
python-semver Documentation

Release 3.0.0

Kostiantyn Rybnikov and all

Apr 02, 2023

CONTENTS

1 Quickstart 3

1.1 Building semver 4

1.2 Installing semver 5

1.3 Using semver 7

1.4 Migrating to semver3 17

1.5 Advanced topics 20

1.6 Contributing to semver 26

1.7 API Reference 32

1.8 pysemver 3.0.0 46

1.9 Change Log 48

1.10 Changelog semver3 development 52

2 Indices and Tables 59

Python Module Index 61

Index 63

If you are searching for how to stay compatible with semver3, refer to [*Migrating from semver2 to semver3*](#).

QUICKSTART

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

Note: This project works for Python 3.7 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 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.Version.parse` function:

```
>>> ver = semver.Version.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.Version.bump_major` leaves the original object untouched, but returns a new `semver.Version` instance with the raised major part:

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

It is allowed to concatenate different “bump functions”:

```
>>> ver.bump_major().bump_minor()
Version(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 Building semver

This project changed slightly its way how it is built. The reason for this was to still support the “traditional” way with **setup.py**, but at the same time try out the newer way with `pyproject.toml`. As Python 3.6 got deprecated, this project does support from now on only `pyproject.toml`.

1.1.1 Background information

Skip this section and head over to *Building with pyproject-build* if you just want to know how to build semver. This section gives some background information how this project is set up.

The traditional way with **setup.py** in this project uses a [Declarative config](#). With this approach, the **setup.py** is stripped down to its bare minimum and all the metadata is stored in `setup.cfg`.

The new `pyproject.toml` contains only information about the build backend, currently `setuptools.build_meta`. The idea is taken from [A Practical Guide to Setuptools and Pyproject.toml](#). Setuptools-specific configuration keys as defined in [PEP 621](#) are currently not used.

1.1.2 Building with pyproject-build

To build semver you need:

- The build module which implements the [PEP 517](#) build frontend. Install it with:

```
pip install build
```

Some Linux distributions has already packaged it. If you prefer to use the module with your package manager, search for `python-build` or `python3-build` and install it.

- The command **pyproject-build** from the build module.

To build semver, run:


```
pyproject-build
```

After the command is finished, you can find two files in the `dist` folder: a `.tar.gz` and a `.whl` file.

1.2 Installing semver

1.2.1 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.

Same applies for semver v3, if you want to get all updates for the semver v3 development line, but not a major update to semver v4:

```
semver>=3,<4
```

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

1.2.2 Pip

```
pip3 install semver
```

If you want to install this specific version (for example, 3.0.0), use the command **pip** with an URL and its version:

```
pip3 install git+https://github.com/python-semver/python-semver.git@3.0.0
```

1.2.3 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
```

1.3 Using semver

1.3.1 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'
```

1.3.2 Getting the Version of semver

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

```
>>> semver.__version__
'3.0.0'
```

1.3.3 Creating a Version

Changed in version 3.0.0: The former *VersionInfo* class has been renamed to *Version*.

The preferred way to create a new version is with the class *Version*.

Note: In the previous major release semver 2 it was possible to create a version with module level functions. However, module level functions are marked as *deprecated* since version 2.x.y now. These functions will be removed. For details, see the sections *Replacing Deprecated Functions* and *Displaying Deprecation Warnings*.

A *Version* instance can be created in different ways:

- From a Unicode string:

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

- From a byte string:

```
>>> Version.parse(b"2.3.4")
Version(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'}
>>> Version(**d)
Version(major=3, minor=4, patch=5, prerelease='pre.2', build='build.4')
```

Keep in mind, the major, minor, patch parts has to be positive integers or strings:

```
>>> d = {'major': -3, 'minor': 4, 'patch': 5, 'prerelease': 'pre.2', 'build':
↳ 'build.4'}
>>> Version(**d)
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)
>>> Version(*t)
Version(major=3, minor=5, patch=6, prerelease=None, build=None)
```

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

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

The old, deprecated module level functions are still available but using them are discouraged. They are available to convert old code to `semver3`.

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 `Version` 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
```

1.3.4 Parsing a Version String

“Parsing” in this context means to identify the different parts in a string. Use the function `parse()`:

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

Set the parameter `optional_minor_and_patch=True` to allow optional minor and patch parts. Optional parts are set to zero. By default (False), the version string to parse has to follow the semver specification:

```
>>> Version.parse("1.2", optional_minor_and_patch=True)
Version(major=1, minor=2, patch=0, prerelease=None, build=None)
```

1.3.5 Checking for a Valid Semver Version

If you need to check a string if it is a valid semver version, use the classmethod `is_valid()`:

```
>>> Version.is_valid("1.0.0")
True
>>> Version.is_valid("invalid")
False
```

1.3.6 Checking for a Compatible Semver Version

To check if a *change* from a semver version *a* to a semver version *b* is *compatible* according to semver rule, use the method `is_compatible()`.

The expression `a.is_compatible(b)` is `True` if one of the following statements is true:

- both versions are equal, or
- both majors are equal and higher than 0. The same applies for both minor parts. Both pre-releases are equal, or
- both majors are equal and higher than 0. The minor of *b*’s minor version is higher then *a*’s. Both pre-releases are equal.

In all other cases, the result is false.

Keep in mind, the method *does not* check patches!

- Two different majors:

```
>>> a = Version(1, 1, 1)
>>> b = Version(2, 0, 0)
>>> a.is_compatible(b)
False
>>> b.is_compatible(a)
False
```

- Two different minors:

```
>>> a = Version(1, 1, 0)
>>> b = Version(1, 0, 0)
>>> a.is_compatible(b)
False
```

(continues on next page)

(continued from previous page)

```
>>> b.is_compatible(a)
True
```

- The same two majors and minors:

```
>>> a = Version(1, 1, 1)
>>> b = Version(1, 1, 0)
>>> a.is_compatible(b)
True
>>> b.is_compatible(a)
True
```

- Release and pre-release:

```
>>> a = Version(1, 1, 1)
>>> b = Version(1, 0, 0, 'rc1')
>>> a.is_compatible(b)
False
>>> b.is_compatible(a)
False
```

- Different pre-releases:

```
>>> a = Version(1, 0, 0, 'rc1')
>>> b = Version(1, 0, 0, 'rc2')
>>> a.is_compatible(b)
False
>>> b.is_compatible(a)
False
```

- Identical pre-releases:

```
>>> a = Version(1, 0, 0, 'rc1')
>>> b = Version(1, 0, 0, 'rc1')
>>> a.is_compatible(b)
True
```

- All major zero versions are incompatible with anything but itself:

```
>>> Version(0, 1, 0).is_compatible(Version(0, 1, 1))
False

# Only identical versions are compatible for major zero versions:
>>> Version(0, 1, 0).is_compatible(Version(0, 1, 0))
True
```

1.3.7 Accessing Parts of a Version Through Names

The `Version` class contains attributes to access the different parts of a version:

```
>>> v = Version.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 any of the above attributes. 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 `Version` instance:

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

1.3.8 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 `Version` 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 = Version.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:

```
>>> s1 = slice(0,3)
>>> ver[s1]
(10, 3, 2)
```

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

```
>>> ver = Version.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
```

1.3.9 Replacing Parts of a Version

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

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

If you pass invalid keys you get an exception:

```
>>> version = semver.Version.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
```

1.3.10 Converting a Version instance into Different Types

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

It is possible to convert a `Version` instance:

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

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

- Into a dictionary with `to_dict()`:

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

- Into a tuple with `to_tuple()`:


```
>>> v = Version(major=5, minor=4, patch=2)
>>> v.to_tuple()
(5, 4, 2, None, None)
```

1.3.11 Raising Parts of a Version

Note: Keep in mind, “raising” the pre-release only will make your complete version *lower* than before.

For example, having version 1.0.0 and raising the pre-release will lead to 1.0.0-rc.1, but 1.0.0-rc.1 is smaller than 1.0.0.

If you search for a way to take into account this behavior, look for the method `next_version()` in section *Increasing Parts of a Version Taking into Account Prereleases*.

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

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

```
>>> str(Version.parse("3.4.5-pre.2+build.4").bump_major())
'4.0.0'
>>> str(Version.parse("3.4.5-pre.2+build.4").bump_minor())
'3.5.0'
>>> str(Version.parse("3.4.5-pre.2+build.4").bump_patch())
'3.4.6'
>>> str(Version.parse("3.4.5-pre.2+build.4").bump_prerelease())
'3.4.5-pre.3'
>>> str(Version.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()`.

For the methods `bump_prerelease()` and `bump_build()` it’s possible to pass an empty string or None. However, it gives different results:

```
>>> str(Version.parse("3.4.5").bump_prerelease(''))
'3.4.5-1'
>>> str(Version.parse("3.4.5").bump_prerelease(None))
'3.4.5-rc.1'
```

An empty string removes any prefix whereas None is the same as calling the method without any argument.

If you already have a prerelease, the argument for the method is not taken into account:

```
>>> str(Version.parse("3.4.5-rc.1").bump_prerelease(None))
'3.4.5-rc.2'
>>> str(Version.parse("3.4.5-rc.1").bump_prerelease(''))
'3.4.5-rc.2'
```

1.3.12 Increasing Parts of a Version Taking into Account Prereleases

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

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

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

1.3.13 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 `Version` instances**

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

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

- **A `Version` type and a `tuple()` or `list()`**

Use the operator as with two `Version` types:

```
>>> v = Version.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 *Version* 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 *Version* 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 Version 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()`.

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

This also applies when a *Version* is a member of a set, or a dictionary key:

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

1.3.15 Comparing Versions through an Expression

If you need a more fine-grained approach of comparing two versions, use the *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:

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

If no operator is specified, the match expression is interpreted as a version to be compared for equality. This allows handling the common case of version compatibility checking through either an exact version or a match expression very easy to implement, as the same code will handle both cases:

```
>>> Version.parse("2.0.0").match("2.0.0")
True
>>> Version.parse("1.0.0").match("3.5.1")
False
```

1.3.16 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 *Version* implements `__gt__()` and `__lt__()`, it can be used with builtins requiring:

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

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

```
>>> max(['1.1.0', '1.2.0', '2.1.0', '0.5.10', '0.4.99'], key=Version.parse)
'2.1.0'
>>> min(['1.1.0', '1.2.0', '2.1.0', '0.5.10', '0.4.99'], key=Version.parse)
'0.4.99'
```

And the same can be done with tuples:

```
>>> max(map(lambda v: Version(*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: Version(*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 Version instance into Different Types*.

1.4 Migrating to semver3

1.4.1 Migrating from semver2 to semver3

This section describes the visible differences for users and how your code stays compatible for semver3. Some changes are backward incompatible.

Although the development team tries to make the transition to semver3 as smooth as possible, at some point change is inevitable.

For a more detailed overview of all the changes, refer to our *Change Log*.

Use Version instead of VersionInfo

The `VersionInfo` has been renamed to `Version` to have a more succinct name. An alias has been created to preserve compatibility but using the old name has been deprecated and will be removed in future versions.

If you still need the old version, use this line:

```
from semver.version import Version as VersionInfo
```

Use semver.cli instead of semver

All functions related to CLI parsing are moved to `semver.cli`. If you need such functions, like `cmd_bump()`, import it from `semver.cli` in the future:

```
from semver.cli import cmd_bump
```

Use semver.Version.is_valid instead of semver.Version.isvalid

The pull request [PR #284](#) introduced the method `is_compatible()`. To keep consistency, the development team decided to rename the `isvalid()` to `is_valid()`.

1.4.2 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 `Version` 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 `Version` class. For example, the function `semver.bump_major()` is replaced by `bump_major()` and calling the `str(versionobject)`:

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

Likewise with the other module level functions.

- `semver.Version.isvalid()`

Replace it with `semver.version.Version.is_valid()`:

- `semver.finalize_version()`

Replace it with `semver.version.Version.finalize_version()`:

```
>>> s1 = semver.finalize_version('1.2.3-rc.5')
>>> s2 = str(semver.Version.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.Version(5, 4, 3, 'pre.2', 'build.1'))
>>> s1 == s2
True
```

- `semver.max_ver()`

Replace it with `max(version1, version2, ...)` or `max([version1, version2, ...], key=Version.parse)` and a key:

```
>>> s1 = semver.max_ver("1.2.3", "1.2.4")
>>> s2 = max("1.2.3", "1.2.4", key=Version.parse)
>>> s1 == s2
True
```

- `semver.min_ver()`

Replace it with `min(version1, version2, ...)` or `min([version1, version2, ...], key=Version.parse)`:

```
>>> s1 = semver.min_ver("1.2.3", "1.2.4")
>>> s2 = min("1.2.3", "1.2.4", key=Version.parse)
>>> s1 == s2
True
```

- `semver.parse()`

Replace it with `semver.version.Version.parse()` and call `semver.version.Version.to_dict()`:

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

- `semver.parse_version_info()`

Replace it with `semver.version.Version.parse()`:

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

- `semver.replace()`

Replace it with `semver.version.Version.replace()`:

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

(continues on next page)

(continued from previous page)

```
>>> s1 == s2
True
```

1.5 Advanced topics

1.5.1 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
from semver import Version
from typing import Optional, Tuple

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: str) -> Tuple[Version, Optional[str]]:
    """
    Convert an incomplete version string into a semver-compatible Version
    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:`Version` 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:`Version` | None, str)
    """
    match = BASEVERSION.search(version)
    if not match:
```

(continues on next page)

(continued from previous page)

```

    return (None, version)

    ver = {
        key: 0 if value is None else value for key, value in match.groupdict().items()
    }
    ver = Version(**ver)
    rest = match.string[match.end() :] # noqa:E203
    return ver, rest

```

The function returns a *tuple*, containing a *Version* 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")
(Version(major=1, minor=2, patch=0, prerelease=None, build=None), '')
>>> coerce("v2.5.2-bla")
(Version(major=2, minor=5, patch=2, prerelease=None, build=None), '-bla')

```

1.5.2 Creating Subclasses from Version

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 *Version* 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(Version):
    """
    A subclass of Version which allows a "v" prefix
    """

    @classmethod
    def parse(cls, version: str) -> "SemVerWithVPrefix":
        """
        Parse version string to a Version instance.

        :param version: version string with "v" or "V" prefix
        :raises ValueError: when version does not start with "v" or "V"
        :return: a new instance
        """
        if not version[0] in ("v", "V"):
            raise ValueError(
                f"{version!r}: not a valid semantic version tag. "
                "Must start with 'v' or 'V'"
            )
        return super().parse(version[1:], optional_minor_and_patch=True)

    def __str__(self) -> str:
        # Reconstruct the tag
        return "v" + super().__str__()

```

The derived class `SemVerWithVPrefix` can be used like the original class. Additionally, you can pass “incomplete” version strings like `v2.3`:

```
>>> v1 = SemVerWithVPrefix.parse("v1.2.3")
>>> assert str(v1) == "v1.2.3"
>>> print(v1)
v1.2.3
>>> v2 = SemVerWithVPrefix.parse("v2.3")
>>> 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'
```

1.5.3 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.

1.5.4 Combining Pydantic and semver

According to its homepage, [Pydantic](#) “enforces type hints at runtime, and provides user friendly errors when data is invalid.”

To work with Pydantic, use the following steps:

1. Derive a new class from `Version` first and add the magic methods `__get_validators__()` and `__modify_schema__()` like this:

```
from semver import Version

class PydanticVersion(Version):
    @classmethod
    def _parse(cls, version):
```

(continues on next page)

(continued from previous page)

```

        return cls.parse(version)

    @classmethod
    def __get_validators__(cls):
        """Return a list of validator methods for pydantic models."""
        yield cls._parse

    @classmethod
    def __modify_schema__(cls, field_schema):
        """Inject/mutate the pydantic field schema in-place."""
        field_schema.update(examples=[
            "1.0.2",
            "2.15.3-alpha",
            "21.3.15-beta+12345",
        ])
    )

```

2. Create a new model (in this example `MyModel`) and derive it from `pydantic.BaseModel`:

```

import pydantic

class MyModel(pydantic.BaseModel):
    version: PydanticVersion

```

3. Use your model like this:

```
model = MyModel.parse_obj({"version": "1.2.3"})
```

The attribute `model.version` will be an instance of `Version`. If the version is invalid, the construction will raise a `pydantic.ValidationError`.

