

---

# **python-semver Documentation**

***Release 3.0.0-dev.1***

**Kostiantyn Rybnikov and all**

**Oct 26, 2020**



CONTENTS

1 Quickstart 3

1.1 Installing semver . . . . . 4

1.2 Using semver . . . . . 6

1.3 Contributing to semver . . . . . 20

1.4 API . . . . . 24

1.5 pysemver 3.0.0-dev.1 . . . . . 33

1.6 Change Log . . . . . 35

1.7 Older Change Log . . . . . 42

2 Indices and Tables 49

Python Module Index 51

Index 53



**Warning:** This is a development version. Do **NOT** use it in production before the final 3.0.0 is released.



## QUICKSTART

A Python module for [semantic versioning](#). Simplifies comparing versions.

---

**Note:** This project works for Python 3.6 and greater only. If you are looking for a compatible version for Python 2, use the maintenance branch `maint/v2`.

The last version of semver which supports Python 2.7 to 3.5 will be 2.x.y However, keep in mind, the major 2 release 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.

---

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.

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.VersionInfo.parse` function:

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

To raise parts of a version, there are a couple of functions available for you. 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.VersionInfo.parse("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=1, patch=0, 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!

## 1.1 Installing semver

### 1.1.1 Release Policy

As semver uses [Semantic Versioning](#), breaking changes are only introduced in major releases (incremented X in “X.Y.Z”).

For users who want to stay with major 2 releases only, add the following version restriction:

```
semver>=2,<3
```

This line avoids surprises. You will get any updates within the major 2 release like 2.11.0 or above. However, you will never get an update for semver 3.0.0.

Keep in mind, as this line avoids any major version updates, you also will never get new exciting features or bug fixes.

You can add this line in your file `setup.py`, `requirements.txt`, or any other file that lists your dependencies.

### 1.1.2 Pip

```
pip3 install semver
```

If you want to install this specific version (for example, 2.10.0), use the command **pip** with an URL and its version:

```
pip3 install git+https://github.com/python-semver/python-semver.git@2.11.0
```



### 1.1.3 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**.
- 

#### Arch Linux

1. Enable the community repositories first:

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

2. Install the package:

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

#### Debian

1. Update the package index:

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

2. Install the package:

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

#### Fedora

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

#### FreeBSD

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

## openSUSE

1. Enable the `devel:languages:python` repository of the Open Build Service:

```
$ sudo zypper addrepo --refresh obs://devel:languages:python devel_languages_
↪python
```

2. Install the package:

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

## Ubuntu

1. Update the package index:

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

2. Install the package:

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

## 1.2 Using semver

The `semver` module can store a version in the `semver.VersionInfo` class. For historical reasons, a version can be also stored as a string or dictionary.

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

### 1.2.1 Getting the Implemented semver.org Version

The `semver.org` page is the authoritative specification of how semantic versioning is defined. To know which version of `semver.org` is implemented in the `semver` library, use the following constant:

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

### 1.2.2 Getting the Version of semver

To know the version of `semver` itself, use the following construct:

```
>>> semver.__version__
'3.0.0-dev.1'
```

### 1.2.3 Creating a Version

Due to historical reasons, the semver project offers two ways of creating a version:

- through an object oriented approach with the `semver.VersionInfo` class. This is the preferred method when using semver.
- through module level functions and builtin datatypes (usually string and dict). This method is still available for compatibility reasons, but are marked as deprecated. Using it will emit a `DeprecationWarning`.

#### Warning: Deprecation Warning

Module level functions are marked as *deprecated* in version 2.x.y now. These functions will be removed in semver 3. For details, see the sections [Replacing Deprecated Functions](#) and [Displaying Deprecation Warnings](#).

A `semver.VersionInfo` instance can be created in different ways:

- From a string (a Unicode string in Python 2):

```
>>> semver.VersionInfo.parse("3.4.5-pre.2+build.4")
VersionInfo(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
>>> semver.VersionInfo.parse(u"5.3.1")
VersionInfo(major=5, minor=3, patch=1, prerelease=None, build=None)
```

- From a byte string:

```
>>> semver.VersionInfo.parse(b"2.3.4")
VersionInfo(major=2, minor=3, patch=4, prerelease=None, build=None)
```

- From individual parts by a dictionary:

```
>>> 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')
```

Keep in mind, the major, minor, patch parts has to be positive.

```
>>> semver.VersionInfo(-1)
Traceback (most recent call last):
...
ValueError: 'major' is negative. A version can only be positive.
```

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 the keys major, minor, patch, prerelease, and build are allowed.

- From a tuple:

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

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

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

The old, deprecated module level functions are still available. If you need them, they return different builtin objects (string and dictionary). Keep in mind, once you have converted a version into a string or dictionary, it's an ordinary builtin object. It's not a special version object like the `semver.VersionInfo` class anymore.

Depending on your use case, the following methods are available:

- From individual version parts into a string

In some cases you only need a string from your version data:

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

- From a string into a dictionary

To access individual parts, you can use the function `semver.parse()`:

```
>>> semver.parse("3.4.5-pre.2+build.4")
OrderedDict([('major', 3), ('minor', 4), ('patch', 5), ('prerelease', 'pre.2'), (
↪ 'build', 'build.4')])
```

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
```

## 1.2.4 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'}
True
```

## 1.2.5 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()`:

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

## 1.2.6 Accessing Parts of a Version Through Names

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

```
>>> v = semver.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
```

If you need to replace different parts of a version, refer to section [Replacing Parts of a Version](#).

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

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

## 1.2.7 Accessing Parts Through Index Numbers

New in version 2.10.0.

Another way to access parts of a version is to use an index notation. The underlying `VersionInfo` object allows to access its data through the magic method `__getitem__`.

