python-semver Documentation

Release 2.9.1

Kostiantyn Rybnikov and all

Feb 16, 2020

Contents

CHAPTER 1

Quickstart

A Python module for semantic versioning. Simplifies comparing versions.

Note: With version 2.9.0 we've moved the GitHub project. The project is now located under the organization `python-semver`. The complete URL is:

```
https://github.com/python-semver/python-semver
```

If you still have an old repository, correct your upstream URL to the new URL:

```
$ git remote set-url upstream git@github.com:python-semver/python-semver.git
```

The module follows the `MAJOR.MINOR.PATCH` style:

- `MAJOR` version when you make incompatible API changes,
- `MINOR` version when you add functionality in a backwards compatible manner, and
- `PATCH` version when you make backwards compatible bug fixes.

Additional labels for pre-release and build metadata are supported.

Warning: Major version 3.0.0 of semver will remove support for Python 2.7 and 3.4.

As anything comes to an end, this project will focus on Python 3.x. New features and bugfixes will be integrated only into the 3.x.y branch of semver.

The last version of semver which supports Python 2.7 and 3.4 will be 2.9.x. However, keep in mind, version 2.9.x is frozen: no new features nor backports will be integrated.

We recommend to upgrade your workflow to Python 3.x to gain support, bugfixes, and new features.

To import this library, use:

```
>>> import semver
```

Working with the library is quite straightforward. To turn a version string into the different parts, use the *semver.parse* function:

```
>>> ver = semver.parse('1.2.3-pre.2+build.4')
>>> ver['major']
1
>>> ver['minor']
2
>>> ver['patch']
3
>>> ver['prerelease']
'pre.2'
>>> ver['build']
'build.5'
```

To raise parts of a version, there are a couple of functions available for you. The *semver.parse_version_info* function converts a version string into a *semver.VersionInfo* class. The function *semver.VersionInfo.bump_major* leaves the original object untouched, but returns a new *semver.VersionInfo* instance with the raised major part:

```
>>> ver = semver.parse_version_info("3.4.5")
>>> ver.bump_major()
VersionInfo(major=4, minor=0, patch=0, prerelease=None, build=None)
```

It is allowed to concatenate different "bump functions":

```
>>> ver.bump_major().bump_minor()
VersionInfo(major=4, minor=0, patch=1, prerelease=None, build=None)
```

To compare two versions, semver provides the *semver.compare* function. The return value indicates the relationship between the first and second version:

```
>>> semver.compare("1.0.0", "2.0.0")
-1
>>> semver.compare("2.0.0", "1.0.0")
1
>>> semver.compare("2.0.0", "2.0.0")
0
```

There are other functions to discover. Read on!

CHAPTER 2

Installing semver

2.1 Pip

For Python 2:

```
pip install semver
```

For Python 3:

```
pip3 install semver
```

2.2 Linux Distributions

Note: Some Linux distributions can have outdated packages. These outdated packages does not contain the latest bug fixes or new features. If you need a newer package, you have these option:

- Ask the maintainer to update the package.
- Update the package for your favorite distribution and submit it.
- Use a Python virtual environment and **`pip install`**.

2.2.1 Arch Linux

1. Enable the community repositories first:

   ```
   [community]
   Include = /etc/pacman.d/mirrorlist
   ```

2. Install the package:

```
$ pacman -Sy python-semver
```

2.2.2 Debian

1. Update the package index:

```
$  sudo apt-get update
```

2. Install the package:

```
$ sudo apt-get install python3-semver
```

2.2.3 Fedora

```
$ dnf install python3-semver
```

2.2.4 FreeBSD

```
$ pkg install py36-semver
```

2.2.5 openSUSE

1. Enable the the `devel:languages:python` repository on the Open Build Service (replace `<VERSION>` with the preferred openSUSE release):

```
$ zypper addrepo https://download.opensuse.org/repositories/devel:/languages:/
↪python/openSUSE_Leap_<VERSION>/devel:languages:python.repo
```

2. Install the package:

```
$ zypper --repo devel_languages_python python3-semver
```

2.2.6 Ubuntu

1. Update the package index:

```
$ sudo apt-get update
```

2. Install the package:

```
$ sudo apt-get install python3-semver
```

CHAPTER 3

Using semver

The `semver` module can store a version in different types:

- as a string.
- as *`semver.VersionInfo`*, a dedicated class for a version type.
- as a dictionary.

Each type can be converted into the other, if the minimum requirements are met.

