colifrapy Documentation

Release 0.5.0

Yomguithereal

April 09, 2014

Colifrapy is a **Command Line Framework for Python**. Its aim is to provide several tools to build robust and structured command line tools very easily.

Its logic is very similar to a MVC framework and is therefore easy to use.

The github repository can be found there.

CHAPTER 1

Summary

1.1 Quickstart

1.1.1 Creating New Project

Having installed colifrapy, initialize a new project with the *Scaffolder* by typing into your console:

```
colifrapy new [name-of-project]

options :
    {-a/--author: Author of the project}
    {-o/--organization: Organization of the author}
```

To test your new project, enter your project's directory and type:

```
python [name-of-project].py test
```

It should launch it and output a header and some strings in the console telling you everything is going to be OK.

1.1.2 Command Line Hub

[name-of-project].py

A colifrapy project relies on a command line hub extending Colifrapy base class. This is the file you have to call in your shell to launch the program. The duty of this hub is to initialize your tool, analyze the arguments given to it and call upon the relevant controller methods.

In fact, this hub can be compared to a basic router for web frameworks.

This is the hub as generated by the scaffolder

```
# Dependencies
#=============
from colifrapy import Colifrapy
from model.controller import Controller

# Hub
#======
class NameOfYourProject(Colifrapy):

    # From this hub, you can access several things :
    #    self.settings (Settings Instance)
```

```
    #    self.log (Logger Instance)
    #    self.opts (Options passed to your hub)
    #    self.controller (Your Controller)

    def launch(self):

        # Welcoming visitors
        self.log.header('main:title')

        # Calling upon the controller
        self.controller.test()

# Launching
#===========
if __name__ == '__main__':
    # By default, the hub will load config/settings.yml
    hub = NameOfYourProject(
        controller=Controller,
        settings_path='path/to/your/settings.yml'
    )
    hub.launch()
```

Note that if you just want to use colifrapy features but don't want to be tied to its architecture, you can just use this hub which can access any critical utilities as well as any colifrapy Model would.

1.1.3 Settings

config/settings.yml

The Settings class is the first class loaded by colifrapy to perform its magic. It will parse your settings.yml file and configure your logger, cacher, arguments and every other configuration you want for your application.

```
# Basic Informations
version: '[project-name] 0.1.0'
description: 'Description of the program.'
usage: 'How to deal with your program'
arguments:
- [ ['-t', '--test'], {'help' : 'Test', 'type' : 'int'} ]
- [ ['positionnal'] ]

# Logger Settings
logger:
    strings: 'config/strings.yml'
    flavor: 'default'

# Generic Settings needed by your program
settings:
    hello: 'world'
    bonjour: 3.4
    hash: {'test' : 2}
```

Also, note that paths are automatically considered by colifrapy either as relative (config/test.yml) or absolute ones (/var/usr/test.yml).

For further information see *Settings*.

1.1.4 Arguments

config/settings.yml['arguments']

Settings Usage

Arguments are to be defined as for the python ArgParser class. In fact, the colifrapy Commander class extends the ArgParser one, so if you need complicated things not handled by colifrapy, just use the Commander class like the ArgParser one.

```
arguments:
- [ ['-t', '--test'], {'help' : 'Test', 'type' : 'int', 'default' : 5} ]
- [ ['-b', '--blue'], {'help' : 'Blue option', 'type' : 'int', 'required' : 'True'} ]
- [ ['some_positionnal_argument'] ]
```

In the command hub and in your models, you can access the options passed to your commander through self.opts . However, even if those are accessible in models for commodity, only the main hub should use them and one should restrain their usage in models.

Special Arguments

Help, Version, Verbose and Settings

As for standard python command line tool, yours will accept three default arguments you should not try to override (verbose is the only one you can override because it is not one of ArgumentParser defaults):

```
-v/--version (outputting your program's version)

-h/--help (displaying your program's help)

-V/--verbose (overriding settings to enable the logger to display every messages)

--settings (overriding settings file if needed)
```

1.1.5 Controller

model/controller.py

The controller is a class whose goal is to call upon other models. The controller itself is in fact also a colifrapy model and is more a convention that something enforced by colifrapy's code.

The controller is totally optional and just illustrate a way to organize your code. If you don't want to follow this logic, just don't pass a controller to your hub instance.