For example, the major part can be accessed by index number 0 (zero). Likewise the other parts:

```
>>> ver = semver.VersionInfo.parse("10.3.2-pre.5+build.10")
>>> ver[0], ver[1], ver[2], ver[3], ver[4]
(10, 3, 2, 'pre.5', 'build.10')
```

If you need more than one part at the same time, use the slice notation:

```
>>> ver[0:3]
(10, 3, 2)
```

Or, as an alternative, you can pass a `slice()` object:

```
>>> sl = slice(0,3)
>>> ver[sl]
(10, 3, 2)
```

Negative numbers or undefined parts raise an `IndexError` exception:

```
>>> ver = semver.VersionInfo.parse("10.3.2")
>>> ver[3]
Traceback (most recent call last):
...
IndexError: Version part undefined
>>> ver[-2]
Traceback (most recent call last):
...
IndexError: Version index cannot be negative
```

## 1.2.8 Replacing Parts of a Version

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

- 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')
```

- From a version string:

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

If you pass invalid keys you get an exception:

```
>>> semver.replace("1.2.3", invalidkey=2)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```

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

```

## 1.2.9 Converting a VersionInfo instance into Different Types

Sometimes it is needed to convert a `semver.VersionInfo` instance into a different type. For example, for displaying or to access all parts.

It is possible to convert a `semver.VersionInfo` instance:

- Into a string with the builtin function `str()`:

```

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

```

- Into a dictionary with `semver.VersionInfo.to_dict()`:

```

>>> v = semver.VersionInfo(major=3, minor=4, patch=5)
>>> v.to_dict()
OrderedDict([('major', 3), ('minor', 4), ('patch', 5), ('prerelease', None), (
↪ 'build', None)])

```

- Into a tuple with `semver.VersionInfo.to_tuple()`:

```

>>> v = semver.VersionInfo(major=5, minor=4, patch=2)
>>> v.to_tuple()
(5, 4, 2, None, None)

```

## 1.2.10 Raising Parts of a Version

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

- `semver.VersionInfo.bump_major()`: raises the major part and set all other parts to zero. Set prerelease and build to None.
- `semver.VersionInfo.bump_minor()`: raises the minor part and sets patch to zero. Set prerelease and build to None.
- `semver.VersionInfo.bump_patch()`: raises the patch part. Set prerelease and build to None.
- `semver.VersionInfo.bump_prerelease()`: raises the prerelease part and set build to None.
- `semver.VersionInfo.bump_build()`: raises the build part.

```

>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4").bump_major())
'4.0.0'
>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4").bump_minor())
'3.5.0'
>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4").bump_patch())
'3.4.6'
>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4").bump_prerelease())

```

(continues on next page)

(continued from previous page)

```
'3.4.5-pre.3'
>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4").bump_build())
'3.4.5-pre.2+build.5'
```

Likewise the module level functions `semver.bump_major()`.

### 1.2.11 Increasing Parts of a Version Taking into Account Prereleases

New in version 2.10.0: Added `semver.VersionInfo.next_version()`.

If you want to raise your version and take prereleases into account, the function `semver.VersionInfo.next_version()` would perhaps a better fit.

```
>>> v = semver.VersionInfo.parse("3.4.5-pre.2+build.4")
>>> str(v.next_version(part="prerelease"))
'3.4.5-pre.3'
>>> str(semver.VersionInfo.parse("3.4.5-pre.2+build.4").next_version(part="patch"))
'3.4.5'
>>> str(semver.VersionInfo.parse("3.4.5+build.4").next_version(part="patch"))
'3.4.5'
>>> str(semver.VersionInfo.parse("0.1.4").next_version("prerelease"))
'0.1.5-rc.1'
```

### 1.2.12 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` instances**

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

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

- **A `semver.VersionInfo` type and a `tuple()` or `list()`**

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



```
>>> v = semver.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
```

- A `semver.VersionInfo` type and a `str()`

You can use also raw strings to compare:

```
>>> v > "1.0.0"
True
>>> v < "3.5.0"
True
```

The opposite does also work:

```
>>> "1.0.0" < v
True
>>> "3.5.0" > v
True
```

However, if you compare incomplete strings, you get a `ValueError` exception:

```
>>> v > "1.0"
Traceback (most recent call last):
...
ValueError: 1.0 is not valid SemVer string
```

- A `semver.VersionInfo` type and a `dict()`

You can also use a dictionary. In contrast to strings, you can have an “incomplete” version (as the other parts are set to zero):

```
>>> v > dict(major=1)
True
```

The opposite does also work:

```
>>> dict(major=1) < v
True
```

If the dictionary contains unknown keys, you get a `TypeError` exception:

```
>>> v > dict(major=1, unknown=42)
Traceback (most recent call last):
...
TypeError: ... got an unexpected keyword argument 'unknown'
```

Other types cannot be compared.

If you need to convert some types into others, refer to *Converting a `VersionInfo` instance into Different Types*.

The use of these comparison operators also implies that you can use builtin functions that leverage this capability; builtins including, but not limited to: `max()`, `min()` (for examples, see [Getting Minimum and Maximum of Multiple Versions](#)) and `sorted()`.

### 1.2.13 Determining Version Equality

Version equality means for semver, that major, minor, patch, and prerelease parts are equal in both versions you compare. The build part is ignored. For example:

```
>>> v = semver.VersionInfo.parse("1.2.3-rc4+1e4664d")
>>> v == "1.2.3-rc4+dedbeef"
True
```

This also applies when a `semver.VersionInfo` is a member of a set, or a dictionary key:

```
>>> d = {}
>>> v1 = semver.VersionInfo.parse("1.2.3-rc4+1e4664d")
>>> v2 = semver.VersionInfo.parse("1.2.3-rc4+dedbeef")
>>> d[v1] = 1
>>> d[v2]
1
>>> s = set()
>>> s.add(v1)
>>> v2 in s
True
```

### 1.2.14 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
```

### 1.2.15 Getting Minimum and Maximum of Multiple Versions

Changed in version 2.10.2: The functions `semver.max_ver()` and `semver.min_ver()` are deprecated in favor of their builtin counterparts `max()` and `min()`.

