performed when files are detected to be modified.

Limitations

- Currently this extension only re-parses configuration files into the config handler. Some applications may need further work in-order to truly honor those changes. For example, if a configuration settings toggles something on or off, or triggers something else to happen (like making an API connection, etc)... this extension does not currently handle that, and it is left up to the application developer to tie into the events via the provided hooks.
- Only available on Linux based systems.

Configuration

This extension does not currently honor any configuration settings.

Hooks

This extension defines the following hooks:

pre_reload_config

Run right before any framework actions are performed once modifications to any of the watched files are detected. Expects a single argument, which is the `app` object, and does not expect anything in return.

```
def my_pre_reload_config_hook(app):
    # do something with app?
    pass
```

post_reload_config

Run right after any framework actions are performed once modifications to any of the watched files are detected. Expects a single argument, which is the `app` object, and does not expect anything in return.

```
def my_post_reload_config_hook(app):
    # do something with app?
    pass
```

Usage

The following example shows how to add the reload_config extension, as well as perform an arbitrary action any time configuration changes are detected.

```
from time import sleep
from cement.core.exc import CaughtSignal
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

def print_foo(app):
    print "Foo => %s" % app.config.get('myapp', 'foo')

class Base(CementBaseController):
    class Meta:
        label = 'base'

    @expose(hide=True)
    def default(self):
        print('Inside Base.default()')

        # simulate a long running process
        while True:
            sleep(30)

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = Base
        extensions = ['reload_config']


with MyApp() as app:
    # run this anytime the configuration has changed
    app.hook.register('post_reload_config', print_foo)

    try:
        app.run()
    except CaughtSignal as e:
        # maybe do something... but catch it regardless so app.close() is
        # called when exiting `with` cleanly.
        print(e)
```

The following would output something like the following when `~/.myapp.conf` or any other configuration file is modified (spaces added for clarity):

```
Inside Base.default()


2015-05-05 03:00:32,023 (DEBUG) cement.ext.ext_reload_config : config
                                 path modified: mask=IN_CLOSE_WRITE,
                                 path=/home/vagrant/.myapp.conf
```

```
2015-05-05 03:00:32,023 (DEBUG) cement.core.config : config file
                                '/home/vagrant/.myapp.conf' exists,
                                loading settings...
2015-05-05 03:00:32,023 (DEBUG) cement.core.hook : running hook
                                'post_reload_config' (<function print_foo
                                at 0x7f1b52a5ab70>) from __main__
Foo => bar


2015-05-05 03:00:32,023 (DEBUG) cement.ext.ext_reload_config : config
                                path modified: mask=IN_CLOSE_WRITE,
                                path=/home/vagrant/.myapp.conf
2015-05-05 03:00:32,023 (DEBUG) cement.core.config : config file
                                '/home/vagrant/.myapp.conf' exists,
                                loading settings...
2015-05-05 03:00:32,023 (DEBUG) cement.core.hook : running hook
                                'post_reload_config' (<function print_foo
                                at 0x7f1b52a5ab70>) from __main__
Foo => bar2


2015-05-05 03:00:32,023 (DEBUG) cement.ext.ext_reload_config : config
                                path modified: mask=IN_CLOSE_WRITE,
                                path=/home/vagrant/.myapp.conf
2015-05-05 03:00:32,023 (DEBUG) cement.core.config : config file
                                '/home/vagrant/.myapp.conf' exists,
                                loading settings...
2015-05-05 03:00:32,023 (DEBUG) cement.core.hook : running hook
                                'post_reload_config' (<function print_foo
                                at 0x7f1b52a5ab70>) from __main__
Foo => bar3
```

cement.ext.ext_smtp

The SMTP Extension provides the ability for applications to send email based on the smtplib standard library.

Requirements

- No external depencies

Configuration

This extension honors the following configuration settings:

- **to** - Default `to` addresses (list, or comma separated depending on the ConfigHandler in use)
- **from_addr** - Default `from_addr` address
- **cc** - Default `cc` addresses (list, or comma separated depending on the ConfigHandler in use)
- **bcc** - Default `bcc` addresses (list, or comma separated depending on the ConfigHandler in use)
- **subject** - Default `subject`
- **subject_prefix** - Additional string to prepend to the `subject`

- **host** - The SMTP host server
- **port** - The SMTP host server port
- **timeout** - The timeout in seconds before terminating a connection
- **ssl** - Whether to initiate SSL or not
- **tls** - Whether to use TLS or not (requires SSL)
- **auth** - Whether or not to initiate SMTP authentication
- **username** - SMTP authentication username
- **password** - SMTP authentication password

You can add these to any application configuration file under a `[mail.smtp]` section, for example:

~/.myapp.conf

```
[myapp]

# set the mail handler to use
mail_handler = smtp


[mail.smtp]

# default to addresses (comma separated list)
to = me@example.com

# default from address
from = someone_else@example.com

# default cc addresses (comma separated list)
cc = jane@example.com, rita@example.com

# default bcc addresses (comma separated list)
bcc = blackhole@example.com, someone_else@example.com

# default subject
subject = This is The Default Subject

# additional prefix to prepend to the subject
subject_prefix = MY PREFIX >

# smtp host server
host = localhost

# smtp host port
port = 465

# timeout in seconds
timeout = 30

# whether or not to establish an ssl connection
ssl = 1

# whether or not to use start tls
tls = 1

# whether or not to initiate smtp auth
```

```
auth = 1

# smtp auth username
username = john.doe

# smtp auth password
password = oober_secure_password
```

Usage

```
class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        mail_handler = 'smtp'

with MyApp() as app:
    app.mail.send('This is my fake message',
        subject='This is my subject',
        to=['john@example.com', 'rita@example.com'],
        from_addr='me@example.com',
        )
```

class cement.ext.ext_smtp.**SMTPMailHandler**(**args*, ***kw*)

Bases: *cement.core.mail.CementMailHandler*

This class implements the *IMail* interface, and is based on the smtplib standard library.

class Meta

Handler meta-data.

config_defaults = {'username': None, 'cc': [], 'subject_prefix': None, 'auth': False, 'ssl': False, 'host': 'localho

Configuration default values

label = 'smtp'

Unique identifier for this handler

SMTPMailHandler.**send**(*body*, ***kw*)

Send an email message via SMTP. Keyword arguments override configuration defaults (cc, bcc, etc).

Parameters

- **body** (multiline string) – The message body to send
- **to** (list) – List of recipients (generally email addresses)
- **from_addr** (str) – Address (generally email) of the sender
- **cc** (list) – List of CC Recipients
- **bcc** (list) – List of BCC Recipients
- **subject** (str) – Message subject line

Returns Boolean (True if message is sent successfully, False otherwise)

Usage

```
# Using all configuration defaults
app.mail.send('This is my message body')
```

```
# Overriding configuration defaults
app.mail.send('My message body'
    from_addr='me@example.com',
    to=['john@example.com'],
    cc=['jane@example.com', 'rita@example.com'],
    subject='This is my subject',
    )
```

cement.ext.ext_tabulate

The Tabulate Extension provides output handling based on the Tabulate library. It's format is familiar to users of MySQL, Postgres, etc.

Requirements

- Tabulate (`pip install tabulate`)

Configuration

This extension does not support any configuration settings.

Usage

```
from cement.core import foundation

class MyApp(foundation.CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['tabulate']
        output_handler = 'tabulate'

with MyApp() as app:
    app.run()

    # create a dataset
    headers = ['NAME', 'AGE', 'ADDRESS']
    data = [
        ["Krystin Bartoletti", 47, "PSC 7591, Box 425, APO AP 68379"],
        ["Cris Hegan", 54, "322 Reubin Islands, Leylabury, NC 34388"],
        ["George Champlin", 25, "Unit 6559, Box 124, DPO AA 25518"],
        ]

    app.render(data, headers=headers)
```

Looks like:

```
| NAME               | AGE | ADDRESS                                  |
|--------------------+-----+------------------------------------------|
| Krystin Bartoletti |  47 | PSC 7591, Box 425, APO AP 68379          |
| Cris Hegan         |  54 | 322 Reubin Islands, Leylabury, NC 34388  |
| George Champlin    |  25 | Unit 6559, Box 124, DPO AA 25518         |
```

class cement.ext.ext_tabulate.**TabulateOutputHandler**(**args*, ***kw*)

Bases: *cement.core.output.CementOutputHandler*

This class implements the *IOutput* interface. It provides tabularized text output using the Tabulate module. Please see the developer documentation on *Output Handling*.

Note This extension has an external dependency on `tabulate`. You must include `tabulate` in your applications dependencies as Cement explicitly does **not** include external dependencies for optional extensions.

class Meta

Handler meta-data.

float_format = 'g'

String format to use for float values.

format = 'orgtbl'

Default template format. See the `tabulate` documentation for all supported template formats.

headers = []

Default headers to use.

interface

alias of `IOutput`

missing_value = ''

Default replacement for missing value.

numeric_alignment = 'decimal'

Default alignment for numeric columns. See the `tabulate` documentation for all supported `numalign` options.

overridable = False

Whether or not to include `tabulate` as an available to choice to override the `output_handler` via command line options.

padding = True

Whether or not to pad the output with an extra pre/post 'n'

string_alignment = 'left'

Default alignment for string columns. See the `tabulate` documentation for all supported `stralign` options.

TabulateOutputHandler.**render**(*data*, ***kw*)

Take a data dictionary and render it into a table. Additional keyword arguments are passed directly to `tabulate.tabulate`.

Required Arguments:

Parameters **data_dict** – The data dictionary to render.

Returns str (the rendered template text)

cement.ext.ext_yaml

The Yaml Extension adds the *YamlOutputHandler* to render output in pure Yaml, as well as the *YamlConfigHandler* that allows applications to use Yaml configuration files as a drop-in replacement of the default *cement.ext.ext_configparser.ConfigParserConfigHandler*.

Requirements

- pyYaml (`pip install pyYaml`)

Configuration

This extension does not honor any application configuration settings.

Usage

myapp.conf

```
---
myapp:
    foo: bar
```

myapp.py

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['yaml']
        config_handler = 'yaml'

        # you probably don't want to do this.. but you can
        # output_handler = 'yaml'

with MyApp() as app:
    app.run()

    # create some data
    data = dict(foo=app.config.get('myapp', 'foo'))

    app.render(data)
```

In general, you likely would not set `output_handler` to `yaml`, but rather another type of output handler that displays readable output to the end-user (i.e. Mustache, Genshi, or Tabulate). By default Cement adds the `-o` command line option to allow the end user to override the output handler. For example: passing `-o yaml` will override the default output handler and set it to `YamlOutputHandler`.

See `CementApp.Meta.handler_override_options`.

```
$ python myapp.py -o yaml
{foo: bar}
```

class `cement.ext.ext_yaml.`**`YamlConfigHandler`**(**args*, ***kw*)

Bases: *cement.ext.ext_configparser.ConfigParserConfigHandler*

This class implements the *IConfig* interface, and provides the same functionality of *ConfigParserConfigHandler* but with Yaml configuration files. See pyYaml for more information on pyYaml.

Note This extension has an external dependency on *pyYaml*. You must include *pyYaml* in your application's dependencies as Cement explicitly does *not* include external dependencies for optional extensions.

_parse_file(*file_path*)
Parse Yaml configuration file settings from file_path, overwriting existing config settings. If the file does not exist, returns False.

Parameters **file_path** – The file system path to the Yaml configuration file.

Returns boolean

class cement.ext.ext_yaml.**YamlOutputHandler**(**args*, ***kw*)
Bases: *cement.core.output.CementOutputHandler*

This class implements the *IOutput* interface. It provides Yaml output from a data dictionary and uses pyYaml to dump it to STDOUT. Please see the developer documentation on *Output Handling*.

This handler forces Cement to suppress console output until `app.render` is called (keeping the output pure Yaml). If troubleshooting issues, you will need to pass the `--debug` option in order to unsuppress output and see what's happening.

class Meta
Handler meta-data.

interface
alias of `IOutput`

overridable = True
Whether or not to include `yaml` as an available to choice to override the `output_handler` via command line options.

YamlOutputHandler.**render**(*data_dict*, *template=None*, ***kw*)
Take a data dictionary and render it as Yaml output. Note that the template option is received here per the interface, however this handler just ignores it. Additional keyword arguments passed to `yaml.dump()`.

Parameters

- **data_dict** – The data dictionary to render.
- **template** – Ignored in this output handler implementation.

Returns A Yaml encoded string.

Return type `str`

cement.ext.ext_yaml.**suppress_output_after_render**(*app*, *out_text*)
This is a `post_render` hook that suppresses console output again after rendering, only if the `YamlOutputHandler` is triggered via command line.

Parameters **app** – The application object.

cement.ext.ext_yaml.**suppress_output_before_run**(*app*)
This is a `post_argument_parsing` hook that suppresses console output if the `YamlOutputHandler` is triggered via command line.

Parameters **app** – The application object.

cement.ext.ext_yaml.**unsuppress_output_before_render**(*app*, *data*)
This is a `pre_render` that unsuppresses console output if the `YamlOutputHandler` is triggered via command line so that the Yaml is the only thing in the output.

Parameters **app** – The application object.

cement.ext.ext_yaml_configobj

The Yaml ConfigObj Extension is a combination of the `YamlConfigHandler` and `ConfigObjConfigHandler` which allows the application to read Yaml configuration files into a ConfigObj based configuration handler.

Requirements

- ConfigObj (`pip install configobj`)
- pyYaml (`pip install pyYaml`)

Configuration

This extension does not honor any application configuration settings.

Usage

myapp.conf

```
---
    myapp:
        foo: bar
```

myapp.py

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['yaml_configobj']
        config_handler = 'yaml_configobj'

with MyApp() as app:
    app.run()

    # get config settings
    app.config['myapp']['foo']

    # set config settings
    app.config['myapp']['foo'] = 'bar2'

    # etc...
```

class cement.ext.ext_yaml_configobj.**YamlConfigObjConfigHandler**(**args*, ***kw*)

Bases: *cement.ext.ext_configobj.ConfigObjConfigHandler*

This class implements the *IConfig* interface, and provides the same functionality of *ConfigObjConfigHandler* but with YAML configuration files. See pyYAML for more information on pyYAML

Note This extension has an external dependency on *pyYAML* and *ConfigObj*. You must include *pyYAML* and *configobj* in your application's dependencies as Cement explicitly does *not* include external dependencies for optional extensions.

class Meta
Handler meta-data.

YamlConfigObjConfigHandler.**_parse_file**(*file_path*)
Parse YAML configuration file settings from file_path, overwriting existing config settings. If the file does not exist, returns False.

Parameters **file_path** – The file system path to the YAML configuration file.

Returns boolean

cement.ext.ext_watchdog

Developer Documentation

This page contains documentation for developers building their application on Cement. It outlines the features of the Cement Framework and how to use them in your applications.

Installation

It is recommended to work out of a VirtualENV during development of your application, which is reference throughout this documentation. VirtualENV is easily installed on most platforms either via PIP or by your OS distributions packaging system (Yum, Apt, Brew, etc).

