How To - Versioning Strategy - Python

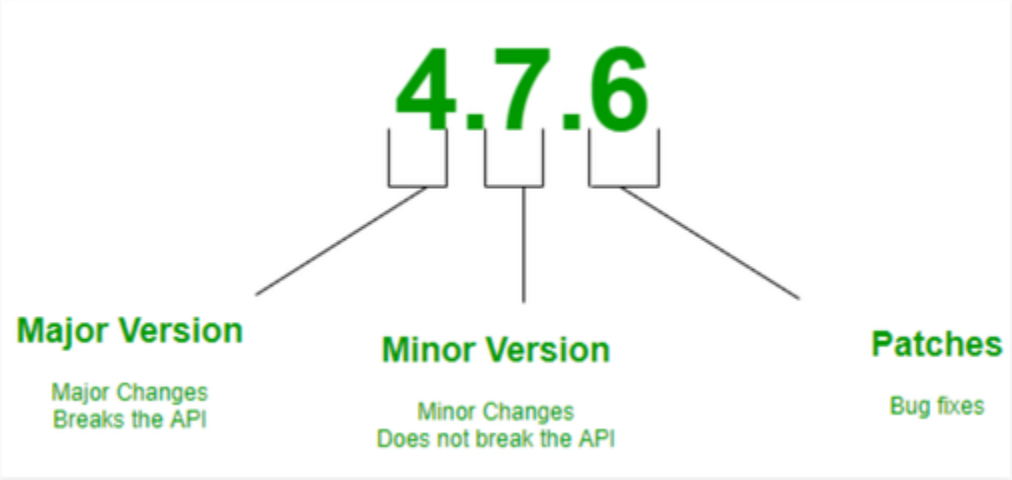
Introduction to Semantic Versioning

Semantic versioning (also referred as **SemVer**) is a versioning system that has been on the rise over the last few years. It has always been a problem for software developers, release managers and consumers. Having a universal way of versioning the software development projects is the best way to track what is going on with the software as new plugins, addons, libraries and extensions are being built almost everyday.

Semantic Versioning is a 3-component number in the format of **X.Y.Z**, where :

- X stands for a major version.
- Y stands for a minor version.
- Z stands for a patch.

So, SemVer is of the form **Major.Minor.Patch**.



Working : The goal of SemVer was to bring some sanity to the management of rapidly moving software release targets. As discussed above, 3 numbers i.e, Major, Minor and Patch are required to identify a software version. For example, if we take version 5.12.2, then it has a major version of 5, a minor version of 12 and a patch version of 2. Below given are the scenarios when you should bump the value of X, Y and Z.

- Bump the value of **X** when breaking the existing API.
- Bump the value of **Y** when implementing new features in a backward-compatible way.
- Bump the value of **Z** when fixing bugs.

Valid identifiers are in the set **[A-Za-z0-9]** and cannot be empty. Pre-release metadata is identified by appending a hyphen to the end of the SemVer sequence. Thus a pre-release for version 1.0.0 could be 1.0.0-alpha.1. Then if another build is needed, it would become 1.0.0-alpha.2, and so on. Note that names cannot contain leading zeros, but hyphens are allowed in names for pre-release identifiers.

Advantages of SemVer :

- You can keep track of every transition in the software development phase.
- Versioning can do the job of explaining the developers about what type of changes have taken place and the possible updates that should take place in the software.
- It helps to keep things clean and meaningful.
- It helps other people who might be using your project as a dependency.

Points to keep in mind :

- The first version starts at 0.1.0 and not at 0.0.1, as no bug fixes have taken place, rather we start off with a set of features as first draft of the project.
- Before 1.0.0 is only the Development Phase, where you focus on getting stuff done.
- SemVer does not cover libraries tagged **0.*.***. The first stable version is v1.0.0.

Automatic semantic versioning using Gitlab.

1. Pick the most recent version and bump it.

We can extract the most recent tag by running `git describe --tags`. The semver library can then be used to bump the version.

```
def main():
    try:
        latest = git("describe", "--tags").decode().strip()
    except subprocess.CalledProcessError:
        # No tags in the repository
        version = "1.0.0"
    else:
        # Skip already tagged commits
        if '-' not in latest:
            print(latest)
            return 0

        version = bump(latest)

    tag_repo(version)
    print(version)

    return 0
```

There is one catch, though - how will we decide whether to bump patch, minor or major version?

```
def bump(latest):
    # TODO decide what to bump
    # Then use bump_patch, bump_minor or bump_major
    return semver.bump_patch(latest)
```

- Make bumping patch the default behavior, as it is the most common operation.
- An intention to bump minor or major version can be marked in the commit message, e.g., by using some phrase like `#minor` or `bump-minor`.
- Similarly, you could attach labels to the Merge Requests called `bump-minor` and `bump-major`.
- You could come up with a script that detects whether any breaking changes were introduced or new functionalities added.