1.5.5 Converting versions between PyPI and semver

When packaging for PyPI, your versions are defined through [PEP 440](#). This is the standard version scheme for Python packages and implemented by the `packaging.version.Version` class.

However, these versions are different from semver versions (cited from [PEP 440](#)):

- The “Major.Minor.Patch” (described in this PEP as “major.minor.micro”) aspects of semantic versioning (clauses 1-8 in the 2.0.0 specification) are fully compatible with the version scheme defined in this PEP, and abiding by these aspects is encouraged.
- Semantic versions containing a hyphen (pre-releases - clause 10) or a plus sign (builds - clause 11) are *not* compatible with this PEP and are not permitted in the public version field.

In other words, it’s not always possible to convert between these different versioning schemes without information loss. It depends on what parts are used. The following table gives a mapping between these two versioning schemes:

PyPI Version	Semver version
epoch	n/a
major	major
minor	minor
micro	patch
pre	prerelease
dev	build
post	n/a

From PyPI to semver

We distinguish between the following use cases:

- **“Incomplete” versions**

If you only have a major part, this shouldn’t be a problem. The initializer of `semver.Version` takes care to fill missing parts with zeros (except for major).

```
>>> from packaging.version import Version as PyPIVersion
>>> from semver import Version

>>> p = PyPIVersion("3.2")
>>> p.release
(3, 2)
>>> Version(*p.release)
Version(major=3, minor=2, patch=0, prerelease=None, build=None)
```

- **Major, minor, and patch**

This is the simplest and most compatible approach. Both versioning schemes are compatible without information loss.

```
>>> p = PyPIVersion("3.0.0")
>>> p.base_version
'3.0.0'
>>> p.release
(3, 0, 0)
>>> Version(*p.release)
Version(major=3, minor=0, patch=0, prerelease=None, build=None)
```

- **With pre part only**

A prerelease exists in both versioning schemes. As such, both are a natural candidate. A prerelease in PyPI version terms is the same as a “release candidate”, or “rc”.

```
>>> p = PyPIVersion("2.1.6.pre5")
>>> p.base_version
'2.1.6'
>>> p.pre
('rc', 5)
>>> pre = "".join([str(i) for i in p.pre])
>>> Version(*p.release, pre)
Version(major=2, minor=1, patch=6, prerelease='rc5', build=None)
```

- **With only development version**

Semver doesn't have a "development" version. However, we could use Semver's build part:

```
>>> p = PyPIVersion("3.0.0.dev2")
>>> p.base_version
'3.0.0'
>>> p.dev
2
>>> Version(*p.release, build=f"dev{p.dev}")
Version(major=3, minor=0, patch=0, prerelease=None, build='dev2')
```

- **With a post version**

Semver doesn't know the concept of a post version. As such, there is currently no way to convert it reliably.

- **Any combination**

There is currently no way to convert a PyPI version which consists of, for example, development *and* post parts.

You can use the following function to convert a PyPI version into semver:

```
def convert2semver(ver: packaging.version.Version) -> semver.Version:
    """Converts a PyPI version into a semver version

    :param ver: the PyPI version
    :return: a semver version
    :raises ValueError: if epoch or post parts are used
    """
    if not ver.epoch:
        raise ValueError("Can't convert an epoch to semver")
    if not ver.post:
        raise ValueError("Can't convert a post part to semver")

    pre = None if not ver.pre else "".join([str(i) for i in ver.pre])
    return semver.Version(*ver.release, prerelease=pre, build=ver.dev)
```

From semver to PyPI

We distinguish between the following use cases:

- **Major, minor, and patch**

```
>>> from packaging.version import Version as PyPIVersion
>>> from semver import Version

>>> v = Version(1, 2, 3)
>>> PyPIVersion(str(v.finalize_version()))
<Version('1.2.3')>
```

- **With pre part only**

```
>>> v = Version(2, 1, 4, prerelease="rc1")
>>> PyPIVersion(str(v))
<Version('2.1.4rc1')>
```

- **With only development version**

```
>>> v = Version(3, 2, 8, build="dev4")
>>> PyPIVersion(f"{v.finalize_version()}{v.build}")
<Version('3.2.8.dev4')>
```

If you are unsure about the parts of the version, the following function helps to convert the different parts:

```
def convert2pypi(ver: semver.Version) -> packaging.version.Version:
    """Converts a semver version into a version from PyPI

    A semver prerelease will be converted into a
    prerelease of PyPI.
    A semver build will be converted into a development
    part of PyPI
    :param semver.Version ver: the semver version
    :return: a PyPI version
    """
    v = ver.finalize_version()
    prerelease = ver.prerelease if ver.prerelease else ""
    build = ver.build if ver.build else ""
    return PyPIVersion(f"{v}{prerelease}{build}")
```

1.5.6 Reading versions from file

In cases where a version is stored inside a file, one possible solution is to use the following function:

```
import os
from typing import Union
from semver.version import Version

def get_version(path: Union[str, os.PathLike]) -> semver.Version:
    """
    Construct a Version object from a file

    :param path: A text file only containing the semantic version
    :return: A :class:`Version` object containing the semantic
             version from the file.
    """
    version = open(path, "r").read().strip()
    return Version.parse(version)
```

1.6 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
```

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](https://semver.org/) specification.
- Check if your feature is covered by the Semantic Versioning specification. If not, ask on its GitHub project <https://github.com/semver/semver>.

1.6.1 Reporting Bugs and Asking Questions

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!

There are several options to participate:

- Open a new topic on our [GitHub discussion](#) page. Tell us our ideas or ask your questions.
- Look into our GitHub [issues](#) tracker or open a new issue.

1.6.2 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 one or more specific Python versions. Use the `-e` option and one or more abbreviations (`py37` for Python 3.7, `py38` for Python 3.8 etc.):