Installation

Installing Stable Versions from PyPI:

```
$ pip install cement
```

Installing Development Versions from Git:

```
$ pip install -e git+git://github.com/datafolklabs/cement.git#egg=cement
```

Running Tests

To run tests, you will need to ensure any dependent services are running (for example Memcached), and then do the following from the root of the source:

```
# Start services
$ memcached &

# Python 2.x
$ pip install -r requirements-dev.txt

# Python 3.x
$ pip install -r requirements-dev-py3.txt

$ python setup.py nosetests
```

Building Documentation

To build this documentation, do the following from the root of the source:

```
$ python setup.py build_sphinx
```

Quick Start

The following creates and runs a sample 'helloworld' application.

helloworld.py

```
from cement.core.foundation import CementApp

with CementApp('helloworld') as app:
    app.run()
    print('Hello World')
```

The above is equivalent to (should you need more control over setup and closing an application):

```
from cement.core.foundation import CementApp

app = CementApp('helloworld')
app.setup()
app.run()
print('Hello World')
app.close()
```

Running the application looks like:

```
$ python helloworld.py
Hello World
```

Oh, I can just hear you saying, "Whoa whoa... hang on a minute. This is a joke right, all you did was print 'Hello World' to STDOUT. What kind of framework is this?". Well obviously this is just an introduction to show that the creation of an application is dead simple. Lets take a look further:

```
$ python helloworld.py --help
usage: helloworld.py [-h] [--debug] [--quiet]

optional arguments:
  -h, --help  show this help message and exit
  --debug     toggle debug output
  --quiet     suppress all output
```

Oh nice, ok... ArgParse is already setup with a few options I see. What else?

```
$ python helloworld.py --debug
2014-04-15 12:28:24,705 (DEBUG) cement.core.foundation : laying cement for the
↪'helloworld' application
2014-04-15 12:28:24,705 (DEBUG) cement.core.hook : defining hook 'pre_setup'
2014-04-15 12:28:24,705 (DEBUG) cement.core.hook : defining hook 'post_setup'
2014-04-15 12:28:24,705 (DEBUG) cement.core.hook : defining hook 'pre_run'
2014-04-15 12:28:24,705 (DEBUG) cement.core.hook : defining hook 'post_run'
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'pre_argument_parsing
↪'
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'post_argument_
↪parsing'
```

```
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'pre_close'
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'post_close'
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'signal'
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'pre_render'
2014-04-15 12:28:24,706 (DEBUG) cement.core.hook : defining hook 'post_render'
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'extension
↪' (IExtension)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'log'␣
↪(ILog)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'config'␣
↪(IConfig)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'plugin'␣
↪(IPlugin)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'output'␣
↪(IOutput)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'argument
↪' (IArgument)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type
↪'controller' (IController)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : defining handler type 'cache'␣
↪(ICache)
2014-04-15 12:28:24,706 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.core.extension.CementExtensionHandler'>' into handlers['extension']['cement
↪']
2014-04-15 12:28:24,706 (DEBUG) cement.core.foundation : now setting up the
↪'helloworld' application
2014-04-15 12:28:24,706 (DEBUG) cement.core.foundation : adding signal handler for␣
↪signal 15
2014-04-15 12:28:24,712 (DEBUG) cement.core.foundation : adding signal handler for␣
↪signal 2
2014-04-15 12:28:24,712 (DEBUG) cement.core.foundation : setting up helloworld.
↪extension handler
2014-04-15 12:28:24,712 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪dummy' framework extension
2014-04-15 12:28:24,712 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_dummy.DummyOutputHandler'>' into handlers['output']['null']
2014-04-15 12:28:24,712 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪plugin' framework extension
2014-04-15 12:28:24,713 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_plugin.CementPluginHandler'>' into handlers['plugin']['cement']
2014-04-15 12:28:24,713 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪configparser' framework extension
2014-04-15 12:28:24,713 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_configparser.ConfigParserConfigHandler'>' into handlers['config'][
↪'configparser']
2014-04-15 12:28:24,713 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪logging' framework extension
2014-04-15 12:28:24,713 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_logging.LoggingLogHandler'>' into handlers['log']['logging']
2014-04-15 12:28:24,713 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪argparse' framework extension
2014-04-15 12:28:24,714 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_argparse.ArgParseArgumentHandler'>' into handlers['argument']['
↪'argparse']
2014-04-15 12:28:24,714 (DEBUG) cement.core.foundation : setting up helloworld.config␣
↪handler
2014-04-15 12:28:24,714 (DEBUG) cement.ext.ext_configparser : config file '/etc/
↪helloworld/helloworld.conf' does not exist, skipping...
```

```
2014-04-15 12:28:24,714 (DEBUG) cement.ext.ext_configparser : config file '/Users/
↪derks/.helloworld/config' does not exist, skipping...
2014-04-15 12:28:24,715 (DEBUG) cement.core.foundation : no cache handler defined,␣
↪skipping.
2014-04-15 12:28:24,715 (DEBUG) cement.core.foundation : setting up helloworld.log␣
↪handler
2014-04-15 12:28:24,715 (DEBUG) cement.core.handler : merging config defaults from '
↪<cement.ext.ext_logging.LoggingLogHandler object at 0x1015c4ed0>' into section 'log.
↪logging'
2014-04-15 12:28:24,715 (DEBUG) helloworld:None : logging initialized for
↪'helloworld:None' using LoggingLogHandler
2014-04-15 12:28:24,715 (DEBUG) cement.core.foundation : setting up helloworld.plugin␣
↪handler
2014-04-15 12:28:24,715 (DEBUG) cement.ext.ext_plugin : plugin config dir /Users/
↪derks/Development/boss/tmp/helloworld/config/plugins.d does not exist.
2014-04-15 12:28:24,716 (DEBUG) cement.core.foundation : setting up helloworld.arg␣
↪handler
2014-04-15 12:28:24,716 (DEBUG) cement.core.foundation : setting up helloworld.output␣
↪handler
2014-04-15 12:28:24,716 (DEBUG) cement.core.foundation : setting up application␣
↪controllers
Hello World
2014-04-15 12:28:24,716 (DEBUG) cement.core.foundation : closing the application
```

Damn son, WTF? Don't worry, we'll explain everything in the rest of the doc.

Getting Warmer

The following is a more advanced example that showcases some of the default application features. Notice the creation of command line arguments, default config creation, and logging.

myapp.py

```
from cement.core.foundation import CementApp
from cement.core import hook
from cement.utils.misc import init_defaults

# define our default configuration options
defaults = init_defaults('myapp')
defaults['myapp']['debug'] = False
defaults['myapp']['some_param'] = 'some value'

# define any hook functions here
def my_cleanup_hook(app):
    pass

# define the application class
class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        config_defaults = defaults
        extensions = ['daemon', 'memcached', 'json', 'yaml']
        hooks = [
            ('pre_close', my_cleanup_hook),
        ]

with MyApp() as app:
```

```
    # add arguments to the parser
    app.args.add_argument('-f', '--foo', action='store', metavar='STR',
                          help='the notorious foo option')

    # log stuff
    app.log.debug("About to run my myapp application!")

    # run the application
    app.run()

    # continue with additional application logic
    if app.pargs.foo:
        app.log.info("Received option: foo => %s" % app.pargs.foo)
```

And execution:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

optional arguments:
  -h, --help         show this help message and exit
  --debug            toggle debug output
  --quiet            suppress all output
  -f STR, --foo STR  the notorious foo option

$ python myapp.py --foo=bar
INFO: Received option: foo => bar
```

Diving Right In

This final example demonstrates the use of application controllers that handle command dispatch and rapid development.

myapp.py

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

class MyBaseController(CementBaseController):
    class Meta:
        label = 'base'
        description = "My Application does amazing things!"
        arguments = [
            ( ['-f', '--foo'],
              dict(action='store', help='the notorious foo option') ),
            ( ['-C'],
              dict(action='store_true', help='the big C option') ),
            ]

    @expose(hide=True)
    def default(self):
        self.app.log.info('Inside MyBaseController.default()')
        if self.app.pargs.foo:
            print("Recieved option: foo => %s" % self.app.pargs.foo)

    @expose(help="this command does relatively nothing useful")
    def command1(self):
```

```
        self.app.log.info("Inside MyBaseController.command1()")

    @expose(aliases=['cmd2'], help="more of nothing")
    def command2(self):
        self.app.log.info("Inside MyBaseController.command2()")


class MySecondController(CementBaseController):
    class Meta:
        label = 'second'
        stacked_on = 'base'

    @expose(help='this is some command', aliases=['some-cmd'])
    def second_cmd1(self):
        self.app.log.info("Inside MySecondController.second_cmd1")


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = 'base'
        handlers = [MyBaseController, MySecondController]



with MyApp() as app:
    app.run()
```

As you can see, we're able to build out the core functionality of our app such as arguments and sub-commands via controller classes.

Lets see what this looks like:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

My Application does amazing things!

commands:

  command1
    this command does relatively nothing useful

  command2 (aliases: cmd2)
    more of nothing

  second-cmd1 (aliases: some-cmd)
    this is some command

optional arguments:
  -h, --help          show this help message and exit
  --debug             toggle debug output
  --quiet             suppress all output
  -f FOO, --foo FOO   the notorious foo option
  -C                  the big C option

$ python myapp.py
INFO: Inside MyBaseController.default()
```

```
$ python myapp.py command1
INFO: Inside MyBaseController.command1()

$ python myapp.py command2
INFO: Inside MyBaseController.command2()

$ python myapp.py second-cmd1
INFO: Inside MySecondController.second_cmd1()
```

Interfaces and Handlers

Cement has a unique interface and handler system that is used to break up pieces of the framework and allow customization of how Cement handles everything from logging to config file parsing, and almost every action in between.

The Cement Interface code is loosely modeled after Zope Interface which allows a developer to define an interface that other developers can then create implementations for. For example, an interface might define that a class have a function called `_setup()`. Any implementation of that interface must provide a function called `_setup()`, and perform the expected actions when called.

In Cement, we call the implementation of interfaces `handlers` and provide the ability to easily register, and retrieve them via the app.

API References:

- *Cement Interface Module*
- *Cement Handler Module*

Defining an Interface

Cement uses interfaces and handlers extensively to manage the framework, however developers can also make use of this system to provide a clean, and standardized way of allowing other developers to customize their application.

The following defines a basic interface:

```
from cement.core.foundation import CementApp
from cement.core.interface import Interface, Attribute

class MyInterface(Interface):
    class IMeta:
        label = 'myinterface'

    # Must be provided by the implementation
    Meta = Attribute('Handler Meta-data')
    my_var = Attribute('A variable of epic proportions.')

    def _setup(app_obj):
        """
        The setup function is called during application initialization and
        must 'setup' the handler object making it ready for the framework
        or the application to make further calls to it.

        Required Arguments:

            app_obj
                The application object.
```

```
        Returns: n/a

        """

    def do_something():
        """
        This function does something.

        """

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        define_handlers = [MyInterface]
```

Alternatively, if you need more control you might define a handler this way:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    # define interfaces after app is created
    app.handler.define(MyInterface)

    app.run()
```

The above simply defines the interface. It does *not* implement any functionality, and can't be used directly. This is why the class functions do not have an argument of `self`, nor do they contain any code other than comments.

That said, what is required is an `IMeta` class that is used to interact with the interface. At the very least, this must include a unique `label` to identify the interface. This can also be considered the 'handler type'. For example, the `ILog` interface has a label of `log` and any handlers registered to that interface are stored in `HandlerManager.__handlers__['log']`.

Notice that we defined `Meta` and `my_var` as Interface Attributes. This is a simple identifier that describes an attribute that an implementation is expected to provide.

Validating Interfaces

A validator call back function can be defined in the interfaces IMeta class like this:

```
from cement.core import interface

def my_validator(klass, obj):
    members = [
        '_setup',
        'do_something',
        'my_var',
        ]
    interface.validate(MyInterface, obj, members)

class MyInterface(interface.Interface):
    class IMeta:
        label = 'myinterface'
        validator = my_validator
    ...
```

When `CementApp.handler.register()` is called to register a handler to an interface, the validator is called and the handler object is passed to the validator. In the above example, we simply define what members we want to validate for and then call `interface.validate()` which will raise `cement.core.exc.InterfaceError` if validation fails. It is not necessary to use `interface.validate()` but it is useful and recommended. In general, the key thing to note is that a validator either raises `InterfaceError` or does nothing if validation passes.

Registering Handlers to an Interface

An interface simply defines what an implementation is expected to provide, where a handler actually implements the interface. The following example is a handler that implements the MyInterface above:

```
from cement.core.foundation import CementApp
from cement.core.handler import CementBaseHandler
from myapp.interfaces import MyInterface

class MyHandler(CementBaseHandler):
    class Meta:
        interface = MyInterface
        label = 'my_handler'
        description = 'This handler implements MyInterface'
        config_defaults = dict(
            foo='bar'
            )

    my_var = 'This is my var'

    def __init__(self, *args, **kw):
        super(MyHandler, self).__init__(*args, **kw)

    def _setup(self, app_obj):
        super(MyHandler, self)._setup(app_obj)

    def do_something(self):
        print "Doing work!"

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        handlers = [MyHandler]
```

Note that you have to invoke *super()* in `__init__()` and `_setup` to correctly initialize and setup your Handler.

Alternatively, if you need more control you might use this approach:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    # register handler after the app is created
    app.handler.register(MyHandler)

    app.run()
```

The above is a simple class that meets all the expectations of the interface. When calling `CementApp.handler.register()`, `MyHandler` is passed to the validator (if defined in the interface) and if it passes validation will be registered into `HandlerManager.__handlers__`.

Using Handlers

The following are a few examples of working with handlers:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    # Get a log handler called 'logging'
    lh = app.handler.get('log', 'logging')

    # Instantiate the handler class, passing any keyword arguments that
    # the handler supports.
    log = log_handler()

    # Setup the handler, passing it the app object.
    log._setup(app)

    # List all handlers of type 'config'
    app.handler.list('config')

    # Check if an interface called 'output' is defined
    app.handler.defined('output')

    # Check if the handler 'argparse' is registered to the 'argument'
    # interface
    app.handler.registered('argument', 'argparse')
```

It is important to note that handlers are stored with the app as uninstantiated objects. Meaning you must instantiate them after retrieval, and call `_setup(app)` when using handlers directly (as in the above example).

Overriding Default Handlers

Cement sets up a number of default handlers for logging, config parsing, etc. These can be overridden in a number of ways. The first way is by passing them as keyword arguments to `CementApp`:

```
from cement.core.foundation import CementApp
from myapp.log import MyLogHandler

# Create the application
app = CementApp('myapp', log_handler=MyLogHandler)
app.setup()
app.run()
app.close()
```

The second way to override a handler is by setting it directly in the `CementApp` meta data:

```
from cement.core.foundation import CementApp
from myapp.log import MyLogHandler

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        log_handler = MyLogHandler

with MyApp() as app:
    app.run()
```

There are times that you may want to pre-instantiate handlers before passing them to CementApp(). The following works just the same:

```
from cement.core.foundation import CementApp
from myapp.log import MyLogHandler

my_log = MyLogHandler(some_param='some_value')

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        log_handler = my_log

with MyApp() as app:
    app.run()
```

To see what default handlers can be overridden, see the *cement.core.foundation* documentation.

Multiple Registered Handlers

All handlers and interfaces are unique. In most cases, where the framework is concerned, only one handler is used. For example, whatever is configured for the `log_handler` will be used and setup as `app.log`. However, take for example an Output Handler. You might have a default `output_handler` of `mustache`‘ (a text templating language) but may also want to override that handler with the `json` output handler when `-o json` is passed at command line. In order to allow this functionality, both the `mustache` and `json` output handlers must be registered.

Any number of handlers can be registered to an interface. You might have a use case for an Interface/Handler that may provide different compatibility base on the operating system, or perhaps based on simply how the application is called. A good example would be an application that automates building packages for Linux distributions. An interface would define what a build handler needs to provide, but the build handler would be different based on the OS. The application might have an `rpm` build handler, or a `dpkg` build handler to perform the build process differently.

Customizing Handlers

The most common way to customize a handler is to subclass it, and then pass it to `CementApp`:

```
from cement.core.foundation import CementApp
from cement.lib.ext_logging import LoggingLogHandler

class MyLogHandler(LoggingLogHandler):
    class Meta:
        label = 'mylog'

    def info(self, msg):
        # do something to customize this function, here...
        super(MyLogHandler, self).info(msg)

app = CementApp('myapp', log_handler=MyLogHandler)
```

Handler Default Configuration Settings

All handlers can define default config file settings via their `config_defaults` meta option. These will be merged into the `app.config` under the `[handler_interface].[handler_label]` section. These settings are overridden in the following order.

- The config_defaults dictionary passed to `CementApp`
- Via any application config files with a `[handler_interface].[handler_type]` block (i.e. `cache.memcached`)

The following shows how to override defaults by passing them with the defaults dictionary to `CementApp`:

```
from cement.core import foundation
from cement.utils.misc import init_defaults

defaults = init_defaults('myinterface.myhandler')
defaults['myinterface.myhandler'] = dict(foo='bar')
app = foundation.CementApp('myapp', config_defaults=defaults)
```

Cement will use all defaults set via `MyHandler.Meta.config_defaults` (for this example), and then override just what is passed via `config_defaults['myinterface.myhandler']`. You should use this approach only to modify the global defaults for your application. The second way is to then set configuration file defaults under the `[myinterface.myhandler]` section. For example:

my.config

```
[myinterface.myhandler]
foo = bar
```

In the real world this may look like `[cache.memcached]`, or `[database.mysql]` depending on what the interface label, and handler label's are. Additionally, individual handlers can override their config section by setting `Meta.config_section`.

Overriding Handlers Via Command Line

In some use cases, you will want the end user to have access to override the default handler of a particular interface. For example, Cement ships with multiple Output Handlers including `json`, `yaml`, and `mustache`. A typical application might default to using `mustache` to render console output from text templates. That said, without changing any code in the application, the end user can simply pass the `-o json` command line option and output the same data that is rendered to template, out in pure JSON.

The only built-in handler override that Cement includes is for the above mentioned example, but you can add any that your application requires.

