scd Documentation

Release 1.2.2

Sergey Arkhipov

February 14, 2017

Contents

SCD (Something Completely Different) is yet another implementation of the tools called bumpversions. There are many such tools available in the wild and I thoroughly looked through them. And decided to reinvent the wheel. You have a legit question: WHY THE BLOODY HELL DOES THIS WORLD NEED YET ANOTHER BUMPVERSION? Because I wanted the tool which works better at slightly bigger scale and I wanted the tool which I won't fight against immediately after adoption.

All bumpversion-like tools alllow you to manage versions of your software within a project. If you have a version number in your config file, documentation title, somewhere in the code, you know that it is irritating to update them manually to the new version. So there is whole set of tools which can manage them with one command.

For example, there is well-known and probably standard de-facto bumpversion. Unfortunately, bumpversion seems stale and seriously limited in its capabilities (this is the main reason why scd was born). For example, there are no regular expressions and replacement patterns look cumbersome (why do we need that `serialize` block if we can use templates? Templates are everywhere!). Also, I wanted to have a possibility to use several replacement blocks without dancing around INI syntax which never works on practice (probably I tend to complicate things, but with bigger project INI starts to irritate a lot).

Please find more rants in Rationale.

CHAPTER 1

Why Something Completely Different?

I usually find myself and my team in situation when we are over optimistic about future releases. “This time we make things right”. And everytime, when we release, I feel myself as John Cleese:

CHAPTER 2

License and Legal Stuff

Software is distributed using MIT license.

```
MIT License

Copyright (c) 2016 Sergey Arkhipov

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

You can find source codes on GitHub: https://github.com/9seconds/scd.

CHAPTER 3

Contents

3.1 Rationale

scd was created when I worked on project which is slightly better than simple library. This project has several services (you may add "micro" prefix if you want) and a lot of plugins. This project has so many plugins that we even created cookiecutter template for that. Overall more than 10 Python packages.

These packages have dependencies and some of these dependencies were project dependencies. Such as common and some api package depend on common, you get it. And we have to pin version or put a range like >=,< or ~= (at that time pip/setuptools even do not understand ~=). Also, we had documentation and we had to manage documentation via long running stable branches. So we had to support docs for version 1.0 and 2.0. Oh, and we had DEBs/RPMs and later Docker images where we put versions in labels!

As you understand, version numbers were hardcoded everywhere. In some places this was the only way to put version number (like in this doc).

We'd been trying to use bumpversion. If worked fine for some files but become a nightmare if we wanted to make complicated replacement where regular expression will fit best. Also, it was totally impossible to use bumpversion for files where we have to put ranges like `dep>={major}.{minor},<{major}.{next_minor}` (no next_minor, what a pity). Yes, these files were not understand ~= at that time and please remember that not all package managers recognize such concept. It had no regular expressions and several replacements for a file. We could fork bumpversion but it was as complicated as create our own.

So here is rationale. We wanted bumpversion which:

1. Support several search/replacement pairs for a file
2. Support searching with regular expressions
3. Have a named sets of replacement/search patterns because in a lot of files these could repeat a lot
4. Have some default search/replacement pairs.
5. Have a more reasonable configuration format than INI.
6. Templates.
7. Possibility to set current version and understand that numbering in files can vary even if current development version persist.
8. Possibility to extract some information from Git to version numbers.

Let's elaborate on those items

3.1.1 Reasonable Configuration File

scd has to support different configuration formats out of box. Currently it supports JSON, YAML and TOML. These formats are not ideal, but at least it is more reasonable to use them, then struggling with INI limitations.

Also, there should be autodiscovery of such files. Please check Configuration to get more details.

3.1.2 Regular Expression Search

It was the biggest limit of bumpversion: using a literal string search. Seriously, I do not want to keep precise literal structure of some string in file. Developer who modifies the file, can forget about **bumpversion**, reindent things or replace some quotes from single to doubles.

I do not want optional third-party tool to dictate how to keep precise line in file. This irritates. That's why I need to have regular expression search. Seriously, it is that simple. To have flexibility to not remember about **scd** or **bumpversion** at all. These tools are optional and should never be implicit dependencies.

3.1.3 Several Search/Replacement Pairs

Okay, you have a package *X* which dependend on *Y* and *Z*. *X*, *Y* and *Z* all are parts of your project. Fine, and you need to bump version. Now solve problem: how to replace version range of *Y* and *Z* in `setup.py` of *X*? In a single replacement literal pattern. Yes, constant. Just because your version bumper is dumb enough to force you to simplify its life. Or with giant unsupporable regexp, yes.

We need to have a support of multiple search/replacement pairs per file. Dixi.

3.1.4 Named sets of Search/Replacement Pairs

If you have a lot of files where to manage version, you will quickly realise that those files are not individual, you will have ~5-6 different search and replacement patterns overall. To avoid a long list of copying and pasting, you need to have a possibility to assign pattern with some name and use it later.

For example, you can have `(?<=version\s=\s")\d+\.\d+\.\d+` named as `setuppy`. In that case, if you will replace double quotes with single ones, you won't sed whole file, you can do it in one place.

3.1.5 Templates

Why the hell on the world do you need to implement confusing `serialize` blocks if world already has templates? **scd** uses Jinja2 as templating engine.

3.1.6 Git and Development Releases

We live in the world where development releases exist and we need something to support them. It is great to have some base version for a current developing release but we need to have a possiblity to generate development version identifiers. Prerelases. Include build numbers.

In Python there are several projects to do that. For example, there is widely used pbr which generates development release numbers for you. There is setuptools_scm which is seriously great and I highly recommend everyone to use it.

The only problem about setuptools-scm is its extensibility. It is extendable by entrypoints and it is reasonable. But if you want to have another version numbering policy, you need to implement your own entrypoint. And put it somewhere. And set `setup_requires` to that package. It works, but it is slightly inconvenient to use that, having additional depenency you have to put somewhere and install before any other package. But seriously, this project

rocks. And available for Python packages only so there is no way to update docs or RPM specs. And it is irritating to have 2 schemes of versioning, they will fail one day.

3.2 Installation

scd is simple Python package which hosted on Cheese Shop so if you are familiar with Python package installation, it would be really straightforward.

Tool works with Python>=2.7 and PyPy2.

3.2.1 Prerequisites

To install scd, you need to have pip or setuptools installed. Pip is required if you want to install it from Cheese Shop and setuptools if you prefer source code installation.

To install Pip follow these guides:

- Installation with package managers
- Installation without package managers

To install setuptools follow official guide and please check repository of your OS: there is a great possiblity that you already have it installed.

3.2.2 Install from Cheese Shop

If you want to install system-wide or in virtualenv then do

