versioningit

Release 1.1.1

John Thorvald Wodder II

2022 Apr 08

CONTENTS

GitHub | PyPI | Documentation | Issues | *Changelog*

CHAPTER

ONE

HOW IT WORKS

`versioningit` divides its operation into six *steps*: `vcs`, `tag2version`, `next-version`, `format`, `write`, and `onbuild`. The first four steps make up the actual version calculation, while the `write` and `onbuild` steps normally only happen while building with setuptools.

1.1 Version Calculation

The version for a given project is determined as follows:

- `vcs` step: The version control system specified in the project's `versioningit` configuration is queried for information about the project's working directory: the most recent tag, the number of commits since that tag, whether there are any uncommitted changes, and other data points.
- `tag2version` step: A version is extracted from the tag returned by the `vcs` step
- If there have been no commits or uncommitted changes since the most recent tag, the version returned by the `tag2version` step is used as the project version. Otherwise:
  - `next-version` step: The next version after the most recent version is calculated
  - `format` step: The results of the preceding steps are combined to produce a final project version.

1.2 Setuptools Integration

1.2.1 Setting the Version

`versioningit` registers a `setuptools.finalize_distribution_options` entry point that causes it to be run whenever setuptools computes the metadata for a project in an environment in which `versioningit` is installed. If the project in question has a `pyproject.toml` file with a `[tool.versioningit]` table, then `versioningit` performs the version calculations described above and sets the project's version to the final value. (If a version cannot be determined because the project is not in a repository or repository archive, then `versioningit` will assume the project is an unpacked sdist and will look for a `PKG-INFO` file to fetch the version from instead.) If the `pyproject.toml` contains a `[tool.versioningit.write]` table, then the `write` step will also be run at this time; the default `write` method creates a file at a specified path containing the project's version.

1.2.2 `onbuild` Step

When a project is built that uses `versioningit`'s custom setuptools commands, the `onbuild` step becomes added to the build process. The default `onbuild` method updates one of the files in the built distribution to contain the project version while leaving the source files in the actual project alone. See "*The [tool.versioningit.onbuild] Subtable*" for more information.

CHAPTER

TWO

CONFIGURATION

The `[tool.versioningit]` table in `pyproject.toml` is divided into six subtables, each describing how one of the six steps of the version extraction & calculation should be carried out. Each subtable consists of an optional `method` key specifying the *method* (entry point or function) that should be used to carry out that step, plus zero or more extra keys that will be passed as parameters to the method when it's called. If the `method` key is omitted, the default method for the step is used.

2.1 Specifying the Method

A method can be specified in two different ways, depending on where it's implemented. A method that is built in to `versioningit` or provided by an installed third-party extension is specified by giving its name as a string, e.g.:

```toml
[tool.versioningit.vcs]
# The method key:
method = "git"  # <- The method name

# Parameters to pass to the method:
match = ["v*"]
default-tag = "1.0.0"
```

Alternatively, a method can be implemented as a function in a Python source file in your project directory (either part of the main Python package or in an auxiliary file); see "*Writing Your Own Methods*" for more information. To tell `versioningit` to use such a method, set the `method` key to a table with a `module` key giving the dotted name of the module in which the method is defined and a `value` key giving the name of the callable object in the module that implements the method. For example, if you created a custom `next-version` method that's named `my_next_version()` and is located in `mypackage/mymodule.py`, you would write:

```toml
[tool.versioningit.next-version]
method = { module = "mypackage.module", value = "my_next_version" }
# Put any parameters here
```

Note that this assumes that `mypackage/` is located at the root of the project directory (i.e., the directory containing the `pyproject.toml` file); if is located inside another directory, like `src/`, you will need to add a `module-dir` key to the method table giving the path to that directory relative to the project root, like so:

```toml
[tool.versioningit.next-version]
method = { module = "mypackage.module", value = "my_next_version", module-dir = "src" }
# Put any parameters here
```

As a special case, if there are no parameters for a given step, the respective subtable can be replaced by the method specification, e.g.:

```toml
[tool.versioningit]
# Use the "git" method for the vcs step with no parameters:
vcs = "git"
# Use a custom function for the next-version step with no parameters:
next-version = { module = "mypackage.module", value = "my_next_version" }
```

2.2 The `[tool.versioningit.vcs]` Subtable

The `vcs` subtable specifies the version control system used by the project and how to extract the tag and related information from it. `versioningit` provides three `vcs` methods: `"git"` (the default), `"git-archive"`, and `"hg"`.

2.2.1 `"git"`

The `"git"` method relies on the project directory being located inside a Git repository with one or more commits. Git 1.8.0 or higher must be installed, though some optional features require more recent Git versions.

The `"git"` method takes the following parameters, all optional:

`match` [list of strings] A set of fileglob patterns to pass to the `--match` option of `git describe` to make Git only consider tags matching the given pattern(s). Defaults to an empty list.

> **Note:** Specifying more than one match pattern requires Git 2.13.0 or higher.

`exclude` [list of strings] A set of fileglob patterns to pass to the `--exclude` option of `git describe` to make Git not consider tags matching the given pattern(s). Defaults to an empty list.

> **Note:** This option requires Git 2.13.0 or higher.

`default-tag` [string] If `git describe` cannot find a tag, `versioningit` will raise a *versioningit.errors.NoTagError* unless `default-tag` is set, in which case it will act as though the initial commit is tagged with the value of `default-tag`

2.2.2 `"git-archive"`

This method is experimental and may change in future releases.

The `"git-archive"` method is a variation of the `"git"` method that also supports determining the version when installing from a properly-prepared Git archive. The method takes the following parameters:

`describe-subst` [string] *(required)* Set this to `"$Format:%(describe)$"` and add the line `pyproject.toml export-subst` to your repository's `.gitattributes` file. This will cause any Git archive made from your repository from this point forward to contain the minimum necessary information to determine a version.

