# Cement Framework

***Release 3.0.6***

**Data Folk Labs, LLC**

**Dec 18, 2021**

# Contents


**1 Core Modules** **3**
- 1.1 cement.core.arg . . . 3
- 1.2 cement.core.backend . . . 4
- 1.3 cement.core.cache . . . 4
- 1.4 cement.core.config . . . 5
- 1.5 cement.core.controller . . . 7
- 1.6 cement.core.exc . . . 8
- 1.7 cement.core.extension . . . 8
- 1.8 cement.core.foundation . . . 10
- 1.9 cement.core.handler . . . 19
- 1.10 cement.core.hook . . . 22
- 1.11 cement.core.interface . . . 24
- 1.12 cement.core.log . . . 26
- 1.13 cement.core.mail . . . 26
- 1.14 cement.core.meta . . . 28
- 1.15 cement.core.output . . . 28
- 1.16 cement.core.template . . . 29
- 1.17 cement.core.plugin . . . 31

**2 Utility Modules** **33**
- 2.1 cement.utils.fs . . . 33
- 2.2 cement.utils.shell . . . 35
- 2.3 cement.utils.misc . . . 40
- 2.4 cement.utils.test . . . 42

**3 Extension Modules** **43**
- 3.1 cement.ext.ext_alarm . . . 43
- 3.2 cement.ext.ext_argparse . . . 43
- 3.3 cement.ext.ext_colorlog . . . 48
- 3.4 cement.ext.ext_configparser . . . 49
- 3.5 cement.ext.ext_daemon . . . 50
- 3.6 cement.ext.ext_dummy . . . 52
- 3.7 cement.ext.ext_generate . . . 55
- 3.8 cement.ext.ext_jinja2 . . . 55
- 3.9 cement.ext.ext_json . . . 56
- 3.10 cement.ext.ext_logging . . . 58
- 3.11 cement.ext.ext_memcached . . . 60

3.12 cement.ext.ext_mustache . . . . . 61
3.13 cement.ext.ext_plugin . . . . . 62
3.14 cement.ext.ext_print . . . . . 63
3.15 cement.ext.ext_redis . . . . . 64
3.16 cement.ext.ext_scrub . . . . . 65
3.17 cement.ext.ext_smtp . . . . . 66
3.18 cement.ext.ext_tabulate . . . . . 67
3.19 cement.ext.ext_yaml . . . . . 68
3.20 cement.ext.ext_watchdog . . . . . 69

**Python Module Index** **71**

**Index** **73**

**Note:** This documentation is strictly for API reference. For more complete developer documentation, please visit the official site http://builtoncement.com.

CHAPTER 1

# Core Modules

## 1.1 cement.core.arg

Cement core argument module.

**class** cement.core.arg.**ArgumentHandler**(**kw*)
Bases: *cement.core.arg.ArgumentInterface*, *cement.core.handler.Handler*

Argument handler implementation

**class** cement.core.arg.**ArgumentInterface**(**kw*)
Bases: *cement.core.interface.Interface*

This class defines the Argument Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *ArgumentHandler* base class as a starting point.

**class Meta**
Bases: object

Interface meta-data options.

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

**add_argument**(*args*, **kw*)
Add arguments to the parser.

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*) – List of option arguments. Generally something like ['-h', '--help'].

**Keyword Arguments**

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

**Returns** None

**parse**(**args*)

Parse the argument list (i.e. `sys.argv`). Can return any object as long as its' 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** **args** (*list*) – A list of command line arguments

**Returns** A callable object whose member reprisent the available arguments

**Return type** object

## 1.2 cement.core.backend

Cement core backend module.

## 1.3 cement.core.cache

Cement core cache module.

**class** cement.core.cache.**CacheHandler**(***kw*)

Bases: *cement.core.cache.CacheInterface*, *cement.core.handler.Handler*

Cache handler implementation.

**class** cement.core.cache.**CacheInterface**(***kw*)

Bases: *cement.core.interface.Interface*

This class defines the Cache Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *CacheHandler* base class as a starting point.

**class Meta**

Bases: `object`

Handler meta-data.

**interface = 'cache'**

The string identifier of the interface.

**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** bool

**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** (*str*) – The key of the value stored in cache

**Keyword Arguments** **fallback** – Optional value that is returned if the cache is expired or the key does not exist.

**Returns** Whatever the value is in the cache, or the `fallback`

**Return type** Unknown

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

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

**Parameters**

- **key** (*str*) – The key of the value to store in cache
- **value** (*unknown*) – The value of that key to store in cache

**Keyword Arguments** **time** (*int*) – A one-off expire time in seconds (or `None`. If no time is given, then a default value is used (determined by the implementation).

Returns: None

## 1.4 cement.core.config

Cement core config module.

**class** cement.core.config.**ConfigHandler**(***kw*)
Bases: *cement.core.config.ConfigInterface*, *cement.core.handler.Handler*

Config handler implementation.

**_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** (*str*) – The file system path to the configuration file.

**Returns** `True` if file was read properly, `False` otherwise

**Return type** bool

**parse_file**(*file_path*)
Ensure we are using the absolute/expanded path to `file_path`, and then call `self._parse_file` to parse config file settings from it, overwriting existing config settings.

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** (*str*) – The file system path to the configuration file.

**Returns**

**True if the given file_path was parsed, and False** otherwise.

**Return type** bool

**class** cement.core.config.**ConfigInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Config Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *ConfigHandler* base class as a starting point.

**class Meta**
Bases: object

Handler meta-data.

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

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

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

**Returns** None

**get**(*section*, *key*)
Return a configuration value based on section.key. Must honor environment variables if they exist to override the config... for example config['myapp']['foo']['bar'] must be overridable by the environment variable MYAPP_FOO_BAR.... Note that MYAPP_ must prefix all vars, therefore config['redis']['foo'] would be overridable by MYAPP_REDIS_FOO ... but config['myapp']['foo']['bar'] would not have a double prefix of MYAPP_MYAPP_FOO_BAR.

**Parameters**

- **section** (*str*) – The section of the configuration to pull key values from.
- **key** (*str*) – The configuration key to get the value for.

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

**Return type** unknown

**get_dict**()
Return a dict of the entire configuration.

**Returns** A dictionary of the entire config.

**Return type** dict

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

**Parameters** **section** (*str*) – The config section to generate a dict from (using that sections' keys).

**Returns** A dictionary of the config section.

**Return type** dict

**get_sections**()
Return a list of configuration sections.

**Returns** A list of config sections.

**Return type** list

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

**Parameters** **section** (*str*) – The section to test for.

**Returns**

**True if the configuration section exists, False** otherwise.

**Return type** bool

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

**Parameters** **section** (*list*) – The config section to pull keys from.

**Returns** A list of keys in `section`.

**Return type** list

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

**Parameters**

- **dict_obj** (*dict*) – The dictionary to merge into the config
- **override** (*bool*) – Whether to override existing values or not.

**Returns** None

**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** (*str*) – The path to the config file to parse.

**Returns** `True` if the file was parsed, `False` otherwise.

**Return type** bool

**set**(*section*, *key*, *value*)
Set a configuration value based at `section.key`.

**Parameters**

- **section** (*str*) – The `section` of the configuration to pull key value from.
- **key** (*str*) – The configuration key to set the value at.
- **value** – The value to set.

**Returns** None

## 1.5 cement.core.controller

Cement core controller module.

**class** cement.core.controller.**ControllerHandler**(***kw*)
Bases: *cement.core.controller.ControllerInterface*, *cement.core.handler.Handler*

Controller handler implementation.

**class** cement.core.controller.**ControllerInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Controller Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *ControllerHandler* base class as a starting point.

**class Meta**
Bases: object

Interface meta-data.

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

**_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** The result of the executed controller function, or `None` if no controller function is called.

**Return type** unknown

## 1.6 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** (*int*) – The signal number
- **frame** – The signal frame object

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

General framework (non-application) related errors.

**Parameters** **msg** (*str*) – The error message

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

Interface related errors.

## 1.7 cement.core.extension

Cement core extensions module.

**class** cement.core.extension.**ExtensionHandler**(***kw*)
Bases: *cement.core.extension.ExtensionInterface*, *cement.core.handler.Handler*

This handler implements the Extention Interface, which handles loading framework extensions. All extension handlers should sub-class from here, or ensure that their implementation meets the requirements of this base class.

**class Meta**
Bases: object

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

**label = 'cement'**
The string identifier of the handler.

**get_loaded_extensions**()
Get all loaded extensions.

**Returns** A list of loaded extensions.

**Return type** list

**list**()
Synonymous with get_loaded_extensions().

**Returns** A list of loaded extensions.

**Return type** list

**load_extension**(*ext_module*)
Given an extension module name, load or in other-words import the extension.

**Parameters** **ext_module** (*str*) – The extension module name. For example: cement.ext.ext_logging.

**Raises** *cement.core.exc.FrameworkError* – Raised if ext_module can not be loaded.

**load_extensions**(*ext_list*)
Given a list of extension modules, iterate over the list and pass individually to self.load_extension().

**Parameters** **ext_list** (*list*) – A list of extension module names (str).

**class** cement.core.extension.**ExtensionInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Extension Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *ExtensionHandler* base class as a starting point.

**class Meta**
Bases: object

Handler meta-data.

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

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

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

**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_json', 'cement.ext.ext_logging']`

## 1.8 cement.core.foundation

Cement core foundation module.

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

The primary application object class.

**class Meta**
Bases: object

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

**alternative_module_mapping = {}**
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 `App.__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.
**Type** EXPERIMENTAL FEATURE

**argument_handler = 'argparse'**
A handler class that implements the Argument interface.

**argv = None**
A list of arguments to use for parsing command line arguments and options.

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

**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 hard-coded application code. It is also called before plugins are loaded.

**cache_handler = None**
A handler class that implements the Cache interface.

**catch_signals = [<Signals.SIGTERM: 15>, <Signals.SIGINT: 2>, <Signals.SIGHUP: 1>]**
List of signals to catch, and raise `cement.core.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 (appended to the builtin list of directories defined by Cement). For each directory cement will load all files that ends with `.conf`. Note: Though `App.Meta.config_dirs` defaults to `None`, Cement will set this to a default list based on `App.Meta.label` (or in other words, the name of the application). This will equate to:

```
[
    '/etc/myapp/ext.d/',
    '/etc/myapp/plugins.d/',
    '~/.myapp/config/ext.d/',
    '~/.myapp/config/plugins.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_file_suffix`.

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

**config_files = None**
List of config files to parse (appended to the builtin list of config files defined by Cement).

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

