



# pygitversionhelper

---

User Manual

prune

CC BY-NC-SA 4.0

## Table of contents

---

1. pyGitVersionHelper	3
1.1 Features	3
1.2 Options	3
1.3 Process	3
2. Usage	4
2.1 Installation	4
2.2 Import in your project	4
2.3 Basic API	4
2.4 Limitations	6
3. Reference	7
3.1 Pygitversionhelper	7



# 1. pyGitVersionHelper

---

A tiny library to help versioning management of git python projects

Because a good developer is a lazy developer and version management in CI/CD can be very time consuming.

Checkout [Latest Documentation](#).

## 1.1 Features

---

- list tags
- get last tag
- get last version
- get current version (bumped)
- convert / switch from SemVer to PEP440 (both ways)
- automatic version format detection (SemVer by default)
- get commit message history

## 1.2 Options

---

- restrict to same branch
- both SemVer and PEP440 support
- custom output format
- configurable default bump type: major, minor, patch or dev
- configurable default bump strategy: post, pre-patch, pre-minor, pre-major
- ignore non-version tag
- force version format

## 1.3 Process

---

- full CI/CD developpment: Gitea / Jenkins + few python libs (pylint, coverage, unittest, mkdocs)
- documentation generated mkdocs and self-hosted
- CI/CD on Linux, manually tested in Windows environnement

## 2. Usage

---

### 2.1 Installation

---

From pypi repository (prefered):

```
/> python -m pip install pygitversionhelper
```

From downloaded .whl file:

```
/> python -m pip install pygitversionhelper-<VERSION>-py3-none-any.whl
```

From master git repository:

```
/> python -m pip install git+https://chacha.ddns.net/gitea/chacha/pygitversionhelper.git@master
```

### 2.2 Import in your project

---

Add this line on the top of your python script:

```
from pygitversionhelper import gitversionhelper
```

[optionnal] If you need to catch exception from this module:

```
from pygitversionhelper import gitversionhelperException
```

### 2.3 Basic API

---

All the API commands are static so it is not needed to create instantiate any object.

They are all executed in the current active directory.

One easy way to change directory:

```
import os
os.chdir("<YOUR DIRECTORY>")
```

#### 2.3.1 sub-lib: repository

---

To check if a repository is dirty:

```
if gitversionhelper.repository.isDirty():
    print("repository is dirty")
```

#### 2.3.2 sub-lib: tag

---

List all tags [default to taggerdate order]:

```
for tag in gitversionhelper.tag.getTags():
    print(f"found tag: {tag}")
```

List all tags [using git refname order]:

```
for tag in gitversionhelper.tag.getTags("v:refname"):
    print(f"found tag: {tag}")
```

Get the last tag:

```
print(f"most recent repository tag: {gitversionhelper.tag.getLastTag()}")
```

Get the last tag [only on same branch]:

```
print(f"most recent repository tag: {gitversionhelper.tag.getLastTag(same_branch=True)}")
```

Get the distance from HEAD to last tag:

```
print(f"number of commit since last tag: {gitversionhelper.tag.getDistanceFromTag()}")
```

Get the distance from HEAD to last tag [only on same branch]:

```
print(f"number of commit since last tag: {gitversionhelper.tag.getDistanceFromTag(same_branch=True)}")
```

### 2.3.3 sub-lib: version

Get the last found version in the repository [return MetaVersion object]:

```
print(f"most recent repository version: {gitversionhelper.version.getLastVersion()}")
```

Get the last found version in the repository [return formatted string]:

```
print(f"most recent repository version: {gitversionhelper.version.getLastVersion(formated_output=True)}")
```

Others kwargs available to this function:

- `version_std`: string to force a version standard for rendering ["PEP440" or "SemVer"]
- `same_branch`: boolean to force searching on same branch
- `ignore_unknown_tags`: boolean to allow unknown tag to be ignored

Get the current version of the repository, automatically bump it if the last commit is not tagged [returns MetaVersion object]:

```
print(f"current repository version: {gitversionhelper.version.getCurrentVersion()}")
```

Or with formatted output:

```
print(f"current repository version: {gitversionhelper.version.getCurrentVersion(formated_output=True)}")
```

