Facio

Release 2.0.0

June 21, 2014

Contents

Facio is a project skeleton generation tool written in Python. It's designed to make it easy for you to build a single project template, or as many templates as you like, with one command to bootstrap a new project.

```
facio my_new_project -t my_remote_template
```

CHAPTER 1

Existing Templates

- Mike Waites Flask Template: https://github.com/mikeywaites/flask-skeleton

CHAPTER 2

Documentation

2.1 About

Facio: /fa.ki.o/ - Latin, meaning to make, do, act, perform, cause, bring about.

2.1.1 What is it?

Are you forever creating new projects? Re-creating the same standard cruft over and over? Then Facio may be for you. It allows you to create your standard project skeleton once, then easily create new projects from that standard skeleton as many times as you like.

It can be as simple or as advanced as you need it to be. You can write logic into your skeleton, store it in a `git` or `mercurial` repository, or even write hooks to be run before or after your skeleton has been built. You can also have as many templates as you like and quickly reference them by name or pick from a list.

Facio aims to solve your standard project skeleton woes.

```
facio my_new_project -t django_skeleton
```

2.1.2 Features

- Support for multiple templates
- Store templates in `git` or `mercurial` repositories
- Add template logic using `Jinja2`
- Add extra context-variables to your templates
- Ability to add before and after hooks called before or after the project is created.
- Bundled hooks include:
  - Create python virtual environments
  - Run `python setup.py install` (or develop)
  - Generate Django Secret key for usage in Django settings modules

2.1.3 License

See LICENSE file in the Git Repository.

2.1.4 Authors

See AUTHORS file in the Git Repository.

2.1.5 Special Thanks

To the amazing Tech Team at Poke London. And thanks to Jack for helping me name it (and pointing out grammatical errors). <3.

2.2 Installing

Facio can be installed on system using the standard python package installers `pip` or `easy_install`.

Note: `sudo` is used in the following commands for system wide installation.

2.2.1 Requirements

Facio is written in python, the only requirement you need is to have one of the following python versions installed.

- Python 2.6, 2.7, 3.2, 3.3

2.2.2 Pip or Easy Install

```
sudo easy_install facio
```

or

```
sudo pip install facio
```

2.2.3 Manually

```
cd /where/you/want/it/to/live
git clone git@github.com:krak3n/facio.git
cd facio
sudo python setup.py install
```

2.2.4 Verify Install

Once you have installed `Facio` using one of the above methods you can very the install by checking that the `facio` script was installed by running:

```
which facio
> /usr/local/bin/facio
```

If all went well `facio` is now available from your command line.

2.3 Quick Start

Facio is designed to simple and easy and also flexible. Here is how to use it:

2.3.1 Out of the box

Facio is a command line application, after you have installed Facio you should now have a `facio` command available on your command line. You can use it straight away without any configuration. It won't give you anything useful but you can see the basics of how Facio works.

Lets create a new project called foo:

Note: `$` denotes your shell prompt throughout this page

```
$ facio foo
```

The above command will bootstrap a very simple sample project which is bundled with Facio and just contains some HTML and CSS. It will be created in the directory in which the `facio` command was run and be called `foo`.

You should be able to open this in a web browser to see more information about Facio.

But this isn't particularly useful for building your skeleton so let's go on.

2.3.2 Your First Template (Skeleton)

We will call project skeletons templates. These templates are designed for reuse and should be kept maintained and updated as you learn new and better ways of creating your projects.

Let's keep it simple to start with by creating a simple HTML project template.

Create a new directory somewhere on your system and lets call it `html_template`. Inside it make 1 html file, you can call it whatever you like but for sanity we will refer to it as `index.html`. Now inside this file add the following:

```
<html>
    <head>
        <title>{{ PROJECT_NAME }}</title>
    </head>
    <body>
        <h1>Welcome to {{ PROJECT_NAME|upper() }}</h1>
        <p>My first Facio generated project template!</p>
    </body>
</html>
```

Now let's tell Facio to use this template: change to a new directory where you would like the project to be created and run:

```
$ facio bar -t /path/to/html_template
```

A new directory will be created called `bar` in your current working directory and inside you'll find `index.html` with the following content:

```
<html>
    <head>
        <title>bar</title>
    </head>
    <body>
        <h1>Welcome to BAR</h1>
```

```
        <p>My first Facio generated project template!</p>
    </body>
</html>
```

You'll notice that because we used one of Jinja2's builtin filters, `upper` in the `h1` tags, the project name has been capitalised.