```
[
    '/etc/myapp/myapp.conf',
    '~/.config/myapp/myapp.conf',
    '~/.myapp.conf',
]
```

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 `App.Meta.config_file_suffix`.

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

**config_section = None**
The base configuration section for the application.

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

**core_extensions = ['cement.ext.ext_dummy', 'cement.ext.ext_smtp', 'cement.ext.ext_p**
List of Cement core extensions. These are generally required by Cement and should only be modified if you know what you're doing. Use `App.Meta.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 might not need/want `LoggingLogHandler` to be registered).

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

**core_interfaces = [<class 'cement.core.extension.ExtensionInterface'>, <class 'ceme**
List of core interfaces to be defined (by the framework). You should not modify this unless you really know what you're doing... instead, you probably want to add your own interfaces to `App.Meta.interfaces`.

**core_meta_override = ['debug', 'plugin_dir', 'ignore_deprecation_warnings', 'templa**
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 `App.Meta.config_section` but defaults to `App.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 you should use `App.Meta.meta_override`.

**core_system_config_dirs = ['/etc/{label}/ext.d', '/etc/{label}/plugins.d']**
List of builtin system level configuration directories to scan for config files.

**core_system_config_files = ['/etc/{label}/{label}{suffix}']**
List of builtin system level configuration files.

**core_system_plugin_dirs = ['/usr/lib/{label}/plugins']**
List of builtin system level directories to scan for plugins.

**core_system_template_dirs = ['/usr/lib/{label}/templates']**
List of builtin system level directories to scan for templates.

**core_user_config_dirs = ['{home_dir}/.config/{label}/ext.d', '{home_dir}/.config/{l**
List of builtin user level configuration directories to scan for config files.

**core_user_config_files = ['{home_dir}/.config/{label}/{label}{suffix}', '{home_dir}**
List of builtin user level configuration files.

**core_user_plugin_dirs = ['{home_dir}/.config/{label}/plugins', '{home_dir}/.{label}**
List of builtin user level directories to scan for plugins.

**core_user_template_dirs = ['{home_dir}/.config/{label}/templates', '{home_dir}/.{la**
List of builtin user level template directories to scan for templates.

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

**debug_argument_help = 'full application debug mode'**
The debug argument help text that is displayed in `--help`.

**debug_argument_options = ['-d', '--debug']**
The argument option(s) to toggle debug mode via cli.

**define_hooks = []**
List of hook definitions (labels). 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 Extension 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 `App.Meta.core_handler_override_options` but this one has precedence.

Dictionary Format:

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

See `App.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 `App.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.

**interfaces = []**
List of interfaces to be defined. Must be a list of uninstantiated interface base classes.

I.e. `[MyCustomInterface, SomeOtherInterface]`

**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 Log interface.

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

**meta_defaults = {}**
Default meta-data 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 import App

META = {
    'output.json': {
        'json_module': 'ujson',
    }
}

class MyApp(App):
    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 `App.Meta.config_section` but defaults to `App.Meta.label`).

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

**plugin_dir = None**
A directory path where plugin code (modules) can be loaded from. By default, this setting is also overridden by the `myapp.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 (appended to the builtin list of directories defined by Cement).

Note: Though `App.Meta.plugin_dirs` defaults to `None`, Cement will populate this with a default list based on `App.Meta.label`. This will equate to:

```
[
    '~/.myapp/plugins',
    '~/.config/myapp/plugins',
    '/usr/lib/myapp/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 Plugin interface.

**plugin_module = 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/` (directory). 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.

**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.

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

**quiet_argument_help = 'suppress all console output'**
The quiet argument help text that is displayed in `--help`.

**quiet_argument_options = ['-q', '--quiet']**
The argument option(s) to toggle quiet mode via cli.

**signal_handler** (*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 `myapp.template_dir` config setting parsed in any of the application configuration files .

If set, this item will be **prepended** to `App.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 (appended to the builtin list of directories defined by Cement).

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

```
[
    '~/.myapp/templates',
    '~/.config/myapp/templates',
    '/usr/lib/myapp/templates',
]
```

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

**template_handler = 'dummy'**
A handler class that implements the Template interface.

**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 `App.Meta.template_dirs`, and and secondly from `App.Meta.template_module`. The `template_dirs` setting has presedence.

**_lay_cement**()
Initialize the framework.

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

**add_config_dir**(*path*)
Append a directory `path` to the list of directories to parse for config files.

**Parameters** **path** (*str*) – Directory path that contains config files.

Example:

```
app.add_config_dir('/path/to/my/config/')
```

**add_config_file**(*path*)
Append a file `path` to the list of configuration files to parse.

**Parameters** **path** (*str*) – Configuration file path..

Example:

```
app.add_config_file('/path/to/my/config/file')
```

**add_plugin_dir**(*path*)
Append a directory `path` to the list of directories to scan for plugins.

**Parameters** **path** (*str*) – Directory path that contains plugin files.

Example:

```
app.add_plugin_dir('/path/to/my/plugins')
```

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

**Parameters** **path** (*str*) – Directory path that contains template files.

Example:

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

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

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

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

**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
- **then exit with whatever self.exit_code is currently set** (*passed*) –
- **Note** (*to.*) – `sys.exit()` will only be called if
- **App.Meta.exit_on_close==True.** –

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

**Returns** boolean

**extend**(*member_name*, *member_object*)
Extend the `App()` 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
- **App()** –

**Raises** *cement.core.exc.FrameworkError* – If `App().member_name` already exists.

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

**Returns** `(data, output_text)`

**Return type** tuple

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

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

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

**Parameters** **path** (*str*) – Directory path that contains template files.

### Example

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

**render**(*data*, *template=None*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *handler=None*, ***kw*)

This is a simple wrapper around `self.output.render()` which simply returns an empty string if no output handler is defined.

**Parameters** **data** (*dict*) – The data dictionary to render.

**Keyword Arguments**

- **template** (*str*) – The template to render to (note that some output handlers do not use templates).
- **out** – A file like object (i.e. `sys.stdout`, or actual file). Set to `None` if no output is desired (just render and return).
- **handler** – The output handler to use to render. Defaults to `App.Meta.output_handler`.

**Other Parameters** **kw** (*dict*) – Additional keyword arguments will be passed to the output handler when calling `self.output.render()`.

**run**()

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

**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.

**Return type** unknown

**run_forever**(*interval=1*, *tb=True*)

This function wraps `self.run()` with an endless while loop. If any exception is encountered it will be logged and then the application will be reloaded.

**Parameters**

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

**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.

**validate_config**()

Validate application config settings.

Example:

```
import os
from cement import App

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

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

        # test that the log file directory exist, if not create it
```

(continues on next page)

(continued from previous page)

```
        logdir = os.path.dirname(self.config.get('log', 'file'))

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

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

App subclass useful for testing.

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** (*instance*) – 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** (*int*) – The signal number
- **frame** – The signal frame

**Raises** *cement.core.exc.CaughtSignal* – Raised, passing signum, and frame

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

**Parameters** **app** (*instance*) – The application object.

## 1.9 cement.core.handler

Cement core handler module.

**class** cement.core.handler.**Handler**(***kw*)
Bases: abc.ABC, *cement.core.meta.MetaMixin*

Base handler class that all Cement Handlers should subclass from.

**class Meta**
Bases: object

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 App.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 = NotImplemented**
The interface that this class implements.

**label = NotImplemented**
The string identifier of this handler.

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

**_setup**(*app*)
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** (*instance*) – The application object.

**_validate**()
Perform any validation to ensure proper data, meta-data, etc.

**class** cement.core.handler.**HandlerManager**(*app*)
Bases: object

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

**get**(*interface*, *handler_label*, *fallback=None*, ***kwargs*)
Get a handler object.

**Parameters**

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

**Keyword Arguments** **setup** (*bool*) – Whether or not to call `setup()` on the handler before returning. This will not be called on the `fallback` if no the handler given does not exist.

**Returns** An uninstantiated handler object

**Return type** *Handler*

**Raises** *cement.core.exc.InterfaceError* – If the `interface` does not exist, or if the handler itself does not exist.

### Example

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

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

**Parameters** **interface** (*str*) – The interface of the handler (i.e. `output`)

**Returns** Handler labels (str) that match `interface`.

**Return type** list

**Raises** *cement.core.exc.InterfaceError* – If the `interface` does not exist.

### Example

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

**register**(*handler_class*, *force=False*)
Register a handler class to an interface. 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 `InterfaceError` is raised.

**Parameters** **handler_class** (`Handler`) – The uninstantiated handler class to register.

**Keyword Arguments**

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

**Raises**

- *cement.core.exc.InterfaceError* – If the `handler_class` does not implement *Handler*, or if `handler_class` does not properly sub-class it's interface.
- *cement.core.exc.InterfaceError* – If the `handler_class.Meta.interface` does not exist

Usage:

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

    def connect(self):
        # ...

app.handler.register(MyDatabaseHandler)
```

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

**Parameters**

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

**Returns** `True` if the handler is registered, `False` otherwise

**Return type** bool

### Example

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

**resolve**(*interface*, *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**

- **interface** (*str*) – The interface of the handler (ex: `output`)

- **handler_def** (*str, instance, Handler*) – The loose references of the handler, by label, instantiated object, or non-instantiated class.

**Keyword Arguments**

- **raise_error** (*bool*) – Whether or not to raise an exception if unable to resolve the handler.
- **meta_defaults** (*dict*) – Optional meta-data dictionary used as defaults to pass when instantiating uninstantiated handlers. Use `App.Meta.meta_defaults` by default.
- **setup** (*bool*) – Whether or not to call `.setup()` before return. Default: `False`

**Returns** The instantiated handler object.

**Return type** instance

### Example

```
# 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())
```

**setup**(*handler_class*)

Setup a handler class so that it can be used.

**Parameters** **handler_class** (*class*) – An uninstantiated handler class.

Returns: None

### Example

```
for controller in app.handler.list('controller'):
    ch = app.handler.setup(controller)
```

## 1.10 cement.core.hook

Cement core hooks module.

**class** cement.core.hook.**HookManager**(*app*)

Bases: object

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

**define**(*name*)

Define a hook namespace that the application and plugins can register hooks in.

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

**Raises** *cement.core.exc.FrameworkError* – If the hook name is already defined

### Example

```
from cement import App

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

**defined**(*hook_name*)

Test whether a hook name is defined.

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

**Returns** `True` if the hook is defined, `False` otherwise.

**Return type** bool

### Example

```
from cement import App

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

**list**()

List all defined hooks.

**Returns** List of registered hook labels.

**Return type** hooks (list)

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

**Keywork Args:** weight (int): The weight in which to order the hook function.

### Example

```
from cement import App

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

with App('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.

**Parameters**

- **name** (*str*) – The name of the hook function.
- **args** (*tuple*) – Additional arguments to be passed to the hook functions.
- **kwargs** (*dict*) – Additional keyword arguments to be passed to the hook functions.

**Yields** The result of each hook function executed.

**Raises** *cement.core.exc.FrameworkError* – If the hook `name` is not defined

### Example

```
from cement import App

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

with App('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', app):
        # do something with the result?
        pass
```

## 1.11 cement.core.interface

Cement core interface module.

**class** cement.core.interface.**Interface**(***kw*)
Bases: abc.ABC, *cement.core.meta.MetaMixin*

Base interface class that all Cement Interfaces should subclass from.

**class Meta**
Bases: object

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

**interface = NotImplemented**
The string identifier of this interface.

**_validate**()
Perform any validation to ensure proper data, meta-data, etc.

**class** cement.core.interface.**InterfaceManager**(*app*)
Bases: object

Manages the interface system to define, get, list interfaces with the Cement Framework.

**define**(*ibc*)
Define an `ibc` (interface base class).

**Parameters** **ibc** (`Interface`) – The abstract base class that defines the interface

**Raises**

- *cement.core.exc.InterfaceError* – If the interface label is already
- *defined*

### Example

```
app.interface.define(DatabaseInterface)
```

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

**Parameters** **interface** (*str*) – The label of the interface (I.e. `log`, `config`, `output`, etc).

**Returns** `True` if the interface is defined, `False` otherwise

**Return type** bool

### Example

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

**get**(*interface*, *fallback=None*, ***kwargs*)
Get an interface class.

**Parameters**

- **interface** (*str*) – The interface label (i.e. `output`)
- **fallback** (`Handler`) – A fallback value to return if `interface` doesn't exist.

**Returns** An uninstantiated interface class

**Return type** *Interface*

**Raises** *cement.core.exc.InterfaceError* – If the `interface` does not exist.

### Example

```
i = app.interface.get('output')
```

**list**()
Return a list of defined interfaces.

**Returns** Interface labels.

**Return type** list

### Example