`match` and `exclude` options are set by including them in the format placeholder like so:

```toml
# Match 'v*' tags:
describe-subst = "$Format:%(describe:match=v*)$"

# Match 'v*' and 'r*' tags:
```

(continues on next page)

(continued from previous page)

```
describe-subst = "$Format:%(describe:match=v*,match=r*)$"

# Match 'v*' tags, exclude '*-final' tags:
describe-subst = "$Format:%(describe:match=v*,exclude=*-final)$"
```

By default, only annotated tags are considered, and lightweight tags are ignored; this can be changed by including the "tags" option in the placeholder like so:

```
# Honor all tags:
describe-subst = "$Format:%(describe:tags)$"

# Honor all tags, exclude '*rc' tags:
describe-subst = "$Format:%(describe:tags,exclude=*rc)$"
```

Options other than "match", "exclude", and "tags" are not supported by `versioningit` and will result in an error.

`default-tag` [string] *(optional)* If `git describe` cannot find a tag, `versioningit` will raise a *versioningit.errors.NoTagError* unless `default-tag` is set, in which case it will act as though the initial commit is tagged with the value of `default-tag`.

Note that this parameter has no effect when installing from a Git archive; if the repository that the archive was produced from had no relevant tags for the archived commit (causing the value of `describe-subst` to be set to the empty string), `versioningit` will raise an error when trying to install the archive.

Note that, in order to provide a consistent set of information regardless of whether installing from a repository or an archive, the `"git-archive"` method provides the `format` step with only a subset of the fields that the `"git"` method does; *see below* for more information.

Changed in version 1.0.0: The "match" and "exclude" settings are now parsed from the `describe-subst` parameter, which is now required, and the old `match` and `exclude` parameters are now ignored. Also, support for the "tags" option was added.

A note on Git version requirements

- The `%(describe)s` placeholder was only added to Git in version 2.32.0, and so a Git repository archive must be created using at least that version in order to be installable with this method. Fortunately, GitHub repository ZIP downloads support `%(describe)`, and so **`pip`**-installing a "git-archive"-using project from a URL of the form `https://github.com/$OWNER/$REPO/archive/$BRANCH.zip` will work.
- The `%(describe)s` placeholder only gained support for the "tags" option in Git 2.35.0, and so, if this option is included in the `describe-subst` parameter, that Git version or higher will be required when creating a repository archive in order for the result to be installable. Unfortunately, as of 2022-02-05, GitHub repository Zips do not support this option.
- When installing from a Git repository rather than an archive, the "git-archive" method parses the `describe-subst` parameter into the equivalent `git describe` options, so a bleeding-edge Git is not required in that situation (but see the version requirements for the "git" method above).

Note: In order to avoid DOS attacks, Git will not expand more than one `%(describe)s` placeholder per archive, and so you should not have any other `$Format:%(describe)$` placeholders in your repository.

Note: This method will not work correctly if you have a tag that resembles `git describe` output, i.e., that is of the form `<anything>-<number>-g<hex-chars>`. So don't do that.

2.2.3 "hg"

The `"hg"` method supports installing from a Mercurial repository or archive. When installing from a repository, Mercurial 5.2 or higher must be installed.

The `"hg"` method takes the following parameters, all optional:

`pattern` [string] A revision pattern (See `hg help revisions.patterns`) to pass to the `latesttag()` template function. Note that this parameter has no effect when installing from a Mercurial archive.

`default-tag` [string] If there is no latest tag, `versioningit` will raise a *`versioningit.errors.NoTagError`* unless `default-tag` is set, in which case it will act as though the initial commit is tagged with the value of `default-tag`

2.3 The `[tool.versioningit.tag2version]` Subtable

The `tag2version` subtable specifies how to extract the version from the tag found by the `vcs` step. `versioningit` provides one `tag2version` method, `"basic"` (the default), which proceeds as follows:

- If the `rmprefix` parameter is set to a string and the tag begins with that string, the given string is removed from the tag.
- If the `rmsuffix` parameter is set to a string and the tag ends with that string, the given string is removed from the tag.
- If the `regex` parameter is set to a string (a Python regex) and the regex matches the tag (using `re.search`), the tag is replaced with the contents of the capturing group named "`version`", or the entire matched text if there is no group by that name. If the regex does not match the tag, the behavior depends on the `require-match` parameter: if true, an error is raised; if false or unset, the tag is left as-is.
- Finally, any remaining leading `v`'s are removed from the tag.

A warning is emitted if the resulting version is not **PEP 440**-compliant.

2.4 The `[tool.versioningit.next-version]` Subtable

The `next-version` subtable specifies how to calculate the next release version from the version extracted from the VCS tag. `versioningit` provides the following `next-version` methods; none of them take any parameters.

`minor` *(default)* Strips the input version down to just the epoch segment (if any) and release segment (i.e., the `N(.N)*` bit), increments the second component of the release segment, and replaces the following components with a single zero. For example, if the version extracted from the VCS tag is `1.2.3.4`, the `"minor"` method will calculate a new version of `1.3.0`.

`minor-release` Like `minor`, except that if the input version is a prerelease or development release, the base version is returned; e.g., `1.2.3a0` becomes `1.2.3`. This method requires the input version to be **PEP 440**-compliant.

`smallest` Like `minor`, except that it increments the last component of the release segment. For example, if the version extracted from the VCS tag is `1.2.3.4`, the `"smallest"` method will calculate a new version of `1.2.3.5`.

smallest-release Like `smallest`, except that if the input version is a prerelease or development release, the base version is returned; e.g., `1.2.3a0` becomes `1.2.3`. This method requires the input version to be **PEP 440**-compliant.

null Returns the input version unchanged. Useful if your repo version is something horrible and unparseable.

A warning is emitted if the resulting version is not **PEP 440**-compliant.

2.5 The `[tool.versioningit.format]` Subtable

The `format` subtable specifies how to format the project's final version based on the information calculated in previous steps. (Note that, if the repository's current state is an exact tag match, this step will be skipped and the version returned by the `tag2version` step will be used as the final version.) `versioningit` provides one `format` method, `"basic"` (the default).

The data returned by the `vcs` step includes a repository *state* (describing the relationship of the repository's current contents to the most recent tag) and a collection of *format fields*. The `"basic"` `format` method takes the name of that state, looks up the `format` parameter with the same name (falling back to a default, given below) to get a format template string, and formats the template using the given format fields plus `{version}`, `{next_version}`, and `{branch}` fields. A warning is emitted if the resulting version is not **PEP 440**-compliant.