```
pip install scd
```

Otherwise, please do

```
pip install --user scd
```

Also, it is possible to use following extras to add some optional features to your installation.

Name	Description
yaml	Enable support of YAML configuration files.
toml	Enable support of TOML configuration files.
simplejson	Use simplejson for JSON parsing.
colors	Enable support of colors in output.

So if you want to install scd with YAML support and colors enabled, please do following:

```
pip install scd[yaml,colors]
```

3.2.3 Install from sources

```
git clone https://github.com/9seconds/scd
cd scd
python setup.py install
```

Verify that tool is installed with `scd --help`.

3.3 Configuration

scd uses configuration file to get information on settings, search/replacement patterns and files to manage.

As you got from *Install from Cheese Shop*, scd can parse TOML, YAML and JSON configuration files. So first we need to elaborate a little bit on how to create required configuration.

3.3.1 Configuration Formats

Yes, 3 formats, but in most cases all three formats are possible to reduce to equialent JSON. Here are examples of all 3 formats, which are totally equialent.

YAML:

```yaml
version:
  number: 1.2.3
  scheme: semver

search_patterns:
  full: "{{ full }}"

replacement_patterns:
  full: "{{ full }}"

defaults:
  search: full
  replace: full

files:
  setup.py:
    - default
```

TOML:

```toml
[version]
number = "1.2.3"
scheme = "semver"

[search_patterns]
full = "{{ full }}"

[replacement_patterns]
full = "{{ full }}"

[defaults]
search = "full"
replace = "full"

[files]
"setup.py" = ["default"]
```

JSON:

```json
{
    "version": {
        "number": "1.2.3",
        "scheme": "semver"
    },
```

```
    "search_patterns": {
        "full": "{{ full }}"
    },
    "replacement_patterns": {
        "full": "{{ full }}"
    },
    "defaults": {
        "search": "full",
        "replace": "full"
    },
    "files": {
        "setup.py": ["default"]
    }
}
```

I hope you get an idea: all these formats are representing the same datastructure. If you are familiar with JSON Schema, you may find that useful:

```
{
    "$schema": "http://json-schema.org/draft-04/schema",
    "type": "object",
    "required": ["version", "defaults", "files"],
    "properties": {
        "config": {
            "type": "number",
            "minimum": 1,
            "multipleOf": 1.0
        },
        "version": {
            "type": "object",
            "required": ["scheme", "number"],
            "properties": {
                "scheme": {
                    "type": "string",
                    "enum": ["pep440", "semver", "git_pep440", "git_semver"]
                },
                "number": {
                    "oneOf": [
                        {"type": "number"},
                        {"type": "string"}
                    ]
                }
            }
        },
        "files": {
            "type": "object",
            "additionalProperties": {
                "type": "array",
                "items": {
                    "oneOf": [
                        {"type": "string", "enum": ["default"]},
                        {
                            "type": "object",
                            "properties": {
                                "search": {"type": "string"},
                                "search_raw": {"type": "string"},
                                "replace": {"type": "string"},
                                "replace_raw": {"type": "string"}
```

```
                                },
                                "anyOf": [
                                    {
                                        "required": ["search"],
                                        "not": {"required": ["search_raw"]}
                                    },
                                    {
                                        "required": ["search_raw"],
                                        "not": {"required": ["search"]}
                                    },
                                    {
                                        "required": ["replace"],
                                        "not": {"required": ["replace_raw"]}
                                    },
                                    {
                                        "required": ["replace_raw"],
                                        "not": {"required": ["replace"]}
                                    }
                                ]
                            }
                        ]
                    }
                }
            },
            "search_patterns": {
                "type": "object",
                "additionalProperties": {"type": "string"}
            },
            "replacement_paterns": {
                "type": "object",
                "additionalProperties": {"type": "string"}
            },
            "groups": {
                "type": "object",
                "additionalProperties": {"type": "string"}
            },
            "defaults": {
                "type": "object",
                "properties": {
                    "search": {"type": "string"},
                    "replacement": {"type": "string"}
                },
                "additionalProperties": false
            }
        }
}
```

Please be noticed that it is possible to extend allowed schemes with external entrypoints but **PEP 440** and SemVer are supported out of box.

3.3.2 Examples

For simplicity, I will put examples here in YAML but as you already understand, they could be easily made with any other format.

Full Example

```
config: 1

version:
  number: 1.0.1
  scheme: semver

search_patterns:
  full: "{{ semver }}"
  vfull: "v{{ semver }}"
  major_minor_block: "\\d+\\.\\d+(?=\\s\\#\\sBUMPVERSION)"

replacement_patterns:
  full: "{{ full }}"
  major_minor: "{{ major }}.{{ minor }}"
  major_minor_p: "{{ major }}.{{ minor }}{% if patch %}.{{ patch }}{% endif %}"

defaults:
  search: full
  replace: full

groups:
  code: 'scd/.*?\.py'
  docs: 'docs/.*?'

files:
  setup.py:
    - search_raw: "(?>=version\\s=\\s\\\"){{ full }}"
  docs/conf.py:
    - default
    - search: vfull
      replace: major_minor_p
    - search: major_minor_block
      replace_raw: "{{ next_major }}"
```

Shortest Example