```
$ tox -e py37
$ tox -e py37,py38
```

To get a complete list and a short description, run:

```
$ tox -av
```

- 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_bump.py` for all Python versions:

```
$ tox -e py37 -- tests/test_bump.py::test_should_bump_major
```

By default, pytest prints only a dot for each test function. 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 3.7 and 3.8 only:

```
$ tox -e py37,py38 -- tests/test_bump.py::test_should_bump_major
```

Our code is checked against formatting, style, type, and docstring issues ([black](#), [flake8](#), [mypy](#), and [docformatter](#)). It is recommended to run your tests in combination with **checks**, for example:

```
$ tox -e checks,py37,py38
```

1.6.3 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 docs/_build/html.

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 Reference](#) section. An appropriate docstring should look like this:

```
def to_tuple(self) -> VersionTuple:
    """
    Convert the Version object to a tuple.

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

    :return: a tuple with all the parts

    >>> semver.Version(5, 3, 1).to_tuple()
    (5, 3, 1, None, None)

    """
```

- **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, as shown above.

- **The documentation**

A docstring is good, but in most cases it's too dense. API documentation cannot replace a good user documentation. 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.

1.6.4 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.

1.6.5 Release Procedure

The following procedures gives a short overview of what steps are needed to create a new release.

These steps are interesting for the release manager only.

Prepare the Release

1. Verify that:
 - all issues for a new release are closed: <https://github.com/python-semver/python-semver/issues>.
 - all pull requests that should be included in this release are merged: <https://github.com/python-semver/python-semver/pulls>.
 - continuous integration for latest build was passing: <https://github.com/python-semver/python-semver/actions>.
2. Create a new branch `release/<VERSION>`.
3. If one or several supported Python versions have been removed or added, verify that the following files have been updated:
 - `setup.cfg`
 - `tox.ini`
 - `.git/workflows/pythonpackage.yml`
 - `.github/workflows/python-testing.yml`
4. Verify that the version in file `src/semver/__about__.py` has been updated and follows the [Semver](#) specification.
5. Add eventually new contributor(s) to [CONTRIBUTORS](#).
6. Check if all changelog entries are created. If some are missing, [create them](#).
7. Show the new draft [CHANGELOG](#) entry for the latest release with:

```
$ tox -e changelog
```

Check the output. If you are not happy, update the files in the `changelog.d/` directory. If everything is okay, build the new `CHANGELOG` with:

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

8. Build the documentation and check the output:

```
$ tox -e docs
```

9. Commit all changes, push, and create a pull request.

Create the New Release

1. Ensure that long description ([README.rst](#)) can be correctly rendered by Pypi using `restview --long-description`
2. Clean up your local Git repository. Be careful, as it **will remove all files** which are not versioned by Git:

```
$ git clean -xfd
```

Before you create your distribution files, clean the directory too:

```
$ rm dist/*
```

3. Create the distribution files (wheel and source):

```
$ tox -e prepare-dist
```

4. Upload the wheel and source to TestPyPI first:

```
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

If you have a `~/.pypirc` with a `testpypi` section, the upload can be simplified:

```
$ twine upload --repository testpypi dist/*
```

5. Check if everything is okay with the wheel. Check also the web site <https://test.pypi.org/project/<VERSION>/>
6. If everything looks fine, merge the pull request.
7. Upload to PyPI:

```
$ git clean -xfd
$ tox -e prepare-dist
$ twine upload dist/*
```

8. Go to <https://pypi.org/project/semver/> to verify that new version is online and the page is rendered correctly.

1.6.6 Finish the Release

1. Create a tag:


```
$ git tag -a x.x.x
```

It's recommended to use the generated Tox output from the Changelog.
2. Push the tag:


```
$ git push --tags
```
3. In [GitHub Release page](#) document the new release. Select the tag from the last step and copy the content of the tag description into the release description.
4. Announce it in <https://github.com/python-semver/python-semver/discussions/categories/announcements>.

You're done! Celebrate!

1.7 API Reference

1.7.1 Metadata `semver.__about__`

Metadata about semver.

Contains information about semver's version, the implemented version of the semver specification, author, maintainers, and description.

```
semver.__about__.__author__ = 'Kostiantyn Rybnikov'
```

Original semver author

```
semver.__about__.__description__ = 'Python helper for Semantic Versioning  
(https://semver.org)'
```

Short description about semver

```
semver.__about__.__maintainer__ = ['Sebastien Celles', 'Tom Schraitle']
```

Current maintainer

```
semver.__about__.__version__ = '3.0.0'
```

Semver version

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

Supported semver specification

1.7.2 Deprecated Functions in `semver._deprecated`

Contains all deprecated functions.

```
semver._deprecated.compare(ver1, ver2)
```

Compare two versions strings.

Deprecated since version 3.0.0: The situation of this function is unclear and it might disappear in the future. If possible, use `semver.version.Version.compare()`. See #258 for details.

Parameters

- **ver1** (str) – first version string
- **ver2** (str) – second version string

Return type

int

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._deprecated.bump_build(version, token='build')`

Raise the build part of the version string.

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

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._deprecated.bump_major(version)`

Raise the major part of the version string.

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

Param

version string

Returns

the raised version string

Return type

str

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

`semver._deprecated.bump_minor(version)`

Raise the minor part of the version string.