For the built-in `vcs` methods, the repository states are:

`distance`	One or more commits have been made on the current branch since the latest tag
`dirty`	No commits have been made on the branch since the latest tag, but the repository has uncommitted changes
`distance-dirty`	One or more commits have been made on the branch since the latest tag, and the repository has uncommitted changes

For the built-in `vcs` methods, the available format fields are:

`{author_date}`	The author date of the HEAD commit[1] (`"git"` only)
`{branch}`	The name of the current branch (with non-alphanumeric characters converted to periods), or `None` if the branch cannot be determined
`{build_date}`	The current date & time, or the date & time specified by the environment variable `SOURCE_DATE_EPOCH` if it is set[1]
`{committer_date}`	The committer date of the HEAD commit[1] (`"git"` only)
`{distance}`	The number of commits since the most recent tag
`{next_version}`	The next release version, calculated by the `next-version` step
`{rev}`	The abbreviated hash of the HEAD commit
`{revision}`	The full hash of the HEAD commit (`"git"` and `"hg"` only)
`{vcs}`	The first letter of the name of the VCS (i.e., "`g`" or "`h`")
`{vcs_name}`	The name of the VCS (i.e., "`git`" or "`hg`")
`{version}`	The version extracted from the most recent tag

The default parameters for the `format` step are:

```
[tool.versioningit.format]
distance = "{version}.post{distance}+{vcs}{rev}"
dirty = "{version}+d{build_date:%Y%m%d}"
distance-dirty = "{version}.post{distance}+{vcs}{rev}.d{build_date:%Y%m%d}"
```

[1] These fields are UTC `datetime.datetime` objects. They are formatted with `strftime()` formats by writing `{fieldname:format}`, e.g., `{build_date:%Y%m%d}`.

2.6 The [tool.versioningit.write] Subtable

The `write` subtable enables an optional feature, writing the final version to a file. Unlike the other subtables, if the `write` subtable is omitted, the corresponding step will not be carried out.

`versioningit` provides one `write` method, `"basic"` (the default), which takes the following parameters:

`file` [string] *(required)* The path to the file to which to write the version, relative to the root of your project directory. This path should use forward slashes (`/`) as the path separator, even on Windows.

> **Note:** This file should not be committed to version control, but it should be included in your project's built sdists and wheels.

`encoding` [string] *(optional)* The encoding with which to write the file. Defaults to UTF-8.

`template`: string *(optional)* The content to write to the file (minus the final newline, which `versioningit` adds automatically), as a string containing a `{version}` placeholder. If this parameter is omitted, the default is determined based on the `file` parameter's file extension. For `.txt` files and files without an extension, the default is:

```
{version}
```

while for `.py` files, the default is:

```
__version__ = "{version}"
```

If `template` is omitted and `file` has any other extension, an error is raised.

Note: When testing out your configuration with the `versioningit` command (See *Command*), you will need to pass the `--write` option if you want the `[tool.versioningit.write]` subtable to take effect.

2.7 The [tool.versioningit.onbuild] Subtable

New in version 1.1.0.

The `onbuild` subtable configures an optional feature, inserting the project version into built project trees when building an sdist or wheel. Specifically, this feature allows you to create sdists & wheels in which some file has been modified to contain the line `__version__ = "<project version>"` or similar while leaving your repository alone.

In order to use this feature, in addition to filling out the subtable, your project must include a `setup.py` file that passes *versioningit.get_cmdclasses()* as the `cmdclass` argument to `setup()`, e.g.:

```
from setuptools import setup
from versioningit import get_cmdclasses

setup(
    cmdclass=get_cmdclasses(),
    # Other arguments go here
)
```

`versioningit` provides one `onbuild` method, `"replace-version"` (the default). It scans a given file for a line matching a given regex and inserts the project version into the first line that matches. The method takes the following parameters:

`source-file` [string] *(required)* The path to the file to modify, relative to the root of your project directory. This path should use forward slashes (`/`) as the path separator, even on Windows.

`build-file` [string] *(required)* The path to the file to modify when building a wheel. This path should be the location of the file when your project is installed, relative to the root of the installation directory. For example, if `source-file` is `"src/mypackage/__init__.py"`, where `src/` is your project dir, set `build-file` to `"mypackage/__init__.py`. If you do not use a `src/`-layout or other remapping of package files, set `build-file` to the same value as `source-file`.

This path should use forward slashes (`/`) as the path separator, even on Windows.

`encoding` [string] *(optional)* The encoding with which to read & write the file. Defaults to UTF-8.

`regex` [string] *(optional)* A Python regex that is tested against each line of the file using `re.search`. The first line that matches is updated as follows:

- If the regex contains a capturing group named "`version`", the substring matched by the group is replaced with the expansion of `replacement` (see below). If `version` did not participate in the match, an error is raised.
- Otherwise, the entire substring of the line matched by the regex is replaced by the expansion of `replacement`.

The default regex is:

```
^\s*__version__\s*=\s*(?P<version>.*)
```

`require-match` [boolean] *(optional)* If `regex` does not match any lines in the file and `append-line` is not set, an error will be raised if `require-match` is true (default: false).

`replacement` [string] *(optional)* The string used to replace the relevant portion of the matched line. The string is first expanded by replacing any occurrences of `{version}` with the project version, and then any backreferences to capturing groups in the regex are expanded.

The default value is `"{version}"` (that is, the version enclosed in double quotes).

`append-line` [string] *(optional)* If `regex` does not match any lines in the file and `append-line` is set, any occurrences of `{version}` in `append-line` are replaced with the project version, and the resulting line is appended to the end of the file.

Thus, with the default settings, `"replace-version"` finds the first line in the given file of the form "`__version__ = ...`" and replaces the part after the = with the project version in double quotes; if there is no such line, the file is left unmodified.

Note: Because the `onbuild` step runs both when building an sdist from the repository and when building a wheel from an sdist, the configuration should be such that running the step a second time doesn't change the file any further. (The technical term for this is "idempotence")

Note: If you use this feature and run **`python setup.py`** directly (as opposed to building with `build` or similar), you must invoke `setup.py` from the root project directory (the one containing your `setup.py`).