```
version:
  number: 1.0.1
  scheme: semver

defaults:
  search: semver
  replace: base

files:
  setup.py:
    - default
```

So, as you can see, config can be large and can be small. It is up to you what to choose.

3.3.3 Parameters

From examples above you may get an idea that some parameters are optionals, some mandatory. Mandatory parameters are `version`, `defaults` and `files`. All others are optionals.

Also, you may notice Mustache-like strings like `{{ something }}`. Your guessing is correct, it is Jinja2 templates. Template context variables are depended on choosen version scheme, you can get a list of them in *Predefined Template Context*.

config

`config` is a numeric version (integers, please) of the config format. This is the first field processed by scd therefore it is possible to have absolutely different schemas in future.

This field is responsible for config schema version. Sometimes (probably in future) we will bring (definitely will) some non-backward compatible changes in schema and we will differ configs by numbers.

This field is optional in 1.x versions, it implicitly equal to 1.

version

Version block defines a settings, related to versioning strategy.

scd won't calculate version for you, you need to set base version by your own. Some may consider that as inconvenience (if you have latest version 0.1.0, it is good to have next one as 0.1.1 calculated automatically), but I belive this is for the greatest good (struggling to force your smartass versioner to have next version 0.2 is way more inconvenient, than setting explicit one).

This block has 2 mandatory parameters and 0 optionals.

Parameter	Type	Example	Description
number	string	1.2.3	This parameter defines basic version you are developing. Upcoming planned version. For example, you've just released version 1.3.0. What is the next version? Basically, nobody knows. It might be 1.3.1, it might be 1.4.0 or even 2.0.0. Seriously, it is totally up to your release management and branching strategy. This number is *planned* version, not *released* one. Planned. And all versions, calculated by scd will use that number as a base. So in templates you may find `{{ major }}` as `1`, `{{ minor }}` as `2` etc.
scheme	string	semver	The name of the scheme your are using for versioning. scd will parse version numbers according to that parameter. So, all these `major`, `minor` etc won't appear magically, they coming from parsed `version/number` parameter. Please check *Predefined Template Context* to get a list of parsed context variables. by default, scd supports **PEP 440** and semver schemes. Their codenames are `pep440` and `semver` accordingly. Also, there are Git-flavored schemes `git_pep440` and `git_semver`: these flavors more or less the same as their prefixless variants, but scd will use git to calculate some parameters like putting git tag in local part of **PEP 440** or distance from latest version tag as prerelase in semver. User can define his own schemes using entrypoints-based plugin mechanism. Please check documentation for `scd.version` for that.

search_patterns

Search patterns defines regular expression which are used to search a place in file where to replace.

scd works in line-mode fashion, similar to sed, so all expressions applied to the line. Also, please be noticed that due to some implementation details, all expression will be compiled with `re.VERBOSE` and `re.UNICODE`. If you are not from Python world, please check re documentation.

Important: Please check documentation on re.VERBOSE. Seriously, if you do not know what it is, go and read.

This block should have a simple mapping, where key is the name of the pattern and value is regular expression, understandable by Python.

There are several predefined search templates are available:

- `pep440`
- `semver`
- `git_pep440`
- `git_semver`

They are matching version in the format, allowed by semver or PEP440. If you have your own versioning available as plugin, it will be here also. Since all of them are defined, there is no need to define them on your own. But if you define pattern with such name in that section, default one will be, obviously, overriden.

Also, to simplify composition of your own patterns, these names are available as template context variables in search patterns. In other words, pattern like `v{{ semver }}` is perfectly fine.

Important: scd will replace group 0 of the pattern. This is done intentionally to avoid possible ambiguity. In other word, it replaces whole pattern, not only some group. If you want to define regular expression more presicely, please use look-ahead and look-behind expressions.

---

replacement_patterns

Replacement patterns are used to express version for the search pattern.

The same thing, this parameter is key/value mapping where key if the name of the pattern and value is Jinja2 template, used for replacement. For available context variables please check *PEP440* and SemVer

There are 2 predefined replacement patterns:

Name	Equialent	Description
base	{{ base }}	Base version. Literally, the same stuff as you have in *version/number* block
full	{{ full }}	Full version, generated by your scheme. The most complete and precise as possible.

Of course, it is possible to override them in that section.

groups

Sometimes you want to change versions only in some subset of files. This why you can group them in some optional groups and filter by these groups. So, let's say you've defined groups *code* and *docs*. In that case, you can modify versions in docs only, without touching the code.

This is a mapping parameter. Key is the group name, value is regular expression. Each expression sets a path (or pathes) relative to the position of config file. The same story, as in *files*.

Important: scd will implicitly append `$` to the pattern. Please do not use `^` and `$` as start/end of the line - it just makes no sense.

---

defaults

If you have a lot of files, sometimes you want to have some default replacement or search. This is because it is possible to postpone some parameter having default one.

This block has 2 mandatory parameters and 2 optionals.

Name	Description
search	This is a name of search pattern which should be used by default.
replace	This is a name of default replacement pattern should be used by default.

Please be noticed, that values are *names*, not raw patterns. Keys from `search_patterns` and `replacement_patterns`.

files

Files are the list of file structures which scd should worry about. If scd does not have a section in config file, it will ignore file even if it explicitly set in CLI. Well, because nobody knows how to manage unknown file.

This is a mapping between filenames and a list of search/replacements.

Filename is rather simple: it is POSIX path to the file, relative to the config. POSIX means that separator is `/`, not `\`. So if you have a filename `docs/source/conf.py`, it will work perfectly on Unix/OS X and Windows. On Windows, actually, scd will interpret this path as `docssourceconf.py` os it is crossplatform. Another mentioned thing about filename is that it is relative to the config file. So with file above and config file path `/home/username/project/.scd.yaml`, scd will process `/home/username/project/docs/source/conf.py`.

Search/replacements are the list with following rules:

Parameter	Description
search	The *name* of the search pattern from `search_patterns` or some globally defined. Please check *search_patterns* for details. **Note**: this is mutually exclusive with `search_raw`. Please define either `search` or `search_raw`.
search_raw	The *pattern* to use. This is actual regular expression which can be used to define some search pattern ad-hoc, without populating `search_patterns` section with patterns which require only once. Please check *search_patterns* for details on how to compose such regular expressions. **Note**: this is mutually exclusive with `search`. Please define either `search` or `search_raw`.
replace	The *name* of the replacement pattern from `replacement_patterns` or some globally defined. Please check *replacement_patterns* for details. **Note**: this is mutually exclusive with `replace_raw`. Please define either `replace` either `replace_raw`
replace_raw	The *replacement* template to use. This is actual Jinja2 template which can be used to define some ad-hoc replacement without populating `replacement_patterns` section with stuff which require only once. Please check *replacement_patterns* for details. **Note**: this is mutually exclusive with `replace`. Please define either `replace_raw` either `replace`.

Please be noticed that at least something has to be defined. You may postpone any parameter (no `search` or `search_raw` for example, but if you define any, please remember about mutual exclusive groups, mentioned in table), then parameters from *defaults* section will be used. But do not keep element empty. There is special placeholder `default` for that. So if you want to use defaults only, please use config like:

```
version:
  number: 1.0.1
  scheme: semver

defaults:
  search: semver
  replace: base

files:
  setup.py:
    - default
```

In that case `semver` search pattern and `base` replacement will be used for `setup.py`.

3.3.4 Predefined Template Context

As it was previously mentioned, there are several predefined context variables which might be used in templates for search and replacements. Also, please remember, that these contexts are different: you cannot use context vars from replacements to make search pattern.

Search Context

Context Variable	Description
pep440	This searches version number, valid according to **PEP 440**.
git_pep440	Same as `pep440`.
semver	This searches version number, valid according to semver.
git_semver	Same as `semver`.

Replacement Context

Replacement context is totally dependend on version scheme provided. Moreover, every scheme provides its own set of context variables, and it is possible that you have a scheme which is not version numbered (I worked with such scheme once, and it was not that bad as one can think).

Of course, there is a number of some predefined context variables for replacements, you may find them in *replacement_patterns* section.

For next sections we need to make some assumptions on versions. Let's pretend that we have version `1.2.0` in our config file, using Git flavor of a scheme, operating on commit `ff5cff170e93ab4f7dd87437951c6646e297c538` which is 5 commits left from latest version tag.

SemVer

Context Variable	Type	Value From Example
base	string	1.2.0
full	string	1.2.0-5+ff5cff1
major	integer	1
next_major	integer	2
prev_major	integer	0
minor	integer	2
next_minor	integer	3
prev_minor	integer	1
patch	integer	0
next_patch	integer	1
prev_patch	integer	0
prerelase	string	5
next_prerelease	string	6
prev_prerelease	string	4
build	string	ff5cff1
next_build	string	ff5cff2
prev_build	string	ff5cff0

As you can see, this is rather trivial. The most interesting parts are build and prerelase management. By default, scd will try to guess next and previous parts (it increments latest number found in the string). Sometimes it make sense (`build5` for example), sometimes not (Git commit hash) so please pay attention to your strategy.

PEP440

To show all possible values, let's consider base version as `1.2.0rc1`.

Context Variable	Type	Value From Example
base	string	1.2.0rc1
full	string	1.2.0rc1.dev5+ff5cff1
maximum	string	0!1.2.0rc1.post0.dev5+ff5cff1
epoch	integer	0
major	integer	1
next_major	integer	2
prev_major	integer	0
minor	integer	2
next_minor	integer	3
prev_minor	integer	1
patch	integer	0
next_patch	integer	1
prev_patch	integer	0
prerelase	integer	1
prerelase_type	string	rc
next_prerelease	integer	2
prev_prerelease	integer	0
dev	integer	5
next_dev	integer	6
prev_dev	integer	4
post	integer	0
next_post	integer	1
prev_post	integer	0
local	string	ff5cff1

So, more or less the same. The only difference is that `full` won't display data which is 0 or empty. `maximum` does.

3.4 Usage

3.4.1 CLI Arguments and Options

```
usage: scd [-h] [-V] [-p] [-n] [-c CONFIG_PATH]
           [-x [CONTEXT_VAR [CONTEXT_VAR ...]]] [-d | -v]
           [FILE_PATH [FILE_PATH ...]]

scd is a tool to manage version strings within your project files.

positional arguments:
  FILE_PATH             Path to the files where to make version bumping. If
                        nothing is set, all filenames in config will be used.

optional arguments:
  -h, --help            show this help message and exit
  -V, --own-version     print version only.
  -p, --replace-version
                        print version to replace to.
  -n, --dry-run         make dry run, do not change anything.
  -c CONFIG_PATH, --config CONFIG_PATH
                        path to the config. By default autodiscovery will be
                        performed.
  -x [CONTEXT_VAR [CONTEXT_VAR ...]], --extra-context [CONTEXT_VAR [CONTEXT_VAR ...]]
                        Additional context variables. Format is key=value.
```

```
  -s {git_pep440,git_semver,pep440,semver}, --version-scheme {git_pep440,git_semver,pep440,semver}
                        override version-scheme from config.
  -d, --debug           run in debug mode
  -v, --verbose         run tool in verbose mode
```

I have no idea what to add here. You can get this output with `scd -h`.

3.4.2 Explicit Scheme

Sometimes you need to override scheme from config file. For example, you may want to use `pep440` for versioning but in CI system (or any build system) you need to use `git_pep440`. This option is for you.

3.4.3 Debug and Verbose Mode

By default, scd won't notify you about anything. And won't print. But somethimes you want to know about some details. There are 2 ways how to do that: using debug and verbose mode.

Verbose output should be used if you are worrying about how scd is processing your files. Debug output - if you have some issue and want to yell on developer having something in your hands. If suspect you absolutely do not need to execute debug mode if you are not author of the tool.

Here are examples:

Verbose mode:

```
>>> Use /home/sergey/dev/pvt/scd/.scd.yaml as config file
>>> Parsed config as YAML
>>> Version is 0.1.0.dev24+3177b4e
>>> Start to process /home/sergey/dev/pvt/scd/setup.py
>>> Modify 'version="0.0.1",' to 'version="0.1.0.dev24+3177b4e",'
>>> Start to process /home/sergey/dev/pvt/scd/docs/source/conf.py
>>> Modify "version = '1.0'" to "0.1'"
>>> Modify "release = '1.0.0b1'" to "0.1.0'"
>>> Start to process /home/sergey/dev/pvt/scd/scd/__init__.py
>>> Modify '__version__ = "0.1.0"' to '0.1.0.dev24"'
```

Debug mode:

```
149 [DEBUG  ] (      main:69 ) Options: Namespace(config=None, debug=True, dry_run=True, files=[], ve
149 [DEBUG  ] (      main:169) Search configfile in /home/sergey/dev/pvt/scd
149 [INFO   ] (      main:177) Use /home/sergey/dev/pvt/scd/.scd.yaml as config file
150 [DEBUG  ] (    config:197) Use default json as JSON config parser.
164 [DEBUG  ] (    config:218) Use PyYAML for YAML config parser.
165 [DEBUG  ] (    config:228) Use toml for TOML config parser.
165 [DEBUG  ] (    config:244) Cannot parse JSON: Expecting value: line 1 column 1 (char 0)
169 [INFO   ] (    config:240) Parsed config as YAML
169 [DEBUG  ] (    config:242) Parsed config content:
{
    "defaults": {
        "replacement": "full",
        "search": "pep440"
    },
    "files": {
        "docs/source/conf.py": [
            {
                "replace_raw": "{{ major }}.{{ minor }}",
                "search_raw": "^version\\s=\\s'{{ pep440 }}'"
```