Controller as generated by the scaffolder

```
# Dependencies
#=============
from colifrapy import Model
from example_model import ExampleModel

# Main Class
#=============
class Controller(Model):

    # Properties
```

```
    example_model = None

    def __init__(self):
        self.example_model = ExampleModel()

    # Example of controller action
    def test(self):
        self.log.write('main:controller')
        self.example_model.hello()
```

1.1.6 Model

model/example_model.py

Models are the bulk of Colifrapy. You can extend them to access your settings and commands easily.

A standard model is generated for you by the Scaffolder when you create a new project.

Minimalist example of a model usage

```
from colifrapy import Model

class MyModel(Model):
    def test(self):
        print self.settings.hello

m = MyModel()
m.test()
>>> 'world'
```

Reserved attributes names are:

- **cache** (access to cache)
- **log** (access to the logger described hereafter)
- **opts** (access to the command line options)
- **settings** (access to the program's settings)

1.1.7 Logger

Basic

The logger is the outputting class of colifrapy. It should be loaded with some strings by the settings. If no strings are given, the logger will just output normally the argument strings you give to it.

For full logger documentation, see *Logger*.

Levels

The logger accepts five levels :

- INFO (green output)
- VERBOSE (cyan output)
- DEBUG (blue output)

- WARNING (yellow ouput)
- ERROR (red output)
- CRITICAL (purple output) –> will throw an exception for you to catch or not

By default, if no level is specified for a message, DEBUG will always be taken.

Strings

config/strings.yml

Colifrapy offers to externalize your strings in order to enable you to quickly modify them if needed, or even translate them easily.

The string format used is a mustache-like one, so variables come likewise : {{some_variable}}

Strings given must follow this yaml layout

```
main:
    process:

        # String with a variable contained within the mustaches
        start: 'Starting corpus analysis (path : {{path}})//INFO'

        # Simply write two slashes at the end to specify the level of the message
        end: 'Exiting//WARNING'
        test_line_break: '\nBonjour'

    title: 'Colifrapy'

other_string_category:
    test: 'Hello everyone//INFO'
    you:
        can:
            make: 'any levels that you want'
            so: 'you can organize your strings however you need.'
```

Usage

This is how you would use the logger in a colifrapy model

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # Main method
        #------------

        # Outputting a message
        self.log.write('main:process:end')
        >>> '[WARNING] :: Exiting'

        # Overriding the message level
        self.log.write('main:process:end', level='INFO')
        >>> '[INFO] :: Exiting'
```

```
        # Passing variables
        self.log.write('main:protocol:start', {'path' : 'test'})
        >>> '[INFO] :: Starting corpus analysis (path : test)'

        # Variables can be passed to the logger as:
        # a hash, a list, a tuple, a single string or integer or float

        # Examples
        self.log.write('{{variable}}', 'test')
        >>> '[DEBUG] :: test'

        self.log.write('{{var1}} is {{var2}}', ['python', 'cool'])
        >>> '[DEBUG] :: python is cool'



        # When yml file is not specified or if message does not match
        self.log.write('Test string')
        >>> '[DEBUG] :: Test string'

        # Named arguments of write
        # variables --> mixed
        # level --> log level

        # Helper methods
        #---------------

        # Printing a header (yellow color by default)
        self.log.header('main:title', [optional]color)
        >>> Colifrapy
        >>> ---------

        # Write methods shorteners
        self.log.critical(message, vars)
        self.log.error(...)
        self.log.warning(...)
        self.log.info(...)
        self.log.debug(...)
        self.log.verbose(...)
```

Asking for confirmation

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # Confirmation
        #---------------

        # 'y' will be taken by default in arg 2
        # will return True for y and False for n
        response = self.log.confirm('Are you sure you want to continue?')
        >>> 'Are you sure you want to continue? (Y/n)'
        >>> y --> True

        response = self.log.confirm('Are you sure you want to continue?', 'n')
        >>> 'Are you sure you want to continue? (y/N)'
```

```
        >>> n --> False
```

Getting user input

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # User Input
        #---------------

        response = self.log.input('What up ?')
        >>> 'What up ?'
        >>> 'feeling fine' --> 'feeling fine'

        # You can also provide a lambda to the function as second argument
        # This lambda will affect the input given
        response = self.log.input('What up ?', lambda x: x.upper())
        >>> 'What up ?'
        >>> 'feeling fine' --> 'FEELING FINE'
```

1.1.8 Cacher

Colifrapy gives you acces, in your hub and models to a caching class able to store data on files for long-term access. There are currently to types of cacher : line and yaml. The first one consist in a text file containing one line read by the cacher while the second archive any python key-value data in a yaml file.

To enable the cacher in the settings.yml file