A full list of built in Jinja2 filters can be found here.

2.4 Configuration & Command Line

Facio can be configured using a file called `~/.facio.cfg` and how Facio runs can also be defined by the command line interface, we will take a look at the command line first.

2.4.1 Command Line Options

Facio is a command line application. Here is how you use it:

Note: Throughout this document `$` represents a terminal prompt.

Required Arguments

The first argument you pass to the `facio` command must always be a project name. For example:

```
$ facio hello_world
```

If no additional optional arguments are supplied then Facio will use its default template to create the project.

Optional Arguments

Template

For Facio to use a different template than its default you must pass either of the following arguments:

- `--template | -t` or `--select | -s`

`--template` The `--template` or its short hand equivalent `-t` takes a string which can be either of the following:

- A file system path the template, for example `/home/me/template/template1`
- A `git` or `mercurial` repository, for examople:
  - `git+git@github.com:me/template.git`
  - `git+/path/to/local/template`
  - `hg+user@someremote.com:me/template`
  - `hg+/path/to/local/template`
- A template name defined in `~/.facio.cfg`, see *Configuration File*.

For example:

```
$ facio foo -t /my/local/template
$ facio bar -t git+git@github.com:krak3n/Facio-Django-Template.git
$ facio baz -t django
```

--select The `--select` or it's `-s` short hand is used for selecting a template which is defined in the `'.facio.cfg` configuration file, see the *Configuration File* section for more information on how to define multiple templates.

For example:

```
$ facio foo --select
```

You will be given a prompt asking you to choose a template, once chosen `facio` will process the selected template.

Variables

You may also need to define more variables to be used when rendering your template. You can do this using the *–vars* argument.

--vars You can add extra variables to the context using `--vars` optional argument. This argument takes a string which should contain a comma delimited list of key value pairs separated by an = operator.

For example:

```
$ facio foo -t bar --vars x=1,y=2,z=3
```

This above example would define 3 new context variables when rendering the template with the fllowing values:

- x = 1
- y = 2
- z = 3

And could be used in templates as follows:

```
<html>
    </head>
        <title>{{ PROJECT_NAME }}</title>
    </head>
    <body>
        <h1>{{ PROJECT_NAME }}</h1>
        <ul>
            <li>X = {{ x ]}</li>
            <li>Y = {{ y ]}</li>
            <li>Z = {{ z ]}</li>
        </ul>
    </body>
</html>
```

Other

--help The `--help` or `-h` will trigger the `facio` help message briefly describing all the options available to you.

--version The `--version` argument will allow to see the current version of Facio you are using.

2.4.2 Configuration File

You can also define a configuration file called `.facio.cfg`. This configuration file should live in your home directory with your other `.` (dot) files. This configuration file should be in an `ini` style format.

For example:

```
[section1]
option = value

[section2]
option = value
```

`[template]` Section

The `[template]` section allows you to define in the `.facio.cfg` file multiple templates you use on a regular basis so you can access them quickly from `facio`.

For example:

```
[template]
django = git+git@github.com:me/django-template.git
rails = git+git@github.com:me/rails-template.git
```

`[files]` Section

The `[files]` section allows you to specify files from your template to skip when copying or skip rendering by `jinja2`.

The `files` section takes 2 options:

- `copy_ignore`: A comma separated list of glob patterns of files **not** to copy, for example you might not want to copy `pyc` files or `.git` files that may be on the file system or in the repository. The default values for this are:
  - `.git`
  - `.hg`
  - `.svn`
  - `.DS_Store`
  - `Thumbs.db`
- `render_ignore`: A comma separated list of glob patterns of files **not** to render with the template engine, for example images such as `jpeg`, `gif` and `png` files. The default values for this are:
  - `*.png`
  - `*.gif`
  - `*.jpeg`
  - `*.jpg`

For example:

```
[files]
copy_ignore = .env,*.pyc
render_ignore = .coverage,*.ico
```

In addition to the defaults `facio` would not copy over any file named `.env` or any file name ending in `.pyc`. It would also not render with the template engine, in addition to the defaults, any file named `.coverage` or any file name ending in `.ico`.