Tip: You are encouraged to test your `onbuild` configuration by building an sdist and wheel for your project and examining the files within to ensure that they look how you want. An sdist can be expanded by running `tar zxf`

filename, and a wheel can be expanded by running `unzip` *filename*.

2.8 `tool.versioningit.default-version`

The final key in the `[tool.versioningit]` table is `default-version`, which is a string rather than a subtable. When this key is set and an error occurs during version calculation, `versioningit` will set your package's version to the given default version. When this key is not set, any errors that occur inside `versioningit` will cause the build/install process to fail.

Note that `default-version` is not applied if an error occurs while parsing the `[tool.versioningit]` table; however, such errors can be caught ahead of time by running the `versioningit` command (See "*Command*").

2.9 Log Level Environment Variable

When `versioningit` is invoked via the setuptools plugin interface, it logs various information to stderr. By default, only messages at `WARNING` level or higher are displayed, but this can be changed by setting the `VERSIONINGIT_LOG_LEVEL` environment variable to the name of a Python logging level (case insensitive) or the equivalent integer value.

CHAPTER

THREE

GETTING PACKAGE VERSION AT RUNTIME

Automatically setting your project's version is all well and good, but you usually also want to expose that version at runtime, usually via a `__version__` variable. There are three options for doing this:

1. Use the `version()` function in `importlib.metadata` to get your package's version, like so:

   ```
   from importlib.metadata import version

   __version__ = version("mypackage")
   ```

   Note that `importlib.metadata` was only added to Python in version 3.8. If you wish to support older Python versions, use the importlib-metadata backport available on PyPI for those versions, e.g.:

   ```
   try:
       from importlib.metadata import version
   except ImportError:
       from importlib_metadata import version

   __version__ = version("mypackage")
   ```

   If relying on the backport, don't forget to include `importlib-metadata; python_version < "3.8"` in your project's `install_requires`!

2. Fill out the `[tool.versioningit.write]` subtable in `pyproject.toml` so that the project version will be written to a file in your Python package which you can then import or read. For example, if your package is named `mypackage` and is stored in a `src/` directory, you can write the version to a Python file `src/mypackage/_version.py` like so:

   ```
   [tool.versioningit.write]
   file = "src/mypackage/_version.py"
   ```

   Then, within `mypackage/__init__.py`, you can import the version like so:

   ```
   from ._version import __version__
   ```

   Alternatively, you can write the version to a text file, say, `src/mypackage/VERSION`:

   ```
   [tool.versioningit.write]
   file = "src/mypackage/VERSION"
   ```

   and then read the version in at runtime with:

   ```
   from pathlib import Path
   __version__ = Path(__file__).with_name("VERSION").read_text().strip()
   ```

3. *(New in version 1.1.0)* Fill out the `[tool.versioningit.onbuild]` subtable in `pyproject.toml` and configure your `setup.py` to use `versioningit`'s custom setuptools commands. This will allow you to create sdists & wheels in which some file has been modified to contain the line `__version__ = "<project version>"` or similar while leaving your repository alone. See "*The [tool.versioningit.onbuild] Subtable*" for more information.

Tip: Wondering which of `write` and `onbuild` is right for your project? See this table for a comparison:

	`write`	`onbuild`
Should affected file be under version control?	**No**	**Yes**
Affected file must already exist?	**No**	**Yes**
Modifies working tree?[1]	**Yes**	**No**
Requires configuration in `setup.py`?	**No**	**Yes**
Run when installing in editable mode?	**Yes**	**No**

---

[1] That is, the `write` method causes a file to be present (though likely ignored) in your repository after running, while the `onbuild` method only modifies a file inside sdists & wheels and leaves the original copy in your repository unchanged.

CHAPTER

FOUR

COMMAND

```
versioningit [<options>] [<project-dir>]
```

When `versioningit` is installed in the current Python environment, a command of the same name will be available that prints out the version for a given `versioningit`-enabled project (by default, the project rooted in the current directory). This can be used to test out your `versioningit` setup before publishing.

4.1 Options

`-n, --next-version`

Instead of printing the current version of the project, print the value of the next release version as computed by the `next-version` step

`--traceback`

Normally, any library errors are shown as just the error message. Specify this option to show the complete error traceback.

`-v, --verbose`

Increase the amount of log messages displayed. Specify twice for maximum information.

The logging level can also be set via the `VERSIONINGIT_LOG_LEVEL` environment variable. If both `-v` and `VERSIONINGIT_LOG_LEVEL` are specified, the more verbose log level of the two will be used, where one `-v` corresponds to `INFO` level and two or more correspond to `DEBUG` level. (If neither are specified, the default level of `WARNING` is used.)

`-w, --write`

Write the version to the file specified in the `[tool.versioningit.write]` subtable, if so configured

CHAPTER

FIVE

LIBRARY API

5.1 High-Level Functions

versioningit.**get_version**(*project_dir: Union[str, pathlib.Path] = '.'*, *config: Optional[dict] = None*, *write: bool = False*, *fallback: bool = True*) → str

Determine the version for the project at `project_dir`.

If `config` is `None`, then `project_dir` must contain a `pyproject.toml` file containing a `[tool.versioningit]` table; if it does not, a *NotVersioningitError* is raised. If `config` is not `None`, then any `pyproject.toml` file in `project_dir` will be ignored, and the configuration will be taken from `config` instead; see "*Passing Explicit Configuration*".

If `write` is true, then the file specified in the `[tool.versioningit.write]` subtable, if any, will be updated.

If `fallback` is true, then if `project_dir` is not under version control (or if the VCS executable is not installed), `versioningit` will assume that the directory is an unpacked sdist and will read the version from the `PKG-INFO` file.