```
app.interface.list()
```

## 1.12 cement.core.log

Cement core log module.

**class** cement.core.log.**LogHandler**(***kw*)
Bases: *cement.core.log.LogInterface*, *cement.core.handler.Handler*

Log handler implementation.

**class** cement.core.log.**LogInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Log Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *LogHandler* base class as a starting point.

**class Meta**
Bases: object

Handler meta-data.

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

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

**Parameters** **msg** (*str*) – The message to log.

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

**Parameters** **msg** (*str*) – The message to log.

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

**Parameters** **msg** (*str*) – The message to log.

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

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

**Parameters** **msg** (*str*) – The message to log.

**set_level**()

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

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

**Parameters** **msg** (*str*) – The message to log.

## 1.13 cement.core.mail

Cement core mail module.

**class** cement.core.mail.**MailHandler**(***kw*)
Bases: *cement.core.mail.MailInterface*, *cement.core.handler.Handler*

Mail handler implementation.

**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`

**class Meta**
Bases: `object`

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

**config_defaults = {'bcc': [], 'cc': [], 'from_addr': 'noreply@example.com', 'sub**
Configuration default values

**_setup**(*app_obj*)
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** (*instance*) – The application object.

**class** cement.core.mail.**MailInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Mail Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *MailHandler* base class as a starting point.

**class Meta**
Bases: `object`

Handler meta-data.

**interface = 'mail'**
The label identifier of the interface.

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

**Parameters** **body** (*str*) – The message body to send

**Keyword Arguments**

- **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** `True` if message was sent successfully, `False` otherwise

**Return type** bool

### Example

```
# 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',
    )
```

## 1.14 cement.core.meta

Cement core meta functionality.

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

Container class for meta attributes of a larger class. Keyword arguments are set as attributes of `self`.

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

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

## 1.15 cement.core.output

Cement core output module.

**class** cement.core.output.**OutputHandler**(***kw*)
Bases: *cement.core.output.OutputInterface*, *cement.core.handler.Handler*

Output handler implementation.

**class** cement.core.output.**OutputInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Output Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *OutputHandler* base class as a starting point.

**class Meta**
Bases: object

Handler meta-data.

**interface = 'output'**
The string identifier of the interface

**render**(*data*, **args*, ***kwargs*)
Render the `data` dict into output in some fashion. This function must accept 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** The rendered output string, or `None` if no output is rendered

**Return type** str, None

## 1.16 cement.core.template

Cement core template module.

**class** cement.core.template.**TemplateHandler**(**args*, ***kwargs*)
Bases: *cement.core.template.TemplateInterface*, *cement.core.handler.Handler*

Base class that all template implementations should sub-class from. Keyword arguments passed to this class will override meta-data options.

**copy**(*src*, *dest*, *data*, *force=False*, *exclude=None*, *ignore=None*)
Render `src` directory as template, including directory and file names, and copy to `dest` directory.

**Parameters**

- **src** (*str*) – The source directory path.
- **dest** (*str*) – The destination directory path.
- **data** (*dict*) – The data dictionary to interpolate in the template.
- **force** (*bool*) – Whether to overwrite existing files.
- **exclude** (*list*) – List of regular expressions to match files that should only be copied, and not rendered as template.
- **ignore** (*list*) – List of regular expressions to match files that should be completely ignored and not copied at all.

**Returns** Returns `True` if the copy completed successfully.

**Return type** bool

**Raises** `AssertionError` – If the `src` template directory path does not exists, and when a `dest` file already exists and `force is not True`.

**load**(*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** (*str*) – The secondary path of the template **after** either `template_module` or `template_dirs` prefix (set via `App.Meta`)

**Returns** The content of the template (`str`), the type of template (`str: directory`, or `module`), and the path (`str`) of the directory or module)

**Return type** tuple

**Raises** *cement.core.exc.FrameworkError* – If the template does not exist in either the `template_module` or `template_dirs`.

**render**(*content*, *data*)
Render `content` as template using using the `data` dictionary.

**Parameters**

- **content** (*str*) – The content to render.
- **data** (*dict*) – The data dictionary to interpolate in the template.

**Returns** The rendered content.

**Return type** str

**class** cement.core.template.**TemplateInterface**(***kw*)

Bases: *cement.core.interface.Interface*

This class defines the Template Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *TemplateHandler* base class as a starting point.

**class Meta**

Bases: object

Handler meta-data.

**interface = 'template'**

The string identifier of the interface

**copy**(*src*, *dest*, *data*)

Render the src directory path, and copy to dest. This method must render directory and file **names** as template content, as well as the contents of files.

**Parameters**

- **src** (*str*) – The source template directory path.
- **dest** (*str*) – The destination directory path.
- **data** (*dict*) – The data dictionary to render with template.

Returns: None

**load**(*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** **path** (*str*) – The secondary path of the template **after** either template_module or template_dirs prefix (set via App.Meta)

**Returns** The content of the template (str), the type of template (str: directory, or module), and the path (str) of the directory or module)

**Return type** tuple

**Raises** *cement.core.exc.FrameworkError* – If the template does not exist in either the template_module or template_dirs.

**render**(*content*, *data*)

Render content as a template using the data dict.

**Parameters**

- **content** (*str*) – The content to be rendered as a template.
- **data** (*dict*) – The data dictionary to render with template.

**Returns** The rendered template string.

**Return type** str

## 1.17 cement.core.plugin

Cement core plugins module.

**class** cement.core.plugin.**PluginHandler**(***kw*)
Bases: *cement.core.plugin.PluginInterface*, *cement.core.handler.Handler*

Plugin handler implementation.

**class** cement.core.plugin.**PluginInterface**(***kw*)
Bases: *cement.core.interface.Interface*

This class defines the Plugin Interface. Handlers that implement this interface must provide the methods and attributes defined below. In general, most implementations should sub-class from the provided *PluginHandler* base class as a starting point.

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

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

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

**Parameters** **plugins** (*list*) – A list of plugin names to load.

CHAPTER 2

# Utility Modules

## 2.1 cement.utils.fs

Common File System Utilities.

**class** cement.utils.fs.**Tmp**(***kwargs*)

Bases: object

Provides creation and cleanup of a separate temporary directory, and file.

**Keyword Arguments**

- **cleanup** (*bool*) – Whether or not to delete the temporary directory and file on exit (when used with the `with` operator).
- **suffix** (*str*) – The suffix that the directory and file will end with. Default: *no suffix*
- **prefix** (*str*) – The prefix that the directory and file will start with. Default: *no prefix*
- **dir** (*str*) – The parent directory path that the temp directory and file will be created in. Default: *system default temporary path*

### Example

```
from cement.utils import fs

with fs.Tmp() as tmp:
    # do something with a temporary directory
    os.path.listdir(tmp.dir)

    # do something with a temporary file
    with open(tmp.file, 'w') as f:
        f.write('some data')
```

**remove**()

Remove the temporary directory (and file) if it exists, and `self.cleanup` is `True`.

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

**Parameters** **path** (*str*) – The original path to expand.

**Returns** The fully expanded, absolute path to the given `path`

**Return type** str

### Example

```
from cement.utils import fs

fs.abspath('~/some/path')
fs.abspath('./some.file')
```

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

**Parameters**

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

**Returns** The new path of backed up file/directory

**Return type** str

### Example

```
from cement.utils import fs

fs.backup('/path/to/original/file')
```

cement.utils.fs.**ensure_dir_exists**(*path*)
Ensure the directory `path` exists, and if not create it.

**Parameters** **path** (*str*) – The filesystem path of a directory.

**Raises**

- `AssertionError` – If the directory `path` exists, but is not a
- `directory.`

Returns: None

cement.utils.fs.**ensure_parent_dir_exists**(*path*)
Ensure the parent directory of `path` (file, or directory) exists, and if not create it.

**Parameters** **path** (*str*) – The filesystem path of a file or directory.

Returns: None

cement.utils.fs.**join**(**args*, ***kwargs*)
Return a complete, joined path, by first calling `abspath()` on the first item to ensure the final path is complete.

**Parameters** **paths** (*list*) – A list of paths to join together.

**Returns** The complete and absolute joined path.

**Return type** list

#### Example

```
from cement.utils import fs

fs.join('~/some/path', 'some/other/relevant/paht')
```

cement.utils.fs.**join_exists**(**paths*)

Wrapper around `os.path.join()`, `os.path.abspath()`, and `os.path.exists()`.

**Parameters** **paths** (*list*) – List of paths to join, and then return `True` if that path exists, or `False` if it does not.

**Returns**

**First item is the fully joined absolute path, and the second** is `bool` (whether that path exists or not).

**Return type** tuple

## 2.2 cement.utils.shell

Common Shell Utilities.

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

Bases: *cement.core.meta.MetaMixin*

A wrapper around `input` 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** (*str*) – The text displayed at the input prompt.

#### Example

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**

Bases: object

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.

**text = 'Tell me someting interesting:'**
The text that is displayed to prompt the user

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

`cement.utils.shell.`**cmd**(*command*, *capture=True*, **args*, ***kwargs*)
Wrapper around `exec_cmd` and `exec_cmd2` depending on whether capturing output is desired. Defaults to setting the Popen `shell` keyword argument to `True` (string command rather than list of command and arguments).

**Parameters**

- **command** (*str*) – The command (and arguments) to run.
- **capture** (*bool*) – Whether or not to capture output.

**Other Parameters**

- **args** – Additional arguments are passed to `Popen()`.
- **kwargs** – Additional keyword arguments are passed to `Popen()`.

**Returns**

**When capture==True, returns the ‘‘(stdout, stderror,** return_code)‘‘ of the command.

**int: When capture==False, returns only the exitcode of the** command.

**Return type** tuple

### Example

```
from cement.utils import shell

# execute a command and capture output
stdout, stderr, exitcode = shell.cmd('echo helloworld')
```

(continues on next page)

(continued from previous page)

```
# execute a command but do not capture output
exitcode = shell.cmd('echo helloworld', capture=False)
```

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

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.

**Other Parameters**

- **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

### Example

```
from cement.utils import shell

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

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

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

**Other Parameters**

- **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

### Example

```
from cement.utils import shell

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

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

Wrapper around `spawn_process` and `spawn_thread` depending on desired execution model.

**Parameters** **target** (*function*) – The target function to execute in the sub-process.

**Keyword Arguments**

- **start** (*bool*) – Call `start()` on the process before returning the process object.
- **join** (*bool*) – Call `join()` on the process before returning the process object. Only called if `start == True`.

- **thread** (*bool*) – Whether to spawn as thread instead of process.

**Other Parameters**

- **args** – Additional arguments are passed to `Process()`
- **kwargs** – Additional keyword arguments are passed to `Process()`.

**Returns** The process object returned by Process().

**Return type** object

### Example

```
from cement.utils import shell

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

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

`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** (*function*) – The target function to execute in the sub-process.

**Keyword Arguments**

- **start** (*bool*) – Call `start()` on the process before returning the process object.
- **join** (*bool*) – Call `join()` on the process before returning the process object. Only called if `start == True`.

**Other Parameters**

- **args** – Additional arguments are passed to `Process()`
- **kwargs** – Additional keyword arguments are passed to `Process()`.

**Returns** The process object returned by Process().

**Return type** object

### Example

```
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** (*function*) – The target function to execute in the thread.

**Keyword Arguments**

- **start** (*bool*) – Call `start()` on the thread before returning the thread object.
- **join** (*bool*) – Call `join()` on the thread before returning the thread object. Only called if `start == True`.

**Other Parameters**

- **args** – Additional arguments are passed to `Thread()`.
- **kwargs** – Additional keyword arguments are passed to `Thread()`.

**Returns** The thread object returned by `Thread()`.

**Return type** object

### Example

```
from cement.utils import shell

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

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

## 2.3 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. This is sometimes more useful, or cleaner than creating an entire dict set (often used in testing).

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

**Returns** Dictionary of nested dictionaries (sections)

**Return type** dict

### Example

```
from cement import App, init_defaults

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

app = App('myapp', config_defaults=defaults)
```

cement.utils.misc.**is_true**(*item*)

Given a value, determine if it is one of `[True, 'true', 'yes', 'y', 'on', '1', 1,]` (note: strings are converted to lowercase before comparison).

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

**Returns**

**True if item equates to a true-ish value, False** otherwise

**Return type** bool

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** (*str*) – The logging namespace. This is generally __name__ or anything you want.