Since `semver.VersionInfo` implements `__gt__()` and `__lt__()`, it can be used with builtins requiring

```
>>> max([semver.VersionInfo(0, 1, 0), semver.VersionInfo(0, 2, 0), semver.
↳ VersionInfo(0, 1, 3)])
VersionInfo(major=0, minor=2, patch=0, prerelease=None, build=None)
>>> min([semver.VersionInfo(0, 1, 0), semver.VersionInfo(0, 2, 0), semver.
↳ VersionInfo(0, 1, 3)])
VersionInfo(major=0, minor=1, patch=0, prerelease=None, build=None)
```

Incidentally, using `map()`, you can get the min or max version of any number of versions of the same type (convertible to `semver.VersionInfo`).

For example, here are the maximum and minimum versions of a list of version strings:

```
>>> str(max(map(semver.VersionInfo.parse, ['1.1.0', '1.2.0', '2.1.0', '0.5.10', '0.4.
↳ 99'])))
'2.1.0'
>>> str(min(map(semver.VersionInfo.parse, ['1.1.0', '1.2.0', '2.1.0', '0.5.10', '0.4.
↳ 99'])))
'0.4.99'
```

And the same can be done with tuples:

```
>>> max(map(lambda v: semver.VersionInfo(*v), [(1, 1, 0), (1, 2, 0), (2, 1, 0), (0, 5,
↳ 10), (0, 4, 99)]))
(2, 1, 0, None, None)
>>> min(map(lambda v: semver.VersionInfo(*v), [(1, 1, 0), (1, 2, 0), (2, 1, 0), (0, 5,
↳ 10), (0, 4, 99)]))
(0, 4, 99, None, None)
```

For dictionaries, it is very similar to finding the max version tuple: see [Converting a VersionInfo instance into Different Types](#).

The “old way” with `semver.max_ver()` or `semver.min_ver()` is still available, but not recommended:

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

### 1.2.16 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
import semver
```

(continues on next page)

(continued from previous page)

```
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() :] # noqa:E203
    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')
```

## 1.2.17 Replacing Deprecated Functions

Changed in version 2.10.0: The development team of semver has decided to deprecate certain functions on the module level. The preferred way of using semver is through the `semver.VersionInfo` class.

The deprecated functions can still be used in version 2.10.0 and above. In version 3 of semver, the deprecated functions will be removed.

The following list shows the deprecated functions and how you can replace them with code which is compatible for future versions:

- `semver.bump_major()`, `semver.bump_minor()`, `semver.bump_patch()`, `semver.bump_prerelease()`, `semver.bump_build()`

Replace them with the respective methods of the `semver.VersionInfo` class. For example, the function `semver.bump_major()` is replaced by `semver.VersionInfo.bump_major()` and calling the `str(versionobject)`:

```
>>> s1 = semver.bump_major("3.4.5")
>>> s2 = str(semver.VersionInfo.parse("3.4.5").bump_major())
>>> s1 == s2
True
```

Likewise with the other module level functions.

- `semver.finalize_version()`

Replace it with `semver.VersionInfo.finalize_version()`:

```
>>> s1 = semver.finalize_version('1.2.3-rc.5')
>>> s2 = str(semver.VersionInfo.parse('1.2.3-rc.5').finalize_version())
>>> s1 == s2
True
```

- `semver.format_version()`

Replace it with `str(versionobject)`:

```
>>> s1 = semver.format_version(5, 4, 3, 'pre.2', 'build.1')
>>> s2 = str(semver.VersionInfo(5, 4, 3, 'pre.2', 'build.1'))
>>> s1 == s2
True
```

- `semver.max_ver()`

Replace it with `max(version1, version2, ...)` or `max([version1, version2, ...])`:

```
>>> s1 = semver.max_ver("1.2.3", "1.2.4")
>>> s2 = str(max(map(semver.VersionInfo.parse, ("1.2.3", "1.2.4"))))
>>> s1 == s2
True
```

- `semver.min_ver()`

Replace it with `min(version1, version2, ...)` or `min([version1, version2, ...])`:

```
>>> s1 = semver.min_ver("1.2.3", "1.2.4")
>>> s2 = str(min(map(semver.VersionInfo.parse, ("1.2.3", "1.2.4"))))
>>> s1 == s2
True
```

- `semver.parse()`

Replace it with `semver.VersionInfo.parse()` and `semver.VersionInfo.to_dict()`:

```
>>> v1 = semver.parse("1.2.3")
>>> v2 = semver.VersionInfo.parse("1.2.3").to_dict()
>>> v1 == v2
True
```

- `semver.parse_version_info()`

Replace it with `semver.VersionInfo.parse()`:

```
>>> v1 = semver.parse_version_info("3.4.5")
>>> v2 = semver.VersionInfo.parse("3.4.5")
>>> v1 == v2
True
```

- `semver.replace()`

Replace it with `semver.VersionInfo.replace()`:

```
>>> s1 = semver.replace("1.2.3", major=2, patch=10)
>>> s2 = str(semver.VersionInfo.parse('1.2.3').replace(major=2, patch=10))
>>> s1 == s2
True
```

## 1.2.18 Displaying Deprecation Warnings

By default, deprecation warnings are [ignored in Python](#). This also affects `semver`'s own warnings.

It is recommended that you turn on deprecation warnings in your scripts. Use one of the following methods:

- Use the option `-Wd` to enable default warnings:

- Directly running the Python command:

```
$ python3 -Wd scriptname.py
```

- Add the option in the shebang line (something like `#!/usr/bin/python3`) after the command:

```
#!/usr/bin/python3 -Wd
```

- In your own scripts add a filter to ensure that *all* warnings are displayed:

```
import warnings
warnings.simplefilter("default")
# Call your semver code
```

For further details, see the section [Overriding the default filter](#) of the Python documentation.

### 1.2.19 Creating Subclasses from VersionInfo

If you do not like creating functions to modify the behavior of semver (as shown in section *Dealing with Invalid Versions*), you can also create a subclass of the `VersionInfo` class.

For example, if you want to output a “v” prefix before a version, but the other behavior is the same, use the following code:

```
class SemVerWithVPrefix(VersionInfo):
    """
    A subclass of VersionInfo which allows a "v" prefix
    """

    @classmethod
    def parse(cls, version):
        """
        Parse version string to a VersionInfo instance.