Raises

- ***NotVCSError*** – if `fallback` is false and `project_dir` is not under version control
- ***NotSdistError*** – if `fallback` is true, `project_dir` is not under version control, and there is no `PKG-INFO` file in `project_dir`
- ***NotVersioningitError*** –
  - if `config` is `None` and `project_dir` does not contain a `pyproject.toml` file
  - if the `pyproject.toml` file does not contain a `[tool.versioningit]` table
- ***ConfigError*** – if any of the values in `config` are not of the correct type
- ***MethodError*** – if a method returns a value of the wrong type

versioningit.**get_next_version**(*project_dir: Union[str, pathlib.Path] = '.'*, *config: Optional[dict] = None*) → str

New in version 0.3.0.

Determine the next version after the current VCS-tagged version for `project_dir`.

If `config` is `None`, then `project_dir` must contain a `pyproject.toml` file containing a `[tool.versioningit]` table; if it does not, a *NotVersioningitError* is raised. If `config` is not `None`, then any `pyproject.toml` file in `project_dir` will be ignored, and the configuration will be taken from `config` instead; see "*Passing Explicit Configuration*".

Raises

- *NotVCSError* – if `project_dir` is not under version control
- *NotVersioningitError* –
  - if `config` is `None` and `project_dir` does not contain a `pyproject.toml` file
  - if the `pyproject.toml` file does not contain a `[tool.versioningit]` table
- *ConfigError* – if any of the values in `config` are not of the correct type
- *MethodError* – if a method returns a value of the wrong type

versioningit.**get_cmdclasses**(*bases: Optional[Dict[str, Type[Command]]] = None*) → Dict[str, Type[Command]]

New in version 1.1.0.

Return a `dict` of custom setuptools `Command` classes, suitable for passing to the `cmdclass` argument of `setuptools.setup()`, that run the `onbuild` step for the project when building an sdist or wheel. Specifically, the `dict` contains a subclass of `setuptools.command.sdist.sdist` at the `"sdist"` key and a subclass of `setuptools.command.build_py.build_py` at the `"build_py"` key.

A `dict` of alternative base classes can optionally be supplied; if the `dict` contains an `"sdist"` entry, that entry will be used as the base class for the customized `sdist` command, and likewise for `"build_py"`. All other classes in the input `dict` are passed through unchanged.

5.2 Low-Level Class

class versioningit.**Versioningit**

A class for getting a version-controlled project's current version based on its most recent tag and the difference therefrom

classmethod from_project_dir(*project_dir: Union[str, pathlib.Path] = '.', config: Optional[dict] = None*) → *versioningit.core.Versioningit*

Construct a *Versioningit* object for the project rooted at `project_dir` (default: the current directory).

If `config` is `None`, then `project_dir` must contain a `pyproject.toml` file containing a `[tool.versioningit]` table; if it does not, a *NotVersioningitError* is raised. If `config` is not `None`, then any `pyproject.toml` file in `project_dir` will be ignored, and the configuration will be taken from `config` instead. See "*Passing Explicit Configuration*".

Raises

- *NotVersioningitError* –
  - if `config` is `None` and `project_dir` does not contain a `pyproject.toml` file
  - if `config` is `None` and the `pyproject.toml` file does not contain a `[tool.versioningit]` table
- *ConfigError* – if the `tool.versioningit` key, `config`, or any subfields are not of the correct type

get_version() → str

Determine the version for `project_dir`

Raises *MethodError* – if a method returns a value of the wrong type

do_vcs() → *versioningit.core.VCSDescription*

Run the `vcs` step

Raises ***MethodError*** – if the method does not return a *VCSDescription*

do_tag2version(*tag: str*) → str

Run the `tag2version` step

Raises ***MethodError*** – if the method does not return a `str`

do_next_version(*version: str*, *branch: Optional[str]*) → str

Run the `next-version` step

Raises ***MethodError*** – if the method does not return a `str`

do_format(*description:* versioningit.core.VCSDescription, *version: str*, *next_version: str*) → str

Run the `format` step

Raises ***MethodError*** – if the method does not return a `str`

do_write(*version: str*) → None

Run the `write` step

do_onbuild(*build_dir: Union[str, pathlib.Path]*, *is_source: bool*, *version: str*) → None

New in version 1.1.0.

Run the `onbuild` step

5.3 Exceptions

exception versioningit.**Error**

Base class of all `versioningit`-specific errors

exception versioningit.**ConfigError**

Bases: *versioningit.errors.Error*, `ValueError`

Raised when the `versioningit` configuration contain invalid settings

exception versioningit.**InvalidTagError**

Bases: *versioningit.errors.Error*, `ValueError`

Raised by `tag2version` methods when passed a tag that they cannot work with

exception versioningit.**InvalidVersionError**

Bases: *versioningit.errors.Error*, `ValueError`

Raised by `next-version` methods when passed a version that they cannot work with

exception versioningit.**MethodError**

Bases: *versioningit.errors.Error*

Raised when a method is invalid or returns an invalid value

exception versioningit.**NoTagError**

Bases: *versioningit.errors.Error*

Raised when a tag cannot be found in version control

exception versioningit.**NotSdistError**

Bases: *versioningit.errors.Error*

Raised when attempting to read a `PKG-INFO` file from a directory that doesn't have one

exception versioningit.**NotVCSError**

Bases: *versioningit.errors.Error*

Raised when `versioningit` is run in a directory that is not under version control or when the relevant VCS program is not installed

exception versioningit.**NotVersioningitError**

Bases: *versioningit.errors.Error*

Raised when `versioningit` is used on a project that does not have `versioningit` enabled

5.4 Utilities

class versioningit.**VCSDescription**(*tag: str*, *state: str*, *branch: Optional[str]*, *fields: Dict[str, Any]*)

A description of the state of a version control repository

branch: Optional[str]

The name of the repository's current branch, or `None` if it cannot be determined or does not apply

fields: Dict[str, Any]

A `dict` of additional information about the repository state to make available to the `format` method Custom `vcs` methods are advised to adhere closely to the set of fields used by the built-in methods.

state: str

The relationship of the repository's current state to the tag. If the repository state is exactly the tagged state, this field should equal `"exact"`; otherwise, it will be a string that will be used as a key in the `[tool.versioningit.format]` subtable. Recommended values are `"distance"`, `"dirty"`, and `"distance-dirty"`.

tag: str

The name of the most recent tag in the repository (possibly after applying any match or exclusion rules based on user parameters) from which the current repository state is descended

versioningit.**get_version_from_pkg_info**(*project_dir: Union[str, pathlib.Path]*) → str

Return the *`Version`* field from the `PKG-INFO` file in `project_dir`

Raises

- ***`NotSdistError`*** – if there is no `PKG-INFO` file
- **`ValueError`** – if the `PKG-INFO` file does not contain a *`Version`* field

versioningit.**run_onbuild**(**, build_dir: Union[str, pathlib.Path]*, *is_source: bool*, *version: str*, *project_dir: Union[str, pathlib.Path] = '.'*, *config: Optional[dict] = None*) → None

New in version 1.1.0.