3.1 Knowing the Implemented semver.org Version

The semver.org is the authorative specification of how semantical versioning is definied. To know which version of semver.org is implemented in the semver libary, use the following constant:

```
>>> semver.SEMVER_SPEC_VERSION
'2.0.0'
```

3.2 Creating a Version

A version can be created in different ways:

- as a complete version string:

```
>>> semver.parse_version_info("3.4.5-pre.2+build.4")
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
>>> semver.VersionInfo.parse("3.4.5-pre.2+build.4")
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

- with individual parts:

```
>>> semver.format_version(3, 4, 5, 'pre.2', 'build.4')
'3.4.5-pre.2+build.4'
>>> semver.VersionInfo(3, 5)
VersionInfo(major=3, minor=5, patch=0, prerelease=None, build=None)
```

You can pass either an integer or a string for `major`, `minor`, or `patch`:

```
>>> semver.VersionInfo("3", "5")
VersionInfo(major=3, minor=5, patch=0, prerelease=None, build=None)
```

In the simplest form, `prerelease` and `build` can also be integers:

```
>>> semver.VersionInfo(1, 2, 3, 4, 5)
VersionInfo(major=1, minor=2, patch=3, prerelease=4, build=5)
```

If you pass an invalid version string you will get a `ValueError`:

```
>>> semver.parse("1.2")
Traceback (most recent call last)
...
ValueError: 1.2 is not valid SemVer string
```

3.3 Parsing a Version String

"Parsing" in this context means to identify the different parts in a string.

- With *semver.parse_version_info()*:

```
>>> semver.parse_version_info("3.4.5-pre.2+build.4")
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

- With *semver.VersionInfo.parse()* (basically the same as *semver.parse_version_info()*):

```
>>> semver.VersionInfo.parse("3.4.5-pre.2+build.4")
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

- With *semver.parse()*:

```
>>> semver.parse("3.4.5-pre.2+build.4")
{'major': 3, 'minor': 4, 'patch': 5,  'prerelease': 'pre.2', 'build': 'build.4'}
```

3.4 Checking for a Valid Semver Version

If you need to check a string if it is a valid semver version, use the classmethod *semver.VersionInfo.isvalid()*:

```
>>> VersionInfo.isvalid("1.0.0")
True
>>> VersionInfo.isvalid("invalid")
False
```

3.5 Accessing Parts of a Version

The *semver.VersionInfo* contains attributes to access the different parts of a version:

```
>>> v = VersionInfo.parse("3.4.5-pre.2+build.4")
>>> v.major
3
>>> v.minor
4
>>> v.patch
5
>>> v.prerelease
'pre.2'
>>> v.build
'build.4'
```

However, the attributes are read-only. You cannot change an attribute. If you do, you get an `AttributeError`:

```
>>> v.minor = 5
Traceback (most recent call last)
...
AttributeError: attribute 'minor' is readonly
```

In case you need the different parts of a version stepwise, iterate over the *semver.VersionInfo* instance:

```
>>> for item in VersionInfo.parse("3.4.5-pre.2+build.4"):
...     print(item)
3
4
5
pre.2
build.4
>>> list(VersionInfo.parse("3.4.5-pre.2+build.4"))
[3, 4, 5, 'pre.2', 'build.4']
```

3.6 Replacing Parts of a Version

If you want to replace different parts of a version, but leave other parts unmodified, use one of the functions *semver.replace()* or *semver.VersionInfo.replace()*:

- From a version string:

  ```
  >>> semver.replace("1.4.5-pre.1+build.6", major=2)
  '2.4.5-pre.1+build.6'
  ```

- From a *semver.VersionInfo* instance:

  ```
  >>> version = semver.VersionInfo.parse("1.4.5-pre.1+build.6")
  >>> version.replace(major=2, minor=2)
  VersionInfo(major=2, minor=2, patch=5, prerelease='pre.1', build='build.6')
  ```

If you pass invalid keys you get an exception:

```
>>> semver.replace("1.2.3", invalidkey=2)
Traceback (most recent call last)
...
TypeError: replace() got 1 unexpected keyword argument(s): invalidkey
>>> version = semver.VersionInfo.parse("1.4.5-pre.1+build.6")
>>> version.replace(invalidkey=2)
Traceback (most recent call last)
...
TypeError: replace() got 1 unexpected keyword argument(s): invalidkey
```

3.7 Converting Different Version Types

Depending which function you call, you get different types (as explained in the beginning of this chapter).

- From a string into *semver.VersionInfo*:

```
>>> semver.VersionInfo.parse("3.4.5-pre.2+build.4")
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

- From *semver.VersionInfo* into a string:

```
>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4"))
'3.4.5-pre.2+build.4'
```

- From a dictionary into *semver.VersionInfo*:

```
>>> d = {'major': 3, 'minor': 4, 'patch': 5,  'prerelease': 'pre.2', 'build':
↪'build.4'}
>>> semver.VersionInfo(**d)
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

  As a minimum requirement, your dictionary needs at least the `major` key, others can be omitted. You get a `TypeError` if your dictionary contains invalid keys. Only `major`, `minor`, `patch`, `prerelease`, and `build` are allowed.

- From a tuple into *semver.VersionInfo*:

```
>>> t = (3, 5, 6)
>>> semver.VersionInfo(*t)
VersionInfo(major=3, minor=5, patch=6, prerelease=None, build=None)
```

- From a *semver.VersionInfo* into a dictionary:

```
>>> v = semver.VersionInfo(major=3, minor=4, patch=5)
>>> semver.parse(str(v))
{'major': 3, 'minor': 4, 'patch': 5, 'prerelease': None, 'build': None}
```

3.8 Increasing Parts of a Version

The `semver` module contains the following functions to raise parts of a version:

- *semver.bump_major()*: raises the major part and set all other parts to zero. Set `prerelease` and `build` to `None`.

- *semver.bump_minor()*: raises the minor part and sets `patch` to zero. Set `prerelease` and `build` to `None`.
- *semver.bump_patch()*: raises the patch part. Set `prerelease` and `build` to `None`.
- *semver.bump_prerelease()*: raises the prerelease part and set `build` to `None`.
- *semver.bump_build()*: raises the build part.