2.5 Templates

Templates are the bare bones of your project with key parts where you would put things like the project name replaced with **Jinja2** template syntax.

These templates can live locally on your file system or they can live on a remote `git` repository. See *Configuration & Command Line* for more on this.

2.5.1 Basic Example

This is a basic `HTML` project template:

```
<html>
    <head>
        <title>{{ PROJECT_NAME }}</title>
    </head>
    <body>
        <h1>Hello world, I am {{ PROJECT_NAME }}</h1>
    </body>
</html>
```

In the above example `{{ PROJECT_NAME }}` will be replaced with whatever you set the project name to be on the command line, so for example: `$ facio -n foo` would result in `{{ PROJECT_NAME }}` being replaced by `foo`.

Your project can be made up of any file types, any directory structure, it will all be copied and processed.

2.5.2 Custom Variables

Of course project name is not always enough so for these situations you can send extra variables to `facio` for use in the template processing. To do this run `facio` with the `--vars` flag passing a comma separated list, for example:

```
facio hello_world --vars foo=bar,something=else
```

Basic Logic

Accessing these variables in templates is easy:

```
Hello World
foo={{ foo }}
something={{ something }}
```

As Jinja2 is used to render the templates, you can use conditions, and other Jinja2 functionality, for example:

```
{% if foo == 'bar' %}
Foo is bar
{% else %}
Foo is not bar
{% endif %}
```

See the Jinja2 Documentation.

Renaming Files / Directories

You can rename a directory and/or file by using double curly braces around the variable name, for example:

> **Warning:** Do not include spaces, use {{var_name}}.ext and not {{ var_name }}.ext

Below is a file structure of a raw template with 1 directory to be renamed and 1 file to be renamed to the content of `foo`.

```
- /path/to/template/
  - {{foo}}/
    - another.txt
  - {{foo}}.txt
  - some_file.txt
  - some_other_file.txt
```

Below is the rendered content.

```
- /path/to/template/
  - bar/
    - another.txt
  - bar.txt
  - some_file.txt
  - some_other_file.txt
```

2.6 Hooks

Facio has the ability to run hooks. Hooks are pieces of code that are run either before or after the project template is rendered.

Hooks are defined on a per project basis and are set in a file which resides in the project template itself. This file is called `.facio.hooks.yml`. It is a `YAML` formated file consisting of a `before` and an `after` list of python dotted paths to code to run. For example:

```
before:
  - path.to.foo

after:
  - path.to.bar
```

2.6.1 Bundled Hooks

Facio also has some bundled hooks you can use out of the box.

These will continue to grow and improve as Facio matures further. Currently there are only `python` related hooks.

Django Secret Key

- **Path**: `facio.hooks.django.secret`
- **Type**: before
- **Creates Context Variable**: `{{ DJANGO_SECRET_KEY }}`

For `Django` projects a secret key is need to protect your project. Django generates this for you when you run `django-admin.py startproject` for example. Facio can also do this and add a new variable to the template context.

To use this create a `.facio.hooks.yml` file at the root of your project template and add the following:

```
before:
  - facio.hooks.django.secret
```

And in your template you can use the `{{ DJANGO_SECRET_KEY }}` variable, for example the secret key would normally go in `settings.py`:

```
...
SECRET_KEY = '{{ DJANGO_SECRET_KEY }}'
..
```

Python Virtual Environment Creation

- **Path**: `facio.hooks.python.virtualenv`
- **Type**: before or after
- **Creates Context Variable**: none

Facio can automatically create a python virtual environment for your project. Add this hook in either the `before` or `after` list.

```
after:
  - facio.hooks.python.virtualenv
```

Prompts

This hook will ask for the following information with sensible defaults set, so you can just press `enter` to skip.

