auto-version Documentation

Release 0.1.3

Paul Ollivier

March 04, 2013

CONTENTS

Warning: THIS IS ALPHA GRADE SOFTWARE. PLEASE READ CAREFULLY THE PRESENT DOCUMENTATION

Table of Contents

CHAPTER

ONE

INSTALLATION

Just run

```
pip install auto-version
```

CHAPTER

TWO

USAGE

2.1 Manual and semi-manual

You may use it entirely from the CLI, but it may not be advised for projects. The cli is just htere for convenience.

It is intended to be used via a configuration file, by default named *version.conf*

Here is the one used for this module:

```
{
    "files": "auto_version/main.py",
    "current_version": "0.1.0",
    "style": "Triplet"
}
```

The `style` option is a string representing the name of the style class to use. Here, I use the Triplet format, which consists in `<major>.<minor>.<patch>`.

`"files"` may be a simple string, or an array, like this:

```
{
    "files": [
        "path/to/file",
        "other/path/to/file"
    ],
    other stuff,
    blablabla
}
```

See auto_version.styles for more available version string styles.

2.2 VCS Integration

Warning: This is still a rather unstable feature, your workflow may be changed, and possibly destroyed.

If versioning system is detected (via the presence or not of a distinctive versioning directory, like `.git`), `auto-version` uses the informations present in the SCM to determine the version numbers. For git, it is via the `git tag` and `git describe` commands;

This still requires a `version.conf` file, but only three parameters are used:

```
{
    "files": "file_to_manage",
    "style": "Triplet",
    "scm_prefix": "prefix to use for version tagging"
}
```

2.2.1 Sample Usage

```
$ cd /tmp
$ git init testing
$ cd testing
$ echo "0.0.1" > hello.txt
$ echo '{ "files": "hello.txt", "style": "Triplet, "scm_prefix": ""}' > version.conf
$ git add hello.txt version.conf
$ git commit -m "Initial commit"
$ git tag 0.0.1
$ auto-version update
$ cat hello.txt
0.0.1
$ echo "hi\!" >> hello.txt
$ git add hello.txt
$ git commit -m "Modified hello.txt to better reflect my understanding of the world, from a programme
$ auto-version update
$ cat hello.txt
0.0.2-pre1-0bf45de
hi!
$ auto-version patch
$ cat hello.txt
0.0.2
hi!
```

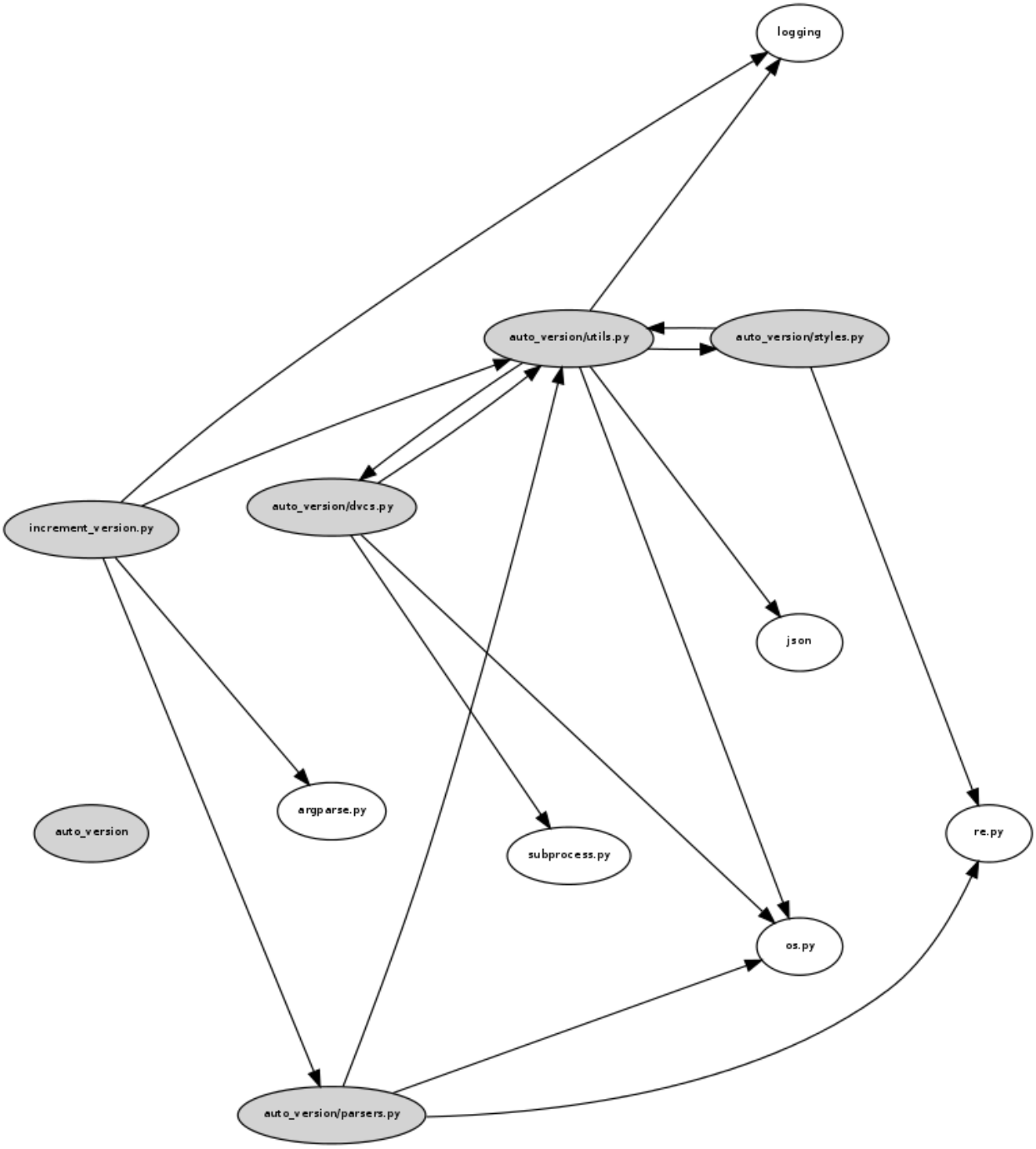
CHAPTER

THREE

IN-DEPTH DOCUMENTATION FOR TRVE L33T HACKERZ OF THE INTERNETS

3.1 Organisation

The organisation is quite simple.