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

Param

version string

Returns

the raised version string

Return type

str

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

`semver._deprecated.bump_patch(version)`

Raise the patch part of the version string.

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

Param

version string

Returns

the raised version string

Return type

str

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

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

Raise the prerelease part of the version string.

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

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

Decorates a function to output a deprecation warning.

Parameters

- **func** (Optional[TypeVar(F, bound= Callable)]) – the function to decorate
- **replace** (Optional[str]) – the function to replace (use the full qualified name like `semver.version.Version.bump_major`).
- **version** (Optional[str]) – the first version when this function was deprecated.
- **category** (Type[Warning]) – 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[... , TypeVar(F, bound= Callable)], partial]

Returns

decorated function which is marked as deprecated

`semver._deprecated.finalize_version(version)`

Remove any prerelease and build metadata from the version string.

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

New in version 2.7.9: Added `finalize_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._deprecated.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(Version(VERSION))` instead.

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._deprecated.match(version, match_expr)`

Compare two versions strings through a comparison.

Deprecated since version 2.10.0: Use `match()` instead.

Parameters

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

Return type

bool

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

`semver._deprecated.max_ver(ver1, ver2)`

Returns the greater version of two versions strings.

Deprecated since version 2.10.2: Use `max()` instead.

Parameters

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

Returns

the greater version of the two

Return type

Version

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

`semver._deprecated.min_ver(ver1, ver2)`

Returns the smaller version of two versions strings.

Deprecated since version 2.10.2: Use `min()` instead.

Parameters

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

Returns

the smaller version of the two

Return type

Version

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

`semver._deprecated.parse(version)`

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

Deprecated since version 2.10.0: Use `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

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


`semver._deprecated.parse_version_info(version)`

Parse version string to a Version instance.

Deprecated since version 2.10.0: Use [`parse\(\)`](#) instead.

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

Parameters

version – version string

Returns

a VersionInfo instance

```
>>> version_info = semver.Version.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._deprecated.replace(version, **parts)`

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

Deprecated since version 2.10.0: Use [`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'
```

1.7.3 CLI Parsing `semver.cli`

CLI parsing for **pysemver** command.

Each command in **pysemver** is mapped to a `cmd_` function. The `main` function calls `createparser` and `process` to parse and process all the commandline options.

The result of each command is printed on stdout.

`semver.cli.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.cli.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.cli.cmd_compare(args)`

Subcommand: Compare two versions

Synopsis: compare <VERSION1> <VERSION2>

Parameters

args (Namespace) – The parsed arguments

Return type

str

`semver.cli.createparser()`

Create an `argparse.ArgumentParser` instance.

Return type

`ArgumentParser`

Returns

parser instance

`semver.cli.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.cli.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

1.7.4 Entry point `semver.__main__`

Module to support call with `__main__.py`. Used to support the following call:

```
$ python3 -m semver ...
```

This makes it also possible to “run” a wheel like in this command:

```
$ python3 semver-3*-py3-none-any.whl/semver -h
```

1.7.5 Version Handling `semver.version`

Version handling by a semver compatible version class.

`semver.version.VersionInfo`

Keep the VersionInfo name for compatibility

class `semver.version.Version(major, minor=0, patch=0, prerelease=None, build=None)`

A semver compatible version class.

See specification at <https://semver.org>.**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

NAMES = ('major', 'minor', 'patch', 'prerelease', 'build')

The names of the different parts of a version

__eq__(*other*)

Return self==value.

Return type

bool

__ge__(*other*)

Return self>=value.

Return type

bool

__getitem__(*index*)

self.__getitem__(*index*) <==> self[*index*] Implement getitem.

If the part requested is undefined, or a part of the range requested is undefined, it will throw an index error.
Negative indices are not supported.

Parameters

index (Union[int, slice]) – a positive integer indicating the offset or a slice() object

Raises

IndexError – if index is beyond the range or a part is None

Return type

Union[int, str, None, Tuple[Union[int, str], ...]]

Returns

the requested part of the version at position index

```
>>> ver = semver.Version.parse("3.4.5")
>>> ver[0], ver[1], ver[2]
(3, 4, 5)
```

__gt__(*other*)

Return self>value.

Return type

bool

__hash__()

Return hash(self).

Return type

int

__iter__()

Return iter(self).

Return type

Iterable[Union[int, str, None]]

__le__(*other*)

Return self<=value.

Return type

bool

__lt__(*other*)

Return self<value.

Return type

bool

`__ne__(other)`

Return self!=value.

Return type

bool

`__repr__()`

Return repr(self).

Return type

str

`__str__()`

Return str(self).

Return type

str

property build: Optional[str]

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

bump_build(token='build')

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

Parameters**token** (Optional[str]) – defaults to 'build'**Return type***Version***Returns**new *Version* object with the raised build part. The original object is not modified.

```
>>> ver = semver.parse("3.4.5-rc.1+build.9")
>>> ver.bump_build()
Version(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*Version***Returns**

new object with the raised major part

```
>>> ver = semver.parse("3.4.5")
>>> ver.bump_major()
Version(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*Version*

Returns

new object with the raised minor part

```
>>> ver = semver.parse("3.4.5")
>>> ver.bump_minor()
Version(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

Version

Returns

new object with the raised patch part

```
>>> ver = semver.parse("3.4.5")
>>> ver.bump_patch()
Version(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 (Optional[str]) – defaults to 'rc'

Return type

Version

Returns

new *Version* object with the raised prerelease part. The original object is not modified.

```
>>> ver = semver.parse("3.4.5")
>>> ver.bump_prerelease().prerelease
'rc.2'
>>> ver.bump_prerelease('').prerelease
'1'
>>> ver.bump_prerelease(None).prerelease
'rc.1'
```

compare(other)

Compare self with other.