The following example shows this in action... note that the following is already setup by Cement, but we're putting it here for clarity:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'

        # define what extensions we want to load
        extensions = ['mustache', 'json', 'yaml']

        # define our default output handler
        output_handler = 'mustache'

        # define our handler override options
        handler_override_options = dict(
            output = (['-o'], dict(help='output format')),
            )
```

```
with MyApp() as app:
    # run the application
    app.run()

    # define some data for the output handler
    data = dict(foo='bar')

    # render something using out output handlers, using mustache by
    # default which will use the default.m templae
    app.render(data, 'default.m')
```

Note what we see at command line:

```
$ python myapp.py --help
usage: myapp.py [-h] [--debug] [--quiet] [-o {yaml,json}]

optional arguments:
  -h, --help      show this help message and exit
  --debug         toggle debug output
  --quiet         suppress all output
  -o {yaml,json}  output format
```

Notice the `-o` command line option, that includes the choices: `yaml` and `json`. This feature will include all Output Handlers that have the `overridable` meta-data option set to `True`. The MustacheOutputHandler does not set this option, therefore it does not show up as a valid choice.

Now what happens when we run it?

```
$ python myapp.py

This text is being rendered via Mustache.
The value of the 'foo' variable is => 'bar'
```

The above is the default output, using `mustache` as our `output_handler`, and rendering the output text from a template called `default.m`. We can now override the output handler using the `-o` option and modify the output format:

```
$ python myapp.py -o json
{"foo": "bar"}
```

Again, any handler can be overridden in this fashion.

Configuration Handling

Cement defines a configuration interface called *IConfig*, as well as the default *ConfigParserConfigHandler* that implements the interface. This handler is built on top of ConfigParser which is included in the Python standard library. Therefor, this class will work much like ConfigParser but with any added functions necessary to meet the requirements of the `IConfig` interface.

Please note that there are other handlers that implement the `IConfig` interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following config handlers are included and maintained with Cement:

- *ConfigParserConfigHandler* (default)

- *ConfigObjConfigHandler*
- *JsonConfigHandler*
- *JsonConfigObjConfigHandler*
- *YamlConfigHandler*
- *YamlConfigObjHandler*

Please reference the *IConfig* interface documentation for writing your own config handler.

Configuration Ordering

An applications configuration is made up of a number of things, including default settings, handler defaults, config file settings, etc. The following is the order in which configurations are discovered and loaded:

- Defaults defined in `CementApp.Meta.config_defaults` or passed as `config_defaults` keyword to `CementApp`
- Extended by `CementBaseHandler.Meta.config_defaults` (not overridden)
- Overridden by configuration files defined in `CementApp.Meta.config_files` in the order they are listed/loaded (last has precedence)
- Overridden by command line options that match the same key name (only if `CementApp.Meta.arguments_override_config == True` or if the argument name is listed in `CementApp.Meta.override_arguments`)

Application Default Settings

Cement does not require default config settings in order to operate. That said, these settings are found under the `app_label` application section of the configuration, and overridden by a `[<app_label>]` block from a configuration file.

A default dictionary is used if no other defaults are passed when creating an application. For example, the following:

```
from cement.core import foundation
app = foundation.CementApp('myapp')
```

Is equivalent to:

```
from cement.core import foundation
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp')
app = foundation.CementApp('myapp', config_defaults=defaults)
```

That said, you can override default settings or add your own defaults like so:

```
from cement.core import foundation
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'section1','section2')
defaults['section1']['foo'] = 'bar'
defaults['section2']['foo2'] = 'bar2'

app = foundation.CementApp('myapp', config_defaults=defaults)
```

It is important to note that the default settings, which is a dict, is parsed by the config handler and loaded into it's own configuration mechanism. Meaning, though some config handlers (i.e. `ConfigObjConfigHandler`) might also be accessible like a dict, not all do (i.e. `ConfigParserConfigHandler`). Please see the documentation for the config handler you use for their full usage when accessing the `app.config` object.

Built-in Defaults

The following are not required to exist in the config defaults, however if they do, Cement will honor them (overriding or appending to built-in defaults).

debug = `False` Toggles debug output. By default, this setting is also overridden by the `[<app_label>] -> debug` config setting parsed in any of the application configuration files.

ignore_deprecation_warnings = `False` Disable deprecation warnings from being logged by Cement.

extensions = `None` List of additional framework extensions to load. Any extensions defined here will be appended to the application's defined extensions.

plugin_config_dir = `None` A directory path where plugin config files can be found. Files must end in `.conf`. By default, this setting is also overridden by the `[<app_label>] -> plugin_config_dir` config setting parsed in any of the application configuration files.

If set, this item will be **appended** to `CementApp.Meta.plugin_config_dirs` so that it's settings will have presedence over other config files.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `plugin_config_dir` without completely trumping the hard-coded list of default `plugin_config_dirs` defined by the app/developer.

plugin_dir = `None` A directory path where plugin code (modules) can be loaded from. By default, this setting is also overridden by the `[<app_label>] -> plugin_dir` config setting parsed in any of the application configuration files.

If set, this item will be **prepended** to `CementApp.Meta.plugin_dirs` so that a user's defined `plugin_dir` has precedence over other `plugin_dirs`.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `plugin_dir` without completely trumping the hard-coded list of default `plugin_dirs` defined by the app/developer.

template_dir = `None` A directory path where template files can be loaded from. By default, this setting is also overridden by the `[<app_label>] -> template_dir` config setting parsed in any of the application configuration files.

If set, this item will be appended to `CementApp.Meta.template_dirs`.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `template_dir` without completely trumping the hard-coded list of default `template_dirs` defined by the app/developer.

Application Configuration Defaults vs Handler Configuration Defaults

There may be slight confusion between the `CementApp.Meta.config_defaults` and the `CementBaseHandler.Meta.config_defaults` options. They both are very similar, however the application level configuration defaults are intended to be used to set defaults for multiple sections. Therefore, the `CementApp.Meta.config_defaults` option is a `dict` with nested `dict`'s under it. Each key of the top level `dict` relates to a config `[section]` and the nested `dict` are the settings for that `[section]`.

The `CementBaseHandler.Meta.config_defaults` only partain to a single `[section]` and therefore is only a single level `dict`, whose settings are applied to the `CementBaseHandler.Meta.config_section` of the application's configuration.

Accessing Configuration Settings

After application creation and setup, you can access the config handler via the `app.config` object. For example:

```
from cement.core import foundation
app = foundation.CementApp('myapp')

# First setup the application
app.setup()

# Get settings
app.config.get('myapp', 'debug')

# Set settings
app.config.set('myapp', 'debug', True)

# Get sections (configuration [blocks])
app.config.get_sections()

# Add a section
app.config.add_section('my_config_section')

# Test if a section exists
app.config.has_section('my_config_section')

# Get configuration keys for the 'myapp' section
app.config.keys('myapp')

# Test if a key exist
if 'debug' in app.config.keys('myapp')

# Merge a dict of settings into the config
other_config = dict()
other_config['myapp'] = dict()
other_config['myapp']['foo'] = 'not bar'
app.config.merge(other_config)
```

Parsing Config Files

Most applications benefit from allowing their users to customize runtime via a configuration file. This can be done by:

```
from cement.core import foundation
app = foundation.CementApp('myapp')

# First setup the application
app.setup()

# Parse a configuration file
app.config.parse_file('/path/to/some/file.conf')
```

Note that Cement automatically parses any config files listed in the `CementApp.Meta.config_files` list. For example:

```
from cement.core import foundation, backend

app = foundation.CementApp('myapp',
        config_files = [
            '/path/to/config1',
            '/path/to/config2'
            ],
        )
```

If no `config_files` meta is provided, Cement will set the defaults to the following common and sane defaults:

- `/etc/<app_label>/<app_label>.conf`
- `~/.<app_label>.conf`
- `~/.<app_label>/config`

Overriding Configurations with Command Line Options

Config settings can **optionally** overridden by a passed command line option if the option name matches a configuration key. Note that this will happen in *all* config sections if enabled:

```
from cement.core.foundation import CementApp
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp')
defaults['myapp']['foo'] = 'bar'

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        config_defaults = defaults
        arguments_override_config = True

with MyApp() as app:
    app.args.add_argument('--foo', action='store', dest='foo')
    app.run()
```

With `arguments_override_config` enabled, running the above application and passing the `--foo=some_value` option will override the `foo` setting under a `[myapp]` configuration section as well as any other section that has a matching `foo` key.

Configuration Options Versus Meta Options

As you will see extensively throughout the Cement code is the use of Meta options. There can be some confusion between the use of Meta options, and application configuration options. The following explains the two:

Configuration Options

Configuration options are application specific. There are config defaults defined by the application developer, that can be (and are intended to be) overridden by user defined settings in a configuration file.

Cement does not rely on the application configuration, though it can honor configuration settings. For example, `CementApp` honors the `debug` config option which is documented, but it doesn't rely on it existing either.

The key things to note about configuration options are:

- They give the end user flexibility in how the application operates.

- Anything that you want users to be able to customize via a config file. For example, the path to a log file or the location of a database server. These are things that you do not want hard-coded into your app, but rather might want sane defaults for.

Meta Options

Meta options are used on the backend by developers to alter how classes operate. For example, the `CementApp` class has a meta option of `log_handler`. The default log handler is `LoggingLogHandler`, however because this is built on an interface definition, Cement can use any other log handler the same way without issue as long as that log handler abides by the interface definition. Meta options make this change seamless and allows the handler to alter functionality, rather than having to change code in the top level class itself.

The key thing to note about Meta options are:

- They give the developer flexibility in how the code operates.
- End users should not have access to modify Meta options via a config file or similar 'dynamic' configuration (unless those specific options are listed in `CementApp.Meta.core_meta_override` or `CementApp.Meta.meta_override` (for example, the `debug` setting under `[<app_label>]` overrides `CementApp.Meta.debug` by default.
- Meta options are used to alter how classes work, however are considered 'hard-coded' settings. If the developer chooses to alter a Meta option, it is for the life of that release.
- Meta options should have a sane default, and be clearly documented.

Argument and Option Handling

Cement defines an argument interface called *IArgument*, as well as the default *ArgParseArgumentHandler* that implements the interface. This handler is built on top of the ArgParse module which is included in the Python standard library.

Please note that there may be other handler's that implement the `IArgument` interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following argument handlers are included and maintained with Cement:

- *ArgParseArgumentHandler*

Please reference the *IArgument* interface documentation for writing your own argument handler.

Adding Arguments

The `IArgument` interface is loosely based on `ArgParse` directly. That said, it only defines a minimal set of params that must be honored by the handler implementation, even though the handler itself may except more than that. The following shows some basic examples of adding arguments based on the interface (meaning, these examples should work regardless of what the handler is):

```
from cement.core import foundation

# create the application
app = foundation.CementApp('myapp')

# then setup the application... which will use our 'mylog' handler
app.setup()

# add any arguments after setup(), and before run()
app.args.add_argument('-f', '--foo', action='store', dest='foo',
                      help='the notorious foo option')
```

```
app.args.add_argument('-V', action='store_true', dest='vendetta',
                      help='v for vendetta')
app.args.add_argument('-A', action='store_const', const=12345,
                      help='the big a option')

# then run the application
app.run()

# access the parsed args from the app.pargs shortcut
if app.pargs.foo:
    print "Received foo option with value %s" % app.pargs.foo
if app.pargs.vendetta:
    print "Received V for Vendetta!"
if app.pargs.A:
    print "Received the A option with value %s" % app.pargs.A

# close the application
app.close()
```

Here we have setup a basic application, and then add a few arguments to the parser.

```
$ python test.py --help
usage: test.py [-h] [--debug] [--quiet] [-f FOO] [-V] [-A]

optional arguments:
  -h, --help         show this help message and exit
  --debug            toggle debug output
  --quiet            suppress all output
  -f FOO, --foo FOO  the notorious foo option
  -V                 v for vendetta
  -A                 the big a option

$ python test.py --foo=bar
Received foo option with value bar

$ python test.py -V
Received V for Vendetta!
```

Accessing Parsed Arguments

The `IArgument` interface defines that the `parse()` function return any type of object that stores the name of the argument as a class member. Meaning, when adding the `foo` option with `action='store'` and the value is stored as the `foo` destination... that would be accessible as `app.pargs.foo`. In the case of the `ArgParseArgumentHandler` the return object is exactly what you would expect by calling `parser.parse_args()`, but may be different with other argument handler implementations.

The parsed arguments are actually stored as `app._parsed_args`, but are exposed as `app.pargs`. Accessing `app.pargs` can be seen in the examples above.

Log Handling

Cement defines a logging interface called *ILog*, as well as the default *LoggingLogHandler* that implements the interface. This handler is built on top of the Logging module which is included in the Python standard library.

Please note that there may be other handler's that implement the ILog interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following log handlers are included and maintained with Cement:

- *LoggingLogHandler*
- *ColorLogHandler*

Please reference the *ILog* interface documentation for writing your own log handler.

Logging Messages

The following shows logging to each of the defined log levels.

```
from cement.core import foundation
app = foundation.CementApp('myapp')

# First setup the application
app.setup()

# Run the application (even though it doesn't do much here)
app.run()

# Log a debug message
app.log.debug('This is a debug message.')

# Log an info message
app.log.info('This is an info message.')

# Log a warning message
app.log.warning('This is a warning message.')

# Log an error message
app.log.error('This is an error message.')

# Log an fatal error message
app.log.fatal('This is a fatal message.')

# Close the application
app.close()
```

The above is displayed in order of 'severity' you can say. If the log level is set to 'INFO', you will receive all 'info' messages and above .. including warning, error, and fatal. However, you will not receive DEBUG level messages. The same goes for a log level of 'WARNING', where you will receive warning, error, and fatal... but you will not receive INFO, or DEBUG level messages.

Changing Log Level

The log level defaults to INFO, based on the 'config_defaults' of the log handler. You can override this via config_defaults:

```
from cement.core import foundation, backend
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'log.logging')
defaults['log.logging']['level'] = 'WARNING'
```

```
app = foundation.CementApp('myapp', config_defaults=defaults)
app.setup()
```

This will also be overridden by the 'level' setting under a '[log.logging]' section in any of the applications configuration files that are parsed.

You should also note that Cement includes a '–debug' command line option by default. This triggers the log level to 'DEBUG' and is helpful for quickly debugging issues:

```
$ python test.py --debug
2012-07-13 02:19:42,270 (DEBUG) cement.core.foundation : laying cement for the 'myapp
↪' application
2012-07-13 02:19:42,270 (DEBUG) cement.core.hook : defining hook 'pre_setup'
2012-07-13 02:19:42,270 (DEBUG) cement.core.hook : defining hook 'post_setup'
2012-07-13 02:19:42,270 (DEBUG) cement.core.hook : defining hook 'pre_run'
2012-07-13 02:19:42,270 (DEBUG) cement.core.hook : defining hook 'post_run'
2012-07-13 02:19:42,271 (DEBUG) cement.core.hook : defining hook 'pre_close'
2012-07-13 02:19:42,271 (DEBUG) cement.core.hook : defining hook 'post_close'
2012-07-13 02:19:42,271 (DEBUG) cement.core.hook : defining hook 'signal'
2012-07-13 02:19:42,271 (DEBUG) cement.core.hook : defining hook 'pre_render'
2012-07-13 02:19:42,271 (DEBUG) cement.core.hook : defining hook 'post_render'
2012-07-13 02:19:42,271 (DEBUG) cement.core.handler : defining handler type 'extension
↪' (IExtension)
2012-07-13 02:19:42,271 (DEBUG) cement.core.handler : defining handler type 'log'␣
↪(ILog)
2012-07-13 02:19:42,271 (DEBUG) cement.core.handler : defining handler type 'config'␣
↪(IConfig)
2012-07-13 02:19:42,271 (DEBUG) cement.core.handler : defining handler type 'plugin'␣
↪(IPlugin)
2012-07-13 02:19:42,272 (DEBUG) cement.core.handler : defining handler type 'output'␣
↪(IOutput)
2012-07-13 02:19:42,272 (DEBUG) cement.core.handler : defining handler type 'argument
↪' (IArgument)
2012-07-13 02:19:42,272 (DEBUG) cement.core.handler : defining handler type
↪'controller' (IController)
2012-07-13 02:19:42,272 (DEBUG) cement.core.handler : defining handler type 'cache'␣
↪(ICache)
2012-07-13 02:19:42,272 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.core.extension.CementExtensionHandler'>' into handlers['extension']['cement
↪']
2012-07-13 02:19:42,272 (DEBUG) cement.core.foundation : now setting up the 'myapp'␣
↪application
2012-07-13 02:19:42,272 (DEBUG) cement.core.foundation : adding signal handler for␣
↪signal 15
2012-07-13 02:19:42,273 (DEBUG) cement.core.foundation : adding signal handler for␣
↪signal 2
2012-07-13 02:19:42,273 (DEBUG) cement.core.foundation : setting up myapp.extension␣
↪handler
2012-07-13 02:19:42,273 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪dummy' framework extension
2012-07-13 02:19:42,273 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_dummy.DummyOutputHandler'>' into handlers['output']['null']
2012-07-13 02:19:42,273 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪plugin' framework extension
2012-07-13 02:19:42,273 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_plugin.CementPluginHandler'>' into handlers['plugin']['cement']
2012-07-13 02:19:42,273 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪configparser' framework extension
```