Typical usage in CI/CD env:

```
bumped_version = gitversionhelper.version.getCurrentVersion(
    formated_output=True, \
    version_std="PEP440", \
    bump_type="dev", \
    bump_dev_strategy="post")
print(f"current repository version: {bumped_version}")
```

kwargs available to this function:

- All same args as `getLastVersion()`
- `bump_type`: if version need to be pump, allow to configure next release update type: **major**, **minor**, **patch**, **dev**
- `bump_dev_strategy`: if `bump_type` is dev, allow to choose dev update strategy: **post**, **pre-patch**, **pre-minor**, **pre-major**

A version object can also be manually formatted:

```
_version = gitversionhelper.tag.getCurrentVersion()
```

Then;

```
_version.doFormatVersion()
```

or;

```
gitversionhelper.version.doFormatVersion(_version)
```

kwargs available to this function:

- `output_format`: string to choose a rendering format ["Auto", "PEP440" or "SemVer"]

## 2.4 Limitations

---

There is unfortunately some technical limitation :

- MultiThreading and async behavior is not tested / supported.
- Multiple tag on the same commit is not supported.
- Branch filter when searching for a version is only tested with -no-ff strategy

## 3. Reference

---

### 3.1 Pygitversionhelper

---

#### 3.1.1 Gitversionhelper

This project try to help doing handy operations with git when dealing with project versioning and tags on python project - at least for project using PEP440 or SemVer standards.

One requirement is to keep it compact and to not cover too much fancy features. This is the reason why it is one single file with nested classes.

This library is made for repository that uses tags as version. Support for non-version tags is optional and not well tested.

This module is the main project file, containing all the code.

Read the read me for more information. Check the unittest s for usage samples.

#### Note

Other Parameters are **\*\*kwargs**

#### ATTRIBUTE

#### DESCRIPTION

TKwargs

Kwargs type definition for type hints

#### Attributes

#### TKwargs module-attribute

```
TKwargs = TypedDict('TKwargs', {'version_std': Optional[str], 'same_branch': Optional[bool], 'formatted_output': Optional[bool], 'bump_type': Optional[str], 'bump_dev_strategy': Optional[str], 'merged_output': Optional[bool], 'ignore_unknown_tags': Optional[bool], 'output_format': Optional[str], 'ignore_merged': Optional[bool]}, total=False)
```

#### Classes

#### gitversionhelper

main gitversionhelper class

Classes

commit

class containing methods focusing on commits

Classes

commitException

Bases: `gitversionhelperException`

generic commit exception

commitNotFound

Bases: `commitException`

tag not found exception

## Functions

`getFromTag` classmethod

```
getFromTag(tag: str) -> str
```

retrieve a commit from repository associated to a tag Args: tag: tag of the commit Returns: the commit Id

`getLast` classmethod

```
getLast(**kwargs: Unpack[TKwargs]) -> str
```

retrieve last commit from repository Keyword Arguments: kwargs/same\_branch (bool): force searching only in the same branch  
kwargs/ignore\_merged (bool): ignore merged commits Returns: the commit Id

`getMessage` classmethod

```
getMessage(commit_hash: str) -> str
```

retrieve a commit message from repository Args: commit\_hash: id of the commit Returns: the commit message

`getMessagesSinceTag` classmethod

```
getMessagesSinceTag(tag: str, **kwargs: Unpack[TKwargs]) -> str | List[str]
```

Retrieve a commits message history from repository.

Start from Latest found commit until the given tag. Args: tag: tag of the commit where search will stop Keyword Arguments:  
kwargs/merged\_output (bool): Output one single merged string kwargs/same\_branch (bool): Force searching only in the same  
branch kwargs/ignore\_merged (bool): ignore merged commits Returns: the commit message

repository

class containing methods focusing on repository

## Classes

`notAGitRepository`

Bases: `repositoryException`

not-a-git-repository repository exception

`repositoryDirty`

Bases: `repositoryException`

dirty-repository repository exception

`repositoryException`

Bases: `gitversionhelperException`

generic repository exception

## Functions

`isDirty` classmethod

```
isDirty() -> bool
```

check if the repository is in dirty state Returns: True if it is dirty

tag

class containing methods focusing on tags

Attributes

`__validGitTagSort` class-attribute instance-attribute

```
__validGitTagSort = ['', 'v:refname', '-v:refname', 'taggerdate', 'committerdate', '-taggerdate', '-committerdate']
```