```
>>> semver.bump_major("3.4.5-pre.2+build.4")
'4.0.0'
>>> semver.bump_minor("3.4.5-pre.2+build.4")
'3.5.0'
>>> semver.bump_patch("3.4.5-pre.2+build.4")
'3.4.6'
>>> semver.bump_prerelease("3.4.5-pre.2+build.4")
'3.4.5-pre.3'
>>> semver.bump_build("3.4.5-pre.2+build.4")
'3.4.5-pre.2+build.5'
```

3.9 Comparing Versions

To compare two versions depends on your type:

- **Two strings**

  Use *semver.compare()*:

  ```
  >>> semver.compare("1.0.0", "2.0.0")
  -1
  >>> semver.compare("2.0.0", "1.0.0")
  1
  >>> semver.compare("2.0.0", "2.0.0")
  0
  ```

  The return value is negative if `version1 < version2`, zero if `version1 == version2` and strictly positive if `version1 > version2`.

- **Two** *semver.VersionInfo* **types**

  Use the specific operator. Currently, the operators <, <=, >, >=, ==, and `!=` are supported:

  ```
  >>> v1 = VersionInfo.parse("3.4.5")
  >>> v2 = VersionInfo.parse("3.5.1")
  >>> v1 < v2
  True
  >>> v1 > v2
  False
  ```

- **A** *semver.VersionInfo* **type and a** `tuple`

  Use the operator as with two *semver.VersionInfo* types:

  ```
  >>> v = VersionInfo.parse("3.4.5")
  >>> v > (1, 0)
  True
  >>> v < (3, 5)
  True
  ```

The opposite does also work:

```
>>> (1, 0) < v
True
>>> (3, 5) > v
True
```

Other types cannot be compared (like dictionaries, lists etc).

If you need to convert some types into other, refer to *Converting Different Version Types*.

3.10 Comparing Versions through an Expression

If you need a more fine-grained approach of comparing two versions, use the `semver.match()` function. It expects two arguments:

1. a version string
2. a match expression

Currently, the match expression supports the following operators:

- < smaller than
- > greater than
- >= greater or equal than
- <= smaller or equal than
- == equal
- != not equal

That gives you the following possibilities to express your condition:

```
>>> semver.match("2.0.0", ">=1.0.0")
True
>>> semver.match("1.0.0", ">1.0.0")
False
```

3.11 Getting Minimum and Maximum of two Versions

```
>>> semver.max_ver("1.0.0", "2.0.0")
'2.0.0'
>>> semver.min_ver("1.0.0", "2.0.0")
'1.0.0'
```

3.12 Dealing with Invalid Versions

As semver follows the semver specification, it cannot parse version strings which are considered "invalid" by that specification. The semver library cannot know all the possible variations so you need to help the library a bit.

For example, if you have a version string `v1.2` would be an invalid semver version. However, "basic" version strings consisting of major, minor, and patch part, can be easy to convert. The following function extract this information and returns a tuple with two items:

```
import re

BASEVERSION = re.compile(
    r"""[vV]?
        (?P<major>0|[1-9]\d*)
        (\.
        (?P<minor>0|[1-9]\d*)
        (\.
            (?P<patch>0|[1-9]\d*)
        )?
        )?
    """,
    re.VERBOSE,
)
def coerce(version):
    """
    Convert an incomplete version string into a semver-compatible VersionInfo
    object

    * Tries to detect a "basic" version string (``major.minor.patch``).
    * If not enough components can be found, missing components are
        set to zero to obtain a valid semver version.

    :param str version: the version string to convert
    :return: a tuple with a :class:`VersionInfo` instance (or ``None``
        if it's not a version) and the rest of the string which doesn't
        belong to a basic version.
    :rtype: tuple(:class:`VersionInfo` | None, str)
    """
    match = BASEVERSION.search(version)
    if not match:
        return (None, version)

    ver = {
        key: 0 if value is None else value
        for key, value in match.groupdict().items()
    }
    ver = semver.VersionInfo(**ver)
    rest = match.string[match.end() :]
    return ver, rest
```

The function returns a *tuple*, containing a `VersionInfo` instance or None as the first element and the rest as the second element. The second element (the rest) can be used to make further adjustments.

For example:

```
>>> coerce("v1.2")
(VersionInfo(major=1, minor=2, patch=0, prerelease=None, build=None), '')
>>> coerce("v2.5.2-bla")
(VersionInfo(major=2, minor=5, patch=2, prerelease=None, build=None), '-bla')
```

CHAPTER 4

pysemver 2.9.1

4.1 Synopsis

```
pysemver compare <VERSION1> <VERSION2>
pysemver bump {major,minor,patch,prerelease,build} <VERSION>
```

4.2 Description

The semver library provides a command line interface with the name **pysemver** to make the functionality accessible for shell scripts. The script supports several subcommands.

4.2.1 Global Options

-h, --help
Display usage summary.

--version
Show program's version number and exit.

4.3 Commands

4.3.1 pysemver bump

Bump a version.