**Keyword Arguments** **debug** (*bool*) – Toggle debug output.

**Returns** A Logger object

**Return type** object

### Example

```
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** (*str*) – Optional salt, if None then random.random() is used.

**Returns** Random MD5 hash

**Return type** str

### Example

```
from cement.utils.misc import rando

rando('dAhn49amvnah3m')
```

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** (*str*) – The text to wrap

**Keyword Arguments**

- **width** (*int*) – The max width of a line before breaking
- **indent** (*str*) – String to prefix subsequent lines after breaking
- **long_words** (*bool*) – Whether or not to break on long words
- **hyphens** (*bool*) – Whether or not to break on hyphens

**Returns** The wrapped string

**Return type** str

## 2.4 cement.utils.test

Cement testing utilities.

CHAPTER 3

# Extension Modules

## 3.1 cement.ext.ext_alarm

Cement alarm extension module.

**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** (*int*) – The time in seconds to set the alarm to.
- **msg** (*str*) – The message to display if the alarm is triggered.

**stop**()
Stop the application alarm.

## 3.2 cement.ext.ext_argparse

Cement argparse extension module.

**class** cement.ext.ext_argparse.**ArgparseArgumentHandler**(**args*, ***kw*)
Bases: argparse.ArgumentParser, *cement.core.arg.ArgumentHandler*

This class implements the Argument Handler 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**
Bases: object

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 = 'argument'**
The interface that this class implements.

**label = 'argparse'**
The string identifier of the handler.

**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.

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

**Returns** Instance object whose members are the arguments parsed.

**Return type** object

**class** cement.ext.ext_argparse.**ArgparseController**(**args*, ***kw*)
Bases: *cement.core.controller.ControllerHandler*

This is an implementation of the Controller handler interface, and is 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.

### Example

```
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'],
                { 'help' : 'my foo option',
                  'dest' : 'foo' }
            ),
        ]

class Second(ArgparseController):
```

(continues on next page)

(continued from previous page)

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

**class Meta**

Bases: object

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**.

**argument_formatter**

alias of argparse.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(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 = {}**

dict) that are merged into the applications config object for the config_section mentioned above.

**Type** Configuration defaults (type

**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. By default this is `_default`, which is equivelant to passing `-h/--help`. It should be noted that this is the only place where having a command function start with _ is OK simply because we treat it as a special case (different that other exposed commands).

If set to `None`, Cement will simply pass and exit 0.

Note: Currently, default function/sub-command only works 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.

**label = None**
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.

**_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** The result of the executed controller function, or `None` if no controller function is called.

**Return type** unknown

**_get_exposed_commands**()
Get a list of exposed commands for this controller

**Returns** List of exposed commands (labels)

**Return type** exposed_commands (list)

**_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`).

**_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`.

#### Example

```
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')
```

**_validate**()
Perform any validation to ensure proper data, meta-data, etc.

cement.ext.ext_argparse.**ex**
alias of *cement.ext.ext_argparse.expose*

**class** cement.ext.ext_argparse.**expose**(*hide=False*, *arguments=[]*, *label=None*, ***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.

**Keyword Arguments**

- **hide** (*bool*) – Whether the command should be visible.
- **arguments** (`list`) – List of tuples that define arguments to add to this commands sub-parser.
- **parser_options** (*dict*) – Additional options to pass to Argparse.

- **label** (*str*) – String identifier for the command.

### Example

```
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()")
```

## 3.3 cement.ext.ext_colorlog

Cement colorlog extension module.

**class** cement.ext.ext_colorlog.**ColorLogHandler**(**args*, ***kw*)

Bases: *cement.ext.ext_logging.LoggingLogHandler*

This class implements the Log Handler 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**

Bases: `object`

Handler meta-data.

**colors = {'CRITICAL': 'red,bg_white', 'DEBUG': 'cyan', 'ERROR': 'red', 'INFO': 'gre**

Color mapping for each log level

**config_defaults = {'colorize_console_log': True, 'colorize_file_log': False, 'fil**

Default configuration settings. Will be overridden by the same settings in any application configuration file under a `[log.colorlog]` block.

**formatter_class**

alias of `builtins.ColoredFormatter`

**formatter_class_without_color**

alias of `logging.Formatter`

**label = 'colorlog'**
The string identifier of the handler.

## 3.4 cement.ext.ext_configparser

Cement configparser extension module.

**class** cement.ext.ext_configparser.**ConfigParserConfigHandler**(***kw*)
Bases: *cement.core.config.ConfigHandler*, configparser.RawConfigParser

This class is an implementation of the *Config* 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**
Bases: object

Handler meta-data.

**label = 'configparser'**
The string identifier of this handler.

**_parse_file**(*file_path*)
Parse a configuration file at file_path and store it.

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

**Returns** True if file was read properly, False otherwise

**Return type** bool

**add_section**(*section*)
Adds a block section to the config.

**Parameters** **section** (*str*) – The section to add.

**get**(*section*, *key*, ***kwargs*)
Return a configuration value based on section.key. Must honor environment variables if they exist to override the config... for example config['myapp']['foo']['bar'] must be overridable by the environment variable MYAPP_FOO_BAR.... Note that MYAPP_ must prefix all vars, therefore config['redis']['foo'] would be overridable by MYAPP_REDIS_FOO ... but config['myapp']['foo']['bar'] would not have a double prefix of MYAPP_MYAPP_FOO_BAR.

**Parameters**

- **section** (*str*) – The section of the configuration to pull key values from.
- **key** (*str*) – The configuration key to get the value for.

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

**Return type** unknown

**get_dict**()
Return a dict of the entire configuration.

**Returns** A dictionary of the entire config.

**Return type** dict

**get_section_dict**(*section*)
Return a dict representation of a section.

**Parameters** **section** – The section of the configuration.

**Returns** Dictionary reprisentation of the config section.

**Return type** dict

**get_sections**()
Return a list of configuration sections.

**Returns** List of sections

**Return type** list

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

**Parameters** **section** (*str*) – The section to test for.

**Returns**

**True if the configuration section exists, False** otherwise.

**Return type** bool

**keys**(*section*)
Return a list of keys within `section`.

**Parameters** **section** (*str*) – The config section

**Returns** List of keys in the `section`.

**Return type** list

**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** (*dict*) – A dictionary of configuration keys/values to merge into our existing config (self).

**Keyword Arguments** **override** (*bool*) – Whether or not to override existing values in the config.

**set**(*section*, *key*, *value*)
Set a configuration value based at `section.key`.

**Parameters**

- **section** (*str*) – The `section` of the configuration to pull key value from.
- **key** (*str*) – The configuration key to set the value at.
- **value** – The value to set.

**Returns** None

## 3.5 cement.ext.ext_daemon

Cement daemon extension module.

**class** cement.ext.ext_daemon.**Environment**(**kw)
Bases: object

This class provides a mechanism for altering the running processes environment.

Optional Arguments:

**Keyword Arguments**

- **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**

W. 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.

## 3.6 cement.ext.ext_dummy

Cement dummy extension module.

**class** cement.ext.ext_dummy.**DummyMailHandler**(***kw*)

Bases: *cement.core.mail.MailHandler*

This class implements the `cement.core.mail.IMail` interface, but is intended for use in development as no email is actually sent.

Example:

```
class MyApp(App):
    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**

Bases: object

Handler meta-data.

**label = 'dummy'**

Unique identifier for this handler

**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** – The message body to send

**Keyword Arguments**

- **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** `True` if message is sent successfully, `False` otherwise