        :param version: version string with "v" or "V" prefix
        :type version: str
        :raises ValueError: when version does not start with "v" or "V"
        :return: a new instance
        :rtype: :class:`SemVerWithVPrefix`
        """
        if not version[0] in ("v", "V"):
            raise ValueError(
                "{v!r}: not a valid semantic version tag. Must start with 'v' or 'V'".
                ↪format(
                    v=version
                )
            )
        self = super(SemVerWithVPrefix, cls).parse(version[1:])
        return self

    def __str__(self):
        # Reconstruct the tag
        return "v" + super(SemVerWithVPrefix, self).__str__()
```

The derived class `SemVerWithVPrefix` can be used like the original class:

```
>>> v1 = SemVerWithVPrefix.parse("v1.2.3")
>>> assert str(v1) == "v1.2.3"
>>> print(v1)
v1.2.3
>>> v2 = SemVerWithVPrefix.parse("v2.3.4")
>>> v2 > v1
True
>>> bad = SemVerWithVPrefix.parse("1.2.4")
Traceback (most recent call last):
...
ValueError: '1.2.4': not a valid semantic version tag. Must start with 'v' or 'V'
```

## 1.3 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
```

### 1.3.1 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.

### 1.3.2 Prerequisites

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>.

### 1.3.3 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 an 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 and create a pull request:
  - a. Write test cases and run the complete test suite, see [Running the Test Suite](#) for details.
  - b. Write a changelog entry, see section [Adding a Changelog Entry](#).
  - c. If you have implemented a new feature, document it into our documentation to help our reader. See section [Documenting semver](#) for further details.
  - d. Create a [pull request](#). Describe in the pull request what you did and why. If you have open questions, ask.
5. Wait for feedback. If you receive any comments, address these.
6. After your pull request got accepted, delete your branch.



### 1.3.4 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
```

### 1.3.5 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 properly documented. A good documentation includes:

- **A docstring**

Each docstring contains a summary line, a linebreak, an optional directive (see next item), 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
    >>> semver.compare("2.0.0", "2.0.0")
    0

    """
```

- **An optional directive**

If you introduce a new feature, change a function/method, or remove something, it is a good practice to introduce Sphinx directives into the docstring. This gives the reader an idea what version is affected by this change.

The first required argument, `VERSION`, defines the version when this change was introduced. You can choose from:

- `.. versionadded:: VERSION`

Use this directive to describe a new feature.

- `.. versionchanged:: VERSION`

Use this directive to describe when something has changed, for example, new parameters were added, changed side effects, different return values, etc.

- `.. deprecated:: VERSION`

Use this directive when a feature is deprecated. Describe what should be used instead, if appropriate.

Add such a directive *after* the summary line, if needed. An appropriate directive could look like this:

```
def to_tuple(self):
    """
    Convert the VersionInfo object to a tuple.

    .. versionadded:: 2.10.0
    Renamed ``VersionInfo._astuple`` to ``VersionInfo.to_tuple`` to
    make this function available in the public API.
    [...]
    """
```

- **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.

### 1.3.6 Adding a Changelog Entry

A “Changelog” is a record of all notable changes made to a project. Such a changelog, in our case the `CHANGELOG.rst`, is read by our *users*. Therefore, any description should be aimed to users instead of describing internal changes which are only relevant to developers.

To avoid merge conflicts, we use the `Towncrier` package to manage our changelog.

The directory `changelog.d` contains “newsfragments” which are short ReST-formatted files. On release, those news fragments are compiled into our `CHANGELOG.rst`.

You don't need to install `towncrier` yourself, use the `tox` command to call the tool.

We recommend to follow the steps to make a smooth integration of your changes:

1. After you have created a new pull request (PR), add a new file into the directory `changelog.d`. Each filename follows the syntax:

```
<ISSUE>.<TYPE>.rst
```

where `<ISSUE>` is the GitHub issue number. In case you have no issue but a pull request, prefix your number with `pr.` `<TYPE>` is one of:

- `bugfix`: fixes a reported bug.
- `deprecation`: informs about deprecation warnings
- `doc`: improves documentation.
- `feature`: adds new user facing features.
- `removal`: removes obsolete or deprecated features.
- `trivial`: fixes a small typo or internal change that might be noteworthy.

For example: `123.feature.rst`, `pr233.removal.rst`, `456.bugfix.rst` etc.

2. Create the new file with the command:

```
tox -e changelog -- create 123.feature.rst
```

The file is created into the `changelog.d/` directory.

3. Open the file and describe your changes in RST format.

- Wrap symbols like modules, functions, or classes into double backticks so they are rendered in a monospace font.

- Prefer simple past tense or constructions with “now”.

4. Check your changes with:

```
tox -e changelog -- check
```

5. Optionally, build a draft version of the changelog file with the command:

```
tox -e changelog
```

6. Commit all your changes and push it.

This finishes your steps.

On release, the maintainer compiles a new `CHANGELOG.rst` file by running:

```
tox -e changelog -- build
```

This will remove all newsfragments inside the `changelog.d` directory, making it ready for the next release.