## Classes

`moreThanOneTag`

Bases: `tagException`

more-than-one-tag tag exception

`tagException`

Bases: `gitversionhelperException`

generic tag exception

`tagNotFound`

Bases: `tagException`

tag-not-found tag exception

## Functions

`getDistanceFromTag` classmethod

```
getDistanceFromTag(tag: Optional[str] = None, **kwargs: Unpack[TKwargs]) -> int
```

retrieve the distance between Latest commit and tag in the repository Arguments: tag: reference tag, if None the most recent one will be used Keyword Arguments: kwargs/same\_branch (bool): force searching only in the same branch Returns: the tag

`getLastTag` classmethod

```
getLastTag(**kwargs: Unpack[TKwargs]) -> str
```

retrieve the Latest tag from a repository Keyword Arguments: kwargs/same\_branch (bool): force searching only in the same branch Returns: the tag

`getTags` classmethod

```
getTags(Sort: str = 'taggerdate', **kwargs: Unpack[TKwargs]) -> List[str]
```

retrieve all tags from a repository Args: Sort: sorting constraints (git format) Keyword Arguments: kwargs/same\_branch (bool): force searching only in the same branch Returns: the tags list

`unknownGITErrors`

Bases: `gitversionhelperException`

unknown git error generic exception

`unknownGITFatalError`

Bases: `unknownGITErrors`

unknown fatal git error generic exception

`version`

class containing methods focusing on versions

## Attributes

`DefaultInputFormat` class-attribute instance-attribute

```
DefaultInputFormat = 'Auto'
```

**TVersionStds** class-attribute instance-attribute

```
TVersionStds = TypedDict('TVersionStds', {'regex': str, 'regex_preversion_num': str, 'regex_build_num': str}, total=False)
```

**VersionStds** class-attribute instance-attribute

```
VersionStds: dict[str, TVersionStds] = {'SemVer': {'regex': '^(?P<major>0|[1-9]\\d*)\\.?(?P<minor>0|[1-9]\\d*)\\.?(?P<patch>0|[1-9]\\d*)(?:(?P<prerelease>(?:[0-9]\\d*|\\d*[a-zA-Z-][0-9a-zA-Z-]*)?(?:\\.(?:0|[1-9]\\d*|\\d*[a-zA-Z-][0-9a-zA-Z-]*)*))?(?!(?P<buildmetadata>[0-9a-zA-Z-]+(?:\\.[0-9a-zA-Z-]+)*))?$', 'regex_preversion_num': '(?:\\.)(?P<num>(?:\\d+(?!\\w)+))', 'regex_build_num': '(?:\\.)(?P<num>(?:\\d+(?!\\w)+))', 'PEP440': {'regex': 'packaging_VERSION_PATTERN'}}
```

**\_\_versionReseted** class-attribute instance-attribute

```
__versionReseted = False
```

## Classes

### MetaVersion

```
MetaVersion(version_std: TVersionStd = 'Auto', major: int = 0, minor: int = 1, patch: int = 0, pre_count: int = 0, post_count: int = 0, raw: str = '0.1.0')
```

generic version object

### Attributes

**BumpDevStrategies** class-attribute instance-attribute

```
BumpDevStrategies: set[TBumpDevStrategies] = {'post', 'pre-patch', 'pre-minor', 'pre-major'}
```

**BumpTypes** class-attribute instance-attribute

```
BumpTypes: set[TBumpTypes] = {'major', 'minor', 'patch', 'dev'}
```

**DefaultBumpDevStrategy** class-attribute instance-attribute

```
DefaultBumpDevStrategy: TBumpDevStrategies = 'post'
```

**DefaultBumpType** class-attribute instance-attribute

```
DefaultBumpType: TBumpTypes = 'patch'
```

**TBumpDevStrategies** class-attribute instance-attribute

```
TBumpDevStrategies = Literal['post', 'pre-patch', 'pre-minor', 'pre-major']
```