- **Virtual environment name**
  - The name of the virtual environment to create
  - Default: Project Name defined when running `$ facio`
- **Virtual environment path**
  - Where the virtual environment should be created on your file system
  - Default: `~/.virtualenvs`

Python Package Installation

- **Path**: `facio.hooks.python.setup`
- **Type**: after
- **Creates Context Variable**: none

This hook allows you to install your project as a python module provided your project has a `setup.py` correctly configured in the templates root directory.

Note: Since your template is required to have been processed before this hook can be run you should only define this hook in the `after` list of `~/.facio.hooks.yml`.

```
after:
  - facio.hooks.python.setup
```

Prompts

This hook will ask you for the following information with sensible defaults set so you can just press `enter` to skip.

- **Python path**
  - Path on the file system to the python executable to run `setup.py` against.
  - Default: The current python executable running `facio` or if the *Python Virtual Environment Creation* hook has been run it will be the path to the virtual environments python executable.
- **install or develop**
  - Install the package onto the python path or install it as an editable module
  - Default: `develop`

2.6.2 Custom Hooks

You can write your own hook if you need to. Your hook will need to meet the following criteria:

- Be available on your python path so it can imported
- Contain a `run` function.

Hello World

Warning: How to add your custom hook onto the python path is beyond the scope of this documentation. If your hook cannot be imported it will not work. See http://www.scotttorborg.com/python-packaging/ for a very helpful guide on python packaging.

Let's make a simple hook that prints "hello world". Create a file in your home directory a new directory called `my_hooks` and inside create 2 files:

- `__init__.py`
- `hello.py`

And add the following content into `hello.py`.

```
def run():
    print 'hello world'
```

This has created a new python module called `my_hooks` and inside we have a `hello.py` python file that can be imported containing our `run` function.

That's it, now all we need to do is get it on the python path. How to add your custom hook onto the python path is beyond the scope of this document, see http://www.scotttorborg.com/python-packaging/ for a very helpful guide on python packaging.

Accessing State

Facio has a `state` module where the current state of Facio is stored. Simply import it:

```
from facio.state import state
```

Updating Context

You can also add extra context variables in hooks for use in your template.

```
# my_hooks.foo

from facio.state import state

def run():
    state.update_context_variables({'FOO': 'bar'})
```

The hook above adds a new `FOO` context variable with the value of `bar` so you can use `{{ FOO }}` in your templates.

Accessing other hook data

You can also access returned values from other hooks that have run. This can be useful to provide sensible defaults in prompts or even to not run the currently executing hook unless another in the chain has run.

```
# my_hooks.bar

from facio.state import state

def run():
    call = state.get_hook_call('my_hooks.foo')
    if call:
        print 'my_hooks.foo has run'
```

Saving hook data

As described above hook data can be stored for use by other hooks later in the chain. To save data all you need to do is have your `run` function return something.

```
# my_hooks.baz

def run():
    return "foobar"
```

When `baz` is called the value `foobar` will be stored so later hooks can use it:

```
# my_hooks.faff

def run():
    print state.get_hook_call('my_hooks.baz')
```

This will print `"foobar"`.

Output and Prompting

If you want your hook to prompt a user for information or print out helpful colored messages you can extend from the `FacioBase` class. You can find more information about this in the `API` documentation.

```
# my_hooks.flop

from facio.base import FacioBase

class Flop(FacioBase):

    def __init__(self):
        val = self.gather('Enter a number: ')
        self.out('You entered: {0}'.format(val))

def run():
    flop = Flop()
    return flop
```

Examples

As mentioned Facio has several bundled hooks, you can use these as templates to writting your own.

2.7 API

Here you can find documentation for Facio's API.

2.7.1 facio.base

class `facio.base.`**`BaseFacio`**

`error`(*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

`gather`(*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

`out`(*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

`success`(*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning (*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

2.7.2 facio.config

class facio.config.**CommandLineInterface**
Facio

Facio is a project scaffolding tool originally developed for Django and expanded to be framework agnostic. You can use Facio to bootstrap any sort of project.