```
2012-07-13 02:19:42,274 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_configparser.ConfigParserConfigHandler'>' into handlers['config'][
↪'configparser']
2012-07-13 02:19:42,274 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪logging' framework extension
2012-07-13 02:19:42,274 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_logging.LoggingLogHandler'>' into handlers['log']['logging']
2012-07-13 02:19:42,274 (DEBUG) cement.core.extension : loading the 'cement.ext.ext_
↪argparse' framework extension
2012-07-13 02:19:42,276 (DEBUG) cement.core.handler : registering handler '<class
↪'cement.ext.ext_argparse.ArgParseArgumentHandler'>' into handlers['argument'][
↪'argparse']
2012-07-13 02:19:42,276 (DEBUG) cement.core.foundation : setting up myapp.config␣
↪handler
2012-07-13 02:19:42,276 (DEBUG) cement.ext.ext_configparser : config file '/etc/myapp/
↪myapp.conf' does not exist, skipping...
2012-07-13 02:19:42,277 (DEBUG) cement.core.foundation : no cache handler defined,␣
↪skipping.
2012-07-13 02:19:42,277 (DEBUG) cement.core.foundation : setting up myapp.log handler
2012-07-13 02:19:42,277 (DEBUG) cement.core.handler : merging config defaults from '
↪<cement.ext.ext_logging.LoggingLogHandler object at 0x100588dd0>'
2012-07-13 02:19:42,277 (DEBUG) myapp : logging initialized for 'myapp' using␣
↪LoggingLogHandler
2012-07-13 02:19:42,278 (DEBUG) cement.core.foundation : setting up myapp.plugin␣
↪handler
2012-07-13 02:19:42,278 (DEBUG) cement.ext.ext_plugin : plugin config dir /etc/myapp/
↪plugins.d does not exist.
2012-07-13 02:19:42,278 (DEBUG) cement.core.foundation : setting up myapp.arg handler
2012-07-13 02:19:42,279 (DEBUG) cement.core.foundation : setting up myapp.output␣
↪handler
2012-07-13 02:19:42,279 (DEBUG) cement.core.foundation : setting up application␣
↪controllers
2012-07-13 02:19:42,279 (DEBUG) cement.core.foundation : no controller could be found.
2012-07-13 02:19:42,280 (DEBUG) cement.core.foundation : closing the application
```

You can see that debug logging is extremely verbose. In the above you will note the message format is:

```
TIMESTAMP - LEVEL - MODULE - MESSAGE
```

The Cement framework only logs to DEBUG, where the MODULE is displayed as 'cement.core.whatever'. Note that Cement uses a minimal logger that is separate from the application log, therefore settings you change in your application do not affect it.

Logging to Console

The default log handler configuration enables logging to console. For example:

```
from cement.core import foundation
app = foundation.CementApp('myapp')
app.setup()
app.run()
app.log.info('This is my info message')
app.close()
```

When running this script at command line you would get:

```
$ python test.py
INFO: This is my info message
```

This can be disabled by setting 'to_console=False' in either the application defaults, or in an application configuration file under the '[log.logging]' section.

Logging to a File

File logging is disabled by default, but is just one line to enable. Simply set the 'file' setting under the '[log.logging]' config section either by application defaults, or via a configuration file.

```
from cement.core import foundation, backend
from cement.utils.misc import init_defaults

defaults = init_defaults('myapp', 'log.logging')
defaults['log.logging']['file'] = 'my.log'

app = foundation.CementApp('myapp', config_defaults=defaults)
app.setup()
app.run()
app.log.info('This is my info message')
app.close()
```

Running this we will see:

```
$ python test.py
INFO: This is my info message

$ cat my.log
2011-08-26 17:50:16,306 (INFO) myapp : This is my info message
```

Notice that the logging is a bit more verbose when logged to a file.

Tips on Debugging

Note: The following is specific to the default *LoggingLogHandler* only, and is not an implementation of the ILog interface.

Logging to 'app.log.debug()' is pretty straight forward, however adding an additional parameter for the 'namespace' can greatly increase insight into where that log is happening. The 'namespace' defaults to the application name which you will see in every log like this:

```
2012-07-30 18:05:11,357 (DEBUG) myapp : This is my message
```

For debugging, it might be more useful to change this to __name__:

```
app.log.debug('This is my info message', __name__)
```

Which looks like:

```
2012-07-30 18:05:11,357 (DEBUG) myapp.somepackage.test : This is my message
```

Or even more verbose, the __file__ and a line number of the log:

```
app.log.debug('This is my info message', '%s,L2734' % __file__)
```

Which looks like:

```
2012-07-30 18:05:11,357 (DEBUG) myapp/somepackage/test.py,L2345 : This is my message
```

You can override this with anything... it doesn't have to be just for debugging.

Output Handling

Cement defines an output interface called *cement.core.output.IOutput*, as well as the default *cement.ext.ext_dummy.DummyOutputHandler* that implements the interface. This handler is part of Cement, and actually does nothing to produce output. Therefore it can be said that by default a Cement application does not handle rendering output to the console, but can if another output handler be used.

Please note that there may be other handler's that implement the `IOutput` interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following output handlers are included and maintained with Cement:

- *cement.ext.ext_dummy.DummyOutputHandler*
- *cement.ext.ext_json.JsonOutputHandler*
- *cement.ext.ext_yaml.YamlOutputHandler*
- *cement.ext.ext_genshi.GenshiOutputHandler*
- cement.ext.ext_handlebars.HandlebarsOutputHandler
- *cement.ext.ext_jinja2.Jinja2OutputHandler*
- *cement.ext.ext_mustache.MustacheOutputHandler*
- *cement.ext.ext_tabulate.TabulateOutputHandler*

Please reference the *cement.core.output.IOutput* interface documentation for writing your own output handler.

Rending Output

Cement applications do not need to use an output handler by any means. Most small applications can get away with simple `print()` statements. However, anyone who has ever built a bigger application that produces a lot of output will know that this can get ugly very quickly in your code.

Using an output handler allows the developer to keep their logic clean, and offload the display of relevant data to an output handler, possibly by templates or other means (GUI?).

An output handler has a `render()` function that takes a data dictionary to produce output. Some output handlers may also accept a `template` or other parameters that define how output is rendered. This is easily accessible by the application object.

```
from cement.core import foundation, output

# Create the application
app = foundation.CementApp('myapp')

# Setup the application
app.setup()
```

```
# Run the application
app.run()

# Add application logic
data = dict(foo='bar')
app.render(data)

# Close the application
app.close()
```

The above example uses the default output handler, therefore nothing is displayed on screen. That said, if we write our own quickly we can see something happen:

```
from cement.core import foundation, handler, output

# Create a custom output handler
class MyOutput(output.CementOutputHandler):
    class Meta:
        label = 'myoutput'

    def render(self, data):
        for key in data:
            print "%s => %s" % (key, data[key])

app = foundation.CementApp('myapp', output_handler=MyOutputHandler)
...
```

Which looks like:

```
$ python test.py
foo => bar
```

Rendering Output Via Templates

An extremely powerful feature of Cement is the ability to offload console output to a template output handler. Several are inluded with Cement but not enabled by default (listed above). The following example shows the use of the Mustache templating langugage, as well as Json output handling.

myapp.py

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose


class MyBaseController(CementBaseController):
    class Meta:
        label = 'base'
        description = 'MyApp Does Amazing Things'

    @expose(hide=True)
    def default(self):
        data = dict(foo='bar')
        self.app.render(data, 'default.m')

        # always return the data, some output handlers require this
```

```
        # such as Json/Yaml (which don't use templates)
        return data


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = MyBaseController
        extensions = ['mustache', 'json']

        # default output handler
        output_handler = 'mustache'


with MyApp() as app:
    app.run()
```

/usr/lib/myapp/templates/default.m

```
This is the output of the MyBaseController.default() command.

The value of the 'foo' variable is => '{{foo}}'
```

And this looks like:

```
$ python myapp.py

This is the output of the MyBaseController.default() command.

The value of the 'foo' variable is => 'bar'
```

Optionally, we can use the `JsonOutputHandler` via `-o json` to trigger just Json output (supressing all other output) using our return dictionary:

```
$ python myapp.py -o json
{"foo": "bar"}
```

Application Controllers

Cement defines a controller interface called *IController*, but does not enable any default handlers that implement the interface.

Using application controllers is not necessary, but enables rapid development by wrapping pieces of the framework like adding arguments, and linking commands with functions to name a few. The examples below use the `CementBaseController` for examples. It is important to note that this class also requires that your application's argument_handler be the `ArgParseArgumentHandler`. That said, the `CementBaseController` is relatively useless when used directly and therefore should be used as a Base class to create your own application controllers from.

The following controllers are included and maintained with Cement:

- *CementBaseController*

Please reference the *IController* interface documentation for writing your own controller.

Example Application Base Controller

This example demonstrates the use of application controllers that handle command dispatch and rapid development.

```
from cement.core import backend
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

# define an application base controller
class MyAppBaseController(CementBaseController):
    class Meta:
        label = 'base'
        description = "My Application does amazing things!"
        epilog = "This is the text at the bottom of --help."

        config_defaults = dict(
            foo='bar',
            some_other_option='my default value',
            )

        arguments = [
            (['-f', '--foo'],
             dict(action='store', help='the notorious foo option')),
            (['-C'],
             dict(action='store_true', help='the big c option')),
            ]

    @expose(hide=True, aliases=['run'])
    def default(self):
        self.app.log.info('Inside base.default function.')
        if self.app.pargs.foo:
            self.app.log.info("Recieved option 'foo' with value '%s'." % \
                              self.app.pargs.foo)

    @expose(help="this command does relatively nothing useful.")
    def command1(self):
        self.app.log.info("Inside base.command1 function.")

    @expose(aliases=['cmd2'], help="more of nothing.")
    def command2(self):
        self.app.log.info("Inside base.command2 function.")


class MyApp(CementApp):
    class Meta:
        label = 'example'
        base_controller = MyAppBaseController


with MyApp() as app:
    app.run()
```

As you can see, we're able to build out the core functionality of our app via a controller class. Lets see what this looks like:

```
$ python example.py --help
usage: example.py <CMD> -opt1 --opt2=VAL [arg1] [arg2] ...
```

```
My Application does amazing things!

commands:

  command1
    this command does relatively nothing useful.

  command2 (aliases: cmd2)
    more of nothing.

optional arguments:
  -h, --help  show this help message and exit
  --debug     toggle debug output
  --quiet     suppress all output
  --foo FOO   the notorious foo option
  -C          the big C option

This is the text at the bottom of --help.


$ python example2.py
INFO: Inside base.default function.

$ python example2.py command1
INFO: Inside base.command1 function.

$ python example2.py cmd2
INFO: Inside base.command2 function.
```

Additional Controllers and Namespaces

Any number of additional controllers can be added to your application after a base controller is created. Additionally, these controllers can be `stacked` onto the base controller (or any other controller) in one of two ways:

- `embedded` - The controllers commands and arguments are included under the parent controllers name space.
- `nested` - The controller label is added as a sub-command under the parent controllers namespace (effectively this is a sub-command with additional sub-sub-commands under it)

For example, The `base` controller is accessed when calling `example.py` directly. Any commands under the `base` controller would be accessible as `example.py <cmd1>`, or `example.py <cmd2>`, etc. An `embedded` controller will merge its commands and options into the `base` controller namespace and appear to be part of the base controller... meaning you would still access the `embedded` controllers commands as `example.py <embedded_cmd1>`, etc (same for options).

For `nested` controllers, a prefix will be created with that controllers label under its parents namespace. Therefore you would access that controllers commands and options as `example.py <controller_label> <controller_cmd1>`.

See the *Multiple Stacked Controllers* example for more help.

Framework Extensions

Cement defines an extension interface called *IExtension*, as well as the default *CementExtensionHandler* that implements the interface. Its purpose is to manage loading framework extensions and making them usable by the application. Extensions are similar to *Application Plugins*, but at the framework level (application agnostic).

Please note that there may be other handler's that implement the `IExtension` interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following extension handlers are included and maintained with Cement:

- *CementExtensionHandler*

Please reference the *IExtension* interface documentation for writing your own extension handler. Additionally, more information on available extensions and their use can be found in the *Cement API Documentation*

Important Note: As of Cement 2.1.3, optional extensions with external dependencies are now being shipped along with mainline sources. This means, that Cement Core continues to maintain a 100% zero dependency policy, however Framework Extensions *can* rely on external deps. It is the responsibility of the application developer to include these dependencies in their application (as the Cement package does not include these dependencies).

Extension Configuration Settings

The following Meta settings are honored under the CementApp:

extension_handler A handler class that implements the IExtension interface. This can be a string (label of a registered handler), an uninstantiated class, or an instantiated class object. Default: CementExtensionHandler.

core_extensions List of Cement core extensions. These are generally required by Cement and should only be modified if you know what you're doing. Use `extensions` to add to this list, rather than overriding core extensions. That said if you want to prune down your application, you can remove core extensions if they are not necessary (for example if using your own log handler extension you likely don't want/need `LoggingLogHandler` to be registered, but removing it really doesn't buy you much).

extensions List of additional framework extensions to load.

The following example shows how to alter these settings for your application:

```
from cement.core.foundation import CementApp
from cement.core.ext import CementExtensionHandler

class MyExtensionHandler(CementExtensionHandler):
    pass

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extension_handler = MyExtensionHandler
        extensions = ['myapp.ext.ext_something_fancy']

with MyApp() as app:
    app.run()
```

Creating an Extension

The extension system is a mechanism for dynamically loading code to extend the functionality of the framework. In general, this includes the registration of interfaces, handlers, and/or hooks.

The following is an example extension that provides an *Output Handler*. We will assume this extension is part of our `myapp` application, and the extension module will be `myapp.ext.ext_myoutput` (or whatever you want to call it).

```
from cement.core import handler, output
from cement.utils.misc import minimal_logger

LOG = minimal_logger(__name__)

class MyOutputHandler(output.CementOutputHandler):
    class Meta:
        label = 'myoutput'

    def render(self, data_dict, template=None):
        LOG.debug("Rendering output via MyAppOutputHandler")
        for key in data_dict.keys():
            print "%s => %s" % (key, data_dict[key])

def load(app):
    handler.register(MyOutputHandler)
```

Take note of two things. One is, the `LOG` we are using is from `cement.utils.misc.minimal_logger(__name__)`. Framework extensions do not use the application log handler, ever. Use the `minimal_logger()`, and only log to 'DEBUG' (recommended).

Secondly, in our extension file we need to define any interfaces, and register handlers and/or hooks if necessary. In this example we only needed to register our output handler (which happens when the extension is loaded by the application).

Last, notice that all `bootstrapping` code goes in a `load()` function. This is where registration of handlers/hooks should happen. For convenience, and certain edge cases, the `app` object is passed here in its current state at the time that `load()` is called.

You will notice that extensions are essentially the same as application plugins, however the difference is both when/how the code is loaded, as well as the purpose of that code. Framework extensions add functionality to the framework for the application to utilize, where application plugins extend the functionality of the application itself.

Loading an Extension

Extensions are loaded when `setup()` is called on an application. Cement automatically loads all extensions listed under the applications `core_extensions` and `extensions` meta options.

To load the above example into our application, we just add it to the list of `extensions` (not core extensions). Lets assume the extension code lives in `myapp/ext/ext_something_fancy.py`:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['myapp.ext.ext_something_fancy']

with MyApp() as app:
    app.run()
```

Note that Cement provides a shortcut for Cement extensions. For example, the following:

```
CementApp('myapp', extensions=['json', 'daemon'])
```

Is equivalent to:

```
CementApp('myapp',
    extensions=[
        'cement.ext.ext_json',
        'cement.ext.ext_daemon',
        ]
    )
```

For non-cement extensions you need to use the full python 'dotted' module path.

Loading Extensions Via a Configuration File

Some use cases require that end-users are able to modify what framework extensions are loaded via a configuration file. The following gives an example of how an application can support an optional `extensions` configuration setting that will **append** extensions to `CementApp.Meta.extensions`.

Note that extensions loaded in this way will happen **after** the config handler is setup. Normally, extensions are loaded just before the configuration files are read. Therefore, some extensions may not be compatible with this method if they attempt to perform any actions before `app.setup()` completes (such as in early framework hooks before configuration files are loaded).

myapp.py

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        config_files = [
            './myapp.conf',
            ]

def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

myapp.conf

```
[myapp]
extensions = json, yaml
```

Which looks like:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

MyApp Does Amazing Things

optional arguments:
  -h, --help     show this help message and exit
  --debug        toggle debug output
  --quiet        suppress all output
  -o {json,yaml} output format
```

Note the `-o` command line option that are provided by Cement allowing the end user to override the output handler with the available/loaded extensions (that support this feature).