```
pysemver bump <PART> <VERSION>
```

<PART>
The part to bump. Valid strings can be `major`, `minor`, `patch`, `prerelease`, or `build`. The part has the following effects:

- `major`: Raise the major part of the version and set minor and patch to zero, remove prerelease and build.
- `minor`: Raise the minor part of the version and set patch to zero, remove prerelease and build.
- `patch`: Raise the patch part of the version and remove prerelease and build.
- `prerelease` Raise the prerelease of the version and remove the build part.
- `build`: Raise the build part.

<VERSION>
The version to bump.

To bump a version, you pass the name of the part (`major`, `minor`, `patch`, `prerelease`, or `build`) and the version string. The bumped version is printed on standard out:

```
$ pysemver bump major 1.2.3
2.0.0
$ pysemver bump minor 1.2.3
1.3.0
```

If you pass a version string which is not a valid semantical version, you get an error message and a return code != 0:

```
$ pysemver bump build 1.5
ERROR 1.5 is not valid SemVer string
```

4.3.2 pysemver check

Checks if a string is a valid semver version.

```
pysemver check <VERSION>
```

<VERSION>
The version string to check.

The *error code* returned by the script indicates if the version is valid (=0) or not (!=0):

```
$ pysemver check 1.2.3; echo $?
0
$ pysemver check 2.1; echo $?
ERROR Invalid version '2.1'
2
```

4.3.3 pysemver compare

Compare two versions.

```
pysemver compare <VERSION1> <VERSION2>
```

<VERSION1>
First version

<VERSION2>
Second version

When you compare two versions, the result is printed on *standard out*, to indicates which is the bigger version:

- `-1` if first version is smaller than the second version,
- `0` if both versions are the same,
- `1` if the first version is greater than the second version.

4.4 Return Code

The *return code* of the script (accessible by `$?` from the Bash) indicates if the subcommand returned successfully nor not. It is *not* meant as the result of the subcommand.

The result of the subcommand is printed on the standard out channel ("stdout" or `0`), any error messages to standard error ("stderr" or `2`).

For example, to compare two versions, the command expects two valid semver versions:

```
$ pysemver compare 1.2.3 2.4.0
-1
$ echo $?
0
```

The return code is zero, but the result is `-1`.

However, if you pass invalid versions, you get this situation:

```
$ pysemver compare 1.2.3 2.4
ERROR 2.4 is not valid SemVer string
$ echo $?
2
```

If you use the **`pysemver`** in your own scripts, check the return code first before you process the standard output.

4.5 See also

Documentation https://python-semver.readthedocs.io/

Source code https://github.com/python-semver/python-semver

Bug tracker https://github.com/python-semver/python-semver/issues

CHAPTER 5

Contributing to semver

The semver source code is managed using Git and is hosted on GitHub:

```
git clone git://github.com/python-semver/python-semver
```

5.1 Reporting Bugs and Feedback

If you think you have encountered a bug in semver or have an idea for a new feature? Great! We like to hear from you.

First, take the time to look into our GitHub issues tracker if this already covered. If not, changes are good that we avoid double work.

5.2 Fixing Bugs and Implementing New Features

Before you make changes to the code, we would highly appreciate if you consider the following general requirements:

- Make sure your code adheres to the Semantic Versioning specification.
- Check if your feature is covered by the Semantic Versioning specification. If not, ask on its GitHub project https://github.com/semver/semver.
- Write test cases if you implement a new feature.
- Test also for side effects of your new feature and run the complete test suite.
- Document the new feature, see *Documenting semver* for details.

5.3 Modifying the Code

We recommend the following workflow:

1. Fork our project on GitHub using this link: https://github.com/python-semver/python-semver/fork
2. Clone your forked Git repository (replace `GITHUB_USER` with your account name on GitHub):

```
$ git clone git@github.com:GITHUB_USER/python-semver.git
```

3. Create a new branch. You can name your branch whatever you like, but we recommend to use some meaningful name. If your fix is based on a existing GitHub issue, add also the number. Good examples would be:
   - `feature/123-improve-foo` when implementing a new feature in issue 123
   - `bugfix/234-fix-security-bar` a bugfixes for issue 234

   Use this **`git`** command:

```
$ git checkout -b feature/NAME_OF_YOUR_FEATURE
```

4. Work on your branch. Commit your work.
5. Write test cases and run the test suite, see *Running the Test Suite* for details.
6. Create a pull request. Describe in the pull request what you did and why. If you have open questions, ask.
7. Wait for feedback. If you receive any comments, address these.
8. After your pull request got accepted, delete your branch.
9. Use the `clean` command to remove build and test files and folders:

```
$ python setup.py clean
```

5.4 Running the Test Suite

We use pytest and tox to run tests against all supported Python versions. All test dependencies are resolved automatically.

You can decide to run the complete test suite or only part of it:

- To run all tests, use:

```
$ tox
```

  If you have not all Python interpreters installed on your system it will probably give you some errors (`InterpreterNotFound`). To avoid such errors, use:

```
$ tox --skip-missing-interpreters
```

  It is possible to use only specific Python versions. Use the `-e` option and one or more abbreviations (`py27` for Python 2.7, `py34` for Python 3.4 etc.):

```
$ tox -e py34
$ tox -e py27,py34
```

  To get a complete list, run:

```
$ tox -l
```

- To run only a specific test, pytest requires the syntax `TEST_FILE::TEST_FUNCTION`.

  For example, the following line tests only the function `test_immutable_major()` in the file `test_semver.py` for all Python versions:

  ```
  $ tox test_semver.py::test_immutable_major
  ```

  By default, pytest prints a dot for each test function only. To reveal the executed test function, use the following syntax:

  ```
  $ tox -- -v
  ```

  You can combine the specific test function with the `-e` option, for example, to limit the tests for Python 2.7 and 3.6 only:

  ```
  $ tox -e py27,py36 test_semver.py::test_immutable_major
  ```

Our code is checked against flake8 for style guide issues. It is recommended to run your tests in combination with **`flake8`**, for example:

```
$ tox -e py27,py36,flake8
```

5.5 Documenting semver

Documenting the features of semver is very important. It gives our developers an overview what is possible with semver, how it "feels", and how it is used efficiently.

Note: To build the documentation locally use the following command:

```
$ tox -e docs
```

The built documentation is available in `dist/docs`.

A new feature is *not* complete if it isn't proberly documented. A good documentation includes:

- **A docstring**

  Each docstring contains a summary line, a linebreak, the description of its arguments in Sphinx style, and an optional doctest. The docstring is extracted and reused in the *API* section. An appropriate docstring should look like this:

```
def compare(ver1, ver2):
    """Compare two versions

    :param ver1: version string 1
    :param ver2: version string 2
    :return: The return value is negative if ver1 < ver2,
            zero if ver1 == ver2 and strictly positive if ver1 > ver2
    :rtype: int

    >>> semver.compare("1.0.0", "2.0.0")
    -1
    >>> semver.compare("2.0.0", "1.0.0")
    1
```

(continues on next page)

(continued from previous page)

```
    >>> semver.compare("2.0.0", "2.0.0")
    0
    """
```

- **The documentation**

  A docstring is good, but in most cases it's too dense. Describe how to use your new feature in our documentation. Here you can give your readers more examples, describe it in a broader context or show edge cases.

CHAPTER 6

API

Python helper for Semantic Versioning (http://semver.org/)

semver.**SEMVER_SPEC_VERSION = '2.0.0'**
Contains the implemented semver.org version of the spec

class semver.**VersionInfo**(*major*, *minor=0*, *patch=0*, *prerelease=None*, *build=None*)
A semver compatible version class.

Parameters

- **major** (*int*) – version when you make incompatible API changes.
- **minor** (*int*) – version when you add functionality in a backwards-compatible manner.
- **patch** (*int*) – version when you make backwards-compatible bug fixes.
- **prerelease** (*str*) – an optional prerelease string
- **build** (*str*) – an optional build string

build
The build part of a version.

bump_build(*token='build'*)
Raise the build part of the version, return a new object but leave self untouched.

Parameters **token** – defaults to 'build'

Returns new object with the raised build part

Return type str

```
>>> ver = semver.parse_version_info("3.4.5-rc.1+build.9")
>>> ver.bump_build()
VersionInfo(major=3, minor=4, patch=5, prerelease='rc.1', build='build.10')
```

bump_major()
Raise the major part of the version, return a new object but leave self untouched.

Returns new object with the raised major part

Return type *VersionInfo*

```
>>> ver = semver.parse_version_info("3.4.5")
>>> ver.bump_major()
VersionInfo(major=4, minor=0, patch=0, prerelease=None, build=None)
```

bump_minor()
Raise the minor part of the version, return a new object but leave self untouched.

Returns new object with the raised minor part

Return type *VersionInfo*

```
>>> ver = semver.parse_version_info("3.4.5")
>>> ver.bump_minor()
VersionInfo(major=3, minor=5, patch=0, prerelease=None, build=None)
```

bump_patch()
Raise the patch part of the version, return a new object but leave self untouched.

Returns new object with the raised patch part

Return type *VersionInfo*

```
>>> ver = semver.parse_version_info("3.4.5")
>>> ver.bump_patch()
VersionInfo(major=3, minor=4, patch=6, prerelease=None, build=None)
```

bump_prerelease(*token='rc'*)
Raise the prerelease part of the version, return a new object but leave self untouched.

Parameters **token** – defaults to 'rc'

Returns new object with the raised prerelease part

Return type str

```
>>> ver = semver.parse_version_info("3.4.5-rc.1")
>>> ver.bump_prerelease()
VersionInfo(major=3, minor=4, patch=5, prerelease='rc.2', build=None)
```

classmethod isvalid(*version*)
Check if the string is a valid semver version.

Parameters **version** (*str*) – the version string to check

Returns True if the version string is a valid semver version, False otherwise.

Return type bool

major
The major part of a version.

minor
The minor part of a version.

static parse(*version*)
Parse version string to a VersionInfo instance.

Parameters **version** – version string

Returns a *semver.VersionInfo* instance

Return type *semver.VersionInfo*