## 1.4 API

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

```
semver.SEMVER_SPEC_VERSION = '2.0.0'
```

Contains the implemented [semver.org](http://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** (`SupportsInt`) – version when you make incompatible API changes.
- **minor** (`SupportsInt`) – version when you add functionality in a backwards-compatible manner.
- **patch** (`SupportsInt`) – version when you make backwards-compatible bug fixes.
- **prerelease** (`Union[str, bytes, int, None]`) – an optional prerelease string
- **build** (`Union[str, bytes, int, None]`) – an optional build string

### property build

The build part of a version (read-only).

**Return type** `Optional[str]`

```
bump_build (token='build')
```

Raise the build part of the version, return a new object but leave self untouched.

**Parameters** **token** (`str`) – defaults to ‘build’

**Return type** `VersionInfo`

**Returns** new object with the raised build part

```
>>> ver = semver.VersionInfo.parse("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.

**Return type** *VersionInfo*

**Returns** new object with the raised major part

```
>>> ver = semver.VersionInfo.parse("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.

**Return type** *VersionInfo*

**Returns** new object with the raised minor part

```
>>> ver = semver.VersionInfo.parse("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.

**Return type** *VersionInfo*

**Returns** new object with the raised patch part

```
>>> ver = semver.VersionInfo.parse("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* (str) – defaults to 'rc'

**Return type** *VersionInfo*

**Returns** new object with the raised prerelease part

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

**compare(other)**

Compare self with other.

**Parameters** *other* (Union[*VersionInfo*, Dict[str, Union[int, str, None]], Collection[Union[int, str, None]], str) – the second version

**Return type** int

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

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

(continues on next page)

(continued from previous page)

```
>>> semver.VersionInfo.parse("2.0.0").compare("2.0.0")
0
>>> semver.VersionInfo.parse("2.0.0").compare(dict(major=2, minor=0, patch=0))
0
```

**finalize\_version()**

Remove any prerelease and build metadata from the version.

**Return type** `VersionInfo`

**Returns** a new instance with the finalized version string

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

**classmethod isvalid(version)**

Check if the string is a valid semver version.

New in version 2.9.1.

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

**Return type** `bool`

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

**property major**

The major part of a version (read-only).

**Return type** `int`

**match(match\_expr)**

Compare self to match a match expression.

**Parameters** **match\_expr** (`str`) – operator and version; valid operators are < smaller than > greater than >= greater or equal than <= smaller or equal than == equal != not equal

**Return type** `bool`

**Returns** True if the expression matches the version, otherwise False

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

**property minor**

The minor part of a version (read-only).

**Return type** `int`

**next\_version(part, prerelease\_token='rc')**

Determines next version, preserving natural order.

New in version 2.10.0.

This function is taking prereleases into account. The “major”, “minor”, and “patch” raises the respective parts like the `bump_*` functions. The real difference is using the “prerelease” part. It gives you the next patch version of the prerelease, for example:

```
>>> str(semver.VersionInfo.parse("0.1.4").next_version("prerelease"))
'0.1.5-rc.1'
```

**Parameters**

- **part** (`str`) – One of “major”, “minor”, “patch”, or “prerelease”
- **prerelease\_token** (`str`) – prefix string of prerelease, defaults to ‘rc’

**Return type** `VersionInfo`**Returns** new object with the appropriate part raised**classmethod** `parse` (`version`)Parse version string to a `VersionInfo` instance.

Changed in version 2.11.0: Changed method from static to classmethod to allow subclasses.

**Parameters** **version** (`Union[str, bytes]`) – version string**Return type** `VersionInfo`**Returns** a `VersionInfo` instance**Raises** **ValueError** – if version is invalid

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

**property** `patch`

The patch part of a version (read-only).

**Return type** `int`**property** `prerelease`

The prerelease part of a version (read-only).

**Return type** `Optional[str]`**replace** (`**parts`)Replace one or more parts of a version and return a new `VersionInfo` object, but leave self untouchedNew in version 2.9.0: Added `VersionInfo.replace()`**Parameters** **parts** (`Union[int, str, None]`) – the parts to be updated. Valid keys are:  
major, minor, patch, prerelease, or build**Return type** `VersionInfo`**Returns** the new `VersionInfo` object with the changed parts**Raises** **TypeError** – if parts contains invalid keys**to\_dict** ()Convert the `VersionInfo` object to an `OrderedDict`.New in version 2.10.0: Renamed `VersionInfo._asdict` to `VersionInfo.to_dict` to make this function available in the public API.**Return type** `Dict[str, Union[int, str, None]]`**Returns** an `OrderedDict` with the keys in the order major, minor, patch, prerelease, and build.

```
>>> semver.VersionInfo(3, 2, 1).to_dict()
OrderedDict([('major', 3), ('minor', 2), ('patch', 1), ('prerelease', None), (
↪ 'build', None)])
```

**to\_tuple()**

Convert the VersionInfo object to a tuple.

New in version 2.10.0: Renamed `VersionInfo._astuple` to `VersionInfo.to_tuple` to make this function available in the public API.

**Return type** `Tuple[int, int, int, Optional[str], Optional[str]]`

**Returns** a tuple with all the parts

```
>>> semver.VersionInfo(5, 3, 1).to_tuple()
(5, 3, 1, None, None)
```

**semver.bump\_build(version, token='build')**

Raise the build part of the version string.

Deprecated since version 2.10.0: Use `semver.VersionInfo.bump_build()` instead.