Documentation: https://facio.readthedocs.org

Usage: facio <project_name> [–template <path>|–select] [–vars <variables>]

Options: -h –help Show this help text. –version Show version. -t –template <path> Template path, can be repository link

(git+ / hg+) or a template name defined in ~/.facio.cfg.

-s –select Lists templates in ~/.facio.cfg prompting you to select a template from this list.

--vars <variables> Comma separated key=value pairs of values to be used in processing templates.

Example: facio hello_world -t git+git@github.com:you/django.git –vars foo=bar

start ()

validate_project_name (*name*)

class facio.config.**ConfigurationFile**
Load the ~/.facio.cfg ini style configuration file, providing an easily queryable dict representation of the config attributes.

error (*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather (*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

out (*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

read (*name='.facio.cfg'*)
Parse the config file using ConfigParser module.

Parameters **name** (*str*) – The file name to read in the users home dir – optional

Returns ConfirgParser or bool

success (*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning (*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

class facio.config.**Settings** (*interface*, *config*)

copy_ignore_globs ()
Returns list of of file copy ignore globs from configuration file.

Returns list

default_template_path = '/var/build/user_builds/facio/envs/latest/lib/python2.7/site-packages/facio-2.0.0-py2.7.egg/f

error (*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather (*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

get_template_path ()
Obtain the template with from the command line interface or from prompting the user to choose a template from the config file.

Returns str or bool

get_variables ()
Returns dict of variables passed into command line interface.

Returns dict

out (*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

render_ignore_globs ()
Returns list of of file render ignore globs from configuration file.

Returns list

success (*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning (*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

2.7.3 facio.state

class `facio.state.State`

error (*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather (*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

get_context_variable (*name*)
Return a specific context variable value.

Parameters **name** (*str*) – Context variable name

Returns str or None – None if name not found in var list

get_context_variables ()
Returns the current context variables at time of call.

Retutns dict

get_hook_call (*module_path*)
Returns a hook call result, else returns false if the module path is not in the hook call list.

Parameters **module_path** (*str*) – The python dotted path to the module

Returns Call result

get_project_name ()
Return the project name stored in the state.

Returns str

get_project_root ()
Return the project root, which is the current working directory plus the project name.

Returns str

get_working_directory ()
Use the `sh` library to return the current working directory using the unix command `pwd`.

Returns str

out (*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

save_hook_call (*module_path*, *result*)
Saves a hook call to state

Parameters

- **module_path** (*str*) – The python dotted path to the module

- **result** (*Anything*) – The result of the module run() function

Returns list – The call list or tuples

set_project_name(*name*)
Set the project name to the state.

Parameters **name** (*str*) – The project name from facio.config.CommandLineInterface

success(*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

update_context_variables(*dictionary*)
Update the context varaibles dict with new values.

** Usage: **

```
from facio.state import state
dictionary = {
    'bar': 'baz',
    'fib': 'fab',
}
state.update_context_variables(dictionary)
```

Parameters **dictionary** (*dict*) – Dictionary of new key values

warning(*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

2.7.4 facio.template

class `facio.template.`**Template**(*origin*)

COPY_ATTEMPT = 1

COPY_ATTEMPT_LIMIT = 5

copy(*callback=None*)
Copy template from origin path to `state.get_project_root()`.

Parameters **callback** (*function – default None*) – A callback function to be called after copy is complete

Returns bool

error(*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather(*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

get_copy_ignore_globs()
Returns ignore globs list at time of call.

Returns list

get_render_ignore_files (*files*)
Returns a list of files to ignore for rendering based on `get_render_ignore_globs` patterns.

Parameters **files** (*list*) – List of files to check against

Returns list – list of filenames

get_render_ignore_globs ()
Returns ignore globs list at time of call.

Returns list

out (*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

rename ()
Runs the two rename files and rename directories methods.

rename_direcories ()
Renames directories that are named after context variables, for example: `{{PROJECT_NAME}}`.

Returns generator

rename_files ()
Rename files that are named after context variables, for example: `{{PROJECT_NAME}}.py`

Returns generator

render ()
Reads the template and uses Jinja 2 to replace context variables with their real values.

success (*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

update_copy_ignore_globs (*globs*)
Update the ignore glob patterns to include the list provided.