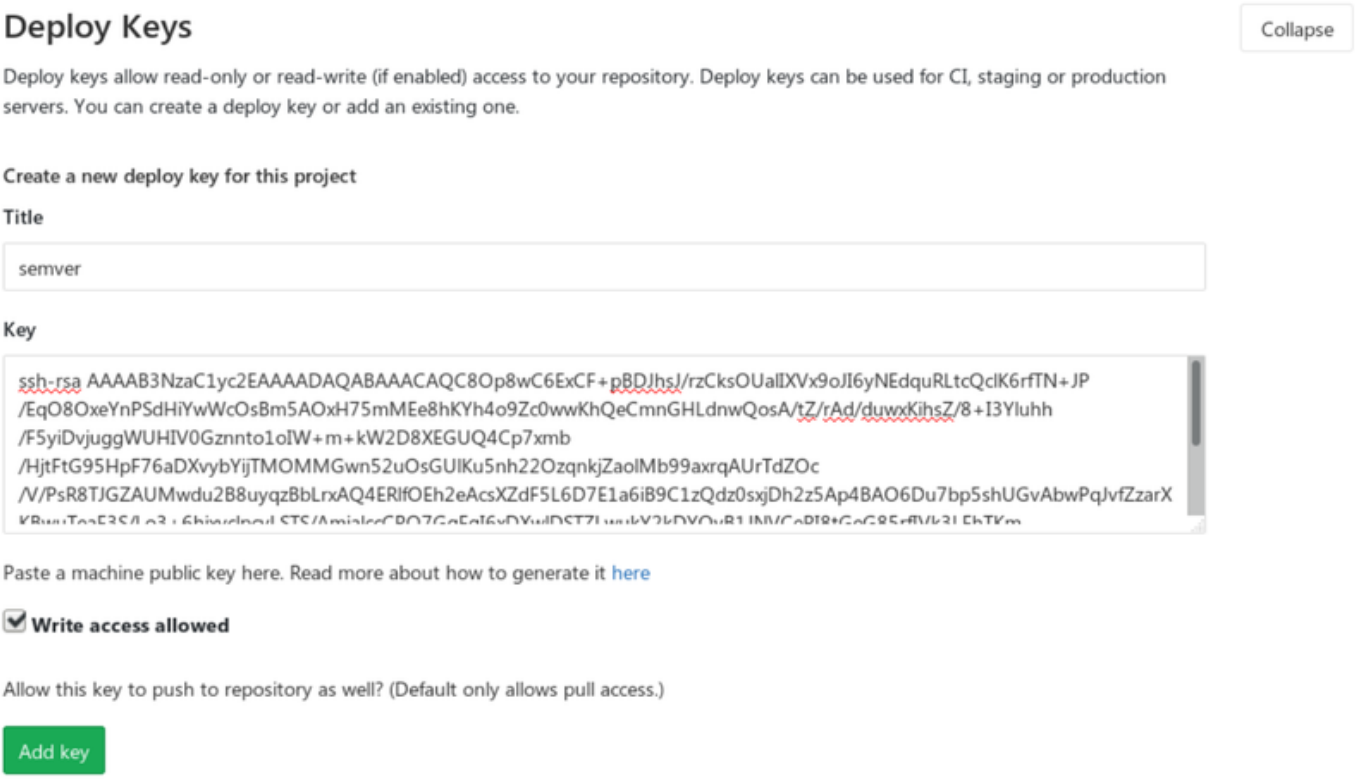
2. Add a new tag and push it to the remote repository.

Authentication

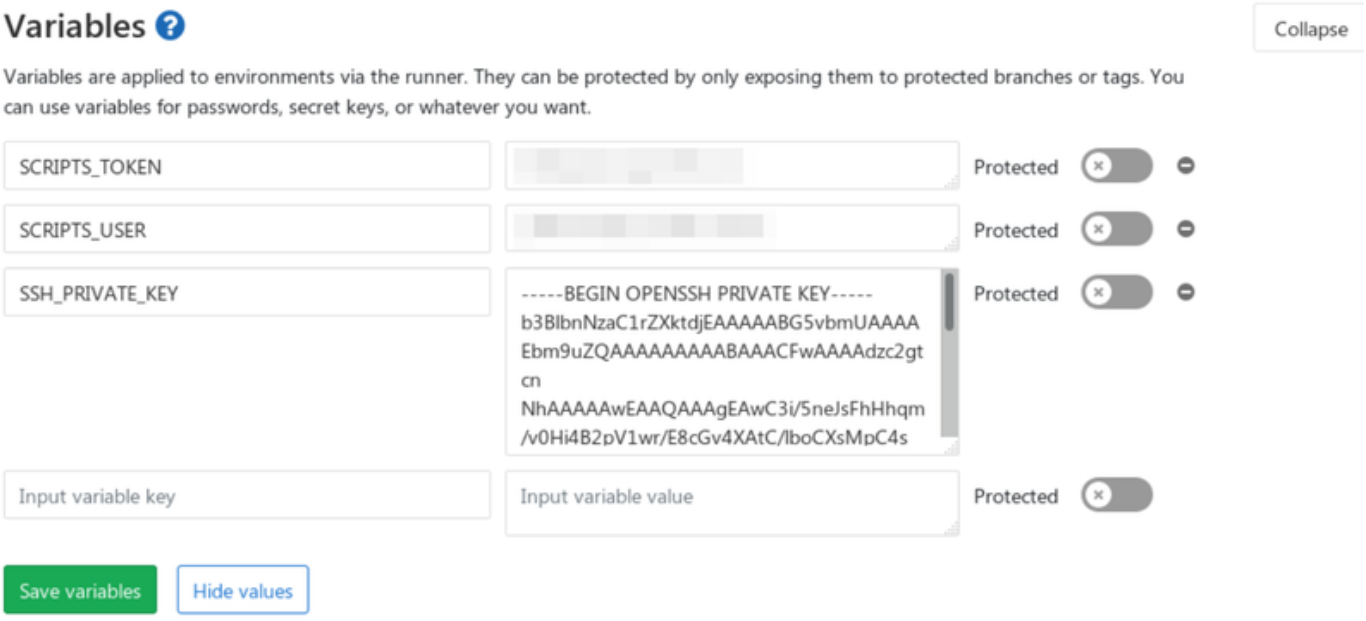
First, generate a new key on your local machine

```
ssh-keygen -t rsa -b 4096
```