Parameters

other (Union[*Version*, 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.compare("2.0.0")
-1
>>> semver.compare("1.0.0")
```

(continues on next page)

(continued from previous page)

```

1
>>> semver.compare("2.0.0")
0
>>> semver.compare(dict(major=2, minor=0, patch=0))
0

```

finalize_version()

Remove any prerelease and build metadata from the version.

Return type

Version

Returns

a new instance with the finalized version string

```

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

```

is_compatible(*other*)

Check if current version is compatible with other version.

The result is True, if either of the following is true:

- both versions are equal, or
- both majors are equal and higher than 0. Same for both minors. Both pre-releases are equal, or
- both majors are equal and higher than 0. The minor of b's minor version is higher then a's. Both pre-releases are equal.

The algorithm does *not* check patches.

New in version 3.0.0.

Parameters

other (*Version*) – the version to check for compatibility

Return type

bool

Returns

True, if *other* is compatible with the old version, otherwise False

```

>>> Version(1, 1, 0).is_compatible(Version(1, 0, 0))
False
>>> Version(1, 0, 0).is_compatible(Version(1, 1, 0))
True

```

classmethod is_valid(*version*)

Check if the string is a valid semver version.

New in version 2.9.1.

Changed in version 3.0.0: Renamed from isvalid()

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: int

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

match(match_expr)

Compare self to match a match expression.

Parameters

match_expr (str) – optional 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.Version.parse("2.0.0").match(">=1.0.0")
True
>>> semver.Version.parse("1.0.0").match(">1.0.0")
False
>>> semver.Version.parse("4.0.4").match("4.0.4")
True
```

property minor: int

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

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.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

Version

Returns

new object with the appropriate part raised

classmethod parse(version, optional_minor_and_patch=False)

Parse version string to a Version instance.

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

Changed in version 3.0.0: Added optional parameter optional_minor_and_patch to allow optional minor and patch parts.

Parameters

- **version** (Union[str, bytes]) – version string
- **optional_minor_and_patch** (bool) – if set to true, the version string to parse can contain optional minor and patch parts. Optional parts are set to zero. By default (False), the version string to parse has to follow the semver specification.

Return type

TypeVar(T, bound= Version)

Returns

a new [Version](#) instance

Raises

- **ValueError** – if version is invalid
- **TypeError** – if version contains the wrong type

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

property patch: int

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

property prerelease: Optional[str]

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

replace(parts)**

Replace one or more parts of a version and return a new [Version](#) object, but leave self untouched

New in version 2.9.0: Added [Version.replace\(\)](#)

Parameters

parts (Union[int, str, None]) – the parts to be updated. Valid keys are: major, minor, patch, prerelease, or build

Return type

[Version](#)

Returns

the new [Version](#) object with the changed parts

Raises

TypeError – if parts contain invalid keys

to_dict()

Convert the Version object to an OrderedDict.

New in version 2.10.0: Renamed `Version._asdict()` to [Version.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.Version(3, 2, 1).to_dict()
OrderedDict([('major', 3), ('minor', 2), ('patch', 1), ('prerelease', None), (
↪ 'build', None)])
```

to_tuple()

Convert the Version object to a tuple.

New in version 2.10.0: Renamed `Version._astuple()` to *`Version.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.Version(5, 3, 1).to_tuple()
(5, 3, 1, None, None)
```

1.8 pysemver 3.0.0

1.8.1 Synopsis

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

1.8.2 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.

1.8.3 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.

1.8.4 Return Code

The *return code* of the script (accessible by `$?` from the Bash) indicates if the subcommand returned successfully or 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.

1.8.5 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>

1.9 Change Log

Changes for the upcoming release can be found in the “[changelog.d](#)” directory in our repository.

This section covers the changes between major version 2 and version 3.

1.9.1 Version 3.0.0

Released

2023-04-02

Maintainer

Tom Schraitle

Bug Fixes

- [#291](#): Disallow negative numbers in `VersionInfo` arguments for `major`, `minor`, and `patch`.
- [#310](#): Rework API documentation. Follow a more “semi-manual” attempt and add auto directives into `docs/api.rst`.
- [#344](#): Allow empty string, a string with a prefix, or `None` as token in `bump_build()` and `bump_prerelease()`.
- [#374](#): Correct Towncrier’s config entries in the `pyproject.toml` file. The old entries `[[tool.towncrier.type]]` are deprecated and need to be replaced by `[tool.towncrier.fragment.<TYPE>]`.
- [PR #384](#): General cleanup, reformat files:
 - Reformat source code with black again as some config options did accidentally exclude the semver source code. Mostly remove some includes/excludes in the black config.
 - Integrate concurrency in GH Action
 - Ignore Python files on project dirs in `.gitignore`
 - Remove unused patterns in `MANIFEST.in`
 - Use `extend-exclude` for flake in `setup.cfg` and adapt list.
 - Use `skip_install=True` in `tox.ini` for black
- [PR #393](#): Fix command `python -m semver` to avoid the error “invalid choice”
- [PR #396](#): Calling `parse()` on a derived class will show correct type of derived class.

Deprecations

- [#169](#): Deprecate CLI functions not imported from `semver.cli`.
- [#234](#): In `setup.py` simplified file and remove `Tox` and `Clean` classes
- [#284](#): Deprecate the use of `isvalid()`.
Rename `isvalid()` to `is_valid()` for consistency reasons with `is_compatible()`.
- [PR #290](#): For semver 3.0.0-alpha0 deprecated:
 - 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
- [#372](#): Deprecate support for Python 3.6.