logging
auto_version/utils.py
auto_version/styles.py
auto_version/dvcs.py
increment_version.py
json
argparse.py
auto_version
subprocess.py
re.py
os.py
auto_version/parsers.py

3.2 Modules detail

3.2.1 auto_version.styles

auto_version.styles is the holder of all the different coding styles.

Each style is built on top of others, as they have multiple features in common. For instance, a Revision-based versioning has a feature used in <major>.<minor>.

I am still thinking about it.

class auto_version.styles.**BaseStyle**(*current_version*)
This is the base style every Style class should inherit from.

static get_pure_version_string(*style_class*, *string*)
Returns only the part matching the version string, so we can isolate it in a string

class auto_version.styles.**Doublet**(*current_version*)
Double format is in the form <major>.<minor>

It is pretty self-explainatory. Mostly used in very small projects, without a lot of dependencies.

<major> begins most of the time at 0, indicating the in-development state of the project.

Examples:

- 0.5
- 1.2
- 1.53

increment(*level='minor'*)
Performs increment of the version number, according to the given "level" parameter. Level may be one of the followings:

•"minor" or 1: increments the <minor> part of version string

•"major" or 0: increments the <major> part of version string, and resets <minor> to 0.

class auto_version.styles.**Full**
Full format, aka. :<major>.<minor>.<patch>+<status>-<build> where <major>, <minor>, <patch> and <build> are numbers (aka, the actual version number. Well, except for the build number), and <status> is one of the following:

•prealpha

•alpha

•beta

•rc

•release

Warning: NOT YET IMPLEMENTED!

class auto_version.styles.**Revision**(*current_version*)
Revision format is a simple, one-number versioning format: r<number> for instance, r7 is the version after r6.

It is used in some DCVS, such as mercurial, or svn.

increment(*level=None*)
Performs the actual incrementation of the version number. The parameter `level` is ignored here, because there is only one.

class auto_version.styles.**Triplet**(*current_version*)
Trpilet format is in the form <major>.<minor>.<patch>

It is the most commonly used versioning 'style'.

Examples:

- 0.0.1
- 1.0.2

increment(*level='patch'*)
Performs increment of version number, according to the given "level" parameter.

Level may be one of the followings:

•"patch" or 2: increments the <patch> part of the version string

•"minor" or 1: increments the <minor> part of the version string, and resets <patch> to 0

•"major" or 0: increments the <major> part of the version string, and resets <minor> and <patch> to 0

3.2.2 auto_version.dvcs

This modules contains all the implementation for versioning system automation.

see Issue#1

The resulting version number will be in the form *<ChosenStyle>+<VCSStyle>*.

For git, VCSStyle is in the form *pre<number_of_commits_since_last_tag>-<sha_hash_of_last_commit>-<is_dirty?>*

If the user wants to use DVCS system, the option –use-vcs should be present, or "use_vcs": True should be present in config. This way, people won't find creepy hashes in their version string.

class auto_version.dvcs.**BaseVCS**
Base VCS class

> **Attention:** Do not use, use the actual vcs implementations instead

get_current_version(*with_status=False*, *increment=True*)
Return the current version, from the state of the repository.

get_status()
Returns the *status* of the repository

set_version(*files*, *version*, *prefix=''*)
When a version increment is made, update the vcs

class auto_version.dvcs.**Git**
Provides git support, via git tags. As many tag their commits with release numbers, it is a good idea to sync auto-version with these tags.

3.2.3 auto_version.parsers

This module contains the main Parser class. This class is the one parsing the given files, and replacing the values by the new ones.

class auto_version.parsers.**BasicParser**(*conf*)

This Class is a basic parser. It takes a list of files in argument, and the format of the versioning system, and performs the replacement.

Expected instanciation arguments are:

- files: an array of pathnames
- current_version: a string containing the current version of the versionned project.
- style: string to the auto_version.styles module class to use.
- action *optionnal*: a string or number representing the action to perform.

perform()

Performs the actual value replacement, according to the given parameters.

> **Attention:** This may be quite long on large files!

3.3 Less usefull stuff

3.3.1 auto_version.utils

Contains some utilities used in the project. You should not have to bother with it.

class auto_version.utils.**ConfManager**(*cli_args*)

Configuration manager. It makes the bridge and the intelligence between the cli args and the configuration file, who may be present. Or not. Whatever.

The conf variable is a dictionnary, loaded from a json file.

get_conf()

return the current state of the configuration dictionnary

save_conf(*d*)

Saves the configuration to file. The configuration is updated with the given dictionnary.

> **Warning:** given keys are overwritten!

auto_version.utils.**detect_vcs**()

Detects the versioning system in use.

> **Attention:** it does not currently handle multi-vcs systems

auto_version.utils.**import_style**(*name*)

Simple utility function, taken from the __import__ docstring to import classes.

CHAPTER

FOUR

INDICES AND TABLES

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

PYTHON MODULE INDEX

a