```
cache:
    kind: 'line'
    directory: 'config'
    filename: 'last_update.txt'

    # Whether you want the cache to be written each time a value is changed
    # Defaults to False
    auto_write: True
```

Then in your model

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # Line Cacher
        #-------------

        # Setting cache
        self.cache.set("test")

        # Getting cache
        self.cache.get()
        >>> "test"

        # Yaml Cacher
        #-------------
```

```
        # Setting cache
        self.cache.set("one", "red")
        self.cache.set("two:deep", "blue")

        # Getting cache
        self.cache.get("one")
        >>> "red"

        self.cache.get("two")
        >>> {"deep" : "blue"}

        self.cache.get("two:deep")
        >>> "blue"

        self.cache.get
        >>> {"two" : "red", {"deep" : "blue"}}
```

Note that the path separator for deep levels in yaml is always ":" in Colifrapy.

For full documentation see *Cacher*.

1.2 Settings

1.2.1 Class

The Settings class' aim is to load a 'settings.yml' file containing every piece of configuration for a colifrapy program. This class is initialized by default by colifrapy's hub and is a singleton so its state won't change throughout your code.

Example of a settings.yml file (created by default in 'config/' by the Scaffolder)

```
# Basic Informations
version: '[project-name] 0.1.0'
description: 'Description of the program.'
usage: 'How to deal with your program'
arguments:
- [ ['-t', '--test'], {'help' : 'Test', 'type' : 'int'} ]
- [ ['positionnal'] ]

# Logger Settings
logger:
    strings: 'config/strings.yml'
    flavor: 'default'
    title_flavor: 'default'
    # Delete the path line not to write the log to a file
    directory: 'logs'
    threshold: ['DEBUG', 'ERROR', 'INFO', 'WARNING', 'VERBOSE']

# Generic Settings needed by your program
settings:
    hello: 'world'
    bonjour: 3.4
    hash: {'test' : 2}
```

1.2.2 Model Usage

In every class that extends colifrapy model, you can access your generic settings with the property settings. The following code assume the precedent yml file was loaded.

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        print self.settings.hello
        >>> "world"

        print self.settings.hash['test']
        >>> 2
```

1.2.3 Standalone Usage

As every colifrapy class, it is possible to use the Settings one as a standalone. The following example presents how you could do that and what you pass to it to make it work.

```
from colifrapy import Settings

# Loading the YAML file
s = Settings()
s.load('path/to/settings.yml')

# Accessing the generic settings
d = s.getDict()
print d.hello
>>> "world"

# Accessing cache
c = s.getCache()
```

Once loaded, you can use it anywhere in your code and even reinstanciate it if convenient with it keeping the same state.

1.2.4 Options

Standard

Standard program options are the following:

- **version** : Name and version of your program (outputted with -v/–version option)
- **description** : Short description of your program and what it does
- **usage** : How to use your program
- **prog** : Program's name to display along usage string.
- **epilog** : A final string to output when help is displayed.

Arguments

A good command line tool often comes with arguments, you can register those in the yaml file for convenience. Once loaded with arguments, the Settings class will load the Commander one with them.

Those are to be defined as for the python ArgParser class.

Example of argument definition (under 'arguments' key).

```
arguments:
- [ ['-t', '--test'], {'help' : 'Test', 'type' : 'int', 'default' : 5} ]
- [ ['-b', '--blue'], {'help' : 'Blue option', 'type' : 'int', 'required' : 'True'} ]
- [ ['some_positionnal_argument'] ]
```

Logger

The Logger class can be given some options through settings. If none are supplied, logger will still be initialized with its default values.

For more precise information see *Logger*.

For more precise information about the logger's styles see *Styles*.