Run the `onbuild` step for the given project.

If `config` is `None`, then `project_dir` must contain a `pyproject.toml` file containing a `[tool.versioningit]` table; if it does not, a *`NotVersioningitError`* is raised. If `config` is not `None`, then any `pyproject.toml` file in `project_dir` will be ignored, and the configuration will be taken from `config` instead; see "*Passing Explicit Configuration*".

Parameters

- **build_dir** – The directory containing the in-progress build
- **is_source** – Set to `True` if building an sdist or other artifact that preserves source paths, `False` if building a wheel or other artifact that uses installation paths
- **version** – The project's version

Raises

- *NotVersioningitError* –
  - if `config` is `None` and `project_dir` does not contain a `pyproject.toml` file
  - if the `pyproject.toml` file does not contain a `[tool.versioningit]` table
- *ConfigError* – if any of the values in `config` are not of the correct type
- *MethodError* – if a method returns a value of the wrong type

5.5 Passing Explicit Configuration

The functions & methods that take a path to a project directory normally read the project's configuration from the `pyproject.toml` file therein, but they can also be passed a `config` argument to take the configuration from instead, in which case `pyproject.toml` will be ignored and need not even exist.

A `config` argument must be a `dict` whose structure mirrors the structure of the `[tool.versioningit]` table in `pyproject.toml`. For example, the following TOML configuration:

```toml
[tool.versioningit.vcs]
method = "git"
match = ["v*"]

[tool.versioningit.next-version]
method = { module = "setup", value = "my_next_version" }

[tool.versioningit.format]
distance = "{next_version}.dev{distance}+{vcs}{rev}"
dirty = "{version}+dirty"
distance-dirty = "{next_version}.dev{distance}+{vcs}{rev}.dirty"
```

corresponds to the following Python `config` value:

```python
{
    "vcs": {
        "method": "git",
        "match": ["v*"],
    },
    "next-version": {
        "method": {
            "module": "setup",
            "value": "my_next_version",
        },
    },
    "format": {
        "distance": "{next_version}.dev{distance}+{vcs}{rev}",
```

(continues on next page)

(continued from previous page)

```
        "dirty": "{version}+dirty",
        "distance-dirty": "{next_version}.dev{distance}+{vcs}{rev}.dirty",
    },
}
```

This is the same structure that you would get by reading from the `pyproject.toml` file like so:

```
import tomli

with open("pyproject.toml", "rb") as fp:
    config = tomli.load(fp)["tool"]["versioningit"]
```

When passing `versioningit` configuration as a `config` argument, an alternative way to specify methods becomes available: in place of a method specification, one can pass a callable object directly.

CHAPTER

SIX

WRITING YOUR OWN METHODS

Changed in version 1.0.0: User parameters, previously passed as keyword arguments, are now passed as a single `params` argument.

If you need to customize how a `versioningit` step is carried out, you can write a custom function in a Python module in your project directory and point `versioningit` to that function as described under "*Specifying the Method*".

When a custom function is called, it will be passed a step-specific set of arguments, as documented below, plus all of the parameters specified in the step's subtable in `pyproject.toml`. (The arguments are passed as keyword arguments, so custom methods need to give them the same names as documented here.) For example, given the below configuration:

```
[tool.versioningit.vcs]
method = { module = "ving_methods", value = "my_vcs", module-dir = "tools" }
tag-dir = "tags"
annotated-only = true
```

`versioningit` will carry out the `vcs` step by calling `my_vcs()` in `ving_methods.py` in the `tools/` directory with the arguments `project_dir` (set to the directory in which the `pyproject.toml` file is located) and `params={"tag-dir": "tags", "annotated-only": True}`.

If a user-supplied parameter to a method is invalid, the method should raise a *versioningit.errors.ConfigError*. If a method is passed a parameter that it does not recognize, it should ignore it.

If you choose to store your custom methods in your `setup.py`, be sure to place the call to `setup()` under an `if __name__ == "__main__":` guard so that the module can be imported without executing `setup()`.

If you store your custom methods in a module other than `setup.py` that is not part of the project's Python package (e.g., if the module is stored in a `tools/` directory), you need to ensure that the module is included in your project's sdists but not in wheels.

If your custom method depends on any third-party libraries, they must be listed in your project's `build-system.requires`.

6.1 vcs

A custom `vcs` method is a callable with the following synopsis:

funcname(*, *project_dir: Union[str, pathlib.Path]*, *params: Dict[str, Any]*) → *versioningit.VCSDescription*

Parameters

- **project_dir** (*path*) – the path to a project directory
- **params** (*dict*) – a collection of user-supplied parameters

Returns a description of the state of the version control repository at the directory

Return type *versioningit.VCSDescription*

Raises

- ***versioningit.errors.NoTagError*** – If a tag cannot be determined for the repository
- ***versioningit.errors.NotVCSError*** – if `project_dir` is not under the expected type of version control

6.2 tag2version

A custom `tag2version` method is a callable with the following synopsis:

funcname(*, *tag: str*, *params: Dict[str, Any]*) → str

Parameters

- **tag** (*str*) – a tag retrieved from version control
- **params** (*dict*) – a collection of user-supplied parameters

Returns a version string extracted from `tag`

Return type str

Raises ***versioningit.errors.InvalidTagError*** – if the tag cannot be parsed

6.3 next-version

A custom `next-version` method is a callable with the following synopsis:

funcname(*, *version: str*, *branch: Optional[str]*, *params: Dict[str, Any]*) → str

Parameters

- **version** (*str*) – a project version (as extracted from a VCS tag)
- **branch** (*Optional[str]*) – the name of the VCS repository's current branch (if any)
- **params** (*dict*) – a collection of user-supplied parameters

Returns a version string for use as the `{next_version}` field in `[tool.versioningit.format]` format templates.