**Return type** bool

#### Example

```
# 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**(***kw*)
Bases: *cement.core.output.OutputHandler*

This class is an internal implementation of the `cement.core.output.OutputHandlerBase` interface. It does not take any parameters on initialization, and does not actually output anything.

**class Meta**
Bases: `object`

Handler meta-data

**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.

**render**(*data*, *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.

**Keyword Arguments** **template** (*str*) – The template parameter is not used by this implementation at all.

**class** cement.ext.ext_dummy.**DummyTemplateHandler**(**args*, ***kwargs*)
Bases: *cement.core.template.TemplateHandler*

This class is an internal implementation of the `cement.core.template.TemplateHandlerBase` interface. It does not take any parameters on initialization, and does not actually render anything.

**class Meta**
Bases: `object`

Handler meta-data

**label = 'dummy'**
The string identifier of this handler.

**copy**(*src*, *dest*, *data*)
This implementation does not actually copy anything, but rather logs it to the debug facility.

**Parameters**

- **src** (*str*) – The source template directory.
- **dest** (*str*) – The destination directory.
- **data** (*dict*) – The data dictionary to render with templates.

**render** (*content*, *data*, **args*, ***kw*)
This implementation does not actually render anything, but rather logs it to the debug facility.

**Parameters**

- **content** (*str*) – The content to render as dictionary
- **data** (*dict*) – The data dictionary to render.

## 3.7 cement.ext.ext_generate

Cement generate extension module.

**class** cement.ext.ext_generate.**Generate** (**args*, ***kw*)
Bases: *cement.ext.ext_argparse.ArgparseController*

**_setup** (*app*)
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** (*instance*) – The application object.

**class** cement.ext.ext_generate.**GenerateTemplateAbstractBase** (**args*, ***kw*)
Bases: *cement.ext.ext_argparse.ArgparseController*

## 3.8 cement.ext.ext_jinja2

Cement jinja2 extension module.

**class** cement.ext.ext_jinja2.**Jinja2OutputHandler** (**args*, ***kw*)
Bases: *cement.core.output.OutputHandler*

This class implements the *OutputHandler* interface. It provides text output from template and uses the Jinja2 Templating Language. Please see the developer documentation on Output Handling.

**class Meta**
Bases: `object`

Handler meta-data.

**_setup** (*app*)
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** (*instance*) – The application object.

**render** (*data*, *template=None*, ***kw*)
Take a data dictionary and render it using the given template file. Additional keyword arguments are ignored.

**Parameters** **data** (*dict*) – The data dictionary to render.

**Keyword Arguments** **template** (*str*) – The path to the template, after the `template_module` or `template_dirs` prefix as defined in the application.

**Returns** The rendered template text

**Return type** str

**class** cement.ext.ext_jinja2.**Jinja2TemplateHandler**(**args*, ***kw*)
Bases: *cement.core.template.TemplateHandler*

This class implements the *Template* Handler interface. It renders content as template, and supports copying entire source template directories using the Jinja2 Templating Language. Please see the developer documentation on Template Handling.

**Note** This extension has an external dependency on `jinja2`. You must include `jinja2` in your applications dependencies as Cement explicitly does **not** include external dependencies for optional extensions.

**class Meta**
Bases: `object`

Handler meta-data.

**load**(**args*, ***kw*)
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** (*str*) – The secondary path of the template **after** either `template_module` or `template_dirs` prefix (set via `App.Meta`)

**Returns** The content of the template (`str`), the type of template (`str`: `directory`, or `module`), and the path (`str`) of the directory or module)

**Return type** tuple

**Raises** *cement.core.exc.FrameworkError* – If the template does not exist in either the `template_module` or `template_dirs`.

**render**(*content*, *data*, **args*, ***kw*)
Render the given `content` as template with the `data` dictionary.

**Parameters**

- **content** (*str*) – The template content to render.
- **data** (*dict*) – The data dictionary to render.

**Returns** The rendered template text

**Return type** str

## 3.9 cement.ext.ext_json

Cement json extension module.

**class** cement.ext.ext_json.**JsonConfigHandler**(**args*, ***kw*)
Bases: *cement.ext.ext_configparser.ConfigParserConfigHandler*

This class implements the *Config* Handler interface, and provides the same functionality of *ConfigParserConfigHandler* but with JSON configuration files.

**class Meta**
Bases: `object`

Handler meta-data.

**json_module = 'json'**
Backend JSON library module to use (*json*, *ujson*).

**_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** (*str*) – The file system path to the JSON configuration
- **file.** –

**Returns** bool

**_setup**(*app*)
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** (*instance*) – The application object.

**class** cement.ext.ext_json.**JsonOutputHandler**(**args*, ***kw*)
Bases: *cement.core.output.OutputHandler*

This class implements the *Output* Handler 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**
Bases: `object`

Handler meta-data

**json_module = 'json'**
Backend JSON library module to use (*json*, *ujson*)

**label = 'json'**
The string identifier of this handler.

**overridable = False**
Whether or not to include `json` as an available choice to override the `output_handler` via command line options.

**_setup**(*app*)
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** (*instance*) – The application object.

**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** (*dict*) – The data dictionary to render.

**Keyword Arguments** **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.

## 3.10 cement.ext.ext_logging

Cement logging extension module.

**class** cement.ext.ext_logging.**LoggingLogHandler**(**args*, ***kw*)
Bases: *cement.core.log.LogHandler*

This class is an implementation of the *Log* interface, and sets up the logging facility using the standard Python logging module.

**class Meta**
Bases: object

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 = {'file': None, 'level': 'INFO', 'max_bytes': 512000, 'max_file**
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**
alias of logging.Formatter

**label = 'logging'**
The string identifier of this handler.

**log_level_argument = None**
List of arguments to use for the cli options (ex: [-l, --list]). If a log-level argument is not wanted, set to None (default).

**log_level_argument_help = 'logging level'**
The help description for the log level argument

**namespace = None**
The logging namespace.

Note: Although Meta.namespace defaults to None, Cement will set this to the application label (App.Meta.label) if not set during setup.

**_setup**(*app_obj*)
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** (*instance*) – The application object.

**_setup_console_log**()
Add a console log handler.

**_setup_file_log**()
Add a file log handler.

**clear_loggers**(*namespace*)
Clear any previously configured loggers for `namespace`.

**debug**(*msg*, *namespace=None*, ***kw*)
Log to the DEBUG facility.

**Parameters** **msg** (*str*) – The message to log.

**Keyword Arguments** **namespace** (*str*) – A log prefix, generally the module `__name__` that the log is coming from. Will default to `self._meta.namespace` if none is passed.

**Other Parameters** **kwargs** – Keyword arguments are passed on to the backend logging system.

**error**(*msg*, *namespace=None*, ***kw*)
Log to the ERROR facility.

**Args** msg (str): The message to log.

**Keyword Arguments** **namespace** (*str*) – A log prefix, generally the module `__name__` that the log is coming from. Will default to `self._meta.namespace` if none is passed.

**Other Parameters** **kwargs** – Keyword arguments are passed on to the backend logging system.

**fatal**(*msg*, *namespace=None*, ***kw*)
Log to the FATAL (aka CRITICAL) facility.

**Parameters** **msg** (*str*) – The message to log.

**Keyword Arguments** **namespace** (*str*) – A log prefix, generally the module `__name__` that the log is coming from. Will default to `self._meta.namespace` if none is passed.

**Other Parameters** **kwargs** – Keyword arguments are passed on to the backend logging system.

**get_level**()
Returns the current log level.

**info**(*msg*, *namespace=None*, ***kw*)
Log to the INFO facility.

**Parameters** **msg** (*str*) – The message to log.

**Keyword Arguments** **namespace** (*str*) – A log prefix, generally the module `__name__` that the log is coming from. Will default to `self._meta.namespace` if none is passed.

**Other Parameters** **kwargs** – Keyword arguments are passed on to the backend logging system.

**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.

**warning**(*msg*, *namespace=None*, ***kw*)
Log to the WARNING facility.

**Parameters** **msg** (*str*) – The message to log.

**Keyword Arguments** **namespace** (*str*) – A log prefix, generally the module `__name__` that the log is coming from. Will default to `self._meta.namespace` if none is passed.

**Other Parameters** **kwargs** – Keyword arguments are passed on to the backend logging system.

## 3.11 cement.ext.ext_memcached

Cement memcached extension module.

**class** cement.ext.ext_memcached.**MemcachedCacheHandler**(**args*, ***kw*)
Bases: *cement.core.cache.CacheHandler*

This class implements the *Cache* Handler 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**
Bases: `object`

Handler meta-data.

**_config**(*key*)
This is a simple wrapper, and is equivalent to: `self.app.config.get('cache.memcached', <key>)`.

**Parameters** **key** (*str*) – The key to get a config value from the 'cache.memcached' config section.

**Returns** The value of the given key.

**Return type** unknown

**_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`

**_setup**(**args*, ***kw*)
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** (*instance*) – The application object.

**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** (*str*) – The key to delete from the cache.

**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** (*str*) – The key of the item in the cache to get.

**Keyword Arguments** **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.

**Return type** unknown

**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.

**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** (*str*) – The key of the item in the cache to set.
- **value** – The value of the item to set.

**Keyword Arguments** **time** (*int*) – The expiration time (in seconds) to keep the item cached. Defaults to *expire_time* as defined in the applications configuration.

## 3.12 cement.ext.ext_mustache

Cement mustache extension module.

**class** cement.ext.ext_mustache.**MustacheOutputHandler** (**args*, ***kw*)
Bases: *cement.core.output.OutputHandler*

This class implements the *Output* Handler 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**
Bases: `object`

Handler meta-data.

**overridable = False**
Whether or not to include `mustache` as an available to choice to override the `output_handler` via command line options.

**_setup** (*app*)
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** (*instance*) – The application object.

**render** (*data*, *template=None*, ***kw*)
Take a data dictionary and render it using the given template file. Additional keyword arguments passed to `stache.render()`.

**Parameters** **data** (*dict*) – The data dictionary to render.

**Keyword Arguments** **template** (*str*) – The path to the template, after the `template_module` or `template_dirs` prefix as defined in the application.

**Returns** The rendered template text

**Return type** str

**class** cement.ext.ext_mustache.**MustacheTemplateHandler**(**args*, ***kw*)

Bases: *cement.core.template.TemplateHandler*

This class implements the *Template* Handler interface. It renderd content as template, and supports copying entire source template directories using the Mustache Templating Language. Please see the developer documentation on Template 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**

Bases: `object`

Handler meta-data.

**render**(*content*, *data*)

Render the given `content` as template with the `data` dictionary.

**Parameters**

- **content** (*str*) – The template content to render.
- **data** (*dict*) – The data dictionary to render.

**Returns** The rendered template text

**Return type** str

## 3.13 cement.ext.ext_plugin

Cement plugin extension module.

**class** cement.ext.ext_plugin.**CementPluginHandler**

Bases: *cement.core.plugin.PluginHandler*

This class is an internal implementation of the *IPlugin* interface. It does not take any parameters on initialization.

**class Meta**

Bases: `object`

Handler meta-data.

**label = 'cement'**

The string identifier for this class.

**_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** – 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

**Return type** bool

**Raises** `ImportError` – If the plugin can not be imported

**_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** (*str*) – The name of the plugin.
- **plugin_dir** (*str*) – The filesystem directory path where the plugin exists.

**_setup**(*app_obj*)
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** (*instance*) – The application object.

**get_disabled_plugins**()
List of disabled plugins

**get_enabled_plugins**()
List of plugins that are enabled (not necessary loaded yet).

**get_loaded_plugins**()
List of plugins that have been loaded.

**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 Python module determined by `App.Meta.plugin_module`.

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* – If the plugin can not be loaded

**load_plugins**(*plugin_list*)
Load a list of plugins. Each plugin name is passed to `self.load_plugin()`.

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

## 3.14 cement.ext.ext_print

Cement print extension module.

**class** cement.ext.ext_print.**PrintDictOutputHandler**(***kw*)
Bases: *cement.core.output.OutputHandler*

This class implements the *Output* Handler interface. It is intended primarily for development where printing out a string reprisentation of the data dictionary would be useful. Please see the developer documentation on Output Handling.

**class Meta**
Bases: object

Handler meta-data

**label = 'print_dict'**
The string identifier of this handler.

**overridable = False**
Whether or not to include json as an available choice to override the output_handler via command line options.

**render**(*data_dict*, *template=None*, ***kw*)
Take a data dictionary and render it as text output. Note that the template option is received here per the interface, however this handler just ignores it.

**Parameters** **data_dict** (*dict*) – The data dictionary to render.

**Keyword Arguments** **template** – This option is completely ignored.

**Returns** A text string.

**Return type** str

**class** cement.ext.ext_print.**PrintOutputHandler**(***kw*)
Bases: *cement.core.output.OutputHandler*

This class implements the *Output* Handler interface. It takes a dict and only prints out the out key. It is primarily used by the app.print() extended function in order to replace print() so that framework features like pre_render and post_render hooks are honored. Please see the developer documentation on Output Handling.

**class Meta**
Bases: object

Handler meta-data

**label = 'print'**
The string identifier of this handler.

**overridable = False**
Whether or not to include json as an available choice to override the output_handler via command line options.

**render**(*data_dict*, *template=None*, ***kw*)
Take a data dictionary and render only the out key as text output. Note that the template option is received here per the interface, however this handler just ignores it.

**Parameters** **data_dict** (*dict*) – The data dictionary to render.

**Keyword Arguments** **template** – This option is completely ignored.

**Returns** A text string.

**Return type** str

## 3.15 cement.ext.ext_redis

Cement redis extension module.

**class** cement.ext.ext_redis.**RedisCacheHandler**(**args*, ***kw*)
Bases: *cement.core.cache.CacheHandler*

This class implements the *Cache* Handler interface. It provides a caching interface using the redis library.

**Note** This extension has an external dependency on `redis`. You must include `redis` in your applications dependencies as Cement explicitly does *not* include external dependencies for optional extensions.

**class Meta**
Bases: `object`

Handler meta-data.

**_config**(*key*, *default=None*)
This is a simple wrapper, and is equivalent to: `self.app.config.get('cache.redis', <key>)`.

**Parameters** **key** (*str*) – The key to get a config value from the `cache.redis` config section.

**Returns** The value of the given key.

**Return type** unknown

**_setup**(**args*, ***kw*)
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** (*instance*) – The application object.

**delete**(*key*, ***kw*)
Delete an item from the cache for the given `key`. Additional keyword arguments are ignored.

**Parameters** **key** (*str*) – The key to delete from the cache.

**get**(*key*, *fallback=None*, ***kw*)
Get a value from the cache. Additional keyword arguments are ignored.

**Parameters** **key** (*str*) – The key of the item in the cache to get.

**Keyword Arguments** **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.

**Return type** unknown

**purge**(***kw*)
Purge the entire cache, all keys and values will be lost. Any additional keyword arguments will be passed directly to the redis `flush_all()` function.

**set**(*key*, *value*, *time=None*, ***kw*)
Set a value in the cache for the given `key`. Additional keyword arguments are ignored.

**Parameters**

- **key** (*str*) – The key of the item in the cache to set.
- **value** – The value of the item to set.
- **time** (*int*) – The expiration time (in seconds) to keep the item cached. Defaults to `expire_time` as defined in the applications configuration.

## 3.16 cement.ext.ext_scrub

Cement scrub extension module.

**class** cement.ext.ext_scrub.**ScrubController**(**args*, ***kw*)
Bases: *cement.ext.ext_argparse.ArgparseController*

Add embedded options to the base controller to support scrubbing output.

**_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`.

#### Example

```
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')
```

## 3.17 cement.ext.ext_smtp

Cement smtp extension module.

**class** cement.ext.ext_smtp.**SMTPMailHandler**(***kw*)
Bases: *cement.core.mail.MailHandler*

This class implements the *IMail* interface, and is based on the smtplib standard library.

**class Meta**
Bases: object

Handler meta-data.

**config_defaults = {'auth': False, 'bcc': [], 'cc': [], 'from_addr': 'noreply@lo**
Configuration default values

**label = 'smtp'**
Unique identifier for this handler

**send**(*body*, ***kw*)
Send an email message via SMTP. Keyword arguments override configuration defaults (cc, bcc, etc).

**Parameters** **body** – The message body to send

**Keyword Arguments**

- **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** `True` if message is sent successfully, `False` otherwise

**Return type** bool

#### Example