```
logger:
    # {string} [None] YAML string file.
    strings: 'example_strings.yml'

    # {boolean} [True] Should the CRITICAL level trigger exceptions?
    exceptions: False

    # {boolean} [True] Should we activate both logger's handlers?
    activated: True

    # {string} ['VERBOSE'] Threshold for both logger's handlers.
    threshold: 'INFO'

    # {string} The log message formatter for both logger's handlers.
    formatter: '%(msg)s -- %(asctime)'

    # {string|lambda} The flavor for colored levelname.
    flavor: 'elegant'

    # Console specific options
    console:

        # {boolean} [True] Should the console handler be activated?
        activated: True

        # {string} ['VERBOSE'] Threshold.
        threshold: 'DEBUG'

        # {string} ['%(flavored_levelname)s :: %(msg)s'] Formatter.
        formatter: '%(msg)s'

    # File specific options:

        # {boolean} [False] Should the file handler be activated?
        activated: False
```

```
    # {string} ['VERBOSE'] Threshold.
    threshold: 'ERROR'

    # {string} ['%(asctime)s %(levelname)s :: %(msg)s'] Formatter.
    formatter: '%(msg)s'

    # {string} ['.'] Directory where the file handler will write.
    directory: 'logs'

    # {string} ['program.log'] Filename for the log file.
    filename: 'current.log'

    # {string} ['simple'] File logging mode
    mode: 'rotation'

    # Rotation mode options
    # {int} [1048576] Max bytes for a current log file.
    max_bytes: 2097152

    # {int} [5] Max number of log files
    backup_count: 4
```

N.B.: Options passed at the **logger** level such as *activated* or *threshold* override the **console** and **file** one and apply to both.

Cacher

If needed, the Settings class can also handle the initialization of a cacher. Just provide a 'cache' key to the settings and populate it.

For more precise information see *Cacher*.

```
cache:
    # Cache Directory
    # Default: cache
    directory: 'cache'

    # Cache filename
    # Default: 'cache.txt' for line mode and 'cache.yml' for yaml mode
    filename: 'project.log'

    # Kind of cache
    # Default: 'line' (choose between line and yaml)
    type: 'yaml'

    # Auto Write
    # Default: False
    auto_write: True
```

If you need more than one cache instance, just pass an array to the cache key in your YAML settings file. In this case, don't forget to pass a name to your settings to access it. Else it will earn a standardized name like *__cacheInstance0*.

```
cache
    - name: 'infos'
      filename: 'cache1.yml'
      type: 'yaml'

    - name: 'last_update'
```

```
      filename: 'last_update.txt'
      type: 'line'
```

Then access your cache likewise.

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        print self.cache['infos']
        print self.cache['last_update']
```

General

If you need any other settings you consider necessary, just provide a settings key to your yaml file and populate it as in the following example.

```
settings:
    mysql:
        host: localhost
        user: root
        password: foo
    to_index: ["books", "notes"]
    limit: 5
```

It is also possible to include other yaml files into those generic settings by following this procedure.

```
# Syntax is 'include::path/to/file.yml'
# Warning, will only work on the first level (not on a nested one)
settings:
    hello: 'world'
    hello2: 'include::path/to/another_config_file.yml'
```

1.2.5 N.B.

For every path given, colifrapy will try and decide whether it is absolute or relative (unix-style):

```
'/usr/local/settings.yml' is an absolute path
'config/settings.yml' is a relative path (relative to the colifrapy hub file)
```

1.3 Logger

1.3.1 Class

The Logger class is the voice of colifrapy. Its aim is to display feedback from your program into the console and to write it to a file if necessary. It may also feed on externalized strings written in a YAML file.

N.B.: this custom logger is built around the python logging module and spawn a logging instance named "colifrapy".

1.3.2 Model Usage

By default, and even if no settings were initialized before, every colifrapy model is loaded with the Logger so you may use it when convenient.

```
from colifrapy import Model

class MyModel(Model):

    # In every colifrapy model, the Logger is
    # accessible through the "log" property.

    def test(self):
        self.log.write('Hello World!')
        >>> '[DEBUG] :: Hello World!'
```

1.3.3 Standalone Usage

If you want to use the colifrapy Logger without messing with the whole framework, it is obviously possible. Note that as a lot of colifrapy classes, the Logger is actually a singleton. You may then instanciate it in different files, it will always be the same one for convenience.

```
from colifrapy import Logger

logger_instance = Logger()

# Run the configuration method at least one to initialize the logger
logger_instance.config(**options)

logger_instance.write('Hello World!')
>>> '[DEBUG] :: Hello World!'

# To change the logger's configuration, just rerun the config method
logger_instance.config(**options)

# Or for specific options
logger_instance.configConsole(options)
logger_instance.configFile(options)
```

1.3.4 Levels

The logger accepts five levels (ordered by importance):

- VERBOSE (cyan output)
- DEBUG (blue output)
- INFO (green output)
- WARNING (yellow ouput)
- ERROR (red output)
- CRITICAL (violet output) –> will throw an exception for you to catch or not

By default, if no level is specified for a message, DEBUG will always be taken.