Add the *public* part as a new **Deploy Key** in the `Settings -> Repository` section. Make sure to check the “Write access allowed” option.



Add the *private* part as a new Variable in the CI/CD section. The name is up to you; I'll stick with `SSH_PRIVATE_KEY`.



After you've saved the keys in GitLab, it is a good idea to delete the private key file.

All that's left is to add the SSH key to the CI definition.

```
script:
  - mkdir -p ~/.ssh && chmod 700 ~/.ssh
  - ssh-keyscan gitlab.com >> ~/.ssh/known_hosts && chmod 644 ~/.ssh/known_hosts
  - eval $(ssh-agent -s)
  - ssh-add <(echo "$SSH_PRIVATE_KEY")
```

Pushing the tag

A simple `git push` won't work right away, as the repository is cloned using HTTPS, not SSH. The simplest solution is to transform the remote push URL by using a regex.

```
def tag_repo(tag):
    url = os.environ["CI_REPOSITORY_URL"]

    # Transforms the repository URL to the SSH URL
    # Example input: https://gitlab-ci-token:xxxxxxxxxxxxxxxxxxx@gitlab.com/threedotslabs/ci-examples.git
    # Example output: git@gitlab.com:threedotslabs/ci-examples.git
    push_url = re.sub(r'.+@([^/]+)/', r'git@\1:', url)

    git("remote", "set-url", "--push", "origin", push_url)
    git("tag", tag)
    git("push", "origin", tag)
```

3. Pass information about the version to the next build steps.

It's very likely that more than one job in your pipeline will need to know the generated version. The idiomatic way to achieve this is to pass a file with the version as an artifact.

```
version:
  image: python:3.7-stretch
  stage: version
  script:
    - pip install semver
    - $SCRIPTS_DIR/common/gen-semver > version
  artifacts:
    paths:
      - version
  only:
    - branches

build:
  image: golang:1.11
  stage: build
  script:
    - export VERSION="unknown"
    - "[ -f ./version ] && export VERSION=$(cat ./version)"
    - $SCRIPTS_DIR/golang/build-semver . example-server main.Version "$VERSION"
  artifacts:
    paths:
      - bin/
  only:
    - branches
```

The next steps can now read the `./version` file. You can also make it more convenient with a one-liner placed in `before_script`:

```
before_script:
  - [ -f ./version ] && export VERSION=$(cat ./version)
```

Don’t run on tags

Remember to put proper `only` setting in your step definition or your automatic tag pushes will trigger new pipelines. Limiting it to `branches` shoul d be easy enough:

```
only:
  - branches
```

The pipeline

step process is required to set up automatic semantic versioning in any Gitlab repository:

1. Generate a unique version
2. Bump the version
3. Tag the latest tag build as latest

```
stages:
  - generate-env-vars
  - version
  - tag-latest

variables:
  IMAGE_NAME: $CI_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME

generate-env-vars:
  stage: generate-env-vars
  tags:
    - shell-tag
  script:
    - TAG=$(git describe --tags --always)
    - echo "export TAG=$TAG" > .variables
    - echo "export IMAGE=$IMAGE_NAME:$TAG" >> .variables
    - cat .variables
  artifacts:
    paths:
    - .variables

version:
  stage: version
  image: mrooding/gitlab-semantic-versioning:1.0.0
  script:
    - python3 /version-update/version-update.py
  only:
   - master

tag-latest:
  stage: tag-latest
  image: docker:18.06.1-ce
  before_script:
    - source .variables
  script:
    - docker pull $IMAGE
    - docker tag $IMAGE $IMAGE_NAME:latest
    - docker push $IMAGE_NAME:latest
  only:
    - tag
```

The `generate-env-vars` step uses `git describe –tags –always`to create a unique tag for every build. On tag builds, it defaults to the last available Git tag. For any branch build, it results in a combination of the last available Git tag and the short hash for the commit.