```
# 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',
    )
```

## 3.18 cement.ext.ext_tabulate

Cement tabulate extension module.

**class** cement.ext.ext_tabulate.**TabulateOutputHandler**(***kw*)

Bases: *cement.core.output.OutputHandler*

This class implements the *Output* Handler 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**

Bases: `object`

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.

**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.

**render**(*data*, ***kw*)
Take a data dictionary and render it into a table. Additional keyword arguments are passed directly to `tabulate.tabulate`.

**Parameters** **data_dict** (*dict*) – The data dictionary to render.

**Returns** The rendered template text

**Return type** str

## 3.19 cement.ext.ext_yaml

Cement yaml extension module.

**class** cement.ext.ext_yaml.**YamlConfigHandler**(**args*, ***kw*)
Bases: *cement.ext.ext_configparser.ConfigParserConfigHandler*

This class implements the *Config* Handler 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.

Due to changes in pyYaml version 5.1 to deprecate *yaml.load* without specifying a *Loader=...*, this class will attempt to parse the yaml content with the 5.1 sugar method *full_load*, falling back to the "unsafe" call for versions prior to 5.1. The *full_load* method uses the FullLoader, which is the default Loader when none is provided. See the pyYaml message on this deprecation: https://msg.pyyaml.org/load

**_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** (*str*) – The file system path to the Yaml configuration file.

**class** cement.ext.ext_yaml.**YamlOutputHandler**(**args*, ***kw*)
Bases: *cement.core.output.OutputHandler*

This class implements the *Output* Handler 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**
Bases: `object`

Handler meta-data.

**overridable = False**
Whether or not to include `yaml` as an available choice to override the `output_handler` via command line options.

**_setup**(*app_obj*)
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** (*instance*) – The application object.

**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** (*dict*) – The data dictionary to render.

**Keyword Arguments** **template** (*str*) – 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.

## 3.20 cement.ext.ext_watchdog

Cement watchdog extension module.

**class** cement.ext.ext_watchdog.**WatchdogEventHandler**(*app*, **args*, ***kw*)
Bases: `watchdog.events.FileSystemEventHandler`

Default event handler used by Cement, that logs all events to the application's debug log. Additional `*args` and `**kwargs` are passed to the parent class.

**Parameters** **app** – The application object

**on_any_event**(*event*)
Catch-all event handler.

**Parameters** **event** (`FileSystemEvent`) – The event object representing the file system event.

**class** cement.ext.ext_watchdog.**WatchdogManager**(*app*, **args*, ***kw*)
Bases: *cement.core.meta.MetaMixin*

The manager class that is attached to the application object via `App.extend()`.

Usage:

```
with MyApp() as app:
    app.watchdog.start()
    app.watchdog.stop()
    app.watchdog.join()
```

**add**(*path*, *event_handler=None*, *recursive=True*)
Add a directory path and event handler to the observer.

**Parameters** **path** (*str*) – A directory path to monitor (str)

**Keyword Arguments**

- **event_handler** (*class*) – An event handler class used to handle events for path (class)
- **recursive** (*bool*) – Whether to monitor the path recursively

**Returns** True if the path is added, False otherwise.

**Return type** bool

**join**(**args*, ***kw*)
Join the observer with the parent process. All *args and **kwargs are passed down to the backend observer.

**start**(**args*, ***kw*)
Start the observer. All *args and **kwargs are passed down to the backend observer.

**stop**(**args*, ***kw*)
Stop the observer. All *args and **kwargs are passed down to the backend observer.

# Python Module Index

## C


cement.core.arg, 3
cement.core.backend, 4
cement.core.cache, 4
cement.core.config, 5
cement.core.controller, 7
cement.core.exc, 8
cement.core.extension, 8
cement.core.foundation, 10
cement.core.handler, 19
cement.core.hook, 22
cement.core.interface, 24
cement.core.log, 26
cement.core.mail, 26
cement.core.meta, 28
cement.core.output, 28
cement.core.plugin, 31
cement.core.template, 29
cement.ext.ext_alarm, 43
cement.ext.ext_argparse, 43
cement.ext.ext_colorlog, 48
cement.ext.ext_configparser, 49
cement.ext.ext_daemon, 50
cement.ext.ext_dummy, 52
cement.ext.ext_generate, 55
cement.ext.ext_jinja2, 55
cement.ext.ext_json, 56
cement.ext.ext_logging, 58
cement.ext.ext_memcached, 60
cement.ext.ext_mustache, 61
cement.ext.ext_plugin, 62
cement.ext.ext_print, 63
cement.ext.ext_redis, 64
cement.ext.ext_scrub, 65
cement.ext.ext_smtp, 66
cement.ext.ext_tabulate, 67
cement.ext.ext_watchdog, 69
cement.ext.ext_yaml, 68
cement.utils.fs, 33
cement.utils.misc, 40
cement.utils.shell, 35
cement.utils.test, 42

# Index

## Symbols


_config() (*cement.ext.ext_memcached.MemcachedCacheHandler method*), 60
_config() (*cement.ext.ext_redis.RedisCacheHandler method*), 65
_dispatch() (*cement.core.controller.ControllerInterface method*), 8
_dispatch() (*cement.ext.ext_argparse.ArgparseController method*), 46
_fix_hosts() (*cement.ext.ext_memcached.MemcachedCacheHandler method*), 60
_get_exposed_commands() (*cement.ext.ext_argparse.ArgparseController method*), 46
_lay_cement() (*cement.core.foundation.App method*), 16
_load_plugin_from_bootstrap() (*cement.ext.ext_plugin.CementPluginHandler method*), 62
_load_plugin_from_dir() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
_parse_file() (*cement.core.config.ConfigHandler method*), 5
_parse_file() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 49
_parse_file() (*cement.ext.ext_json.JsonConfigHandler method*), 56
_parse_file() (*cement.ext.ext_yaml.YamlConfigHandler method*), 68
_post_argument_parsing() (*cement.ext.ext_argparse.ArgparseController method*), 46
_pre_argument_parsing() (*cement.ext.ext_argparse.ArgparseController method*), 47
_pre_argument_parsing() (*cement.ext.ext_scrub.ScrubController method*), 66
_setup() (*cement.core.handler.Handler method*), 20
_setup() (*cement.core.mail.MailHandler method*), 27
_setup() (*cement.ext.ext_generate.Generate method*), 55
_setup() (*cement.ext.ext_jinja2.Jinja2OutputHandler method*), 55
_setup() (*cement.ext.ext_json.JsonConfigHandler method*), 57
_setup() (*cement.ext.ext_json.JsonOutputHandler method*), 57
_setup() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
_setup() (*cement.ext.ext_memcached.MemcachedCacheHandler method*), 60
_setup() (*cement.ext.ext_mustache.MustacheOutputHandler method*), 61
_setup() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
_setup() (*cement.ext.ext_redis.RedisCacheHandler method*), 65
_setup() (*cement.ext.ext_yaml.YamlOutputHandler method*), 68
_setup_console_log() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
_setup_file_log() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
_validate() (*cement.core.handler.Handler method*), 20
_validate() (*cement.core.interface.Interface method*), 24
_validate() (*cement.ext.ext_argparse.ArgparseController method*), 47
_write_pid_file() (*cement.ext.ext_daemon.Environment method*), 51

## A


abspath() (*in module cement.utils.fs*), 34
add() (*cement.ext.ext_watchdog.WatchdogManager method*), 70
add_arg() (*cement.core.foundation.App method*), 16
add_argument() (*cement.core.arg.ArgumentInterface method*), 3
add_argument() (*cement.ext.ext_argparse.ArgparseArgumentHandler method*), 44
add_config_dir() (*cement.core.foundation.App method*), 16
add_config_file() (*cement.core.foundation.App method*), 16
add_handler_override_options() (*in module cement.core.foundation*), 19
add_plugin_dir() (*cement.core.foundation.App method*), 16
add_section() (*cement.core.config.ConfigInterface method*), 6
add_section() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 49
add_template_dir() (*cement.core.foundation.App method*), 16
AlarmManager (*class in cement.ext.ext_alarm*), 43
aliases (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
alternative_module_mapping (*cement.core.foundation.App.Meta attribute*), 10
App (*class in cement.core.foundation*), 10
App.Meta (*class in cement.core.foundation*), 10
ArgparseArgumentHandler (*class in cement.ext.ext_argparse*), 43
ArgparseArgumentHandler.Meta (*class in cement.ext.ext_argparse*), 43
ArgparseController (*class in cement.ext.ext_argparse*), 44
ArgparseController.Meta (*class in cement.ext.ext_argparse*), 45
argument_formatter (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
argument_handler (*cement.core.foundation.App.Meta attribute*), 10
ArgumentHandler (*class in cement.core.arg*), 3
ArgumentInterface (*class in cement.core.arg*), 3
ArgumentInterface.Meta (*class in cement.core.arg*), 3
arguments (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
argv (*cement.core.foundation.App attribute*), 16
argv (*cement.core.foundation.App.Meta attribute*), 10
auto (*cement.utils.shell.Prompt.Meta attribute*), 36


## B


backup() (*in module cement.utils.fs*), 34
bootstrap (*cement.core.foundation.App.Meta attribute*), 10


## C


cache_handler (*cement.core.foundation.App.Meta attribute*), 11
CacheHandler (*class in cement.core.cache*), 4
CacheInterface (*class in cement.core.cache*), 4
CacheInterface.Meta (*class in cement.core.cache*), 4
case_insensitive (*cement.utils.shell.Prompt.Meta attribute*), 36
catch_signal() (*cement.core.foundation.App method*), 16
catch_signals (*cement.core.foundation.App.Meta attribute*), 11
CaughtSignal, 8
cement.core.arg (*module*), 3
cement.core.backend (*module*), 4
cement.core.cache (*module*), 4
cement.core.config (*module*), 5
cement.core.controller (*module*), 7
cement.core.exc (*module*), 8
cement.core.extension (*module*), 8
cement.core.foundation (*module*), 10
cement.core.handler (*module*), 19
cement.core.hook (*module*), 22
cement.core.interface (*module*), 24
cement.core.log (*module*), 26
cement.core.mail (*module*), 26
cement.core.meta (*module*), 28
cement.core.output (*module*), 28
cement.core.plugin (*module*), 31
cement.core.template (*module*), 29
cement.ext.ext_alarm (*module*), 43
cement.ext.ext_argparse (*module*), 43
cement.ext.ext_colorlog (*module*), 48
cement.ext.ext_configparser (*module*), 49
cement.ext.ext_daemon (*module*), 50
cement.ext.ext_dummy (*module*), 52
cement.ext.ext_generate (*module*), 55
cement.ext.ext_jinja2 (*module*), 55
cement.ext.ext_json (*module*), 56
cement.ext.ext_logging (*module*), 58
cement.ext.ext_memcached (*module*), 60
cement.ext.ext_mustache (*module*), 61
cement.ext.ext_plugin (*module*), 62
cement.ext.ext_print (*module*), 63

cement.ext.ext_redis (*module*), 64
cement.ext.ext_scrub (*module*), 65
cement.ext.ext_smtp (*module*), 66
cement.ext.ext_tabulate (*module*), 67
cement.ext.ext_watchdog (*module*), 69
cement.ext.ext_yaml (*module*), 68
cement.utils.fs (*module*), 33
cement.utils.misc (*module*), 40
cement.utils.shell (*module*), 35
cement.utils.test (*module*), 42
cement_signal_handler() (*in module cement.core.foundation*), 19
CementPluginHandler (*class in cement.ext.ext_plugin*), 62
CementPluginHandler.Meta (*class in cement.ext.ext_plugin*), 62
cleanup() (*in module cement.ext.ext_daemon*), 51
clear (*cement.utils.shell.Prompt.Meta attribute*), 36
clear_command (*cement.utils.shell.Prompt.Meta attribute*), 36
clear_loggers (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
clear_loggers() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
close() (*cement.core.foundation.App method*), 16
cmd() (*in module cement.utils.shell*), 37
ColorLogHandler (*class in cement.ext.ext_colorlog*), 48
ColorLogHandler.Meta (*class in cement.ext.ext_colorlog*), 48
colors (*cement.ext.ext_colorlog.ColorLogHandler.Meta attribute*), 48
config_defaults (*cement.core.foundation.App.Meta attribute*), 11
config_defaults (*cement.core.handler.Handler.Meta attribute*), 19
config_defaults (*cement.core.mail.MailHandler.Meta attribute*), 27
config_defaults (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
config_defaults (*cement.ext.ext_colorlog.ColorLogHandler.Meta attribute*), 48
config_defaults (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
config_defaults (*cement.ext.ext_smtp.SMTPMailHandler.Meta attribute*), 66
config_dirs (*cement.core.foundation.App.Meta attribute*), 11
config_file_suffix (*cement.core.foundation.App.Meta attribute*), 11
config_files (*cement.core.foundation.App.Meta attribute*), 11
config_handler (*cement.core.foundation.App.Meta attribute*), 11
config_section (*cement.core.foundation.App.Meta attribute*), 11
config_section (*cement.core.handler.Handler.Meta attribute*), 19
config_section (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
ConfigHandler (*class in cement.core.config*), 5
ConfigInterface (*class in cement.core.config*), 6
ConfigInterface.Meta (*class in cement.core.config*), 6
ConfigParserConfigHandler (*class in cement.ext.ext_configparser*), 49
ConfigParserConfigHandler.Meta (*class in cement.ext.ext_configparser*), 49
console_format (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
ControllerHandler (*class in cement.core.controller*), 7
ControllerInterface (*class in cement.core.controller*), 7
ControllerInterface.Meta (*class in cement.core.controller*), 8
copy() (*cement.core.template.TemplateHandler method*), 29
copy() (*cement.core.template.TemplateInterface method*), 30
copy() (*cement.ext.ext_dummy.DummyTemplateHandler method*), 54
core_extensions (*cement.core.foundation.App.Meta attribute*), 12
core_handler_override_options (*cement.core.foundation.App.Meta attribute*), 12
core_interfaces (*cement.core.foundation.App.Meta attribute*), 12
core_meta_override (*cement.core.foundation.App.Meta attribute*), 12
core_system_config_dirs (*cement.core.foundation.App.Meta attribute*),

12
core_system_config_files (cement.core.foundation.App.Meta attribute), 12
core_system_plugin_dirs (cement.core.foundation.App.Meta attribute), 12
core_system_template_dirs (cement.core.foundation.App.Meta attribute), 12
core_user_config_dirs (cement.core.foundation.App.Meta attribute), 12
core_user_config_files (cement.core.foundation.App.Meta attribute), 12
core_user_plugin_dirs (cement.core.foundation.App.Meta attribute), 12
core_user_template_dirs (cement.core.foundation.App.Meta attribute), 12

## D

daemonize() (cement.ext.ext_daemon.Environment method), 51
daemonize() (in module cement.ext.ext_daemon), 51
debug (cement.core.foundation.App attribute), 17
debug (cement.core.foundation.App.Meta attribute), 12
debug() (cement.core.log.LogInterface method), 26
debug() (cement.ext.ext_logging.LoggingLogHandler method), 59
debug_argument_help (cement.core.foundation.App.Meta attribute), 12
debug_argument_options (cement.core.foundation.App.Meta attribute), 12
debug_format (cement.ext.ext_logging.LoggingLogHandler.Meta attribute), 58
default (cement.utils.shell.Prompt.Meta attribute), 36
default_func (cement.ext.ext_argparse.ArgparseController.Meta attribute), 45
define() (cement.core.hook.HookManager method), 22
define() (cement.core.interface.InterfaceManager method), 24
define_hooks (cement.core.foundation.App.Meta attribute), 12
defined() (cement.core.hook.HookManager method), 23
defined() (cement.core.interface.InterfaceManager method), 25
delete() (cement.core.cache.CacheInterface method), 4
delete() (cement.ext.ext_memcached.MemcachedCacheHandler method), 60
delete() (cement.ext.ext_redis.RedisCacheHandler method), 65
description (cement.ext.ext_argparse.ArgparseController.Meta attribute), 45
DummyMailHandler (class in cement.ext.ext_dummy), 52
DummyMailHandler.Meta (class in cement.ext.ext_dummy), 53
DummyOutputHandler (class in cement.ext.ext_dummy), 54
DummyOutputHandler.Meta (class in cement.ext.ext_dummy), 54
DummyTemplateHandler (class in cement.ext.ext_dummy), 54
DummyTemplateHandler.Meta (class in cement.ext.ext_dummy), 54

## E

ensure_dir_exists() (in module cement.utils.fs), 34
ensure_parent_dir_exists() (in module cement.utils.fs), 34
Environment (class in cement.ext.ext_daemon), 50
epilog (cement.ext.ext_argparse.ArgparseController.Meta attribute), 45
error() (cement.core.log.LogInterface method), 26
error() (cement.ext.ext_logging.LoggingLogHandler method), 59
ex (in module cement.ext.ext_argparse), 47
exec_cmd() (in module cement.utils.shell), 38
exec_cmd2() (in module cement.utils.shell), 38
exit_on_close (cement.core.foundation.App.Meta attribute), 13
expose (class in cement.ext.ext_argparse), 47
extend() (cement.core.foundation.App method), 17
extend_app() (in module cement.ext.ext_daemon), 51
extension_handler (cement.core.foundation.App.Meta attribute), 13
ExtensionHandler (class in cement.core.extension), 8
ExtensionHandler.Meta (class in cement.core.extension), 9
ExtensionInterface (class in cement.core.extension), 9
ExtensionInterface.Meta (class in cement.core.extension), 9
extensions (cement.core.foundation.App.Meta attribute), 13

## F


fatal() (*cement.core.log.LogInterface method*), 26
fatal() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
file_format (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
float_format (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67
format (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67
formatter_class (*cement.ext.ext_colorlog.ColorLogHandler.Meta attribute*), 48
formatter_class (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
formatter_class_without_color (*cement.ext.ext_colorlog.ColorLogHandler.Meta attribute*), 48
framework_logging (*cement.core.foundation.App.Meta attribute*), 13
FrameworkError, 8


## G


Generate (*class in cement.ext.ext_generate*), 55
GenerateTemplateAbstractBase (*class in cement.ext.ext_generate*), 55
get() (*cement.core.cache.CacheInterface method*), 5
get() (*cement.core.config.ConfigInterface method*), 6
get() (*cement.core.handler.HandlerManager method*), 20
get() (*cement.core.interface.InterfaceManager method*), 25
get() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 49
get() (*cement.ext.ext_memcached.MemcachedCacheHandler method*), 60
get() (*cement.ext.ext_redis.RedisCacheHandler method*), 65
get_dict() (*cement.core.config.ConfigInterface method*), 6
get_dict() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 49
get_disabled_plugins() (*cement.core.plugin.PluginInterface method*), 31
get_disabled_plugins() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
get_enabled_plugins() (*cement.core.plugin.PluginInterface method*), 31
get_enabled_plugins() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
get_level() (*cement.core.log.LogInterface method*), 26
get_level() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