Application Plugins

Cement defines a plugin interface called *IPlugin*, as well as the default *CementPluginHandler* that implements the interface.

Please note that there may be other handlers that implement the `IPlugin` interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following plugin handlers are included and maintained with Cement:

- *CementPluginHandler*

Please reference the *IPlugin* interface documentation for writing your own plugin handler.

Plugin Configuration Settings

There are a few settings related to how plugins are loaded under an applications meta options. These are:

plugins = `[]` A list of plugins to load. This is generally considered bad practice since plugins should be dynamically enabled/disabled via a plugin config file.

plugin_config_dirs = `None` A list of directory paths where plugin config files can be found. Files must end in `.conf` (or the extension defined by `CementApp.Meta.config_extension`), or they will be ignored.

Note: Though `CementApp.Meta.plugin_config_dirs` is `None`, Cement will set this to a default list based on `CementApp.Meta.label`. This will equate to:

```
['/etc/<app_label>/plugins.d', '~/.<app_label>/plugin.d']
```

Files are loaded in order, and have precedence in that order. Therefore, the last configuration loaded has precedence (and overwrites settings loaded from previous configuration files).

plugin_config_dir = `None` A directory path where plugin config files can be found. Files must end in `.conf` (or the extension defined by `CementApp.Meta.config_extension`), or they will be ignored. By default, this setting is also overridden by the `[<app_label>] -> plugin_config_dir` config setting parsed in any of the application configuration files.

If set, this item will be **appended** to `CementApp.Meta.plugin_config_dirs` so that it's settings will have presedence over other configuration files.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `plugin_config_dir` without completely trumping the hard-coded list of default `plugin_config_dirs` defined by the app/developer.

plugin_bootstrap = `None` A python package (dotted import path) where plugin code can be loaded from. This is generally something like `myapp.plugins` where a plugin file would live at `myapp/plugins/myplugin.py` or `myapp/plugins/myplugin/__init__.py`. This provides a facility for applications that have builtin plugins that ship with the applications source code and live in the same Python module.

Note: Though the meta default is `None`, Cement will set this to `<app_label>.plugins` if not set.

plugin_dirs = `None` A list of directory paths where plugin code (modules) can be loaded from.

Note: Though `CementApp.Meta.plugin_dirs` is None, Cement will set this to a default list based on `CementApp.Meta.label` if not set. This will equate to:

```
['~/.<app_label>/plugins', '/usr/lib/<app_label>/plugins']
```

Modules are attempted to be loaded in order, and will stop loading once a plugin is successfully loaded from a directory. Therefore this is the oposite of configuration file loading, in that here the first has precedence.

plugin_dir = None A directory path where plugin code (modules) can be loaded from. By default, this setting is also overridden by the `[<app_label>] -> plugin_dir` config setting parsed in any of the application configuration files.

If set, this item will be **prepended** to `Meta.plugin_dirs` so that a users defined `plugin_dir` has precedence over others.

In general, this setting should not be defined by the developer, as it is primarily used to allow the end-user to define a `plugin_dir` without completely trumping the hard-coded list of default `plugin_dirs` defined by the app/developer.

Creating a Plugin

A plugin is essentially an extension of a Cement application, that is loaded from an internal or external source location. It is a mechanism for dynamically loading code (whether the plugin is enabled or not). It can contain any code that would normally be part of your application, but should be thought of as optional features, where the core application does not rely on that code to operate.

The following is an example plugin (single file) that provides a number of options and commands via an application controller:

myplugin.py

```
from cement.core.controller import CementBaseController, expose

class MyPluginController(CementBaseController):
    class Meta:
        label = 'myplugin'
        description = 'this is my controller description'
        stacked_on = 'base'

        config_defaults = dict(
            foo='bar',
            )

        arguments = [
            (['--foo'],
             dict(action='store', help='the infamous foo option')),
            ]

    @expose(help="this is my command description")
    def mycommand(self):
        print 'in MyPlugin.mycommand()'

def load(app):
    app.handler.register(MyPluginController)
```

As you can see, this is very similar to an application that has a base controller, however as you'll note we do not create an application object via `foundation.CementApp()` like we do in our application. This code/file would then be saved to a location defined by your applications configuration that determines where plugins are loaded from (see the next section).

Notice that all 'bootstrapping' code goes in a `load()` function. This is where registration of handlers/hooks should happen. For convenience, and certain edge cases, the *app* object is passed here in its current state at the time that `load()` is called. You do not need to do anything with the `app` object, but you can.

A plugin also has a configuration file that will be Cement will attempt to find in one of the directories listed in `CementApp.Meta.plugin_config_dirs` as defined by your application's configuration. The following is an example plugin configuration file:

myplugin.conf

```
[myplugin]
enable_plugin = true
foo = bar
```

Loading a Plugin

Plugin modules are looked for first in one of the defined `plugin_dirs`, and if not found then Cement attempts to load them from the `plugin_bootstrap`. The following application shows how to configure an application to load plugins. Take note that these are the **default settings** and will work the same if not defined:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

class MyBaseController(CementBaseController):
    class Meta:
        label = 'base'
        description = 'MyApp Does Amazing Things'

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = MyBaseController
        plugin_bootstrap='myapp.bootstrap',
        plugin_config_dirs=[
            '/etc/myapp/plugins.d',
            '~/.myapp/plugins.d',
            ]
        plugin_dirs=[
            '/usr/lib/myapp/plugins',
            '~/.myapp/plugins',
            ]


def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

We modified the default settings for `plugin_config_dirs` and `plugin_dirs`. These are the default settings under `Cementapp`, however we have put them here for clarity.

Running this application will do nothing particularly special, however the following demonstrates what happens when we add a simple plugin that provides an application controller:

/etc/myapp/plugins.d/myplugin.conf

```
[myplugin]
enable_plugin = true
some_option = some value
```

/usr/lib/myapp/plugins/myplugin.py

```
from cement.core.controller import CementBaseController, expose
from cement.utils.misc import init_defaults

defaults = init_defaults('myplugin')

class MyPluginController(CementBaseController):
    class Meta:
        label = 'myplugin'
        description = 'this is my plugin description'
        stacked_on = 'base'
        config_defaults = defaults
        arguments = [
            (['--some-option'], dict(action='store')),
            ]

    @expose(help="this is my command description")
    def my_plugin_command(self):
        print 'In MyPlugin.my_plugin_command()'

def load(app):
    app.handler.register(MyPluginController)
```

Running our application with the plugin disabled, we see:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

MyApp Does Amazing Things

optional arguments:
  -h, --help  show this help message and exit
  --debug     toggle debug output
  --quiet     suppress all output
```

But if we enable the plugin, we get something a little different:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

MyApp Does Amazing Things

commands:

  my-plugin-command
    this is my command description

optional arguments:
  -h, --help            show this help message and exit
  --debug               toggle debug output
  --quiet               suppress all output
  --some-option SOME_OPTION
```

We can see that the `my-plugin-command` and the `--some-option` option were provided by our plugin, which has been 'stacked' on top of the base controller.

User Defined Plugin Configuration and Module Directories

Most applications will want to provide the ability for the end-user to define where plugin configurations and modules live. This is possible by setting the `plugin_config_dir` and `plugin_dir` settings in any of the applications configuration files. Note that these paths will be **added** to the built-in `plugin_config_dirs` and `plugin_dirs` settings respectively, rather than completely overwriting them. Therefore, your application can maintain it's default list of plugin configuration and module paths while also allowing users to define their own.

/etc/myapp/myapp.conf

```
[myapp]
plugin_dir = /usr/lib/myapp/plugins
plugin_config_dir = /etc/myapp/plugins.d
```

The `plugin_bootstrap` setting is however only configurable within the application itself.

What Can Go Into a Plugin?

The above example shows how to add an optional application controller via a plugin, however a plugin can contain anything you want. This could be as simple as adding a hook that does something magical. For example:

```
from cement.core import hook

def my_magical_hook(app):
    # do something magical
    print('Something Magical is Happening!')

def load(app):
    hook.register('post_setup', my_magical_hook)
```

And with the plugin enabled, we get this when we run the same app defined above:

```
$ python myapp.py
Something Magical is Happening!
```

The primary detail is that Cement calls the *load()* function of a plugin... after that, you can do anything you like.

Single File Plugins vs. Plugin Directories

As of Cement 2.9.x, plugins can be either a single file (i.e `myplugin.py`) or a python module directory (i.e. `myplugin/__init__.py`). Both will be loaded and executed the exact same way.

One caveat however, is that the submodules referenced from within a plugin directory must be relative path. For example:

myplugin/__init__.py

```
from .controllers import MyPluginController

def load(app):
    app.handler.register(MyPluginController)
```

myplugin/controllers.py

```
from cement.core.controller import CementBaseController, expose

class MyPluginController(CementBaseController):
    class Meta:
        label = 'myplugin'
        stacked_on = 'base'
        stacked_type = 'embedded'

    @expose()
    def my_command(self):
        print('Inside MyPluginController.my_command()')
```

Loading Templates From Plugin Directories

A common use case for complex applications is to use an output handler the uses templates, such as Mustache, Genshi, Jinja2, etc. In order for a plugin to use it's own template files it's templates directory first needs to be added to the list of template directories to be parsed. In the future, this will be more streamlined however currently the following is the recommeded way:

myplugin/__init__.py

```
def add_template_dir(app):
    path = os.path.join(os.path.basename(self.__file__, 'templates')
    app.add_template_dir(path)

def load(app):
    app.hook.register('post_setup', add_template_dir)
```

The above will append the directory `/path/to/myplugin/templates` to the list of template directories that the applications output handler with search for template files.

Framework and Application Hooks

Hooks allow the developers to tie into different pieces of the application. A hook can be defined anywhere, be it internally in the application, or in a plugin. Once a hook is defined, functions can be registered to that hook so that when the hook is called, all functions registered to that hook will be run. By defining a hook, you are saying that you are going to honor that hook somewhere in your application. Using descriptive hook names are good for clarity. For example, `pre_database_connect` is obviously a hook that will be run before a database connection is attempted.

The most important thing to remember when defining hooks for your application is to properly document them. Include whether anything is expected in return or what, if any, arguments will be passed to the hook functions when called.

API Reference:

- *Cement Hook Module*

Defining a Hook

A hook can be defined anywhere, however it is generally recommended to define the hook as early as possible. A hook definition simply gives a label to the hook, and allows the developer (or third-party plugin developers) to register functions to that hook. It's label is arbitrary.

The most convenient way to define a hook is via `CementApp.Meta.define_hooks`:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        define_hooks = ['my_example_hook']
```

Alternatively, if you need more control you might do it in `CementApp.setup()`:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'

    def setup(self):
        # always run core setup first
        super(MyApp, self).setup()

        # define application hooks here
        self.hook.define('my_example_hook')
```

Registering Functions to a Hook

A hook is just an identifier, but the functions registered to that hook are what get run when the hook is called. Registering a hook function should also be done early on in the bootstrap process, any time after the application has been created, after the hook is defined, and before the hook is run. Note that every hook is different, and therefore should be clearly documented by the 'owner' of the hook (application developer, plugin developer, etc).

The most convenient way to register a hook function is with `CementApp.Meta.hooks`:

```
from cement.core.foundation import CementApp

def my_hook1(app):
    pass

def my_hook2(app):
    pass

class MyApp(CementApp):
    class Meta:
        hooks = [
            ('post_argument_parsing', my_hook1),
            ('pre_close', my_hook2),
            ]

with MyApp() as app:
    app.run()
```

Where `CementApp.Meta.hooks` is a list of tuples that define the hook label, and the function to register to that hook.

Alternatively, if you need more control you might use:

```
from cement.core.foundation import CementApp

def my_hook1(app):
```

```
    pass

with CementApp('myapp') as app:
    app.hook.register('post_argument_parsing', my_hook1)
    app.run()
```

Or, for a third-party plugin:

```
def my_hook1(app):
    pass

def load(app):
    app.hook.register('post_argument_parsing', my_hook1)
```

What you return depends on what the developer defining the hook is expecting. Each hook is different, and the nature of the hook determines whether you need to return anything or not. That is up to the developer. Also, the `args` and `kwargs` coming in depend on the developer. You have to be familiar with the purpose of the defined hook in order to know whether you are receiving any `args` or `kwargs`.

Running a hook

Now that a hook is defined, and functions have been registered to that hook all that is left is to run it. Keep in mind, you don't want to run a hook until after the application load process... meaning, after all plugins and other code are loaded. If you receive an error that the hook doesn't exist, then you are trying to register a hook too soon before the hook is defined. Likewise, if it doesn't seem like your hook is running and you don't see it mentioned in `--debug` output, you might be registering your hook **after** the hook has already run.

That said, this is how you run a hook:

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    for res in app.hook.run('my_example_hook'):
        # do something with res?
        pass
```

As you can see we iterate over the hook, rather than just calling `app.hook.run()` by itself. This is necessary because `app.hook.run()` yields the results from each hook function as they are run. Hooks can be run anywhere **after** the hook is defined, and hooks are registered to that hook.

Controlling Hook Run Order

Sometimes you might have a very specific purpose in mind for a hook, and need it to run before or after other functions in the same hook. For that reason there is an optional `weight` parameter that can be passed when registering a hook function.

The following is an example application that defines, registers, and runs a custom application hook:

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'

    def setup(self):
```

```
        # always run core setup
        super(MyApp, self).setup()

        # define hooks in setup
        self.hook.define('my_hook')


# the following are the function that will run when ``my_hook`` is called
def func1(app):
    print 'Inside hook func1'

def func2(app):
    print 'Inside hook func2'

def func3(app):
    print 'Inside hook func3'


with MyApp() as app:
    # register all hook functions *after* the hook is defined (setup) but
    # also *before* the hook is called (different for every hook)
    app.hook.register('my_hook', func1, weight=0)
    app.hook.register('my_hook', func2, weight=100)
    app.hook.register('my_hook', func3, weight=-99)

    # run the application
    app.run()

    # run our custom hook
    for res in self.hook.run('my_hook', app):
        pass
```

And the result is:

```
$ python my_hook_test.py
Inside hook func3
Inside hook func1
Inside hook func2
```

As you can see, it doesn't matter what order we register the hook, the weight runs then in order from lowest to highest.

Cement Framework Hooks

Cement has a number of hooks that tie into the framework.

pre_setup

Run first when `CementApp.setup()` is called. The application object is passed as an argument. Nothing is expected in return.

post_setup

Run last when CementApp.setup() is called. The application object is passed as an argument. Nothing is expected in return.

pre_run

Run first when CementApp.run() is called. The application object is passed as an argument. Nothing is expected in return.

post_run

Run last when CementApp.run() is called. The application object is passed as an argument. Nothing is expected in return.

pre_argument_parsing

Run after CementApp.run() is called, just *before* argument parsing happens. The application object is passed as an argument to these hook functions. Nothing is expected in return.

post_argument_parsing

Run after CementApp.run() is called, just *after* argument parsing happens. The application object is passed as an argument to these hook functions. Nothing is expected in return.

This hook is generally useful where the developer needs to perform actions based on the arguments that were passed at command line, but before the logic of *app.run()* happens.

pre_render

Run first when CementApp.render() is called. The application object, and data dictionary are passed as arguments. Must return either the original data dictionary, or a modified one.

Note: This does not affect anything that is 'printed' to console.

post_render

Run last when CementApp.render() is called. The application object, and rendered output text are passed as arguments. Must return either the original output text, or a modified version.

pre_close

Run first when app.close() is called. This hook should be used by plugins and extensions to do any 'cleanup' at the end of program execution. Nothing is expected in return.

post_close

Run last when app.close() is called. Most use cases need pre_close(), however this hook is available should one need to do anything after all other 'close' operations.

signal

Run when signal handling is enabled, and the defined signal handler callback is executed. This hook should be used by the application, plugins, and extensions to perform any actions when a specific signal is caught. Nothing is expected in return.

Extending CementApp

CementApp provides a convenient `extend` mechanism that allows plugins, extensions, or the app itself to add objects/functions to the global application object. For example, a plugin might extend the CementApp with an `api` member allowing developers to call `app.api.get(...)`. The application itself does not provide `app.api` however the plugin does. As plugins are often third party, it is not possible for the plugin developer to simply sub-class the CementApp and add the functionality because the CementApp is already instantiated by the time plugins are loaded.