1.3.5 Options

Colifrapy's logger has three different configuration methods, each one dealing with a particular end. You can therefore configure the logger as a whole or rather one of both its handlers (console and file).

Note that if you want to change one of those options on the fly you can always run the config method one more time with the changed options.

Generic Options

The generic options you may pass to the logger's *config* method (those options are automatically taken care of when the logger is loaded by the Settings class) are the following:

- **strings** {string} Path leading to your externalized YAML strings. *default*: None (the logger won't use externalized strings)
- **exceptions** {boolean} Should the CRITICAL level trigger exceptions. *default*: True
- **flavor** {string|lambda} The flavor to use to format %(flavored_levelname)s. *default*: 'default'
- **console_kwargs** {dict} A configuration dict to be run into the configConsole method. *default*: None (the configConsole method will be called with its defaults)
- **file_kwargs** {dict} A configuration dict to be run into the configFile method. *default*: None (the configFile method will be called with its defaults)

For a list of flavors, see *Styles*. If none of the proposed flavors suit you and you need to create your own, please note that you can pass a lambda taking the levelname variable to the flavor option

Usage example

```
from colifrapy import Logger

logger_instance = Logger()
logger_instance.config(strings='example_string.yml', exceptions=False)
```

Console Options

The console options you may pass to the logger's *configConsole* method (those options are automatically taken care of when the logger is loaded by the Settings class under logger:console) are the following:

- **activated** {boolean} Whether the console handler should be activated or not. *default*: True
- **threshold** {string} Threshold for the console handler. *default*: 'VERBOSE'
- **formatter** {string} Formatter for the console handler. *default*: '%(flavored_levelname)s :: %(msg)s'

Usage example

```
from colifrapy import Logger

logger_instance = Logger()
logger_instance.configConsole(threshold='WARNING', activated=True)
```

File Options

The console options you may pass to the logger's *configFile* method (those options are automatically taken care of when the logger is loaded by the Settings class under logger:file) are the following:

- **activated** {boolean} Whether the file handler should be activated or not. *default*: False
- **threshold** {string} Threshold for the file handler. *default*: 'VERBOSE'
- **formatter** {string} Formatter for the console handler *default*: '%(asctime)s %(levelname)s :: %(msg)s'.
- **directory** {string} Directory where the file handler is supposed to write its logs. *default*: '.'
- **filename** {string} Name of the log files. *default*: 'program.log'
- **mode** {string} File logging mode. See *Modes*. *default*: 'simple'
- **max_bytes** {integer} When in rotation mode, maximum of bytes for a log file before rotating. *default*: 1048576
- **backup_count** {integer} When in rotation mode, maximum number of archived log files. *default*: 5

Note that the file handler is not activated by default.

Usage example

```
from colifrapy import Logger

logger_instance = Logger()
logger_instance.configFile(threshold='ERROR', activated=True, mode='overwrite')
```

1.3.6 Strings

Colifrapy offers to externalize your strings in order to enable you to quickly modify them if needed, or even translate them easily. I you do not provide the logger with some strings, it will simply take normal python strings.

The string format used is a mustache-like one, so variables come likewise : {{some_variable}}

Strings given must follow this yaml layout

```
main:
    process:

        # String with a variable contained within the mustaches
        start: 'Starting corpus analysis (path : {{path}})//INFO'

        # Simply write two slashes at the end to specify the level of the message
        end: 'Exiting//WARNING'
        test_line_break: '\nBonjour'

    title: 'Colifrapy'

other_string_category:
    test: 'Hello everyone//INFO'
    you:
        can:
            make: 'any levels that you want'
            so: 'you can organize your strings however you need.'
```

1.3.7 Modes

The Logger comes with three different outputting modes:

- **simple**: it will write everything to a single specified file.
- **overwrite**: the log will be completely overwritten each time you launch the program.

- **rotation**: each time your log file overcomes a specified number of lines, it will create a new file and archive the old one. E.g. it functions like the apache log.

For more information about file rotation, you can read the python logging module's RotatingFileHandler documentation.