** Usage: **

```
from facio.template import Template
t = Template('foo', '/path/to/foo')
globs = [
    '*.png',
    '*.gif',
]
t.update_copy_ignore_globs(globs)
```

Parameters **globs** (*list*) – A list of globs

update_render_ignore_globs (*globs*)
Update the render ignore glob patterns to include the list provided.

** Usage: **

```
from facio.template import Template
t = Template('foo', '/path/to/foo')
globs = [
    '*.png',
    '*.gif',
]
t.update_render_ignore_globs(globs)
```

Parameters **globs** (*list*) – A list of globs

warning(*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

2.7.5 facio.vcs

class facio.vcs.**BaseVCS**(*path*)
Base Version Control System Class all VCS related classes should extend from, provides common API.

clone()
This class should be overridden in VCS subclass, if not a FacioException will be raised.

error(*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather(*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

get_temp_directory()
Create a temporary directory to clone the template to.

Returns str – Temp directory path

out(*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

remove_tmp_dir(*origin*, *destination*)
Template.copy callback function to remove created temp directory.

success(*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning(*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

class `facio.vcs.GitVCS`(*path*)

Git Version Control System for cloning git repositories.

clone()

Clone the git repository into a temporary directory.

error(*message*)

Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather(*message*)

Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

get_temp_directory()

Create a temporary directory to clone the template to.

Returns str – Temp directory path

out(*message*, *color=<function blue at 0x7f1c04204f50>*)

Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

remove_tmp_dir(*origin*, *destination*)

Template.copy callback function to remove created temp directory.

success(*message*)

Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning(*message*)

Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

class `facio.vcs.MercurialVCS`(*path*)

Mercurial Version Control System for cloning hg repositories.

clone()

Clone the hg repository into a temporary directory.

error(*message*)

Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather(*message*)

Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

get_temp_directory()

Create a temporary directory to clone the template to.

Returns str – Temp directory path

out (*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

remove_tmp_dir (*origin*, *destination*)
Template.copy callback function to remove created temp directory.

success (*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning (*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

2.7.6 facio.hooks

Hook

Below is documentatin for the `facio.hooks.Hook` class.

class `facio.hooks.`**`Hook`**

error (*message*)
Print error that does not result in an exit (Red)

Parameters **message** (*str*) – Message to print to user

gather (*message*)
Common message prompting for gathering input form the end user.

Parameters **message** (*str*) – Message to print to user

has_after ()
Does the hooks file contain a after list.

Returns Bool

has_before ()
Does the hooks file contain a before list.

Returns Bool

has_run (*path*)
Has a hooks module run.

Parameters **path** (*str*) – The hooks python module path

Returns False if not run else the modules returned data

import_module (*path*)
Import module to run in before or post hooks.

Parameters **path** (*str*) – The python path to the module

load(*path*)
Parse the hooks file.

Parameters **path** (*str*) – Path to hooks file, locally

out(*message*, *color=<function blue at 0x7f1c04204f50>*)
Print message information to user (Blue)

Parameters **message** (*str*) – Message to print to user

** Optional Key Word Arguments **

Parameters **color** – Clint color function to use

Type function – default blue

run_after()
Run the after modules.

run_before()
Run the before modules.

run_module(*path*)
Run a before or after module.

Parameters **path** – Path to the module

success(*message*)
Print a success message (Green)

Parameters **message** (*str*) – Message to print to user

warning(*message*)
Print a warning message (Yellow)

Parameters **message** (*str*) – Message to print to user

django.secret

class facio.hooks.django.secret.**GenerateDjangoSecretKey**

characters = 'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)'

generate()
Generate Django secret key

Returns str – The generated key

facio.hooks.django.secret.**run**()
Called by the facio.hooks runner.

python.setup

class facio.hooks.python.setup.**Setup**

get_default_path_to_python()
Returns the default path to python, if virtualenv hooks has been called use that path, else use the current executing python, this should be the systems python in most cases.

Returns str – path to python executable

get_install_arg()
Gets the install args from the user, for example setup.py install or develop.