```
>>> semver.VersionInfo.parse('3.4.5-pre.2+build.4')
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

patch
The patch part of a version.

prerelease
The prerelease part of a version.

replace(***parts*)

Replace one or more parts of a version and return a new *semver.VersionInfo* object, but leave self untouched

Parameters **parts** (*dict*) – the parts to be updated. Valid keys are: `major`, `minor`, `patch`, `prerelease`, or `build`

Returns the new *semver.VersionInfo* object with the changed parts

Raises TypeError, if `parts` contains invalid keys

semver.**bump_build**(*version*, *token='build'*)
Raise the build part of the version.

Parameters

- **version** – version string
- **token** – defaults to 'build'

Returns the raised version string

Return type str

```
>>> semver.bump_build('3.4.5-rc.1+build.9')
'3.4.5-rc.1+build.10'
```

semver.**bump_major**(*version*)
Raise the major part of the version.

Param version string

Returns the raised version string

Return type str

```
>>> semver.bump_major("3.4.5")
'4.0.0'
```

semver.**bump_minor**(*version*)
Raise the minor part of the version.

Param version string

Returns the raised version string

Return type str

```
>>> semver.bump_minor("3.4.5")
'3.5.0'
```

semver.**bump_patch**(*version*)
Raise the patch part of the version.

Param version string

Returns the raised version string

Return type str

```
>>> semver.bump_patch("3.4.5")
'3.4.6'
```

semver.**bump_prerelease**(*version*, *token='rc'*)
Raise the prerelease part of the version.

Parameters

- **version** – version string
- **token** – defaults to 'rc'

Returns the raised version string

Return type str

```
>>> semver.bump_prerelease('3.4.5', 'dev')
'3.4.5-dev.1'
```

semver.**cmd_bump**(*args*)
Subcommand: Bumps a version.

Synopsis: bump <PART> <VERSION> <PART> can be major, minor, patch, prerelease, or build

Parameters **args** (argparse.Namespace) – The parsed arguments

Returns the new, bumped version

semver.**cmd_check**(*args*)
Subcommand: Checks if a string is a valid semver version.

Synopsis: check <VERSION>

Parameters **args** (argparse.Namespace) – The parsed arguments

semver.**cmd_compare**(*args*)
Subcommand: Compare two versions

Synopsis: compare <VERSION1> <VERSION2>

Parameters **args** (argparse.Namespace) – The parsed arguments

semver.**cmp**(*a*, *b*)

semver.**comparator**(*operator*)
Wrap a VersionInfo binary op method in a type-check.

semver.**compare**(*ver1*, *ver2*)
Compare two versions.

Parameters

- **ver1** – version string 1
- **ver2** – version string 2

Returns The return value is negative if ver1 < ver2, zero if ver1 == ver2 and strictly positive if ver1 > ver2

Return type int

```
>>> semver.compare("1.0.0", "2.0.0")
-1
>>> semver.compare("2.0.0", "1.0.0")
1
>>> semver.compare("2.0.0", "2.0.0")
0
```

semver.**createparser**()
Create an argparse.ArgumentParser instance.

Returns parser instance

Return type argparse.ArgumentParser

semver.**finalize_version**(*version*)
Remove any prerelease and build metadata from the version.

Parameters **version** – version string

Returns the finalized version string

Return type str

```
>>> semver.finalize_version('1.2.3-rc.5')
'1.2.3'
```

semver.**format_version**(*major*, *minor*, *patch*, *prerelease=None*, *build=None*)
Format a version according to the Semantic Versioning specification.

Parameters

- **major** (*int*) – the required major part of a version
- **minor** (*int*) – the required minor part of a version
- **patch** (*int*) – the required patch part of a version
- **prerelease** (*str*) – the optional prerelease part of a version
- **build** (*str*) – the optional build part of a version

Returns the formatted string

Return type str

```
>>> semver.format_version(3, 4, 5, 'pre.2', 'build.4')
'3.4.5-pre.2+build.4'
```

semver.**main**(*cliargs=None*)
Entry point for the application script.

Parameters **cliargs** (*list*) – Arguments to parse or None (=use sys.argv)

Returns error code

Return type int

semver.**match**(*version*, *match_expr*)
Compare two versions through a comparison.

Parameters

- **version** (*str*) – a version string

- **match_expr** (*str*) – operator and version; valid operators are < smaller than > greater than >= greator or equal than <= smaller or equal than == equal != not equal

Returns True if the expression matches the version, otherwise False

Return type bool

```
>>> semver.match("2.0.0", ">=1.0.0")
True
>>> semver.match("1.0.0", ">1.0.0")
False
```

semver.**max_ver**(*ver1*, *ver2*)

Returns the greater version of two versions.