Take the following for example:

myapp.py

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    app.run()
```

The above is a very simple Cement application, which obviously doesn't do much. That said, we can add a plugin that extends the application to add an API client object, for example, pretty easily. Note the following is an arbitrary and non-functional example using dRest:

/etc/myapp/plugins.d/api.conf

```
[api]
enable_plugin = true
endpoint = https://example.com/api/v1
user = john.doe
password = XXXXXXXXXXX
```

/var/lib/myapp/plugins/api.py

```
import drest
from cement.core import hook

def extend_api_object(app):
    # get api info from this plugins configuration
    endpoint = app.config.get('api', 'endpoint')
    user = app.config.get('api', 'user')
    password = app.config.get('api', 'password')

    # create an api object and authenticate
    my_api_client = drest.API(endpoint)
    my_api_client.auth(username, password)

    # extend the global app object with an ``api`` member
    app.extend('api', my_api_client)

def load(app):
    hook.register('pre_run', extend_api_object)
```

In the above plugin, we simply created a dRest API client within a `pre_run` hook and then extended the global `app` with it. The developer can now reference `app.api` anywhere that the global `app` object is accessible.

Our application code could now look like:

myapp.py

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    app.run()

    # use the api object that the plugin provides
    app.api.get(...)
```

Signal Handling

Python provides the Signal library allowing developers to catch Unix signals and set handlers for asynchronous events. For example, the 'SIGTERM' (Terminate) signal is received when issuing a 'kill' command for a given Unix process. Via the signal library, we can set a handler (function) callback that will be executed when that signal is received. Some signals however can not be handled/caught, such as the SIGKILL signal (kill -9). Please refer to the Signal library documentation for a full understanding of its use and capabilities.

A caveat when setting a signal handler is that only one handler can be defined for a given signal. Therefore, all handling must be done from a single callback function. This is a slight roadblock for applications built on Cement in that many pieces of the framework are broken out into independent extensions as well as applications that have 3rd party plugins. The trouble happens when the application, plugins, and framework extensions all need to perform some action when a signal is caught. This section outlines the recommended way of handling signals with Cement versus manually setting signal handlers that may.

Important Note

It is important to note that it is not necessary to use the Cement mechanisms for signal handling, what-so-ever. That said, the primary concern of the framework is that app.close() is called no matter what the situation. Therefore, if you decide to disable signal handling all together you *must* ensure that you at the very least catch signal.SIGTERM and signal.SIGINT with the ability to call app.close(). You will likely find that it is more complex than you might think. The reason we put these mechanisms in place is primarily that we found it was the best way to a) handle a signal, and b) have access to our 'app' object in order to be able to call 'app.close()' when a process is terminated.

Signals Caught by Default

By default Cement catches the signals SIGTERM and SIGINT. When these signals are caught, Cement raises the exception 'CaughtSignal(signum, frame)' where 'signum' and 'frame' are the parameters passed to the signal handler. By raising an exception, we are able to pass runtime back to our applications main process (within a try/except block) and maintain the ability to access our 'application' object without using global objects.

A basic application using default handling might look like:

```
import signal
from cement.core.foundation import CementApp
from cement.core.exc import CaughtSignal

with CementApp('myapp') as app:
    try:
        app.run()
    except CaughtSignal as e:
```

```
        # do something with e.signum or e.frame (passed from signal)
        if e.signum == signal.SIGTERM:
            print("Caught SIGTERM...")
        elif e.signum == signal.SIGINT:
            print("Caught SIGINT...")
```

The above provides a very simple means of handling the most common signals, which in turns allowes our application to "exit clean" by running `app.close()` and any `pre_close` or `post_close` hooks. If we don't catch the signals, then the exceptions will be unhandled and the application will not exit clean.

Using The Signal Hook

An alternative way of adding multiple callbacks to a signal handler is by using the Cement signal hook. This hook is called anytime a handled signal is encountered.

```
import signal
from cement.core.foundation import CementApp
from cement.core.exc import CaughtSignal

def my_signal_handler(app, signum, frame):
    # do something with app?
    pass

    # or do someting with signum or frame
    if signum == signal.SIGTERM:
        print("Caught SIGTERM...")
    elif signum == signal.SIGINT:
        print("Caught SIGINT...")

with CementApp('myapp') as app:
    hook.register('signal', my_signal_handler)

    try:
        app.run()
    except CaughtSignal as e:
        # do soemthing with e.signum, e.frame
        pass
```

The key thing to note here is that the main application itself can easily handle the `CaughtSignal` exception without using hooks, however using the `signal` hook is useful for plugins and extensions to be able to tie into the signal handling outside of the main application. Both serve the same purpose.

Regardless of how signals are handled, all extensions or plugins should use the `pre_close` hook for cleanup purposes as much as possible as it is always run when `app.close()` is called.

Configuring Which Signals To Catch

You can define other signals to catch by passing a list of 'catch_signals' to foundation.CementApp():

```
import signal
from cement.core.foundation import CementApp

SIGNALS = [signal.SIGTERM, signal.SIGINT, signal.SIGHUP]

CementApp('myapp', catch_signals=SIGNALS)
...
```

What happens is, Cement iterates over the catch_signals list and adds a generic handler function (the same) for each signal. Because the handler calls the cement 'signal' hook, and then raises an exception which both pass the 'signum' and 'frame' parameters, you are able to handle the logic elsewhere rather than assigning a unique callback function for every signal.

What If I Don't Like Your Signal Handler Callback?

If you want more control over what happens when a signal is caught, you are more than welcome to override the default signal handler callback. That said, please be kind and be sure to atleast run the cement `signal` hook within your callback.

The following is an example taken from the builtin callback handler. Note that there is a bit of hackery in how we are acquiring the `CementApp` from the frame. This is because the signal is picked up outside of our control so we need to find it.

```
import signal
from cement.core.foundation import CementApp

def cement_signal_handler(signum, frame):
    """
    Catch a signal, run the ``signal`` hook, and then raise an exception
    allowing the app to handle logic elsewhere.

    :param signum: The signal number
    :param frame: The signal frame.
    :raises: cement.core.exc.CaughtSignal

    """
    LOG.debug('Caught signal %s' % signum)

    # hackish, but we do not have direct access to the CementApp object
    for f_global in frame.f_globals.values():
        if isinstance(f_global, CementApp):
            app = f_global
            for res in app.hook.run('signal', app, signum, frame):
                pass
    raise exc.CaughtSignal(signum, frame)


with CementApp('myapp') as app:
    try:
        app.run()
    except CaughtSignal as e:
        # do something with e.signum, or e.frame
        pass
```

This Is Stupid, and UnPythonic - How Do I Disable It?

To each their own. If you simply do not want any kind of signal handling performed, just set `catch_signals=None`.

```
from cement.core.foundation import foundation

CementApp('myapp', catch_signals=None)
```

Application Cleanup

The concept of 'cleanup' after application run time is nothing new. What happens during 'cleanup' all depends on the application. This might mean closing and deleting temporary files, removing session data, or deleting a PID (Process ID) file.

To allow for application cleanup not only within your program, but also external plugins and extensions, there is the `app.close()` function that must be called after `app.run()` regardless of any exceptions or runtime errors.

For example:

```
from cement.core.foundation import CementApp

app = CementApp('helloworld')
app.setup()
app.run()
app.close()
```

Calling `app.close()` ensures that the `pre_close` and `post_close` framework hooks are run, allowing extensions/plugins/etc to cleanup after the program runs.

Note that when using the Python `with` operator, the `setup()` and `close()` methods are automatically called. For example, the following is exactly the same as the above example:

```
from cement.core.foundation import CementApp

with CementApp('helloworld') as app:
    app.run()
```

Exit Status and Error Codes

You can optionally set the status code that your application exists with via the meta options `cement.foundation.CementApp.Meta.exit_on_close`.

```
app = CementApp('helloworld', exit_on_close=True)
app.setup()
app.run()
app.close(27)
```

Or Alternatively:

```
class MyApp(CementApp):
    class Meta:
        label = 'helloworld'
        exit_on_close = True

with MyApp() as app:
    app.run()
    app.exit_code = 123
```

Note the use of the `exit_on_close` meta option. Cement **will not** call `sys.exit()` unless `CementApp.Meta.exit_on_close == True`. You will find that calling `sys.exit()` in testing is very problematic, therefore you will likely want to enable `exit_on_close` in production, but not for testing as in this example:

```
class MyApp(CementApp):
    class Meta:
        label = 'helloworld'
```

```
        exit_on_close = True

class MyAppForTesting(MyApp):
    class Meta:
        exit_on_close = False

# ...
```

Also note that the default exit code is `0`, however any uncaught exceptions will cause the application to exit with a code of `1` (error).

Running Cleanup Code

Any extension, or plugin, or even the application itself that has 'cleanup' code should do so within the `pre_close` or `post_close` hooks to ensure that it gets run. For example:

```
from cement.core import hook

def my_cleanup(app):
    # do something when app.close() is called
    pass

hook.register('pre_close', my_cleanup)
```

Caching

Cement defines a cache interface called *ICache*, but does not implement caching by default. The documentation below references usage based on the interface and not the full capabilities of any given implementation.

The following cache handlers are included and maintained with Cement:

- *MemcachedCacheHandler*

Please reference the *ICache* interface documentation for writing your own cache handler.

General Usage

For this example we use the Memcached extension, which requires the `pylibmc` library to be installed, as well as a Memcached server running on localhost.

Example:

/path/to/myapp.conf

```
[myapp]
extensions = memcached

[cache.memcached]
# comma separated list of hosts to use
hosts = 127.0.0.1

# time in milliseconds
expire_time = 300
```

myapp.py

```
from cement.core.foundation import CementApp

with CementApp('myapp') as app:
    # run the application
    app.run()

    # set a cached value
    app.cache.set('my_key', 'my value')

    # get a cached value
    app.cache.get('my_key')

    # delete a cached value
    app.cache.delete('my_key')

    # delete the entire cache
    app.cache.purge()
```

Sending Email Messages

Cement defines a mail interface called *IMail*, as well as the default *DummyMailHandler* that implements the interface.

Please note that there are other handlers that implement the `IMail` interface. The documentation below only references usage based on the interface and not the full capabilities of the implementation.

The following mail handlers are included and maintained with Cement:

- *DummyMailHandler* (default)
- *SMTPMailHandler*

Please reference the *IMail* interface documentation for writing your own mail handler.

Example Usage

```
from cement.core.foundation import CementApp

class MyApp(CementApp):
    class Meta:
        label = 'myapp'

with MyApp() as app:
    app.run()

    # send an email message
    app.mail.send('This is my message',
        subject='This is my subject',
        to=['you@example.com'],
        from_addr='me@example.com',
        cc=['him@example.com', 'her@example.com'],
        bcc=['boss@example.com']
        )
```

Note that the default mail handler simply prints messages to the screen, and does not actually send anything. You can override this pretty easily without changing any code by using the built-in *SMTPMailHandler*. Simply modify the application configuration to something like:

myapp.conf

```
[myapp]
mail_handler = smtp

[mail.smtp]
ssl = 1
tls = 1
auth = 1
username = john.doe
password = oober_secure_password
```

Unit Testing Your Application

Testing is an incredibly important part of the application development process. The Cement framework provides some simple helpers and shortcuts for executing basic tests of your application. Please note that *cement.utils.test* *does* require the 'nose' package. That said, using *cement.utils.test* is not required to test a Cement based application. It is merely here for convenience, and is used by Cement when performing its own Nose tests.

For more information on testing, please see the following:

- UnitTest
- Nose
- Coverage

API Reference:

- *Cement Testing Utility*

An Example Test Case

The following outlines a basic test case using the cement.utils.test module.

```
from cement.utils import test
from myapp.cli.main import MyApp

class MyTestCase(test.CementTestCase):
    app_class = MyApp

    def setUp(self):
        super(MyTestCase, self).setUp()

        # Create a default application for the test functions to use.
        # Note that some tests may require you to perform this in the
        # test function in order to alter functionality.  That's perfectly
        # fine, this is only here for convenience.
        self.app = MyApp(argv=[], config_files=[])

    def test_myapp(self):
        with self.app as app:

            # Perform basic assertion checks.  You can do this anywhere
            # in the test function, depending on what the assertion is
            # checking.
            self.ok(app.config.has_key('myapp', 'debug'))
            self.eq(app.config.get('myapp', 'debug'), False)
```

```
            # Run the applicaion, if necessary
            app.run()

            # Test the last rendered output (if app.render was used)
            data, output = app.get_last_rendered()
            self.eq(data, {'foo':'bar'})
            self.eq(output, 'some rendered output text')

    @test.raises(Exception)
    def test_exception(self):
        try:
            # Perform tests that intentionally cause an exception.  The
            # test passes only if the exception is raised.
            raise Exception('test')
        except Exception as e:
            # Do further checks to ensure the proper exception was raised
            self.eq(e.args[0], 'Some Exception Message')

            # Finally, call raise again which re-raises the exception that
            # we just caught.  This completes our test (to actually
            # verify that the exception was raised)
            raise
```

Cement Testing Caveats

In general, testing Cement applications should be no different than testing anything else in Python. That said, the following are some things to keep in mind.

Command Line Arguments

Never rely on `sys.argv` for command line arguments. The `CementApp()` class accepts the `argv` keyword argument allowing you to pass the arguments that you would like to test for. Using `sys.argv` will cause issues with the calling script (i.e. `nosetests`, etc) and other issues. Always pass `argv` to `CementApp()` in tests.

Config Files

It is recommended to always set your apps `config_files` setting to an empty list, or to something relative to your current working directory. Using default config files settings while testing will introduce unexpected results. For example, if a `~/myapp.conf` user configuration exists it can alter the runtime of your application in a way that might cause tests to fail.

Making Things Easy

The easiest way to accomplish the above is by sub-classing your `CementApp` into a special 'testing' version. For example:

```
from cement.utils import test
from myapp.cli.main import MyApp

class MyTestApp(MyApp):
    class Meta:
        argv = []
        config_files = []

class MyTestCase(test.CementTestCase):
    app_class = MyTestApp
```

```
    def test_myapp_default(self):
        with self.app as app:
            app.run()

    def test_myapp_foo(self):
        with MyTestApp(argv=['--foo', 'bar']) as app:
            app.run()
```

Application Design

Cement does not enforce any form of application layout, or design. That said, there are a number of best practices that can help newcomers get settled into using Cement as a foundation to build their application.

Single File Scripts

Cement can easily be used for quick applications and scripts that are based out of a single file. The following is a minimal example that creates a `CementApp` with several sub-commands:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose


class BaseController(CementBaseController):
    class Meta:
        label = 'base'
        description = "MyApp Does Amazing Things"
        arguments = [
            (['-f', '--foo'], dict(help='notorious foo option')),
            (['-b', '--bar'], dict(help='infamous bar option')),
            ]

    @expose(hide=True)
    def default(self):
        print("Inside MyAppBaseController.default()")

    @expose(help="this is some help text about the cmd1")
    def cmd1(self):
        print("Inside BaseController.cmd1()")

    @expose(help="this is some help text about the cmd2")
    def cmd2(self):
        print("Inside BaseController.cmd2()")

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = BaseController


def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

In this example, we've defined a base controller to handler the heavy lifting of what this script does, while providing sub-commands to handler different tasks. We've also included a number of command line arguments/options that can be used to alter how the script operates, and to allow user input.

Notice that we have defined a `main()` function, and then beyond that where we call `main()` if `__name__` is `__main__`. This essentially says, if the script was called directly (not imported by another Python library) then execute the `main()` function.

Multi-File Applications

Larger applications need to be properly organized to keep code clean, and to keep a high level of maintainability (read: to keep things from getting shitty). The Boss Project provides our recommended application layout, and is a great starting point for anyone new to Cement.

The primary detail about how to layout your code is this: All CLI/Cement related code should live separate from the "core logic" of your application. Most likely, you will have some code that is re-usable by other people and you do not want to mix this with your Cement code, because that will rely on Cement being loaded to function properly (like it is when called from command line).

For this reason, we recommend a structure similar to the following:

```
- myapp/
- myapp/cli
- myapp/core
```

All code related to your CLI, which relies on Cement, should live in `myapp/cli/`, and all code that is the "core logic" of your application should live in a module like `myapp/core`. The idea being that, should anyone wish to re-use your library, they should not be required to run your CLI application to do so. You want people to be able to do the following:

```
from yourapp.core.some_library import SomeClass
```

The `SomeClass` should not rely on `CementApp` (i.e. the `app` object). In this case, the code under `myapp/cli/` would import from `myapp/core/` and add the "CLI" stuff on top of it.