Returns str – The install type

get_path_to_python()
Gets the path to python to run setup.py against. Detect if the virtualenv hooks has run, if so the default path to python should come from the path to this virtual environment, else it should be the system default python path.

Returns str – The path to python

log_errors(*errors*)
Called with errors are encountered running setup.py and are logged to a setup.error.log.

Parameters **errors** (*str*) – Errors from setup.py

run()
Runs the python setup.py command.

Returns bool – Based on return code subprocess call return code

`facio.hooks.python.setup.`**run**()
Called by hooks runner, runs the setup class and returns Bool on status of the run command.

Returns bool – The state of running setup.py

python.virtualenv

class `facio.hooks.python.virtualenv.`**Virtualenv**

create()
Creates a python virtual environment.

get_name()
Returns the name for the virtualenv - gathered from user input with the default value being the project name from facio state.

Returns str – Virtual environment name

get_path()
The path to where the virtual environment should be created, the user is prompted to input this path, default will be ~/.virtualenvs.

Returns str – The path to where the virtual environment

`facio.hooks.python.virtualenv.`**run**()
Called from `facio.hooks` runner.

Returns str – Path to the created virtual environment

2.8 Contributing

Fancy helping out? Fork, commit and issue a pull request :)

I can't guarantee I will accept your pull request, but here some things which will help:

- Your code is to PEP8 standards
- Your pull request adds a useful feature or fixes a bug

- Your code has unit tests to ensure it works as it should
- Your code is documented so documentation can be auto generated

I use Git Flow to develop this project, as such the branch structure is as follows:

- Master: The current stable release, hotfixes come off this branch
- Develop: The current in development code, feature branches come off this branch
- feature/x: Feature branches should be named `feature/my_feature_name`

So please create new features from the **develop** branch. Pull requests onto master directly will **not be accepted** unless it is a hotfix.

2.8.1 Installing the Code

Note: This section assumes familiarity with python virtual environments and `virtualenvwrapper`.

First create a fork of `http://github.com/krak3n/facio` so it's in your own github account, then clone:

```
$ git clone git@github.com:you/facio.git
```

Once cloned switch to the develop branch:

```
$ git fetch --all
$ git checkout develop
```

Create a python virtual environment:

```
$ virtualenv facio --no-site-packages
$ workon facio
```

Now you can install the code as a development egg with the development dependencies, this includes everything you need to run tests and debug code.

```
$ make develop
```

Facio and it's dependencies will now be installed into your virtual environment.

2.8.2 Vagrant

I use Vagrant for development so I have bundled the facio repository with a `Vagrantfile`.

There are the following dependencies:

- Vagrant 1.1+
- Latest VirtualBox
- Vagrant Guest Additions Plugin: `vagrant plugin install vagrant-vbguest`
- Vagrant Salt Provisioner: `vagrant plugin install vagrant-salt`

Once you have all the dependencies installed it should be a simple case of running `vagrant up` at the root of the repository. Once it's finished you should have a development environment with all of the `facio` dependencies installed into a python virtual environment. All you have to do is run:

```
$ make develop
```

On the vagrant box.

2.9 Change Log

2.9.1 Version 2.0 - 1/8/2013

- Added Hook Support
- Refactored large parts of the codebase
- Updated configuration file section for a new `[files]` section
- Python 3 Support
- Mercurial Support
- Cleaned up command line interface

2.9.2 Version 1.1.1 (hotfix) - 5/4/2013

- Fixed Manifest file so bundled template is properly included.

2.9.3 Version 1.1 - 5/4/2013

- Improved output to the user
- Decoupled SCM into separate classes so its easier to add new ones in the future
- Updated bundled template
- Documentation

2.9.4 Version 1.0.1 (hotfix) - 6/12/2012

- Fixed issue where bundled default template was not provided in distribution.

2.9.5 Version 1.0 - 5/12/2012

- Decoupled Git cloning from Template Class into a separate class, laying the foundation for future SCM support.
- Created a bundled default template.

2.9.6 Version 1.0 Beta 1 - 26/11/2012

- Initial Release

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

f