1.3.8 Methods

Writing

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # Main method
        #------------

        # Outputting a message
        self.log.write('main:process:end')
        >>> '[WARNING] :: Exiting'

        # Overriding the message level
        self.log.write('main:process:end', level='INFO')
        >>> '[INFO] :: Exiting'

        # Passing variables
        self.log.write('main:protocol:start', {'path' : 'test'})
        >>> '[INFO] :: Starting corpus analysis (path : test)'

        # Variables can be passed to the logger as:
        # a hash, a list, a tuple, a single string or integer or float

        # Examples
        self.log.write('{{variable}}', 'test')
        >>> '[DEBUG] :: test'

        self.log.write('{{var1}} is {{var2}}', ['python', 'cool'])
        >>> '[DEBUG] :: python is cool'

        # When yml string file is not specified or if message does not exist in the yaml file
        self.log.write('Test string')
        >>> '[DEBUG] :: Test string'

        # Named arguments of write
        # variables --> mixed
        # level --> log level



        # Helper methods
        #---------------

        # Printing a header
        self.log.header('main:title', [optional]flavor='default')
        >>> Colifrapy
        >>> ---------
```

```
        # You can also pass a function as the title flavor rather
        # than a predetermined one.
        self.log.header('main:title', flavor=lambda msg: msg.upper())
        >>> COLIFRAPY

        # Write methods shorteners
        self.log.critical(message, vars)
        self.log.error(...)
        self.log.warning(...)
        self.log.info(...)
        self.log.debug(...)
        self.log.verbose(...)
```

Confirmation

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # Confirmation
        #---------------

        # 'y' will be taken by default in arg 2
        # will return True for y and False for n
        response = self.log.confirm('Are you sure you want to continue?')
        >>> 'Are you sure you want to continue? (Y/n)'
        >>> y --> True

        response = self.log.confirm('Are you sure you want to continue?', 'n')
        >>> 'Are you sure you want to continue? (y/N)'
        >>> n --> False
```

User Input

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # User Input
        #---------------

        response = self.log.input('What up ?')
        >>> 'What up ?'
        >>> 'feeling fine
        >>> 'feeling fine'

        # You can also provide a lambda to the function as second argument
        # This lambda will affect the input given
        response = self.log.input('What up ?', lambda x: x.upper())
        >>> 'What up ?'
        >>> 'feeling fine'
        >>> 'FEELING FINE'
```

1.3.9 Styles

Colifrapy's logger comes with several visual alternatives that you may choose from. Those are called flavors and are available for title and standard messages.

Formatters

Colifrapy's logger accepts a format string the same way as the python logging module, so you can customize your logging output. It also add a custom variable named *flavored_levelname* which is in fact the level name colored and stylized.

```
# Default formatter for console
'%(flavored_levelname)s :: %(msg)s'
>>> [DEBUG] :: message to log

# Default formatter for file
'%(asctime)s %(levelname)s :: %(msg)s'
>>> 2014-01-15 13:56:09,798 DEBUG :: message to log
```

For the full documentation about the variables usable by the formatter, see this page.

Title Flavors

default

```
Title
-----
```

heavy

```
#########
# Title #
#########
```

elegant

```
# Title
#-------
```

bold

```
# Title
#=======
```

Flavors

default

```
[DEBUG]
```

flat

```
debug
```

reverse

```
# With reverse colors
DEBUG
```

elegant

```
Debug
```

underline

```
DEBUG
-----
```

1.4 Models

Models are the bulk of Colifrapy. You can extend them to access your settings and commands easily.

1.4.1 Usage

```
from colifrapy import Model

class MyModel(Model):
    def test(self):
        print self.settings.hello

m = MyModel()
m.test()
>>> 'world'
```

1.4.2 Reserved Attributes

- **cache** (access to cache)
- **log** (access to the logger described right after)
- **opts** (access to the command line options)
- **settings** (access to the program's settings)

1.4.3 Controller

The Controller, as referred in other part of this documentation, is strictly speaking a model no different than any other. It is just a convention to use it as a central point for launching other models.

1.5 Cacher

The Cacher classes' aim is to save some data to files for long-term usage. One could see them as very simplistic databases.

1.5.1 Model Usage

As for the logger, if a cacher was initialized by the settings while running colifrapy, “cache” will be a reserved attribute of any of your models.

It is possible to register more than one cache instance with the Settings class. To achieve this, see *cache settings*.

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # We assume that a line cache file containing
        # the string "Hello" was loaded
        print self.cache.get()
        >>> 'Hello'
```

1.5.2 Standalone Usage

You can also use the Cacher classes as standalones rather than within colifrapy’s architecture.

```
from colifrapy import LineCacher, YAMLCacher

line_cache = LineCacher(options)
yaml_cache = YAMLCacher(options)
```

1.5.3 Modes

Line Cacher

The Line Cacher consists in a strict one-liner file and is aimed at storing really simple data.