In short, the CLI code should handle interaction with the user via the shell, and the core code should handle application logic un-reliant on the CLI being loaded.

See the *Starting Projects from Boss Templates* section for more info on using Boss.

Handling High Level Exceptions

The following expands on the above to give an example of how you might handle exceptions at the highest level (wrapped around the app object). It is very well known that exception handling should happen as close to the source of the exception as possible, and you should do that. However at the top level (generally in your `main.py` or similar) you want to handle certain exceptions (such as argument errors, or user interaction related errors) so that they are presented nicely to the user. End-users don't like stack traces!

The below example catches common framework exceptions that Cement might throw, but you could also catch your own application specific exception this way:

```
import sys

from cement.core.foundation import CementApp
from cement.core.exc import FrameworkError, CaughtSignal
```

```
def main():
    with CementApp('myapp') as app:
        try:
            app.run()

        except CaughtSignal as e:
            # determine what the signal is, and do something with it?
            from signal import SIGINT, SIGABRT

            if e.signum == SIGINT:
                # do something... maybe change the exit code?
                app.exit_code = 110
            elif e.signum == SIGABRT:
                # do something else...
                app.exit_code = 111

        except FrameworkError as e:
            # do something when a framework error happens
            print("FrameworkError => %s" % e)

            # and maybe set the exit code to something unique as well
            app.exit_code = 300

        finally:
            # Maybe we want to see a full-stack trace for the above
            # exceptions, but only if --debug was passed?
            if app.debug:
                import traceback
                traceback.print_exc()

if __name__ == '__main__':
    main()
```

Starting Projects from Boss Templates

The Boss Project provides 'Baseline Open Source Software' templates and development tools. It has similarities to PasteScript with regards to templating, but far easier to extend. The official template repository includes a number of templates specifically for Cement, and are the recommended means of start Cement based projects.

This is just a quick overview of creating Cement Apps, Plugins, and Extensions with Boss.

Creating a Cement App

```
$ boss create ./myapp -t boss:cement-app

$ cd myapp

$ virtualenv /path/to/myapp/env

$ source /path/to/myapp/env/bin/activate

$ pip install -r requirements.txt

$ python setup.py develop
```

```
$ myapp --help

$ pip install nose coverage

$ python setup.py nosetests
```

Creating a Cement Plugin

```
$ source /path/to/myapp/env/bin/activate

$ cd /path/to/myapp

$ mkdir plugins

$ boss create ./plugins/myplugin -t boss:cement-plugin
```

Add the following to ~/myapp.conf (or whereever your config file is):

```
[myapp]
plugin_config_dir = /path/to/myapp/config/plugins.d
plugin_dir = /path/to/myapp/plugins

# Enable the plugin here, or in a plugins.d/myplugin.conf
[myplugin]
enable_plugin = 1
```

And it should be enabled when you run your app (though it doesn't do anything out of the box).

Creating a Cement Extension

3rd party extensions are generally created within the app they are being built with, but do not have to be. In this case we are adding the extension to an existing Cement project:

```
$ boss create ./myapp -t boss:cement-ext
```

At this point you would enable the extension in your app to utilize it.

Contributing

Cement is an open-source project, and is open to any and all contributions that other developers would like to provide. This document provides some guidelines that all contributors must be aware of, and abide by to have their submissions included in the source.

Licensing

The Cement source code is licensed under the BSD three-clause license and is approved by the Open Source Initiative. All contributed source code must be either the original work of the contributing author, which will be contributed under the BSD license, or work taken from another project that is released under a BSD-compatible license.

Submitting Bug Reports and Feature Requests

If you've found a bug, or would like to request a feature please create a detailed issue for it at http://github.com/datafolklabs/cement/issues.

The ideal bug report would include:

- Bug description
- Include the version of Python, Cement, and any dependencies in use
- Steps to reproduce the bug
- Code samples that show the bug in action
- A pull request including code that a) fixes the bug, and b) atleast one test case that tests for the bug specifically

The ideal feature request would include:

- Feature description
- Example code, or pseudo code of how you might use the feature
- Example command line session showing how the feature would be used by the end-user
- **A pull request including:**
  - The feature you would like added
  - At least one test case that tests the feature and maintains 100% code coverage when tests are run (meaning that your tests should cover 100% of your contributed code)
  - Documentation that outlines how to use the feature

Guidelines for Code Contributions

All contributors should attempt to abide by the following:

- Contributors fork the project on GitHub onto their own account
- All changes should be commited, and pushed to their repository
- All pull requests are from a topic branch, not an existing Cement branch
- Contributors make every effort to comply with PEP8
- **Before starting on a new feature, or bug fix, always do the following:**
  - `git pull --rebase` to get latest changes from upstream
  - **Checkout a new branch. For example:**
    - `git checkout -b feature/<feature_name>`
    - `git checkout -b bug/<bug_number>`
- **Code must include the following:**
  - All tests pass successfully
  - Coverage reports 100% code coverage when running tests
  - New features are documented in the appropriate section of the doc
  - Significant changes are mentioned in the ChangeLog
- All contributions must be associated with at least one issue in GitHub. If the issue does not exist, create one (per the guidelines above).

- **Commit comments must include something like the following:**
  - Resolves Issue #1127
  - Partially Resolves Issue #9873
- A single commit per issue.
- Contributors should add their full name, or handle, to the CONTRIBUTORS file.

Regarding git commit messages, please read the following:

- Commit Guidelines

The majority of commits only require a single line commit message. That said, for more complex commits, please use the following as an example (as outlined in the ProGit link above):

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded by a
   single space, with blank lines in between, but conventions vary here
```

Source Code and Versioning

One of the primary goals of Cement is stability in the source code. For this reason we maintain a number of different git branches for focused development.

Development Branches

Active 'forward' development happens out of two branches:

- master - Development for the next minor stable release.
- portland - Development for the next major release.

Additionally, specific development branches might exist in the future for larger releases that may require iterative 'release candidate' handling before an official stable release. These branches will have the format of:

- dev/3.1.x
- dev/3.3.x
- dev/4.1.x
- dev/4.3.x
- etc

Stable Branches

- stable/0.8.x
- stable/1.0.x

- stable/1.2.x
- stable/2.0.x
- stable/2.2.x
- stable/3.0.x
- stable/3.2.x
- etc

There is a system for versioning that may seem complex, and needs some explanation. Version numbers are broken up into three parts:

- <Major>.<Minor>.<Bugfix>

This means:

- Major - The major version of the source code generally relates to extensive incompatible changes, or entire code base rewrites. Applications built on the '1.x.x' version of Cement will need to be completely rewritten for the '2.x.x' versions of Cement.
- Minor - The minor version signifies the addition of new features. It may also indicate minor incompatibilities with the previous stable version, but should be easily resolvable with minimal coding effort.
- Bugfix - During the lifecycle of a stable release such as '2.2.x', the only updates should be bug and/or security related. At times, minor features may be introduced during a 'bugfix' release but that should not happen often.

It should be noted that both the Minor, and Bugfix versions follow a `even == stable`, and `odd == development` scheme. Therefore, the current version in git will always end in an 'odd number'. For example, if the current stable version is `2.0.18`, then the version in `stable/2.0.x` would be `2.0.19`. That said, the `master` branch might then be `2.1.1` which is the first version of the next minor release. Bugfixes would get applied to both branches, however feature updates would only be applied to `master`. The next stable release would then be `2.2.0` and a new git branch of `stable/2.2.x` will be created.

The `portland` branch is always very forward looking, and will contain significant (and likely broken) code changes. It should never be used for anything other than development and testing.

Examples

The following are examples that demonstrate how to handle certain situations with Cement.

Adding a Version Option to Your App

Almost every app out there supports a `--version` option of some sort that provides the end user with version information. This is pretty important to include, so we've added an example below.

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController

VERSION = '0.9.1'

BANNER = """
My Awesome Application v%s
Copyright (c) 2014 John Doe Enterprises
""" % VERSION

class MyBaseController(CementBaseController):
```

```
    class Meta:
        label = 'base'
        description = 'MyApp Does Amazing Things'
        arguments = [
            (['-v', '--version'], dict(action='version', version=BANNER)),
            ]

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = MyBaseController


with MyApp() as app:
    app.run()
```

This looks like:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

MyApp Does Amazing Things

optional arguments:
  -h, --help     show this help message and exit
  --debug        toggle debug output
  --quiet        suppress all output
  -v, --version  show program's version number and exit

$ python myapp.py --version
My Awesome Application v0.9.1
Copyright (c) 2014 John Doe Enterprises
```

Multiple Stacked Controllers

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

# define application controllers
class MyAppBaseController(CementBaseController):
    class Meta:
        label = 'base'
        description = "my application does amazing things"
        arguments = [
            (['--base-opt'], dict(help="option under base controller")),
            ]

    @expose(help="base controller default command", hide=True)
    def default(self):
        print "Inside MyAppBaseController.default()"

    @expose(help="another base controller command")
    def command1(self):
        print "Inside MyAppBaseController.command1()"

class SecondController(CementBaseController):
```

```
    class Meta:
        label = 'second_controller'
        stacked_on = 'base'
        stacked_type = 'nested'
        description = "this is the second controller (stacked/nested on base)"
        arguments = [
            (['--2nd-opt'], dict(help="another option under base controller")),
            ]

    @expose(help="second-controller default command", hide=True)
    def default(self):
        print "Inside SecondController.default()"

    @expose(help="this is a command under the second-controller namespace")
    def command2(self):
        print "Inside SecondController.command2()"

class ThirdController(CementBaseController):
    class Meta:
        label = 'third_controller'
        stacked_on = 'second_controller'
        stacked_type = 'embedded'
        description = "this controller is embedded in the second-controller"
        arguments = [
            (['--3rd-opt'], dict(help="an option only under 3rd controller")),
            ]

    @expose(help="another command under the second-controller namespace")
    def command3(self):
        print "Inside ThirdController.command3()"

class FourthController(CementBaseController):
    class Meta:
        label = 'fourth_controller'
        stacked_on = 'second_controller'
        stacked_type = 'nested'
        description = "this controller is nested on the second-controller"
        arguments = [
            (['--4th-opt'], dict(help="an option only under 4th controller")),
            ]

    @expose(help="a command only under the fourth-controller namespace")
    def command4(self):
        print "Inside FourthController.command4()"

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        handlers = [
            MyAppBaseController,
            SecondController,
            ThirdController,
            FourthController,
            ]

def main():
    with MyApp() as app:
        app.run()
```

```
if __name__ == '__main__':
    main()
```

In the *base* controller output of *–help* notice that the *second-controller* is listed as a sub-command:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

my application does amazing things

commands:

  command1
    another base controller command

  second-controller
    this is the second controller (stacked/nested on base)

optional arguments:
  -h, --help           show this help message and exit
  --debug              toggle debug output
  --quiet              suppress all output
  --base-opt BASE_OPT  option under base controller


$ python myapp.py
Inside MyAppBaseController.default()


$ python myapp.py command1
Inside MyAppBaseController.command1()

$ python myapp.py second-controller
Inside SecondController.default()

$ python myapp.py second-controller --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

this is the second controller (stacked/nested on base)

commands:

  command2
    this is a command under the second-controller namespace

  command3
    another command under the second-controller namespace

  fourth-controller
    this controller is nested on the second-controller

optional arguments:
  -h, --help         show this help message and exit
  --debug            toggle debug output
  --quiet            suppress all output
  --2nd-opt 2ND_OPT  another option under base controller
```

```
  --3rd-opt 3RD_OPT  an option only under 3rd controller
```

Under the *second-controller* you can see the commands and options from the second and third controllers. In this example, the *second-controller* is *nested* on the base controller, and the *third-controller* is *embedded* on the *second-controller*. Finally, we see that the *fourth-controller* is also *nested* on the *second-controller* creating a sub-sub-command.

```
$ python myapp.py second-controller command3
Inside ThirdController.command3()


$ python myapp.py second-controller fourth-controller --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

this controller is nested on the second-controller

commands:

  command4
    a command only under the fourth-controller namespace

optional arguments:
  -h, --help         show this help message and exit
  --debug            toggle debug output
  --quiet            suppress all output
  --4th-opt 4TH_OPT  an option only under 3rd controller


$ python myapp.py second-controller fourth-controller command4
Inside FourthController.command4()
```

Abstract Base Controllers for Shared Arguments and Commands

For larger, complex applications it is often very useful to have abstract base controllers that hold shared arguments and commands that a number of other controllers have in common. Note that in the example below, you can not override the Meta.arguments in a sub-class or you overwrite the shared arguments, but it is possible to *append* to them in order to maintain the defaults while having unique options/arguments for the sub-classed controller. As well, you can add any number of additional commands in the sub-class but still maintain the existing shared commands (or override them as necessary).

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose


class AbstractBaseController(CementBaseController):
    """
    This is an abstract base class that is useless on its own, but used
    by other classes to sub-class from and to share common commands and
    arguments.  This should not be confused with the `MyAppBaseController`
    used as the ``base_controller`` namespace.

    """
    class Meta:
        stacked_on = 'base'
        stacked_type = 'nested'
        arguments = [
```

```
            ( ['-f', '--foo'], dict(help='notorious foo option')),
            ]

    def _setup(self, base_app):
        super(AbstractBaseController, self)._setup(base_app)

        # add a common object that will be used in any sub-class
        self.reusable_dict = dict()

    @expose(hide=True)
    def default(self):
        """
        This command will be shared within all controllers that sub-class
        from here.  It can also be overridden in the sub-class, but for
        this example we are making it dynamic.

        """
        # do something with self.my_shared_obj here?
        if 'some_key' in self.reusable_dict.keys():
            pass

        # or do something with parsed args?
        if self.app.pargs.foo:
            print "Foo option was passed with value: %s" % self.app.pargs.foo

        # or maybe do something dynamically
        print("Inside %s.default()" % self.__class__.__name__)

class MyAppBaseController(CementBaseController):
    """
    This is the application base controller, but we don't want to use our
    abstract base class here.

    """
    class Meta:
        label = 'base'

    @expose(hide=True)
    def default(self):
        print("Inside MyAppBaseController.default()")

class Controller1(AbstractBaseController):
    """
    This controller sub-classes from the abstract base class as to inherite
    shared arguments, and commands.

    """
    class Meta:
        label = 'controller1'

    @expose()
    def command1(self):
        print("Inside Controller1.command1()")

class Controller2(AbstractBaseController):
    """
    This controller also sub-classes from the abstract base class as to
    inherite shared arguments, and commands.
```

```
    """
    class Meta:
        label = 'controller2'

    @expose()
    def command2(self):
        print("Inside Controller2.command2()")


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = 'base'
        handlers = [
            MyAppBaseController,
            Controller1,
            Controller2,
            ]

def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

And:

```
$ python myapp.py
Inside MyAppBaseController.default()

$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

Base Controller

commands:

  controller1
    Controller1 Controller

  controller2
    Controller2 Controller

optional arguments:
  -h, --help  show this help message and exit
  --debug     toggle debug output
  --quiet     suppress all output


$ python myapp.py controller1
Inside Controller1.default()

$ python myapp.py controller1 --foo=bar
Foo option was passed with value: bar
Inside Controller1.default()
```

```
$ python myapp.py controller2
Inside Controller2.default()
```

Multiple Controllers With Same Label

There are many ways to use controllers. In some circumstances you might find that you want to have two controllers with the same label but stacked on different parent controllers. This is a problem because controller labels must be unique.

Take for example the situation where you want to have a 'list' controller, rather than a 'list' function of another controller. You might call this as:

```
$ myapp <controller1> <list_controller>

$ myapp <controller2> <some_other_list_controller>
```

In both cases, you would probably want the 'sub-controller' or sub-command to be 'list'. This is possible with the use of the 'aliases' and 'aliases_only' Meta options. Take the following code as an example where we have a 'users' and a 'hosts' controller and we want to have a 'list' sub-command under both:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

# define application controllers
class MyAppBaseController(CementBaseController):
    class Meta:
        label = 'base'

class UsersController(CementBaseController):
    class Meta:
        label = 'users'
        description = "this is the users controller"
        stacked_on = 'base'
        stacked_type = 'nested'

class HostsController(CementBaseController):
    class Meta:
        label = 'hosts'
        description = "this is the hosts controller"
        stacked_on = 'base'
        stacked_type = 'nested'

class UsersListController(CementBaseController):
    class Meta:
        label = 'users_list'
        description = 'list all available users'
        aliases = ['list']
        aliases_only = True
        stacked_on = 'users'
        stacked_type = 'nested'

    @expose(hide=True)
    def default(self):
        print "Inside UsersListController.default()"

class HostsListController(CementBaseController):
```

```
    class Meta:
        label = 'hosts_list'
        description = 'list all available hosts'
        aliases = ['list']
        aliases_only = True
        stacked_on = 'hosts'
        stacked_type = 'nested'

    @expose(hide=True)
    def default(self):
        print "Inside HostsListController.default()"

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        handlers = [
            MyAppBaseController,
            UsersController,
            HostsController,
            UsersListController,
            HostsListController,
            ]

def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

Base Controller

commands:

  hosts
    this is the hosts controller

  users
    this is the users controller

optional arguments:
  -h, --help  show this help message and exit
  --debug     toggle debug output
  --quiet     suppress all output


$ python myapp.py users --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}

this is the users controller

commands:

  list
    list all available users
```

```
optional arguments:
  -h, --help  show this help message and exit
  --debug     toggle debug output
  --quiet     suppress all output


$ python myapp.py users list
Inside UsersListController.default()

$ python myapp.py hosts list
Inside HostsListController.default()
```

BASH Auto Completion

Auto Completion, or "TAB Completion" is a very common and familiar feature in BASH (and other modern shells). It is possible to auto-complete Cement apps (using BASH for this example) including sub-levels for nested controllers. The difficulty is that this auto-completion code must be maintained outside of Cement and your application code, and be implemented in the shell environment generally by use of an "RC" file, or similar means. This then must be updated anytime your application is modified (or atleast any time the sub-commands/controllers/arguments are modified).