**Parameters**

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

**Returns** the raised version string

```
>>> 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 string.

Deprecated since version 2.10.0: Use `semver.VersionInfo.bump_major()` instead.

**Param** version string

**Returns** the raised version string

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

**semver.bump\_minor(version)**

Raise the minor part of the version string.

Deprecated since version 2.10.0: Use `semver.VersionInfo.bump_minor()` instead.

**Param** version string

**Returns** the raised version string

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

**semver.bump\_patch(version)**

Raise the patch part of the version string.

Deprecated since version 2.10.0: Use `semver.VersionInfo.bump_patch()` instead.

**Param** version string

**Returns** the raised version string

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



`semver.bump_prerelease(version, token='rc')`

Raise the prerelease part of the version string.

Deprecated since version 2.10.0: Use `semver.VersionInfo.bump_prerelease()` instead.

**Parameters**

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

**Returns** the raised version string

```
>>> 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** (Namespace) – The parsed arguments

**Return type** str

**Returns** the new, bumped version

`semver.cmd_check(args)`

Subcommand: Checks if a string is a valid semver version.

Synopsis: check <VERSION>

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

**Return type** None

`semver.cmd_compare(args)`

Subcommand: Compare two versions

Synopsis: compare <VERSION1> <VERSION2>

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

**Return type** str

`semver.compare(ver1, ver2)`

Compare two versions strings.

**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

```
>>> 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.

**Return type** `ArgumentParser`

**Returns** parser instance

`semver.deprecated(func=None, replace=None, version=None, category=<class 'DeprecationWarning'>)`

Decorates a function to output a deprecation warning.

**Parameters**

- **func** (Optional[~F]) – the function to decorate
- **replace** (Optional[str]) – the function to replace (use the full qualified name like `semver.VersionInfo.bump_major`).
- **version** (Optional[str]) – the first version when this function was deprecated.
- **category** – allow you to specify the deprecation warning class of your choice. By default, it's `DeprecationWarning`, but you can choose `PendingDeprecationWarning` or a custom class.

**Return type** `Union[Callable[..., ~F], partial]`

**Returns** decorated function which is marked as deprecated

`semver.finalize_version(version)`

Remove any prerelease and build metadata from the version string.

Deprecated since version 2.10.0: Use `semver.VersionInfo.finalize_version()` instead.

New in version 2.7.9: Added `finalize_version()`

**Parameters** **version** – version string

**Returns** the finalized version string

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

`semver.format_version(major, minor, patch, prerelease=None, build=None)`

Format a version string according to the Semantic Versioning specification.

Deprecated since version 2.10.0: Use `str(VersionInfo(VERSION))` instead.

**Parameters**

- **major** – the required major part of a version
- **minor** – the required minor part of a version
- **patch** – the required patch part of a version
- **prerelease** – the optional prerelease part of a version
- **build** – the optional build part of a version

**Returns** the formatted string

```
>>> 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`)

**Return type** `int`

**Returns** error code

`semver.match(version, match_expr)`

Compare two versions strings through a comparison.

**Parameters**

- **version** – a version string
- **match\_expr** – operator and version; valid operators are < smaller than > greater than >= greater or equal than <= smaller or equal than == equal != not equal

**Returns** True if the expression matches the version, otherwise False

```
>>> 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 strings.

**Parameters**

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

**Returns** the greater version of the two

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

`semver.min_ver(ver1, ver2)`

Returns the smaller version of two versions strings.

**Parameters**

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

**Returns** the smaller version of the two

```
>>> 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.

Deprecated since version 2.10.0: Use `semver.VersionInfo.parse()` instead.

**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

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

(continues on next page)

(continued from previous page)

```
5
>>> ver['prerelease']
'pre.2'
>>> ver['build']
'build.4'
```

`semver.parse_version_info(version)`

Parse version string to a `VersionInfo` instance.

Deprecated since version 2.10.0: Use `semver.VersionInfo.parse()` instead.

New in version 2.7.2: Added `semver.parse_version_info()`

**Parameters** `version` – version string

**Returns** a `VersionInfo` instance

```
>>> version_info = semver.VersionInfo.parse("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** (Namespace) – The parsed arguments
- **parser** – the parser instance

**Return type** `str`

**Returns** result of the selected action

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

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

Deprecated since version 2.10.0: Use `semver.VersionInfo.replace()` instead.

New in version 2.9.0: Added `replace()`

**Parameters**

- **version** – the version string to replace
- **parts** – 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

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

## 1.5 pysemver 3.0.0-dev.1

### 1.5.1 Synopsis

```
pysemver <COMMAND> <OPTION>...
```

### 1.5.2 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.

#### Global Options

**-h, --help**

Display usage summary.

**--version**

Show program's version number and exit.

### 1.5.3 Commands

#### pysemver bump

Bump a version.

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

##### <PART>

The part to bump. Valid strings are 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
```