**TBumpTypes** class-attribute instance-attribute

```
TBumpTypes = Literal['major', 'minor', 'patch', 'dev']
```

**TVersionStd** class-attribute instance-attribute

```
TVersionStd = Literal['Auto', 'PEP440', 'SemVer']
```

**major** class-attribute instance-attribute

```
major: int = major
```

**minor** class-attribute instance-attribute

```
minor: int = minor
```

**patch** class-attribute instance-attribute

```
patch: int = patch
```

**post\_count** class-attribute instance-attribute

```
post_count: int = post_count
```

**pre\_count** class-attribute instance-attribute

```
pre_count: int = pre_count
```

**raw** class-attribute instance-attribute

```
raw: str = raw
```

**version\_std** class-attribute instance-attribute

```
version_std: TVersionStd = version_std
```

## Functions

### bump

```
bump(amount: int = 1, **kwargs: Unpack[TKwargs]) -> gitversionhelper.version.MetaVersion | str
```

bump the version to the next one Args: amount: number of revision to bump Keyword Arguments: kwargs/bump\_type (str): the given bump\_type (can be None) kwargs/bump\_dev\_strategy (str): the given bump\_dev\_strategy (can be None) Returns: the bumped version

### doFormatVersion

```
doFormatVersion(**kwargs: Unpack[TKwargs]) -> str
```

output a formatted version string Keyword Arguments: kwargs/output\_format: output format to render ("Auto" or "PEP440" or "SemVer") Returns: formatted version string

### PreAndPostVersionUnsupported

Bases: `versionException`

pre and post release can not be present at the same time

### noValidVersion

Bases: `versionException`

no valid version found exception

### versionException

Bases: `gitversionhelperException`

generic version exception

## Functions

### doFormatVersion classmethod

```
doFormatVersion(inputversion: MetaVersion, **kwargs: Unpack[TKwargs]) -> str
```

output a formatted version string from a MetaVersion object Keyword Arguments: kwargs/output\_format (str): output format to render ("Auto" or "PEP440" or "SemVer") Args: inputversion: version to be rendered Returns: formatted version string

### getCurrentFormattedVersion classmethod

```
getCurrentFormattedVersion(**kwargs: Unpack[TKwargs]) -> str
```

same as getCurrentVersion() with formatted\_output kwarg forced activated. Keyword Arguments: kwargs/version\_std (str): the given version\_std (can be None) kwargs/same\_branch (bool): force searching only in the same branch kwargs/bump\_type (str): the given bump\_type (can be None) kwargs/bump\_dev\_strategy (str): the given bump\_dev\_strategy (can be None) kwargs/output\_format (str): output format to render ("Auto" or "PEP440" or "SemVer") Returns: the last version

### getCurrentVersion classmethod

```
getCurrentVersion(**kwargs: Unpack[TKwargs]) -> gitversionhelper.version.MetaVersion | str
```

get the current version or bump depending of repository state. Keyword Arguments: `kwargs/version_std` (str): the given `version_std` (can be None) `kwargs/same_branch` (bool): force searching only in the same branch `kwargs/formated_output` (bool): output a formated version string `kwargs/bump_type` (str): the given `bump_type` (can be None) `kwargs/bump_dev_strategy` (str): the given `bump_dev_strategy` (can be None) `kwargs/output_format` (str): output format to render ("Auto" or "PEP440" or "SemVer") Returns: the last version

`getLastVersion` classmethod

```
getLastVersion(**kwargs: Unpack[TKwargs]) -> gitversionhelper.version.MetaVersion | str
```

get the last version from tags Keyword Arguments: `kwargs/version_std` (str): the given `version_std` (can be None) `kwargs/same_branch` (bool): force searching only in the same branch `kwargs/formated_output` (bool): output a formated version string `kwargs/ignore_unknown_tags` (bool): skip tags with not decoded versions (default to False) Returns: the last version in `MetaVersion` object or string

wrongArguments

Bases: `gitversionhelperException`

wrong argument generic exception

## gitversionhelperException

Bases: `Exception`

general Module Exception