```
            },
            {
                "replace_raw": "{{ major }}.{{ minor }}.{{ patch }}",
                "search_raw": "^release\\s=\\s'{{ pep440 }}'"
            }
        ],
        "scd/__init__.py": [
            {
                "replace_raw": "{{ major }}.{{ minor }}.{{ patch }}{% if post %}.post{{ post }}{% end
                "search_raw": "^__version__\\s=\\s\"{{ pep440 }}\""
            }
        ],
        "setup.py": [
            {
                "replace": "full",
                "search": "setuppy"
            }
        ]
    },
    "search_patterns": {
        "setuppy": "(?<=version=\\\"){{ git_pep440 }}"
    },
    "version": {
        "number": "0.1.0",
        "scheme": "git_pep440"
    }
}
175 [INFO   ] (      main:72 ) Version is 0.1.0.dev24+3177b4e
176 [DEBUG  ] (     files:204) File /home/sergey/dev/pvt/scd/docs/source/conf.py is ok
176 [DEBUG  ] (     files:204) File /home/sergey/dev/pvt/scd/setup.py is ok
176 [DEBUG  ] (     files:204) File /home/sergey/dev/pvt/scd/scd/__init__.py is ok
176 [INFO   ] (      main:81 ) Start to process /home/sergey/dev/pvt/scd/docs/source/conf.py
176 [DEBUG  ] (      main:82 ) File object: <File(filename='docs/source/conf.py', path='/home/sergey/
184 [INFO   ] (     files:61 ) Modify "version = '1.0'" to "0.1'"
185 [INFO   ] (     files:61 ) Modify "release = '1.0.0b1'" to "0.1.0'"
186 [DEBUG  ] (      main:149) No need to save /home/sergey/dev/pvt/scd/docs/source/conf.py
186 [INFO   ] (      main:81 ) Start to process /home/sergey/dev/pvt/scd/setup.py
186 [DEBUG  ] (      main:82 ) File object: <File(filename='setup.py', path='/home/sergey/dev/pvt/scd
193 [INFO   ] (     files:61 ) Modify 'version="0.0.1",' to 'version="0.1.0.dev24+3177b4e",'
193 [DEBUG  ] (      main:149) No need to save /home/sergey/dev/pvt/scd/setup.py
193 [INFO   ] (      main:81 ) Start to process /home/sergey/dev/pvt/scd/scd/__init__.py
193 [DEBUG  ] (      main:82 ) File object: <File(filename='scd/__init__.py', path='/home/sergey/dev/
198 [INFO   ] (     files:61 ) Modify '__version__ = "0.1.0"' to '0.1.0.dev24"'
198 [DEBUG  ] (      main:149) No need to save /home/sergey/dev/pvt/scd/scd/__init__.py
```

3.4.4 Dry Run

Sometimes you do not want to do replacement, but to check what it will change. Execute scd with `--dry-run` flag. Also, I advise to run in verbose mode to get details you want.

3.4.5 Config Autodiscovery

It is always possible to set path to your config with `--config`. It is fine but sometimes you do not want to remember where is your config is placed. And you are working within Git repository. And all folks are placing such files in the root of repositories so... this is idea of autodiscovery.

Let's assume that you are working in `./ui` directory of your repository and execuing scd without explicit config path (`--config ../.scd.yaml`). What will happen:

1. scd will try to search within your current directory. It will search configs in following order:
   - `.scd.json`
   - `scd.json`
   - `.scd.yaml`
   - `scd.yaml`
   - `.scd.toml`
   - `scd.toml`
2. If nothing is found, scd will get top level of your repository (`git rev-parse --show-toplevel`) and start to search there. The same file order.

3.4.6 Extra Context

Sometimes you need to have some extra context to propagate into templates or patterns. Here is the flag for that, `-x` (`--extra-context`). If you execute scd like `scd -x name=myname`, you will get `name` variable for replacement and search patterns immediately.

3.5 API Reference

SCD.

This is yet another implementation of the tools called bumpversions. There are many such tools available in the wild and I thoroughly looked through them. And decided to reinvent the wheel. You have a legit question: WHY THE BLOODY HELL DOES THIS WORLD NEED YET ANOTHER BUMPVERSION? Because I wanted the tool which works better at slightly bigger scale and I wanted the tool which I won't fight against immediately after adoption.

All bumpversion-like tools alllow you to manage versions of your software within a project. If you have a version number in your config file, documentation title, somewhere in the code, you know that it is irritating to update them manually to the new version. So there is whole set of tools which can manage them with one command.

For example, there is well-known and probably standard de-facto bumpversion. Unfortunately, bumpversion seems stale and seriously limited in its capabilities (this is the main reason why scd was born). For example, there are no regular expressions and replacement patterns look cumbersome (why do we need that `serialize` block if we can use templates? Templates are everywhere!). Also, I wanted to have a possibility to use several replacement blocks without dancing around INI syntax which never works on practice (probably I tend to complicate things, but with bigger project INI starts to irritate a lot).

scd is extensible with setuptools' entrypoints. It basically means that if you want, you can always create you own implementation of some functions, scd will discover that and can use.

Currently, there is only one entrypoint is defined, `scd.version`. All instances of that entrypoint should be subclasses of *`scd.version.Version`* class. Please check *`scd.version.SemVer`* or *`scd.version.PEP440`* for examples.

3.5.1 Contents

scd.main

Module, which has routines for scd CLI.

scd.main.**catch_exceptions**(*func*)
Decorator which makes function more CLI friendly.

If everything is ok, it returns `os.EX_OK` (code 0), if not - `os.EX_SOFTWARE` (code 70). Also, it is smart enough to differ verbose and debug mode and print accordingly.

scd.main.**configure_logging**()
Configure logging based on `OPTIONS`.

scd.main.**get_options**()
Return parsed commandline arguments.

Returns Parsed commandline arguments

Return type `argparse.Namespace`

scd.main.**guess_configfile**()
Return file-like object, guessing where the hell if config file.

Returns Open config.

Return type file-like object

Raises **ValueError** – if cannot find config file.

scd.main.**main**()
Main function.

Basically, it parses CLI, creates config, traverse files and does modifications. All that scd does is happening with this function.

scd.main.**process_file**(*fileobj*, *config*)
Function, which is responsible for processing of file.

Parameters

- **fileobj** (*scd.files.File*) – File to process.
- **config** (*scd.config.Config*) – Parsed configuration.

scd.main.**search_config_in_directory**(*directory*)
Return config file name if it is found in directory.

Parameters **directory** (*str*) – Path to the directory where to search config files.

Returns Path to the config file (absolute) or `None` if nothing is found

Return type str or None

scd.config

This module contains all routines, related to scd's configuration.

class scd.config.**Config**(*configpath*, *version_scheme*, *config*, *extra_context*)
Wrapper over parsed configuration data.

This wrapper provides methods for internal scd's implementation.

You want to use this class to access configuration data.

Parameters

- **configpath** (*str*) – Path to the configuration file (can be relative).
- **or None version_scheme** (*str*) – Explicit version scheme to use.
- **config** (*dict*) – Parsed configuration.
- **str] extra_context** (*dict[str,*) – Additional context to use in templates.

Raises **ValueError** – if configuration is not valid to schema.

project_directory
Absolute path to the directory with config file.

Returns Absolute path to the directory.

Return type str

static validate_schema(*config*)
Validate parsed content to comply with JSON Schema.

Parameters **config** (*dict*) – Parsed configuration.

Returns A list of errors, found during verification. If list is empty, everyting is valid.

Return type list[str]

class scd.config.**Parser**(*name*, *func*)

func
Alias for field number 1

name
Alias for field number 0

class scd.config.**V1Config**(*configpath*, *version_scheme*, *config*, *extra_context*)