## 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
```

## 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.

## 1.5.4 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.

## 1.5.5 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>

## 1.6 Change Log

Changes for the upcoming release can be found in the “`changelog.d`” directory in our repository.

### 1.6.1 Version 3.0.0-dev.1

**Released** 2020-10-26

**Maintainer** Tom Schraitle

#### Deprecations

- [PR #290](#): For semver 3.0.0-alpha0:
  - Remove anything related to Python2
  - In `tox.ini` and `.travis.yml` Remove targets `py27`, `py34`, `py35`, and `pypy`. Add `py38`, `py39`, and `nightly` (allow to fail)
  - In `setup.py` simplified file and remove `Tox` and `Clean` classes
  - Remove old Python versions (2.7, 3.4, 3.5, and `pypy`) from Travis
- [#234](#): In `setup.py` simplified file and remove `Tox` and `Clean` classes

#### Features

- [PR #290](#): Create semver 3.0.0-alpha0
  - Update `README.rst`, mention maintenance branch `maint/v2`.
  - Remove old code mainly used for Python2 compatibility, adjusted code to support Python3 features.
  - Split test suite into separate files under `tests/` directory
  - Adjust and update `setup.py`. Requires Python `>=3.6.*` Extract metadata directly from source (affects all the `__version__`, `__author__` etc. variables)

- [#270](#): Configure Towncrier ([PR #273](#)):
  - Add `changelog.d/.gitignore` to keep this directory
  - Create `changelog.d/README.rst` with some descriptions
  - Add `changelog.d/_template.rst` as Towncrier template
  - Add `[tool.towncrier]` section in `pyproject.toml`
  - Add “changelog” target into `tox.ini`. Use it like `tox -e changelog -- CMD` whereas `CMD` is a Towncrier command. The default `tox -e changelog` calls Towncrier to create a draft of the changelog file and output it to stdout.
  - Update documentation and add include a new section “Changelog” included from `changelog.d/README.rst`.
- [#276](#): Document how to create a subclass from `VersionInfo` class

## Bug Fixes

- [#291](#): Disallow negative numbers in `VersionInfo` arguments for `major`, `minor`, and `patch`.

## Improved Documentation

- [PR #290](#): Several improvements in the documentation:
  - New layout to distinguish from the semver2 development line.
  - Create new logo.
  - Remove any occurrences of Python2.
  - Describe changelog process with Towncrier.
  - Update the release process.

## Trivial/Internal Changes

- [PR #290](#): Add supported Python versions to **black**.

---

## 1.6.2 Version 2.13.0

**Released** 2020-10-20

**Maintainer** Tom Schraitle



## Features

- [PR #287](#): Document how to create subclass from `VersionInfo`

## Bug Fixes

- [PR #283](#): Ensure equal versions have equal hashes. Version equality means for semver, that major, minor, patch, and prerelease parts are equal in both versions you compare. The build part is ignored.

## Additions

n/a

## Deprecations

n/a

---

### 1.6.3 Version 2.12.0

**Released** 2020-10-19

**Maintainer** Tom Schraitle

#### Bug Fixes

- [#291](#) ([PR #292](#)): Disallow negative numbers of major, minor, and patch for `semver.VersionInfo`
- 

### 1.6.4 Version 2.11.0

**Released** 2020-10-17

**Maintainer** Tom Schraitle

#### Bug Fixes

- [#276](#) ([PR #277](#)): **`VersionInfo.parse` should be a class method** Also add authors and update changelog in [#286](#)
  - [#274](#) ([PR #275](#)): Py2 vs. Py3 incompatibility `TypeError`
-

## 1.6.5 Version 2.10.2

**Released** 2020-06-15

**Maintainer** Tom Schraitle

### Features

#268: Increase coverage

### Bug Fixes

- [#260 \(PR #261\)](#): Fixed `__getitem__` returning None on wrong parts
- [PR #263](#): Doc: Add missing “install” subcommand for openSUSE

### Deprecations

- **#160 (PR #264)**:
    - `semver.max_ver()`
    - `semver.min_ver()`
- 

## 1.6.6 Version 2.10.1

**Released** 2020-05-13

**Maintainer** Tom Schraitle

### Features

- [PR #249](#): Added release policy and version restriction in documentation to help our users which would like to stay on the major 2 release.
- [PR #250](#): Simplified installation semver on openSUSE with `obs://`.
- [PR #256](#): Made docstrings consistent

### Bug Fixes

- [#251 \(PR #254\)](#): Fixed return type of `semver.VersionInfo.next_version` to always return a `VersionInfo` instance.
-

## 1.6.7 Version 2.10.0

**Released** 2020-05-05

**Maintainer** Tom Schraitle

### Features

- [PR #138](#): Added `__getitem__` magic method to `semver.VersionInfo` class. Allows to access a version like `version[1]`.
- [PR #235](#): Improved documentation and shift focus on `semver.VersionInfo` instead of advertising the old and deprecated module-level functions.
- [PR #230](#): Add version information in some functions:
  - Use `.. versionadded::` RST directive in docstrings to make it more visible when something was added
  - Minor wording fix in docstrings (versions -> version strings)

### Bug Fixes

- [#224 \(PR #226\)](#): In `setup.py`, replaced `in class clean, super (CleanCommand, self).run()` with `CleanCommand.run(self)`
- [#244 \(PR #245\)](#): Allow comparison with `VersionInfo`, tuple/list, dict, and string.

### Additions

- [PR #228](#): Added better doctest integration

### Deprecations

- [#225 \(PR #229\)](#): Output a `DeprecationWarning` for the following functions:
  - `semver.parse`
  - `semver.parse_version_info`
  - `semver.format_version`
  - `semver.bump_{major,minor,patch,prerelease,build}`
  - `semver.finalize_version`
  - `semver.replace`
  - `semver.VersionInfo._asdict` (use the new, public available function `semver.VersionInfo.to_dict()`)
  - `semver.VersionInfo._astuple` (use the new, public available function `semver.VersionInfo.to_tuple()`)

These deprecated functions will be removed in semver 3.

## 1.6.8 Version 2.9.1

**Released** 2020-02-16

**Maintainer** Tom Schraitle

### 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

### Bug Fixes

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

## 1.6.9 Version 2.9.0

**Released** 2019-10-30

**Maintainer** Sébastien Celles <[s.celles@gmail.com](mailto:s.celles@gmail.com)>

### 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 `confest.py` to improve doctests
- PR #165: Improved code coverage
- PR #166: Reworked `.gitignore` file
- #167 (PR #168): Introduced global constant `SEMVER_SPEC_VERSION`

### 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

### Removals

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

## 1.6.10 Version 2.8.2

**Released** 2019-05-19

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

Skipped, not released.

---

## 1.6.11 Version 2.8.1

**Released** 2018-07-09

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

## 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.

## Bug Fixes

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

## 1.6.12 Version 2.8.0

**Released** 2018-05-16

**Maintainer** Sébastien Celles <[s.celles@gmail.com](mailto:s.celles@gmail.com)>

## Changes

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