Note that, in the future, we would love to include some form of "BASH RC Generator" that will do this for you, however in the meantime the following is a working example that can be used as a model for adding BASH auto-completion to your app.

Update: As of Cement 2.7.x, the *Argcomplete Framework Extension* can be used as an alternative to this example. Both are viable options though this route is much more manual, and the Argcomplete route might not fit your needs.

Example Cement App

The following application code implements three levels of namespaces, or sub-commands, that are implemented via nested-controllers.

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

class BaseController(CementBaseController):
    class Meta:
        label = 'base'

    @expose()
    def base_cmd1(self):
        print("Inside BaseController.base_cmd1()")

class EmbeddedController(CementBaseController):
    class Meta:
        label = 'embedded'
        description = "embedded with base namespace"
        stacked_on = 'base'
        stacked_type = 'embedded'

    @expose()
    def base_cmd2(self):
        print("Inside EmbeddedController.base_cmd2()")
```

```
    @expose()
    def embedded_cmd3(self):
        print("Inside EmbeddedController.embedded_cmd3()")

class SecondLevelController(CementBaseController):
    class Meta:
        label = 'second'
        description = ''
        stacked_on = 'base'
        stacked_type = 'nested'

    @expose()
    def second_cmd4(self):
        print("Inside SecondLevelController.second_cmd4()")

    @expose()
    def second_cmd5(self):
        print("Inside SecondLevelController.second_cmd5()")

class ThirdLevelController(CementBaseController):
    class Meta:
        label = 'third'
        description = ''
        stacked_on = 'second'
        stacked_type = 'nested'

    @expose()
    def third_cmd6(self):
        print("Inside ThirdLevelController.third_cmd6()")

    @expose()
    def third_cmd7(self):
        print("Inside ThirdLevelController.third_cmd7()")


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        handlers = [
            BaseController,
            EmbeddedController,
            SecondLevelController,
            ThirdLevelController,
            ]


def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

This looks like:

```
$ python myapp.py --help
usage: myapp.py (sub-commands ...) [options ...] {arguments ...}
```

```
Base Controller

commands:

  base-cmd1

  base-cmd2

  embedded-cmd3

  second


$ python myapp.py second --help

commands:

  second-cmd4

  second-cmd5

  third


$ python myapp.py second third --help

commands:

  third-cmd6

  third-cmd7
```

For demonstration purposes, we are going to create a BASH alias here so that we can call our *myapp* command name as if we would in production (not development):

```
$ alias myapp="python ./myapp.py"
```

In the "real world" your actual *myapp* command would be setup/installed by something like this in *setup.py*:

```
entry_points="""
    [console_scripts]
    myapp = myapp.cli.main:main
    """,
```

Or by simply copying *myapp.py* to */usr/bin/myapp*, or similar.

Example BASH RC

The following is a BASH RC script that will setup auto-completiong for the above Cement App *myapp*. You **will** need to modify this, it is just an example and is not intended to be copy and pasted:

```
alias myapp="python ./myapp.py"

_myapp_complete()
{
    local cur prev BASE_LEVEL
```

```
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    # SETUP THE BASE LEVEL (everything after "myapp")
    if [ $COMP_CWORD -eq 1 ]; then
        COMPREPLY=( $(compgen \
                        -W "base-cmd1 base-cmd2 embedded-cmd3 second" \
                        -- $cur) )


    # SETUP THE SECOND LEVEL (EVERYTHING AFTER "myapp second")
    elif [ $COMP_CWORD -eq 2 ]; then
        case "$prev" in

            # HANDLE EVERYTHING AFTER THE SECOND LEVEL NAMESPACE
            "second")
                COMPREPLY=( $(compgen \
                              -W "second-cmd4 second-cmd5 third" \
                              -- $cur) )
                ;;

            # IF YOU HAD ANOTHER CONTROLLER, YOU'D HANDLE THAT HERE
            "some-other-controller")
                COMPREPLY=( $(compgen \
                              -W "some-other-sub-command" \
                              -- $cur) )
                ;;

            # EVERYTHING ELSE
            *)
                ;;
        esac

    # SETUP THE THIRD LEVEL (EVERYTHING AFTER "myapp second third")
    elif [ $COMP_CWORD -eq 3 ]; then
        case "$prev" in
            # HANDLE EVERYTHING AFTER THE THIRD LEVEL NAMESPACE
            "third")
                COMPREPLY=( $(compgen \
                              -W "third-cmd6 third-cmd7" \
                              -- $cur) )
                ;;

            # IF YOU HAD ANOTHER CONTROLLER, YOU'D HANDLE THAT HERE
            "some-other-controller")
                COMPREPLY=( $(compgen \
                              -W "some-other-sub-command" \
                              -- $cur) )
                ;;

            *)
                ;;
        esac
    fi

    return 0
```

```
} &&
complete -F _myapp_complete myapp
```

You would then "source" the RC file:

```
$ source myapp.rc
```

In the "real world" you would probably put this in a system wide location such at `/etc/profile.d` or similar (in a production deployment).

Finally, this is what it looks like:

```
# show all sub-commands at the base level
$ myapp [tab] [tab]
base-cmd1       base-cmd2       embedded-cmd3       second

# auto-complete a partial matching sub-command
$ myapp base [tab]

$ myapp base-cmd [tab] [tab]
base-cmd1       base-cmd2

# auto-complete a full matching sub-command
$ myapp sec [tab]

$ myapp second

# show all sub-commands under the second namespace
$ myapp second [tab] [tab]
second-cmd4       second-cmd5       third

# show all sub-commands under the third namespace
$ myapp second third [tab] [tab]
third-cmd6       third-cmd7
```

Handling Arbitrary Extra Positional Arguments

It is common practice to accept additional positional arguments at command line, rather than option flags. For example:

```
$ myapp some-command some-argument --foo=bar
```

In the above, `some-command` would be the function under whatever controller it is exposed from, and *some-argument* would be just an arbtrary argument. In most cases, the argument within the code is generic, but its uses vary. For example:

```
$ myapp create-user john.doe

$ myapp create-group admins
```

In the above, the sub-commands are `create-user` and `create-group`, and in this use case they are under the same controller. The `argument` however differs for each command, though it is passed to the app the same (the first positional argument, that is not a controller/command).

The following example outlines how you might handle arbitrary (or generic) positional arguments.

Example

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

class MyBaseController(CementBaseController):
    class Meta:
        label = 'base'

class MySecondController(CementBaseController):
    class Meta:
        label = 'second'
        stacked_type = 'nested'
        stacked_on = 'base'
        description = 'this is the second controller namespace'
        arguments = [
            (['-f', '--foo'],
             dict(help='the notorious foo option', action='store')),
            (['extra_arguments'],
             dict(action='store', nargs='*')),
        ]

    @expose()
    def cmd1(self):
        print "Inside MySecondController.cmd1()"

        if self.app.pargs.extra_arguments:
            print "Extra Argument 0: %s" % self.app.pargs.extra_arguments[0]
            print "Extra Argument 1: %s" % self.app.pargs.extra_arguments[1]


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = 'base'
        handlers = [
            MyBaseController,
            MySecondController,
            ]

def main():
    with MyApp() as app:
        app.run()

if __name__ == '__main__':
    main()
```

And this would look something like:

```
$ python argtest.py second cmd1 extra1 extra2
Inside MySecondController.cmd1()
Extra Argument 0: extra1
Extra Argument 1: extra2
```

Auto Reload When Configuration Files Change

Cement 2.5.2 added the **experimental** *Reload Config Extension*, allowing applications built on Cement to automatically reload `app.config` any time configuration files and/or plugin configuration files are modified. Note that the `ext_reload_config` extension requires `pyinotify` and is only supported on Linux.

Additionally, the extension adds the `pre_reload_config` and `post_reload_config` hooks allowing applications to respond to the event and perform actions when config changes are detected.

The following demonstrates how this can be accomplished:

```
from time import sleep
from cement.core.exc import CaughtSignal
from cement.core import hook
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose

def print_foo(app):
    print("Foo => %s" % app.config.get('myapp', 'foo'))


class Base(CementBaseController):
    class Meta:
        label = 'base'

    @expose(hide=True)
    def default(self):
        print('Inside Base.default()')

        # simulate a long running process
        while True:
            sleep(30)

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['reload_config']
        hooks = [('post_reload_config', print_foo)]
        handlers = [Base]


with MyApp() as app:
    try:
        app.run()
    except CaughtSignal as e:
        # maybe do something... but catch it regardless so app.close() is
        # called when exiting `with` cleanly.
        print(e)
```

The output looks something like:

```
$ python myapp.py --debug

Inside Base.default()

2015-05-12 19:05:34,372 (DEBUG) cement.ext.ext_reload_config : config path modified:␣
↪mask=IN_CLOSE_WRITE, path=/home/vagrant/.myapp.conf
2015-05-12 19:05:34,373 (DEBUG) cement.core.config : config file '/home/vagrant/.
↪myapp.conf' exists, loading settings...
```

```
2015-05-12 19:05:34,373 (DEBUG) cement.core.hook : running hook 'post_reload_config' (
↪<function print_foo at 0x7f6b4d401b70>) from __main__
Foo => bar1

2015-05-12 19:05:44,121 (DEBUG) cement.ext.ext_reload_config : config path modified:␣
↪mask=IN_CLOSE_WRITE, path=/home/vagrant/.myapp.conf
2015-05-12 19:05:44,122 (DEBUG) cement.core.config : config file '/home/vagrant/.
↪myapp.conf' exists, loading settings...
2015-05-12 19:05:44,122 (DEBUG) cement.core.hook : running hook 'post_reload_config' (
↪<function print_foo at 0x7f6b4d401b70>) from __main__
Foo => bar2
```

Tabularized Output

Users familiar with MySQL, PGSQL, etc find comfort in table-based output patterns. For one it adds structure, and two generally makes things much more readable. The folloiwing is an example of a simple app using the *Tabulate* extension:

```
from cement.core.foundation import CementApp
from cement.core.controller import expose, CementBaseController


class MyController(CementBaseController):
    class Meta:
        label = 'base'

    @expose(hide=True)
    def default(self):
        headers=['NAME', 'AGE', 'ADDRESS']
        data=[
            ["Krystin Bartoletti", 47, "PSC 7591, Box 425, APO AP 68379"],
            ["Cris Hegan", 54, "322 Reubin Islands, Leylabury, NC 34388"],
            ["George Champlin", 25, "Unit 6559, Box 124, DPO AA 25518"],
            ]
        self.app.render(data, headers=headers)


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        extensions = ['tabulate']
        output_handler = 'tabulate'
        handlers = [MyController]


with MyApp() as app:
    app.run()
```

The output looks like:

```
$ python myapp.py

| NAME               |   AGE | ADDRESS                                  |
|--------------------+-------+------------------------------------------|
| Krystin Bartoletti |    47 | PSC 7591, Box 425, APO AP 68379          |
| Cris Hegan         |    54 | 322 Reubin Islands, Leylabury, NC 34388  |
```

```
| George Champlin    |    25 | Unit 6559, Box 124, DPO AA 25518        |
```

Reload Application on SIGHUP

A common convention in the Linux world is to handle a `SIGHUP` signal, by reloading the current runtime within the same process ID (`pid`). The following example demonstrates how you might achieve that:

```
import signal
from time import sleep
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose
from cement.core.exc import CaughtSignal


class MyController(CementBaseController):
    class Meta:
        label = 'base'

    @expose(hide=True)
    def default(self):
        print('Inside MyController.default()')

        ### inner loop where the application logic happens
        while True:
            print('Inside Inner Loop')
            sleep(5)

class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        base_controller = MyController


with MyApp() as app:
    ### outer loop where signals are handles, and application reload
    ### happens

    keep_alive = True
    while keep_alive is True:
        try:
            app.run()
        except CaughtSignal as e:
            app.log.warning(e.msg)
            if e.signum in [signal.SIGHUP]:
                app.log.warning('Reloading MyApp')
                app.reload()
                keep_alive = True
            else:
                app.exit_code = 1
                keep_alive = False
```

Running the application shows the inner loop in action:

```
$ python myapp.py
Inside MyController.default()
Inside Inner Loop
```

```
Inside Inner Loop
Inside Inner Loop
Inside Inner Loop
Inside Inner Loop
Inside Inner Loop
Inside Inner Loop
```

If we grab the PID in another terminal, and send it a SIGHUP signal we can see it reload:

```
$ ps auxw | grep [m]yapp
derks            1317   0.0  0.1  2425792  12988 s001  S+   10:03PM   0:00.11 python␣
↪myapp.py
```

In the original terminal (running our app) we see that the app reloads and the inner loop continues:

```
Caught signal 1
WARNING: Reloading myapp
Inside MyController.default()
Inside Inner Loop
Inside Inner Loop
```

However, the PID remains the same:

```
$ ps auxw | grep [m]yapp
derks            1317   0.0  0.1  2425792  13012 s001  S+   10:03PM   0:00.11 python␣
↪myapp.py
```

If you'd like to see even more detail on what Cement is doing during the reload try adding `--debug`.

Managing Multiple Environments

Many applications and use-cases call for managing multiple environments and handling common settings between them. These environments might refer to infrastructure such as `production`, `staging`, or `dev`... or perhaps it might be to handle multiple accounts or regions within an account at a service provider.

The following example outlines one possible approach to managing multiple infrastructure accounts from an application Built on Cement:

myapp.py:

```
from cement.core.foundation import CementApp
from cement.core.controller import CementBaseController, expose
from cement.utils.misc import init_defaults

# set default settings for our different environments
defaults = init_defaults('myapp', 'env.production', 'env.staging', 'env.dev')
defaults['myapp']['default_env'] = 'production'
defaults['env.production']['foo'] = 'bar.production'
defaults['env.staging']['foo'] = 'bar.staging'
defaults['env.dev']['foo'] = 'bar.dev'

# do this in a hook so that we can load the default from config
def set_default_env(app):
    if app.pargs.env is None:
        app.pargs.env = app.config.get('myapp', 'default_env')

class MyController(CementBaseController):
```

```
    class Meta:
        label = 'base'
        arguments = [
            (['-E', '--environment'],
             dict(help='environment override',
                  action='store',
                  nargs='?',
                  choices=['production', 'staging', 'dev'],
                  dest='env')),
            ]

    @expose(hide=True)
    def default(self):
        print('Inside MyController.default()')

        # shorten things up a bit for clarity
        env_key = self.app.pargs.env
        env = self.app.config.get_section_dict('env.%s' % env_key)

        print('Current Environment: %s' % env_key)
        print('Foo => %s' % env['foo'])


class MyApp(CementApp):
    class Meta:
        label = 'myapp'
        config_defaults = defaults
        base_controller = MyController

with MyApp() as app:
    app.hook.register('post_argument_parsing', set_default_env)
    app.run()
```

myapp.conf

```
[myapp]
default_env = production

[env.production]
foo = bar.production

[env.staging]
foo = bar.staging

[env.dev]
foo = bar.dev
```

This looks like:

```
$ python myapp.py
Inside MyController.default()
Current Environment: production
Foo => bar.production

$ python myapp.py -E staging
Inside MyController.default()
Current Environment: staging
Foo => bar.staging
```

```
$ python myapp.py -E dev
Inside MyController.default()
Current Environment: dev
Foo => bar.dev
```

The idea being that you can maintain a single set of operations, but modify what or where those operations happen by simply toggling the configuration section (that has the same configuration settings per environment).

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y