Implementation of *Config* for config version 1.

defaults
A mapping of default search/replace patterns from config file.

Returns Raw mapping, as is.

Return type dict[str, str]

files
A list of files defines in config file.

Returns List of file instances

Return type list[*scd.files.File*]

filter_files(*required_groups*, *required_files*)
Filter and return only those files which are required.

This uses groups and `required_files` parameter filtering.

Parameters

- **required_groups** (*list[str]*) – A list of mandatory groups
- **required_files** (*list[str]*) – A list of mandatory files

Returns A list of files after filtering.

Return type list[*scd.files.File*]

groups
A list of groups defined in config file.

Returns List of group names

Return type list[str, str]

replacement_patterns
A mapping of replacement patterns (name/repl) from config file.

Returns Raw mapping, as is.

Return type dict[str, str]

search_patterns
A mapping of search patterns (name/pattern) from config file.

Returns Raw mapping, as is.

Return type dict[str, str]

version
Instance of *scd.version.Version*.

This instance is created based on data from config file.

Returns Version

Return type *scd.version.Version*

version_number
Base version number from config file.

Returns Literal number from config

Return type str

version_scheme
Scheme of the versioning from config file.

For example, it can be `git_pep440`.

Returns Version scheme

Return type str

scd.config.**get_json_parser**()
Function which detects what parser should be used for parsing JSONs.

It uses following logic: if simplejson is available, it would be used, otherwise default `json` will work.

Returns JSON parser

Return type *Parser*

scd.config.**get_parsers**()
Function to detect locally available parsers.

Returns A list of available parsers for config files.

Return type list[*Parser*]

scd.config.**get_toml_parser**()
Function which detects what parser should be used for parsing TOMLs.

It uses following logic: if toml is available, it would be used, otherwise `None` is returned.

Returns TOML parser or `None` if nothing found.

Return type *Parser* or None

scd.config.**get_yaml_parser**()
Function which detects what parser should be used for parsing YAMLs.

It uses following logic: if PyYAML is available, it would be used, otherwise it will try for ruamel.yaml.

Returns YAML parser or `None` if nothing found.

Return type *Parser* or None

scd.config.**make_config**(*filename*, *version_scheme*, *content*, *extra_context*)
Function to generate config based on incoming parameters.

This function does validation of config version.

Parameters

- **filename** (*str*) – Path to the configuration file (can be relative).
- **or None version_scheme** (*str*) – Explicit version scheme to use.
- **content** (*dict*) – Parsed configuration.
- **str] extra_context** (*dict[str,*) – Additional context to use in templates.

Raises **ValueError** – if config version is not supported.

scd.config.**parse**(*fileobj*, *version_scheme*, *extra_context*)
Function which parses given file-like object with config data.

Parameters

- **fileobj** (*file-like object*) – Open file object for parsing.
- **or None version_scheme** (*str*) – Explicit version scheme to use.
- **str] extra_context** (*dict[str,*) – Additional context to use in templates.

Returns Parsed config

Return type *Config*

Raises **ValueError** – if not possible to parse config in any way.

scd.files

All classes and routines related to files.

class scd.files.**File**(*name*, *data*, *config*)
This is a wrapper for a file on FS which should be managed by scd.

The same story as for *scd.config.Config*: this wrapper is used for purposes of conveience mostly. Also, it is required when one need to emit a list of *SearchReplace* instances for a file.

Parameters

- **name** (*str*) – The name of the file from config (as is, not absolute one)
- **config** (*scd.config.Config*) – Instance of used config.
- **data** (*list*) – A contents of search/replacement parts of the config.

all_replacements
Mapping of all known replacements for a file.

This mapping includes default replacements and those, defined in config file.

Key is the name of the replacement, value is an instance of `jinja2.Template`.

Returns Mapping of replacements.

Return type dict[str, str]

all_search_patterns
Mapping of all search patterns for a file.

This mapping includes default patterns and those, defined in config file.

Key is the name of the replacement, value is compiled regular expression.

Returns Mapping of patterns.

Return type dict[str, str]

default_replace_pattern
Property, returns default replacement template from config.

Returns Default replacement pattern

Return type `jinja2.Template`

default_replacements
Mapping of default replacements for a file.

Key is the name of the replacement, value is an instance of `jinja2.Template`.

Returns Mapping of replacements.

Return type dict[str, str]

default_search_pattern
Property, returns default search pattern from config.

Returns Default search pattern

Return type Regular expression

default_search_patterns
Mapping of default search patterns for a file.

Key is the name of the replacement, value is compiled regular expression.

Returns Mapping of patterns.

Return type dict[str, str]

filename
Relative filename of the file.

The most cool part about this property is that such name is platform independent: on Windows it might be `docsconf.py`, on Linux: `docs/conf.py`. That cool.

Returns Native platform filename

Return type str

path
Absolute path to the file for current platform.

Returns Native platform absolute path.

Return type str

patterns
A list of search/replacements for a file, based on config.

Returns List of instances for file management.

Return type list[*SearchReplace*]

class scd.files.**SearchReplace**(*search*, *replace*)
Class, which presents a pair of single search and replacement.

Parameters

- **search** (*regexp*) – Search regular expression.
- **replace** (jinja2.Template) – Replacement template

static get_replacement(*replace*, *version*)
Return rendered template, taken context from version.

Parameters

- **replace** (jinja2.Template) – Template for replacement.
- **version** (*scd.version.Version*) – Version instance, where template takes context.

Returns Rendered template, ready to insert.

Raises **ValueError** – if there is no enough context to render template.

Return type str

process(*version*, *text*)
Process text according to given version.

This does what is expected: search in text (as a rule, line from file) and inserts replacement where required.

Parameters

- **version** (*scd.version.Version*) – Version instance to use.
- **text** (*str*) – Text to process.

Returns Processed line, after inserting replacement if needed. Return original line otherwise.

Return type str

scd.files.**make_pattern**(*base_pattern*, *config*)
Function, which creates regular expression based on given pattern.

Also, it injects all predefined search regexps like pep440 etc.

Parameters **base_pattern** (*str*) – Pattern to transform to regular expression instance.

Returns Regular expression pattern

Return type regexp

Raises **ValueError** – if pattern cannot be parsed.

scd.files.**make_template**(*template*)
Function for creating template instance from text template.

Parameters **template** (*str*) – Text template to process.

Returns Correct template instance, based on given text.

Return type jinja2.Template

scd.files.**validate_access**(*files*)
Function, which validates access to the files.

Parameters **files** (list[*scd.files.File*]) – A list of files to check

Returns Is all files are accessible or not

Return type bool

scd.utils

A set of various utils, used within scd.

scd.utils.**execute**(*command*)
Executor of external command and wrapper for result.

This is a wrapper for `subprocess.Popen` with stdin set to `/dev/null`.

It returns result like:

```
{
    "code": 0,
    "stdout": ["this is a line of stdout", "and this is another"],
    "stderr": []
}
```

Parameters **command** (*list[str]*) – A command for `subprocess.Popen` to execute.

Returns Execution result.

Return type dict

Raises **ValueError** – if command is not possible to execute.

scd.utils.**get_plugins**(*namespace*)
A mapping of plugins (loaded) in given namespace.

Parameters **namespace** (*str*) – The name of namespace to use.

Returns Mapping for plugins (key is the name and value is loaded plugin).

Return type dict

scd.utils.**get_version_plugins**()
A mapping of scd version plugins.

scd.version

Routines for version management.

These module has *Version* class which is a base class for entrypoints `scd.version`. All entrypoints of such class should be subclasses of *Version*.

Currently, it `scd.version` has following defined entrypoints:

Entrypoint	Class
pep440	*PEP440*
semver	*SemVer*
git_pep440	*GitPEP440*
git_semver	*GitSemVer*

class scd.version.**GitMixin**(**args*, ***kwargs*)
Mixin to add Git flavor for *Version* classes.

class scd.version.**GitPEP440**(*config*)
Git flavored *PEP440* implementation.

This implementation does the same, but precalculates local and dev parts based on Git information.

Dev release is the number of commits since latest tag and local will have Git short commit SHA at the first place.

class scd.version.**GitSemVer**(*config*)
Git flavored *SemVer* implementation.

This implementation does the same, but precalculates build and prerelase parts based on Git information.

Prerelase is the number of commits since latest tag and build is short commit hash. Previous and next builds are always empty. Because nobody predicts next commit hash.

class scd.version.**PEP440**(*config*)
Implementation of Python versioning.

For details, please check **PEP 440**.

dev
Dev number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 2.

Returns Development part of the version number

Return type int

epoch
Epoch part of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 1483072998.

Returns Epoch part of the version number

Return type int

local
Local part of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 5afe90c.linux.

Returns Local part of the version number

Return type int

major
Major part of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 1.

Returns Major part of the version number

Return type int

maximum
Maximal representation of the version.

This always has all possible parts (probably except of prerelase, it is still optional, because we have to know context to calculate that) even if it makes no sense. I have no idea about usecase of that except of having this property for completenes.

Example: `0!1.2.3rc3.post0.dev0+1ubuntu1`.

Horrible.

Returns Maximal version number.

Return type str

minor
Minor number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 2.

Returns Minor part of the version number

Return type int

next_dev
Next dev number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 3.

Returns Next development part of the version number

Return type int

next_major
Next major number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 2.

Returns Next major part of the version number

Return type int

next_minor
Next minor number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 3.

Returns Next minor part of the version number

Return type int

next_patch
Next patch number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 4.

Returns Next patch part of the version number

Return type int

next_post
Next post number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 14.

Returns Next post part of the version number

Return type int

next_prerelease
Next prerelease number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 4.

Returns Next prerelease part of the version number

Return type int

patch
Patch number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 3.

Returns Patch part of the version number

Return type int

post
Post number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 13.

Returns Post part of the version number

Return type int

prerelease
Prerelease number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 3.

Returns Prerelease part of the version number

Return type int

prerelease_type
Type of the prerelase.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns rc.

Returns Type of the prerelease

Return type str

prev_dev
Prev dev number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 2.

Returns Previous development part of the version number

Return type int

prev_major
Prev major number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 0.

Returns Previous major part of the version number

Return type int

prev_minor
Prev minor number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 1.

Returns Previous minor part of the version number

Return type int

prev_patch
Prev patch number for the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 2.

Returns Previous patch part of the version number

Return type int

prev_post
Prev post number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 12.

Returns Previous post part of the version number

Return type int

prev_prerelease
Prev prerelease number of the version.

For version `1483072998!1.2.3rc3.post13.dev2+5afe90c.linux` it returns 2.

Returns Previous prerelease part of the version number

Return type int

class `scd.version.`**SemVer**(*config*)
Implementation of semantic version numbering.

For details, please check http://semver.org/.

build
Build version number.

Build version number of version `1.2.3-pre1+build4` is build4.

Returns Build version number.

Return type str

next_build
Next build version number.

Next build version number of version `1.2.3-pre1+build4` is build5.

Returns Next build version number.

Return type str

next_major
Next major version number.

Next major number of version `1.2.3` is 2.

Returns Next major version number.

Return type int

next_minor
Next minor version number.

Next minor number of version `1.2.3` is 3.

Returns Next minor version number.

Return type int

next_patch
Next patch version number.

Next patch number of version `1.2.3` is 4.

Returns Next patch version number.

Return type int

next_prerelease
Next prerelase version number.

Next prerelase version number of version `1.2.3-pre1+build4` is pre2.

Returns Next prerelase version number.

Return type str

classmethod next_text_version(*text*)
Method which returns next number from the string.

From string `build10s` it returns 11.

Parameters **text** (*str*) – Line to search in.

Returns Next number

Return type int

classmethod parse_text_version(*text*)
Method which extracts latest number from the string.

Empty string implies 0. No number also implies 0.

Parameters **text** (*str*) – Line to search in.

Returns Latest number

Return type int

prerelease
Prerelase version number.

Prerelase version number of version `1.2.3-pre1+build4` is pre1.

Returns Prerelase version number.

Return type str

prev_build
Prev build version number.

Previous build version number of version `1.2.3-pre1+build4` is build3.

Returns Previous build version number.

Return type str

prev_major
Prev major version number.

Previous major number of version `1.2.3` is 0.

Returns Previous major version number.

Return type int

prev_minor
Prev minor version number.

Previous minor number of version `1.2.3` is 1.

Returns Previous minor version number.

Return type int

prev_patch
Prev patch version number.

Previous patch number of version `1.2.3` is 4.

Returns Previous patch version number.

Return type int

prev_prerelease
Prev prerelase version number.

Previous prerelase version number of version `1.2.3-pre1+build4` is pre0.

Returns Previous prerelase version number.

Return type str

classmethod prev_text_version(*version*)
Method which returns previous number from the string.

From string `build10s` it returns 9.

Parameters **text** (*str*) – Line to search in.

Returns Next number

Return type int

class scd.version.**Version**(*config*)
Base class for version scheme.

This class is the base of `scd.version` entrypoint and it's main intention is correct version parsing and creating of template context.

Parameters **config** (*scd.config.Config*) – Configuration wrapper

base
Base number from config. Literally, as defined there.

Returns Version number

Return type str

context
Context for `jinja2.Template`.

Returns A mapping of context variables.

Return type dict[str, str or int]

full
Full reference version number, with a lot of details.

Returns Version number

Return type str

scd.version.**git_distance**(*git_dir*, *matcher='v*'*)
Return a number of commits since latest matched tag.

Parameters

- **git_dir** (*str*) – Path to the `.git` directory of repository.
- **matcher** (*str*) – Glob of the tag names to operate with.

Returns The number of commits or `None` if nothing is found.

Return type int or None

scd.version.**git_tag**(*git_dir*)
Return a current Git commit sha for repository.

Parameters **git_dir** (*str*) – Path to the `.git` directory of repository.

Returns Commit SHA in short form or `None` if cannot find any.

Return type str or None

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

S

P

R

S

V