Example of cache file content

```
1245
```

YAML Cacher

The YAML Cacher is designed to store more complex states of data and organized in a key-value fashion. The readability of the YAML file format makes the cache file easy to manually modify if needed. This is also possible to create deep nested data structures that will be accessible by paths.

Example of cache file content

```
number_of_tries: 14
countries:
    albania: True
    united_kingdom: "Hello"
```

1.5.4 Options

Here are the possible options you may pass to the Cacher classes constructors :

- **directory** (string) directory where you want to store your cache *default*: “cache/”

- **filename** (string) name of the cache file *default*: “cache.txt” or “cache.yml”
- **auto_write** (boolean) whether you want your cache to be automatically written when changed or not. If set to False, you’ll have to write the invoke the cache writing manually. *default*: False

1.5.5 Methods

Line Cacher

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

    # Setting cache
    self.cache.set('Hello')

    # Getting cache
    print self.cache.get()
    >>> 'Hello'

    # Writing to cache
    # N.B. : Useless if auto_write is set to True
    self.cache.write()

    # Deleting cache
    self.cache.delete()


    # Reading and writing filters
    # Example of a single date cached
    date_format = "%Y/%m/%d %H:%M:%S"
    self.cache.setReadingFilter(lambda x: datetime.strptime(x, date_format))
    self.cache.setWritingFilter(lambda x: x.strftime(date_format))
```

YAML Cacher

```
from colifrapy import Model

class MyModel(Model):
    def test(self):

        # Setting cache
        self.cache.set("one", "red")
        self.cache.set("two:deep", "blue")

        # Getting cache
        print self.cache.get("one")
        >>> "red"

        print self.cache.get("two")
        >>> {"deep" : "blue"}

        print self.cache.get("two:deep")
        >>> "blue"
```

```
print self.cache
>>> {"one" : "red", "two" : {"deep" : "blue"}}

# Unset path
self.cache.unset("two")
print self.cache
>>> {"one" : "red"}

# Overwriting cache
self.cache.overwrite({'other' : 'structure'})
print self.cache
>>> {'other' : 'structure'}


# Writing to cache
# N.B. : Useless if auto_write is set to True
self.cache.write()

# Deleting cache
self.cache.delete()
```

1.6 Scaffolder

Colifrapy comes with a scaffolder used to generate code boilerplate. Therefore, a command is automatically added when you install colifrapy with pip.

1.6.1 Usage

To use the scaffolder:

```
colifrapy new [name-of-project]

options :
    {-a/--author: Author of the project}
    {-o/--organization: Organization of the author}
```

This will generate a standard colifrapy project containing the following files :

- .gitignore (exluded files for git)
- requirements.txt (base pip dependencies)
- README.md (project documentation)
- [name-of-project].py (command line hub)
- **config/**
  - settings.yml (standard settings for your project)
  - strings.yml (externalized strings)
- **model/**
  - controller.py (basic controller)
  - example_model.py (basic model)

Every relevant folder will of course come along with its __init__.py file.

1.7 Goodies

Colifrapy also gives access to internal functions and helpers that may prove useful.

1.7.1 Colorization

A function used to style console output. Note that not every style work on every consoles.

```
from colifrapy.tools.colorize import colorize

# The colorize function accepts up to four arguments
# 1. Positionnal : the string to style
# 2. fore_color : color of the string
# 3. background_color : color of background
# 4. style : a list or string of style(s)

# Available colors : black, red, green, yellow, blue, magenta, cyan, white
# Available styles : reset, bold, italic, dim, underline, blink-slow, blink-fast, reverse, hidden

# Example
print colorize('hello', fore_color='red', background_color='black', style='bold')
```

1.7.2 Singleton Decorator

Most of colifrapy classes are actually meant to be singletons. To perform this, the framework uses a simplistic decorator.