Parameters

- **ver1** – version string 1
- **ver2** – version string 2

Returns the greater version of the two

Return type *VersionInfo*

```
>>> semver.max_ver("1.0.0", "2.0.0")
'2.0.0'
```

semver.**min_ver**(*ver1*, *ver2*)

Returns the smaller version of two versions.

Parameters

- **ver1** – version string 1
- **ver2** – version string 2

Returns the smaller version of the two

Return type *VersionInfo*

```
>>> semver.min_ver("1.0.0", "2.0.0")
'1.0.0'
```

semver.**parse**(*version*)

Parse version to major, minor, patch, pre-release, build parts.

Parameters **version** – version string

Returns dictionary with the keys 'build', 'major', 'minor', 'patch', and 'prerelease'. The prerelease or build keys can be None if not provided

Return type dict

```
>>> ver = semver.parse('3.4.5-pre.2+build.4')
>>> ver['major']
3
>>> ver['minor']
4
>>> ver['patch']
5
>>> ver['prerelease']
'pre.2'
```

(continues on next page)

(continued from previous page)

```
>>> ver['build']
'build.4'
```

semver.**parse_version_info**(*version*)

Parse version string to a VersionInfo instance.

Parameters **version** – version string

Returns a *VersionInfo* instance

Return type *VersionInfo*

```
>>> version_info = semver.parse_version_info("3.4.5-pre.2+build.4")
>>> version_info.major
3
>>> version_info.minor
4
>>> version_info.patch
5
>>> version_info.prerelease
'pre.2'
>>> version_info.build
'build.4'
```

semver.**process**(*args*)

Process the input from the CLI.

Parameters

- **args** (argparse.Namespace) – The parsed arguments
- **parser** (argparse.ArgumentParser) – the parser instance

Returns result of the selected action

Return type str

semver.**replace**(*version*, ***parts*)

Replace one or more parts of a version and return the new string.

Parameters

- **version** (*str*) – the version string to replace
- **parts** (*dict*) – the parts to be updated. Valid keys are: major, minor, patch, prerelease, or build

Returns the replaced version string

Raises TypeError, if parts contains invalid keys

Return type str

```
>>> import semver
>>> semver.replace("1.2.3", major=2, patch=10)
'2.2.10'
```

CHAPTER 7

Change Log

All notable changes to this code base will be documented in this file, in every released version.

7.1 Version 2.9.1

Released 2020-02-16

Maintainer Tom Schraitle

7.1.1 Features

- #177 (PR #178): Fixed repository and CI links (moved https://github.com/k-bx/python-semver/ repository to https://github.com/python-semver/python-semver/)
- PR #179: Added note about moving this project to the new python-semver organization on GitHub
- #187 (PR #188): Added logo for python-semver organization and documentation
- #191 (PR #194): Created manpage for pysemver
- #196 (PR #197): Added distribution specific installation instructions
- #201 (PR #202): Reformatted source code with black
- #208 (PR #209): Introduce new function `semver.VersionInfo.isvalid()` and extend **`pysemver`** with **`check`** subcommand
- #210 (PR #215): Document how to deal with invalid versions
- PR #212: Improve docstrings according to PEP257

7.1.2 Bug Fixes

- #192 (PR #193): Fixed “pysemver” and “pysemver bump” when called without arguments

7.1.3 Removals

not available

7.2 Version 2.9.0

Released 2019-10-30

Maintainer Sébastien Celles <s.celles@gmail.com>

7.2.1 Features

- #59 (PR #164): Implemented a command line interface
- #85 (PR #147, PR #154): Improved contribution section
- #104 (PR #125): Added iterator to *semver.VersionInfo()*
- #112, #113: Added Python 3.7 support
- PR #120: Improved test_immutable function with properties
- PR #125: Created `setup.cfg` for pytest and tox
- #126 (PR #127): Added target for documentation in `tox.ini`
- #142 (PR #143): Improved usage section
- #144 (PR #156): Added *semver.replace()* and *semver.VersionInfo.replace()* functions
- #145 (PR #146): Added posargs in `tox.ini`
- PR #157: Introduce `conftest.py` to improve doctests
- PR #165: Improved code coverage
- PR #166: Reworked `.gitignore` file
- #167 (PR #168): Introduced global constant `SEMVER_SPEC_VERSION`

7.2.2 Bug Fixes

- #102: Fixed comparison between VersionInfo and tuple
- #103: Disallow comparison between VersionInfo and string (and int)
- #121 (PR #122): Use python3 instead of python3.4 in `tox.ini`
- PR #123: Improved `__repr__()` and derive class name from `type()`
- #128 (PR #129): Fixed wrong datatypes in docstring for *semver.format_version()*
- #135 (PR #140): Converted prerelease and build to string
- #136 (PR #151): Added testsuite to tarball
- #154 (PR #155): Improved README description

7.2.3 Removals

- #111 (PR #110): Droped Python 3.3
- #148 (PR #149): Removed and replaced `python setup.py test`

7.3 Version 2.8.2

Released 2019-05-19

Maintainer Sébastien Celles <s.celles@gmail.com>

Skipped, not released.