Return type str

Raises ***versioningit.errors.InvalidVersionError*** – if `version` cannot be parsed

6.4 format

A custom `format` method is a callable with the following synopsis:

funcname(*, *description: versioningit.VCSDescription*, *version: str*, *next_version: str*, *params: Dict[str, Any]*) → str

Parameters

- **description** – a *versioningit.VCSDescription* returned by a `vcs` method

- **version** (*str*) – a version string extracted from the VCS tag
- **next_version** (*str*) – a "next version" calculated by the `next-version` step
- **params** (*dict*) – a collection of user-supplied parameters

Returns the project's final version string

Return type str

Note that the `format` method is not called if `description.state` is `"exact"`, in which case the version returned by the `tag2version` step is used as the final version.

6.5 write

A custom `write` method is a callable with the following synopsis:

funcname(*, *project_dir: Union[str, pathlib.Path]*, *version: str*, *params: Dict[str, Any]*) → None

Parameters

- **project_dir** (*path*) – the path to a project directory
- **version** (*str*) – the project's final version
- **params** (*dict*) – a collection of user-supplied parameters

6.6 onbuild

A custom `onbuild` method is a callable with the following synopsis:

funcname(*, *build_dir: Union[str, pathlib.Path]*, *is_source: bool*, *version: str*, *params: Dict[str, Any]*) → None

Modifies one or more files in `build_dir`

Parameters

- **build_dir** (*path*) – the path to the directory where the project is being built
- **is_source** (*bool*) – true if an sdist or other artifact that preserves source paths is being built, false if a wheel or other artifact that uses installation paths is being built
- **version** (*str*) – the project's final version
- **params** (*dict*) – a collection of user-supplied parameters

6.7 Distributing Your Methods in an Extension Package

If you want to make your custom `versioningit` methods available for others to use, you can package them in a Python package and distribute it on PyPI. Simply create a Python package as normal that contains the method function, and specify the method function as an entry point of the project. The name of the entry point group is `versioningit.STEP` (though, for `next-version`, the group is spelled with an underscore instead of a hyphen: `versioningit.next_version`). For example, if you have a custom `vcs` method implemented as a `foobar_vcs()` function in `mypackage/vcs.py`, you would declare it in `setup.cfg` as follows:

```
[options.entry_points]
versioningit.vcs =
    foobar = mypackage.vcs:foobar_vcs
```

Once your package is on PyPI, package developers can use it by including it in their `build-system.requires` and specifying the name of the entry point (For the entry point above, this would be `foobar`) as the method name in the appropriate subtable. For example, a user of the `foobar` method for the `vcs` step would specify it as:

```
[tool.versioningit.vcs]
method = "foobar"
# Parameters go here
```

CHAPTER

SEVEN

RESTRICTIONS & CAVEATS

- When building or installing a project that uses `versioningit`, the entire repository history (or at least everything back through the most recent tag) must be available. This means that installing from a shallow clone (the default on most CI systems) will not work. If you are using the `"git"` or `"git-archive"` `vcs` method and have `default-tag` set in `[tool.versioningit.vcs]`, then shallow clones will end up assigned the default tag, which may or may not be what you want.
- If using the `[tool.versioningit.write]` subtable to write the version to a file, this file will only be updated whenever the project is built or installed. If using editable installs, this means that you must re-run `python setup.py develop` or `pip install -e .` after each commit if you want the version to be up-to-date.
- If you define & use a custom method inside your Python project's package, you will not be able to retrieve your project version by calling `importlib.metadata.version()` inside `__init__.py` — at least, not without a `try: ... except ...` wrapper. This is because `versioningit` loads the package containing the custom method before the package is installed, but `importlib.metadata.version()` only works after the package is installed.

CHAPTER

EIGHT

CHANGELOG

8.1 v1.1.1 (2022-04-08)

- Do not import setuptools unless needed (contributed by @jenshnielsen)

8.2 v1.1.0 (2022-03-03)

- Added custom setuptools commands for inserting the project version into a source file at build time
  - New step and subtable: "onbuild"
  - New public *get_cmdclasses()* and *run_onbuild()* functions
- Moved documentation from the README to a Read the Docs site
  - Established external documentation for the public library API
- When falling back to using `tool.versioningit.default-version`, emit a warning if the version is not PEP 440-compliant.
- The `versioningit` command now honors the `VERSIONINGIT_LOG_LEVEL` environment variable

8.3 v1.0.0 (2022-02-06)

- Changes to custom methods:
  - The signatures of the method functions have changed; user-supplied parameters are now passed as a single `params: Dict[str, Any]` argument instead of as keyword arguments.
  - User-supplied parameters with the same names as step-specific method arguments are no longer discarded.
- Changes to the "git-archive" method:
  - Lightweight tags are now ignored (by default, but see below) when installing from a repository in order to match the behavior of the `%(describe)` format placeholder.
  - The "match" and "exclude" settings are now parsed from the `describe-subst` parameter, which is now required, and the old `match` and `exclude` parameters are now ignored.
  - Git 2.35's "tags" option for honoring lightweight tags is now recognized.
  - Added a dedicated error message when an invalid `%(describe)` placeholder is "expanded" into itself in an archive

- The `file` parameter to the "basic" write method is now required when the `[tool.versioningit.write]` table is present. If you don't want to write the version to a file, omit the table entirely.
- Library API:
  - `Config` is no longer exported; it should now be considered private.
  - Merged `Versioningit.from_config()` functionality into *Versioningit.from_project_dir()*
  - Renamed `Versioningit.from_config_obj()` to `Versioningit.from_config()`; it should now be considered private

8.4 v0.3.3 (2022-02-04)

- Git 1.8.0 is now the minimum required version for the git methods, and this is documented. (Previously, the undocumented minimum version was Git 1.8.5.)
- Document the minimum supported Mercurial version as 5.2.