get_loaded_extensions() (*cement.core.extension.ExtensionHandler method*), 9
get_loaded_plugins() (*cement.core.plugin.PluginInterface method*), 31
get_loaded_plugins() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
get_section_dict() (*cement.core.config.ConfigInterface method*), 6
get_section_dict() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 49
get_sections() (*cement.core.config.ConfigInterface method*), 6
get_sections() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 50


## H


Handler (*class in cement.core.handler*), 19
Handler.Meta (*class in cement.core.handler*), 19
handler_override() (*in module cement.core.foundation*), 19
handler_override_options (*cement.core.foundation.App.Meta attribute*), 13
HandlerManager (*class in cement.core.handler*), 20
handlers (*cement.core.foundation.App.Meta attribute*), 13
has_section() (*cement.core.config.ConfigInterface method*), 7
has_section() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 50
headers (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67
help (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
hide (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 45
HookManager (*class in cement.core.hook*), 22
hooks (*cement.core.foundation.App.Meta attribute*), 13

## I


ignore_deprecation_warnings (*cement.core.foundation.App.Meta attribute*), 13
ignore_unknown_arguments (*cement.ext.ext_argparse.ArgparseArgumentHandler.Meta attribute*), 44
info() (*cement.core.log.LogInterface method*), 26
info() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
init_defaults() (*in module cement.utils.misc*), 40
interface (*cement.core.arg.ArgumentInterface.Meta attribute*), 3
interface (*cement.core.cache.CacheInterface.Meta attribute*), 4
interface (*cement.core.config.ConfigInterface.Meta attribute*), 6
interface (*cement.core.controller.ControllerInterface.Meta attribute*), 8
interface (*cement.core.extension.ExtensionInterface.Meta attribute*), 9
interface (*cement.core.handler.Handler.Meta attribute*), 19
interface (*cement.core.interface.Interface.Meta attribute*), 24
interface (*cement.core.log.LogInterface.Meta attribute*), 26
interface (*cement.core.mail.MailInterface.Meta attribute*), 27
interface (*cement.core.output.OutputInterface.Meta attribute*), 28
interface (*cement.core.template.TemplateInterface.Meta attribute*), 30
interface (*cement.ext.ext_argparse.ArgparseArgumentHandler.Meta attribute*), 44
Interface (*class in cement.core.interface*), 24
Interface.Meta (*class in cement.core.interface*), 24
InterfaceError, 8
InterfaceManager (*class in cement.core.interface*), 24
interfaces (*cement.core.foundation.App.Meta attribute*), 13
is_true() (*in module cement.utils.misc*), 40


## J


Jinja2OutputHandler (*class in cement.ext.ext_jinja2*), 55
Jinja2OutputHandler.Meta (*class in cement.ext.ext_jinja2*), 55
Jinja2TemplateHandler (*class in cement.ext.ext_jinja2*), 55
Jinja2TemplateHandler.Meta (*class in cement.ext.ext_jinja2*), 56
join() (*cement.ext.ext_watchdog.WatchdogManager method*), 70
join() (*in module cement.utils.fs*), 34
join_exists() (*in module cement.utils.fs*), 35
json_module (*cement.ext.ext_json.JsonConfigHandler.Meta attribute*), 56
json_module (*cement.ext.ext_json.JsonOutputHandler.Meta attribute*), 57
JsonConfigHandler (*class in cement.ext.ext_json*), 56
JsonConfigHandler.Meta (*class in cement.ext.ext_json*), 56
JsonOutputHandler (*class in cement.ext.ext_json*), 57
JsonOutputHandler.Meta (*class in cement.ext.ext_json*), 57


## K


keys() (*cement.core.config.ConfigInterface method*), 7
keys() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 50


## L


label (*cement.core.extension.ExtensionHandler.Meta attribute*), 9
label (*cement.core.foundation.App.Meta attribute*), 14
label (*cement.core.handler.Handler.Meta attribute*), 19
label (*cement.ext.ext_argparse.ArgparseArgumentHandler.Meta attribute*), 44
label (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46
label (*cement.ext.ext_colorlog.ColorLogHandler.Meta attribute*), 48
label (*cement.ext.ext_configparser.ConfigParserConfigHandler.Meta attribute*), 49
label (*cement.ext.ext_dummy.DummyMailHandler.Meta attribute*), 53
label (*cement.ext.ext_dummy.DummyOutputHandler.Meta attribute*), 54
label (*cement.ext.ext_dummy.DummyTemplateHandler.Meta attribute*), 54
label (*cement.ext.ext_json.JsonOutputHandler.Meta attribute*), 57
label (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
label (*cement.ext.ext_plugin.CementPluginHandler.Meta attribute*), 62
label (*cement.ext.ext_print.PrintDictOutputHandler.Meta attribute*), 64
label (*cement.ext.ext_print.PrintOutputHandler.Meta attribute*), 64
label (*cement.ext.ext_smtp.SMTPMailHandler.Meta attribute*), 66

last_rendered (*cement.core.foundation.App attribute*), 17
list() (*cement.core.extension.ExtensionHandler method*), 9
list() (*cement.core.handler.HandlerManager method*), 20
list() (*cement.core.hook.HookManager method*), 23
list() (*cement.core.interface.InterfaceManager method*), 25
load() (*cement.core.template.TemplateHandler method*), 29
load() (*cement.core.template.TemplateInterface method*), 30
load() (*cement.ext.ext_jinja2.Jinja2TemplateHandler method*), 56
load_extension() (*cement.core.extension.ExtensionHandler method*), 9
load_extension() (*cement.core.extension.ExtensionInterface method*), 9
load_extensions() (*cement.core.extension.ExtensionHandler method*), 9
load_extensions() (*cement.core.extension.ExtensionInterface method*), 9
load_plugin() (*cement.core.plugin.PluginInterface method*), 31
load_plugin() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
load_plugins() (*cement.core.plugin.PluginInterface method*), 31
load_plugins() (*cement.ext.ext_plugin.CementPluginHandler method*), 63
log_handler (*cement.core.foundation.App.Meta attribute*), 14
log_level_argument (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
log_level_argument_help (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
LoggingLogHandler (*class in cement.ext.ext_logging*), 58
LoggingLogHandler.Meta (*class in cement.ext.ext_logging*), 58
LogHandler (*class in cement.core.log*), 26
LogInterface (*class in cement.core.log*), 26
LogInterface.Meta (*class in cement.core.log*), 26

## M

mail_handler (*cement.core.foundation.App.Meta attribute*), 14
MailHandler (*class in cement.core.mail*), 26
MailHandler.Meta (*class in cement.core.mail*), 27
MailInterface (*class in cement.core.mail*), 27
MailInterface.Meta (*class in cement.core.mail*), 27
max_attempts (*cement.utils.shell.Prompt.Meta attribute*), 37
max_attempts_exception (*cement.utils.shell.Prompt.Meta attribute*), 37
MemcachedCacheHandler (*class in cement.ext.ext_memcached*), 60
MemcachedCacheHandler.Meta (*class in cement.ext.ext_memcached*), 60
merge() (*cement.core.config.ConfigInterface method*), 7
merge() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 50
Meta (*class in cement.core.meta*), 28
meta_defaults (*cement.core.foundation.App.Meta attribute*), 14
meta_override (*cement.core.foundation.App.Meta attribute*), 14
MetaMixin (*class in cement.core.meta*), 28
minimal_logger() (*in module cement.utils.misc*), 41
missing_value (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67
MustacheOutputHandler (*class in cement.ext.ext_mustache*), 61
MustacheOutputHandler.Meta (*class in cement.ext.ext_mustache*), 61
MustacheTemplateHandler (*class in cement.ext.ext_mustache*), 62
MustacheTemplateHandler.Meta (*class in cement.ext.ext_mustache*), 62

## N

namespace (*cement.ext.ext_logging.LoggingLogHandler.Meta attribute*), 58
numbered (*cement.utils.shell.Prompt.Meta attribute*), 37
numeric_alignment (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67

## O

on_any_event() (*cement.ext.ext_watchdog.WatchdogEventHandler method*), 69
options (*cement.utils.shell.Prompt.Meta attribute*), 37

options_separator (*cement.utils.shell.Prompt.Meta attribute*), 37
output_handler (*cement.core.foundation.App.Meta attribute*), 14
OutputHandler (*class in cement.core.output*), 28
OutputInterface (*class in cement.core.output*), 28
OutputInterface.Meta (*class in cement.core.output*), 28
overridable (*cement.core.handler.Handler.Meta attribute*), 19
overridable (*cement.ext.ext_dummy.DummyOutputHandler.Meta attribute*), 54
overridable (*cement.ext.ext_json.JsonOutputHandler.Meta attribute*), 57
overridable (*cement.ext.ext_mustache.MustacheOutputHandler.Meta attribute*), 61
overridable (*cement.ext.ext_print.PrintDictOutputHandler.Meta attribute*), 64
overridable (*cement.ext.ext_print.PrintOutputHandler.Meta attribute*), 64
overridable (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67
overridable (*cement.ext.ext_yaml.YamlOutputHandler.Meta attribute*), 68

## P

padding (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 67
pargs (*cement.core.foundation.App attribute*), 17
parse() (*cement.core.arg.ArgumentInterface method*), 4
parse() (*cement.ext.ext_argparse.ArgparseArgumentHandler method*), 44
parse_file() (*cement.core.config.ConfigHandler method*), 5
parse_file() (*cement.core.config.ConfigInterface method*), 7
parser_options (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46
plugin_dir (*cement.core.foundation.App.Meta attribute*), 14
plugin_dirs (*cement.core.foundation.App.Meta attribute*), 14
plugin_handler (*cement.core.foundation.App.Meta attribute*), 15
plugin_module (*cement.core.foundation.App.Meta attribute*), 15
PluginHandler (*class in cement.core.plugin*), 31
PluginInterface (*class in cement.core.plugin*), 31
plugins (*cement.core.foundation.App.Meta attribute*), 15
PrintDictOutputHandler (*class in cement.ext.ext_print*), 63
PrintDictOutputHandler.Meta (*class in cement.ext.ext_print*), 63
PrintOutputHandler (*class in cement.ext.ext_print*), 64
PrintOutputHandler.Meta (*class in cement.ext.ext_print*), 64
process_input() (*cement.utils.shell.Prompt method*), 37
Prompt (*class in cement.utils.shell*), 35
prompt() (*cement.utils.shell.Prompt method*), 37
Prompt.Meta (*class in cement.utils.shell*), 36
purge() (*cement.core.cache.CacheInterface method*), 5
purge() (*cement.ext.ext_memcached.MemcachedCacheHandler method*), 61
purge() (*cement.ext.ext_redis.RedisCacheHandler method*), 65

## Q

quiet (*cement.core.foundation.App.Meta attribute*), 15
quiet_argument_help (*cement.core.foundation.App.Meta attribute*), 15
quiet_argument_options (*cement.core.foundation.App.Meta attribute*), 15

## R

rando() (*in module cement.utils.misc*), 41
random() (*in module cement.utils.misc*), 41
RedisCacheHandler (*class in cement.ext.ext_redis*), 64
RedisCacheHandler.Meta (*class in cement.ext.ext_redis*), 65
register() (*cement.core.handler.HandlerManager method*), 21
register() (*cement.core.hook.HookManager method*), 23
registered() (*cement.core.handler.HandlerManager method*), 21
reload() (*cement.core.foundation.App method*), 17
remove() (*cement.utils.fs.Tmp method*), 33
remove_template_dir() (*cement.core.foundation.App method*), 17
render() (*cement.core.foundation.App method*), 17
render() (*cement.core.output.OutputInterface method*), 28
render() (*cement.core.template.TemplateHandler method*), 29
render() (*cement.core.template.TemplateInterface method*), 30
render() (*cement.ext.ext_dummy.DummyOutputHandler method*), 54
render() (*cement.ext.ext_dummy.DummyTemplateHandler method*), 54

render() (*cement.ext.ext_jinja2.Jinja2OutputHandler method*), 55
render() (*cement.ext.ext_jinja2.Jinja2TemplateHandler method*), 56
render() (*cement.ext.ext_json.JsonOutputHandler method*), 57
render() (*cement.ext.ext_mustache.MustacheOutputHandler method*), 61
render() (*cement.ext.ext_mustache.MustacheTemplateHandler method*), 62
render() (*cement.ext.ext_print.PrintDictOutputHandler method*), 64
render() (*cement.ext.ext_print.PrintOutputHandler method*), 64
render() (*cement.ext.ext_tabulate.TabulateOutputHandler method*), 68
render() (*cement.ext.ext_yaml.YamlOutputHandler method*), 69
resolve() (*cement.core.handler.HandlerManager method*), 21
run() (*cement.core.foundation.App method*), 18
run() (*cement.core.hook.HookManager method*), 23
run_forever() (*cement.core.foundation.App method*), 18

## S

ScrubController (*class in cement.ext.ext_scrub*), 65
selection_text (*cement.utils.shell.Prompt.Meta attribute*), 37
send() (*cement.core.mail.MailInterface method*), 27
send() (*cement.ext.ext_dummy.DummyMailHandler method*), 53
send() (*cement.ext.ext_smtp.SMTPMailHandler method*), 66
set() (*cement.core.cache.CacheInterface method*), 5
set() (*cement.core.config.ConfigInterface method*), 7
set() (*cement.ext.ext_alarm.AlarmManager method*), 43
set() (*cement.ext.ext_configparser.ConfigParserConfigHandler method*), 50
set() (*cement.ext.ext_memcached.MemcachedCacheHandler method*), 61
set() (*cement.ext.ext_redis.RedisCacheHandler method*), 65
set_level() (*cement.core.log.LogInterface method*), 26
set_level() (*cement.ext.ext_logging.LoggingLogHandler method*), 59
setup() (*cement.core.foundation.App method*), 18
setup() (*cement.core.handler.HandlerManager method*), 22
signal_handler() (*cement.core.foundation.App.Meta method*), 15
SMTPMailHandler (*class in cement.ext.ext_smtp*), 66
SMTPMailHandler.Meta (*class in cement.ext.ext_smtp*), 66
spawn() (*in module cement.utils.shell*), 38
spawn_process() (*in module cement.utils.shell*), 39
spawn_thread() (*in module cement.utils.shell*), 39
stacked_on (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46
stacked_type (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46
start() (*cement.ext.ext_watchdog.WatchdogManager method*), 70
stop() (*cement.ext.ext_alarm.AlarmManager method*), 43
stop() (*cement.ext.ext_watchdog.WatchdogManager method*), 70
string_alignment (*cement.ext.ext_tabulate.TabulateOutputHandler.Meta attribute*), 68
subparser_options (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46
suppress_output_after_render() (*in module cement.ext.ext_json*), 57
suppress_output_after_render() (*in module cement.ext.ext_yaml*), 69
suppress_output_before_run() (*in module cement.ext.ext_json*), 58
suppress_output_before_run() (*in module cement.ext.ext_yaml*), 69
switch() (*cement.ext.ext_daemon.Environment method*), 51

## T

TabulateOutputHandler (*class in cement.ext.ext_tabulate*), 67
TabulateOutputHandler.Meta (*class in cement.ext.ext_tabulate*), 67
template_dir (*cement.core.foundation.App.Meta attribute*), 15
template_dirs (*cement.core.foundation.App.Meta attribute*), 15
template_handler (*cement.core.foundation.App.Meta attribute*), 16
template_module (*cement.core.foundation.App.Meta attribute*), 16
TemplateHandler (*class in cement.core.template*), 29
TemplateInterface (*class in cement.core.template*), 30
TemplateInterface.Meta (*class in cement.core.template*), 30
TestApp (*class in cement.core.foundation*), 19

text (*cement.utils.shell.Prompt.Meta attribute*), 37
title (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46
Tmp (*class in cement.utils.fs*), 33

## U

unsuppress_output_before_render() (*in module cement.ext.ext_json*), 58
unsuppress_output_before_render() (*in module cement.ext.ext_yaml*), 69
usage (*cement.ext.ext_argparse.ArgparseController.Meta attribute*), 46

## V

validate_config() (*cement.core.foundation.App method*), 18

## W

warning() (*cement.core.log.LogInterface method*), 26
warning() (*cement.ext.ext_logging.LoggingLogHandler method*), 60
WatchdogEventHandler (*class in cement.ext.ext_watchdog*), 69
WatchdogManager (*class in cement.ext.ext_watchdog*), 69
wrap() (*in module cement.utils.misc*), 41

## Y

YamlConfigHandler (*class in cement.ext.ext_yaml*), 68
YamlOutputHandler (*class in cement.ext.ext_yaml*), 68
YamlOutputHandler.Meta (*class in cement.ext.ext_yaml*), 68