7.4 Version 2.8.1

Released 2018-07-09

Maintainer Sébastien Celles <s.celles@gmail.com>

7.4.1 Features

- #40 (PR #88): Added a static parse method to VersionInfo
- #77 (PR #47): Converted multiple tests into pytest.mark.parametrize
- #87, #94 (PR #93): Removed named tuple inheritance.
- #89 (PR #90): Added doctests.

7.4.2 Bug Fixes

- #98 (PR #99): Set prerelease and build to None by default
- #96 (PR #97): Made VersionInfo immutable

7.5 Version 2.8.0

Released 2018-05-16

Maintainer Sébastien Celles <s.celles@gmail.com>

7.5.1 Changes

- #82 (PR #83): Renamed `test.py` to `test_semver.py` so py.test can autodiscover test file

7.5.2 Additions

- #79 (PR #81, PR #84): Defined and improve a release procedure file
- #72, #73 (PR #75): Implemented `__str__()` and `__hash__()`

7.5.3 Removals

- #76 (PR #80): Removed Python 2.6 compatibility

7.6 Version 2.7.9

Released 2017-09-23

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.6.1 Additions

- #65 (PR #66): Added `semver.finalize_version()` function.

7.7 Version 2.7.8

Released 2017-08-25

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

- #62: Support custom default names for pre and build

7.8 Version 2.7.7

Released 2017-05-25

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

- #54 (PR #55): Added comparision between VersionInfo objects
- PR #56: Added support for Python 3.6

7.9 Version 2.7.2

Released 2016-11-08

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.9.1 Additions

- Added `semver.parse_version_info()` to parse a version string to a version info tuple.

7.9.2 Bug Fixes

- #37: Removed trailing zeros from prelease doesn't allow to parse 0 pre-release version
- Refine parsing to conform more strictly to SemVer 2.0.0.

  SemVer 2.0.0 specification §9 forbids leading zero on identifiers in the prerelease version.

7.10 Version 2.6.0

Released 2016-06-08

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.10.1 Removals

- Remove comparison of build component.

  SemVer 2.0.0 specification recommends that build component is ignored in comparisons.

7.11 Version 2.5.0

Released 2016-05-25

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.11.1 Additions

- Support matching 'not equal' with "!=".

7.11.2 Changes

- Made separate builds for tests on Travis CI.

7.12 Version 2.4.2

Released 2016-05-16

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.12.1 Changes

- Migrated README document to reStructuredText format.
- Used Setuptools for distribution management.
- Migrated test cases to Py.test.
- Added configuration for Tox test runner.

7.13 Version 2.4.1

Released 2016-03-04

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.13.1 Additions

- #23: Compared build component of a version.

7.14 Version 2.4.0

Released 2016-02-12

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.14.1 Bug Fixes

- #21: Compared alphanumeric components correctly.

7.15 Version 2.3.1

Released 2016-01-30

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.15.1 Additions

- Declared granted license name in distribution metadata.

7.16 Version 2.3.0

Released 2016-01-29

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.16.1 Additions

- Added functions to increment prerelease and build components in a version.

7.17 Version 2.2.1

Released 2015-08-04

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.17.1 Bug Fixes

- Corrected comparison when any component includes zero.

7.18 Version 2.2.0

Released 2015-06-21

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.18.1 Additions

- Add functions to determined minimum and maximum version.
- Add code examples for recently-added functions.

7.19 Version 2.1.2

Released 2015-05-23

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.19.1 Bug Fixes

- Restored current README document to distribution manifest.

7.20 Version 2.1.1

Released 2015-05-23

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.20.1 Bug Fixes

- Removed absent document from distribution manifest.

7.21 Version 2.1.0

Released 2015-05-22

Maintainer Kostiantyn Rybnikov <k-bx@k-bx.com>

7.21.1 Additions

- Documented installation instructions.
- Documented project home page.
- Added function to format a version string from components.
- Added functions to increment specific components in a version.

7.21.2 Changes

- Migrated README document to Markdown format.

7.21.3 Bug Fixes

- Corrected code examples in README document.

7.22 Version 2.0.2

Released 2015-04-14

Maintainer Konstantine Rybnikov <k-bx@k-bx.com>

7.22.1 Additions

- Added configuration for Travis continuous integration.
- Explicitly declared supported Python versions.

7.23 Version 2.0.1

Released 2014-09-24

Maintainer Konstantine Rybnikov <k-bx@k-bx.com>

7.23.1 Bug Fixes

- #9: Fixed comparison of equal version strings.

7.24 Version 2.0.0

Released 2014-05-24

Maintainer Konstantine Rybnikov <k-bx@k-bx.com>

7.24.1 Additions

- Grant license in this code base under BSD 3-clause license terms.

7.24.2 Changes

- Update parser to SemVer standard 2.0.0.
- Ignore build component for comparison.

7.25 Version 0.0.2

Released 2012-05-10

Maintainer Konstantine Rybnikov <k-bx@k-bx.com>

7.25.1 Changes

- Use standard library Distutils for distribution management.

7.26 Version 0.0.1

Released 2012-04-28

Maintainer Konstantine Rybnikov <kost-bebix@yandex.ru>

- Initial release.

CHAPTER 8

Indices and Tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

B

C

F

I

M

P

R

S

V