8.5 v0.3.2 (2022-01-16)

- Call `importlib.metadata.entry_points()` only once and reuse the result for a speedup (contributed by @jenshnielsen)

8.6 v0.3.1 (2022-01-02)

- Support Python 3.10
- Support tomli 2.0

8.7 v0.3.0 (2021-09-27)

- Gave the CLI interface an `-n`/`--next-version` option for showing a project's next release version
- Added a *get_next_version()* function
- Added a mention to the README of the existence of exported functionality other than *get_version()*
- Renamed the individual step-calling methods of *Versioningit* to have names of the form `do_$STEP()`

8.8 v0.2.1 (2021-08-02)

- Update for tomli 1.2.0

8.9 v0.2.0 (2021-07-13)

- The log messages displayed for unknown parameters are now at WARNING level instead of INFO and include suggestions for what you might have meant
- "git" `vcs` method: `default-tag` will now be honored if the **`git describe`** command fails (which generally only happens in a repository without any commits)
- Added an experimental "git-archive" method for determining a version when installing from a Git archive
- Project directories under `.git/` are no longer considered to be under version control
- Project directories inside Git working directories that are not themselves tracked by Git are no longer considered to be under version control
- Support added for installing from Mercurial repositories & archives

8.10 v0.1.0 (2021-07-08)

- Add more logging messages
- Changed default version formats to something that doesn't use `{next_version}`
- "basic" `tag2version` method:
  - If `regex` is given and it does not contain a group named "`version`," the entire text matched by the regex will be used as the version
  - Added a `require-match` parameter for erroring if the regex does not match
- "basic" `write` method: `encoding` now defaults to UTF-8
- New `next-version` methods: `"minor-release"`, `"smallest-release"`, and `"null"`
- Replaced `entrypoints` dependency with `importlib-metadata`
- Added `tool.versioningit.default-version` for setting the version to use if an error occurs
- When building a project from a shallow clone or in a non-sdist directory without VCS information, display an informative error message.

8.11 v0.1.0a1 (2021-07-05)

Alpha release

`versioningit` is yet another setuptools plugin for automatically determining your package's version based on your version control repository's tags. Unlike others, it allows easy customization of the version format and even lets you easily override the separate functions used for version extraction & calculation.

Features:

- Installed & configured through **PEP 518**'s `pyproject.toml`
- Supports Git, modern Git archives, and Mercurial
- Formatting of the final version uses format template strings, with fields for basic VCS information and separate template strings for distanced vs. dirty vs. distanced-and-dirty repository states
- Can optionally write the final version to a file for loading at runtime
- Provides custom setuptools commands for inserting the final version into a source file at build time
- The individual methods for VCS querying, tag-to-version calculation, version bumping, version formatting, and writing the version to a file can all be customized using either functions defined alongside one's project code or via publicly-distributed entry points
- Can alternatively be used as a library for use in `setup.py` or the like, in case you don't want to or can't configure it via `pyproject.toml`
- The only thing it does is calculate your version and optionally write it to a file; there's no overriding of your sdist contents based on what's in your Git repository, especially not without a way to turn it off, because that would just be rude.

CHAPTER

NINE

INSTALLATION & SETUP

`versioningit` requires Python 3.6 or higher. Just use pip for Python 3 (You have pip, right?) to install `versioningit` and its dependencies:

```
python3 -m pip install versioningit
```

However, usually you won't need to install `versioningit` in your environment directly. Instead, you specify it in your project's `pyproject.toml` file in the `build-system.requires` key, like so:

```
[build-system]
requires = [
    "setuptools >= 42",  # At least v42 of setuptools required!
    "versioningit ~= 1.0",
    "wheel"
]
build-backend = "setuptools.build_meta"
```

Then, you configure `versioningit` by adding a `[tool.versioningit]` table to your `pyproject.toml`. See "*Configuration*" for details, but you can get up & running with just the minimal configuration, an empty table:

```
[tool.versioningit]
```

`versioningit` replaces the need for (and will overwrite) the `version` keyword to the `setup()` function, so you should remove any such keyword from your `setup.py`/`setup.cfg` to reduce confusion.

Once you have a `[tool.versioningit]` table in your `pyproject.toml` — and once your repository has at least one tag — building your project with `setuptools` while `versioningit` is installed (which happens automatically if you set up your `build-system.requires` as above and you're using a **PEP 517** frontend like build) will result in your project's version automatically being set based on the latest tag in your Git repository. You can test your configuration and see what the resulting version will be using the `versioningit` command (see "*Command*").

CHAPTER

TEN

EXAMPLE CONFIGURATIONS

One of `versioningit`'s biggest strengths is its ability to configure the version format using placeholder strings. The default format configuration looks like this:

```toml
[tool.versioningit.format]

# Format used when there have been commits since the most recent tag:
distance = "{version}.post{distance}+{vcs}{rev}"

# Format used when there are uncommitted changes:
dirty = "{version}+d{build_date:%Y%m%d}"

# Format used when there are both commits and uncommitted changes:
distance-dirty = "{version}.post{distance}+{vcs}{rev}.d{build_date:%Y%m%d}"
```

Other format configurations of interest include:

- The default format used by setuptools_scm:

  ```toml
  [tool.versioningit.next-version]
  method = "smallest"

  [tool.versioningit.format]
  distance = "{next_version}.dev{distance}+{vcs}{rev}"
  dirty = "{version}+d{build_date:%Y%m%d}"
  distance-dirty = "{next_version}.dev{distance}+{vcs}{rev}.d{build_date:%Y%m%d}"
  ```

- The format used by versioneer:

  ```toml
  [tool.versioningit.format]
  distance = "{version}+{distance}.{vcs}{rev}"
  dirty = "{version}+{distance}.{vcs}{rev}.dirty"
  distance-dirty = "{version}+{distance}.{vcs}{rev}.dirty"
  ```

- The format used by vcversioner:

  ```toml
  [tool.versioningit.format]
  distance = "{version}.post{distance}"
  dirty = "{version}"
  distance-dirty = "{version}.post{distance}"
  ```

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

V

INDEX

Symbols

B

C

D

E

F

G

I

M

N

P

R

S

T

V