## Additions

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

## Removals

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

## 1.7 Older Change Log

This changelog contains older entries from 2.7.9 and before.

### 1.7.1 Version 2.7.9

**Released** 2017-09-23

**Maintainer** Kostiantyn Rybnikov <[k-bx@k-bx.com](mailto:k-bx@k-bx.com)>

## Additions

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

### 1.7.2 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
- 

### 1.7.3 Version 2.7.7

**Released** 2017-05-25

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

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

### 1.7.4 Version 2.7.2

**Released** 2016-11-08

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

## Additions

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

## Bug Fixes

- #37: Removed trailing zeros from pre-release 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.
-

### 1.7.5 Version 2.6.0

**Released** 2016-06-08

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

#### Removals

- Remove comparison of build component.  
SemVer 2.0.0 specification recommends that build component is ignored in comparisons.
- 

### 1.7.6 Version 2.5.0

**Released** 2016-05-25

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

#### Additions

- Support matching ‘not equal’ with “!=”.

#### Changes

- Made separate builds for tests on Travis CI.
- 

### 1.7.7 Version 2.4.2

**Released** 2016-05-16

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

#### Changes

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



### 1.7.8 Version 2.4.1

**Released** 2016-03-04

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

#### Additions

- #23: Compared build component of a version.
- 

### 1.7.9 Version 2.4.0

**Released** 2016-02-12

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

#### Bug Fixes

- #21: Compared alphanumeric components correctly.
- 

### 1.7.10 Version 2.3.1

**Released** 2016-01-30

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

#### Additions

- Declared granted license name in distribution metadata.
- 

### 1.7.11 Version 2.3.0

**Released** 2016-01-29

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

#### Additions

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

### 1.7.12 Version 2.2.1

**Released** 2015-08-04

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

#### Bug Fixes

- Corrected comparison when any component includes zero.
- 

### 1.7.13 Version 2.2.0

**Released** 2015-06-21

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

#### Additions

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

### 1.7.14 Version 2.1.2

**Released** 2015-05-23

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

#### Bug Fixes

- Restored current README document to distribution manifest.
- 

### 1.7.15 Version 2.1.1

**Released** 2015-05-23

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

#### Bug Fixes

- Removed absent document from distribution manifest.
-

### 1.7.16 Version 2.1.0

**Released** 2015-05-22

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

#### 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.

#### Changes

- Migrated README document to Markdown format.

#### Bug Fixes

- Corrected code examples in README document.
- 

### 1.7.17 Version 2.0.2

**Released** 2015-04-14

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

#### Additions

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

### 1.7.18 Version 2.0.1

**Released** 2014-09-24

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

## Bug Fixes

- #9: Fixed comparison of equal version strings.
- 

### 1.7.19 Version 2.0.0

**Released** 2014-05-24

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

## Additions

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

## Changes

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

### 1.7.20 Version 0.0.2

**Released** 2012-05-10

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

## Changes

- Use standard library Distutils for distribution management.
- 

### 1.7.21 Version 0.0.1

**Released** 2012-04-28

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

- Initial release.

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### S

semver, [24](#)





## Symbols

--help  
     pysemver command line option, 33  
 --version  
     pysemver command line option, 33  
 -h  
     pysemver command line option, 33  
 <PART>  
     pysemver command line option, 33  
 <VERSION1>  
     pysemver command line option, 34  
 <VERSION2>  
     pysemver command line option, 34  
 <VERSION>  
     pysemver command line option, 33, 34

## B

build() (*semver.VersionInfo* property), 24  
 bump\_build() (*in module semver*), 28  
 bump\_build() (*semver.VersionInfo* method), 24  
 bump\_major() (*in module semver*), 28  
 bump\_major() (*semver.VersionInfo* method), 24  
 bump\_minor() (*in module semver*), 28  
 bump\_minor() (*semver.VersionInfo* method), 25  
 bump\_patch() (*in module semver*), 28  
 bump\_patch() (*semver.VersionInfo* method), 25  
 bump\_prerelease() (*in module semver*), 28  
 bump\_prerelease() (*semver.VersionInfo* method), 25

## C

cmd\_bump() (*in module semver*), 29  
 cmd\_check() (*in module semver*), 29  
 cmd\_compare() (*in module semver*), 29  
 compare() (*in module semver*), 29  
 compare() (*semver.VersionInfo* method), 25  
 createparser() (*in module semver*), 29

## D

deprecated() (*in module semver*), 30

## F

finalize\_version() (*in module semver*), 30  
 finalize\_version() (*semver.VersionInfo* method), 26  
 format\_version() (*in module semver*), 30

## I

isvalid() (*semver.VersionInfo* class method), 26

## M

main() (*in module semver*), 30  
 major() (*semver.VersionInfo* property), 26  
 match() (*in module semver*), 31  
 match() (*semver.VersionInfo* method), 26  
 max\_ver() (*in module semver*), 31  
 min\_ver() (*in module semver*), 31  
 minor() (*semver.VersionInfo* property), 26  
 module  
     semver, 24

## N

next\_version() (*semver.VersionInfo* method), 26

## P

parse() (*in module semver*), 31  
 parse() (*semver.VersionInfo* class method), 27  
 parse\_version\_info() (*in module semver*), 32  
 patch() (*semver.VersionInfo* property), 27  
 prerelease() (*semver.VersionInfo* property), 27  
 process() (*in module semver*), 32  
 pysemver command line option  
     --help, 33  
     --version, 33  
     -h, 33  
     <PART>, 33  
     <VERSION1>, 34  
     <VERSION2>, 34  
     <VERSION>, 33, 34

## R

replace() (*in module semver*), 32  
 replace() (*semver.VersionInfo* method), 27

## S

`semver`

    module, [24](#)

`SEMVER_SPEC_VERSION` (*in module `semver`*), [24](#)

## T

`to_dict()` (*`semver.VersionInfo` method*), [27](#)

`to_tuple()` (*`semver.VersionInfo` method*), [27](#)

## V

`VersionInfo` (*class in `semver`*), [24](#)