```
from colifrapy.tools.decorators import singleton

@singleton
class MySingleton():
    pass
```

1.7.3 Helper Functions

Some functions that may prove useful

```
# You would rather import only functions you need,
# but for the sake of the example I use '*'
from colifrapy.tools.utilities import *

# Is the variable a number ?
is_number('test')
>>> False

# Is the variable a string (python 2/3 compatible) ?
is_string('test')
>>> True

# Is the variable a list or a tuple?
```

```
is_of_list(['red', 'blue'])
>>> True

# Is the variable a function
is_func(lambda x: x.lower())
>>> True

# Parsing a string into a lambda
# WARNING: This isn't very safe
parse_lambda('lambda x: x.upper()')
>>> lambda x: x.upper()

# Get Index with fallback
get_index(['red', 'blue'], 'green', 5)
>>> 5

# Determine whether your path is relative or absolute
# if it happens to be relative, the function will assume
# it is relative to the file called (__main__)

# For those examples, we assume that the file called by the command
# line is /home/user/test/test.py
normalize_path('/home/user/path/to/file.txt')
>>> '/home/user/path/to/file.txt'

normalize_path('/resources/file.txt')
>>> '/home/user/test/resources/file.txt'

# A second boolean argument can be passed to indicate the function if
# the path leads to a directory or a file.
# In case of a directory, the path will be returned with a correct trailing slash
# Default is False (file)
normalize_path('/resources/test_folder', False)
>>> '/home/user/test/resources/test_folder'

normalize_path('/resources/test_folder', True)
>>> '/home/user/test/resources/test_folder/'
```

1.7.4 Simplified Action Hub

If your program is as simple as parsing one positionnal argument given by the user in order to choose the action to perform, you might want to use colifrapy_action argument in you yaml setting file.

Example:

```
python my-program.py action
```

Your settings yaml file

```
version: 'Basic action program'
description: 'Let the user choose the action he wants.'
arguments:
- [ ['colifrapy_action'], {'choices' : ['test', 'hello', 'delete']}]
```

Once this argument setup, just write a simplistic colifrapy hub that will automatically trigger the relevant controller method named after a choice that the use can make.

Command line hub

```
from colifrapy import Colifrapy
from model.controller import Controller

# Hub
class MyProject(Colifrapy):
    pass

# Launching
if __name__ == '__main__':
    hub = MyProject(Controller)
```

Controller

```
from colifrapy import Model

class Controller(Model):

    def test(self):
        self.log.write('test')

    def hello(self):
        self.log.write('Hello World!')

    def delete(self):
        self.log.write('Deleting...')
```

Usage

```
python my-program.py test
>>> '[DEBUG] :: 'test'

python my-program.py hello
>>> '[DEBUG] :: 'Hello World!'

python my-program.py delete
>>> '[DEBUG] :: 'Deleting...'
```

CHAPTER 2

Installation

It is recommanded to use colifrapy under a python virtualenv. (Use the excellent virtualenvwrapper to spare you some painful operations with classic virtualenvs).

Install colifrapy with pip (version up to 0.4.1):

```
pip install colifrapy
```

If you want to use the latest one which is still in development and hosted on github:

```
pip install git+https://github.com/Yomguithereal/colifrapy.git
```

CHAPTER 3

Philosophy

As every framework, colifrapy aims at enable you to work immediately on critical and intersting parts of your code that will tackle the problems you need to solve instead of battling with petty things such as the console output, your settings and the arguments passed to your tool.

However, colifrapy is not a tyrant and does not force you to go its way. As such, every part of colifrapy can be used on its own and you will remain free to code the way you want to.

CHAPTER 4

Concept

When using colifrapy, your tool is called through a command line hub which acts more or less like a router which will call upon a controller using one or several models to perform the job.

Your hub has therefore the task to load a yaml configuration file containing your command line arguments, name, version and other contextual settings.

Once those settings are loaded, every part of your application will remain able to access critical utilities such as argv opts, settings and make use of colifrapy's logger to ouptut nicely to the console and to log files.

So, schematically colifrapy is a YAML configuration file loaded by a command line hub that will call upon a controller and other models.

Every bit of colifrapy can be used as a standalone.

- **Logger** (outputs to console)
- **Settings** (deals with your yml settings)
- **Commander** (deals with argv)
- **Cacher** (saves data to file)

CHAPTER 5

Examples

The project furuikeya is a good example of the usage of colifrapy since the framework was originally designed for it.

CHAPTER 6

Dependencies

- pyyaml
- argparse

CHAPTER 7

License

Colifrapy is under a MIT license.