Python 3.6 reached its end of life and isn’t supported anymore. At the time of writing (Dec 2022), the lowest version is 3.7.

Although the [poll](#) didn’t cast many votes, the majority agreed to remove support for Python 3.6.
- [PR #402](#): Keep `semver.compare`.
Although it breaks consistency with module level functions, it seems it’s a much needed/used function. It’s still unclear if we should deprecate this function or not (that’s why we use `PendingDeprecationWarning`).

As we don’t have a uniform initializer yet, this function stays in the `_deprecated.py` file for the time being until we find a better solution. See [#258](#) for details.

Features

- [#169](#): Create semver package and split code among different modules in the packages:
 - Remove `semver.py`
 - Create `src/semver/__init__.py`
 - Create `src/semver/cli.py` for all CLI methods
 - Create `src/semver/_deprecated.py` for the `deprecated` decorator and other deprecated functions
 - Create `src/semver/__main__.py` to allow calling the CLI using **`python -m semver`**
 - Create `src/semver/_types.py` to hold type aliases
 - Create `src/semver/version.py` to hold the `Version` class (old name `VersionInfo`) and its utility functions
 - Create `src/semver/__about__.py` for all the metadata variables
- [#213](#): Add typing information
- [#284](#): Implement `is_compatible()` to make “is self compatible with X”.
- [#305](#): Rename `VersionInfo` to `Version` but keep an alias for compatibility
- [PR #359](#): Add optional parameter `optional_minor_and_patch` in `parse()` to allow optional minor and patch parts.
- [PR #362](#): Make `match()` accept a bare version string as match expression, defaulting to equality testing.
- [#364](#): Enhance `pyproject.toml` to make it possible to use the **`pyproject-build`** command from the build module. For more information, see [Building semver](#).
- [#365](#): Improve `pyproject.toml`.
 - Use `setuptools`, add metadata. Taken approach from [A Practical Guide to Setuptools and Pyproject.toml](#).
 - Doc: Describe building of semver
 - Remove `.travis.yml` in `MANIFEST.in` (not needed anymore)
 - Distinguish between Python 3.6 and others in `tox.ini`
 - Add `skip_missing_interpreters` option for `tox.ini`
 - GH Action: Upgrade `setuptools` and `setuptools-scm` and test against 3.11.0-rc.2

Improved Documentation

- [#276](#): Document how to create a subclass from `VersionInfo` class
- [#284](#): Document deprecation of `isvalid()`.
- [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.
- [#304](#): Several improvements in documentation:

- Reorganize API documentation.
 - Add migration chapter from semver2 to semver3.
 - Distinguish between changelog for version 2 and 3
- #305: Add note about *Version* rename.
- #312: Rework “Usage” section.
 - Mention the rename of *VersionInfo* to *Version* class
 - Remove semver. prefix in doctests to make examples shorter
 - Correct some references to dunder methods like `__getitem__()`, `__gt__()` etc.
 - Remove inconsistencies and mention module level function as deprecated and discouraged from using
 - Make empty `super()` call in `semverwithvprefix.py` example
- #315: Improve release procedure text
- #335: Add new section “Converting versions between PyPI and semver” the limitations and possible use cases to convert from one into the other versioning scheme.
- #340: Describe how to get version from a file
- #343: Describe combining Pydantic with semver in the “Advanced topic” section.
- #350: Restructure usage section. Create subdirectory “usage/” and splitted all section into different files.
- #351: Introduce new topics for:
 - “Migration to semver3”
 - “Advanced topics”
- PR #392: Fix the example in the documentation for combining semver and pydantic.

Trivial/Internal Changes

- #169: Adapted infrastructure code to the new project layout.
 - Replace `setup.py` with `setup.cfg` because the `setup.cfg` is easier to use
 - Adapt documentation code snippets where needed
 - Adapt tests
 - Changed the deprecated to hardcode the `semver` package name in the warning.

Increase coverage to 100% for all non-deprecated APIs

- PR #290: Add supported Python versions to **black**.
- #304: Support PEP-561 `py.typed`.

According to the mentioned PEP:

“Package maintainers who wish to support type checking of their code MUST add a marker file named `py.typed` to their package supporting typing.”

Add `package_data` to `setup.cfg` to include this marker in dist and whl file.

- #309: Some (private) functions from the `semver.version` module has been changed.

The following functions got renamed:

- function `semver.version.comparator()` got renamed to `semver.version._comparator()` as it is only useful inside the `Version` class.
- function `semver.version.cmp()` got renamed to `semver.version._cmp()` as it is only useful inside the `Version` class.

The following functions got integrated into the `Version` class:

- function `semver.version._nat_cmd()` as a classmethod
- function `semver.version.ensure_str()`
- #313: Correct `tox.ini` for changelog entry to skip installation for semver. This should speed up the execution of towncrier.
- #316: Comparisons of `Version` class and other types return now a `NotImplemented` constant instead of a `TypeError` exception.

The `NotImplemented` section of the Python documentation recommends returning this constant when comparing with `__gt__`, `__lt__`, and other comparison operators to “to indicate that the operation is not implemented with respect to the other type”.

- #319: Introduce stages in `.travis.yml` The config file contains now two stages: check and test. If check fails, the test stage won't be executed. This could speed up things when some checks fails.
- #322: Switch from Travis CI to GitHub Actions.
- #347: Support Python 3.10 in GitHub Action and other config files.
- #378: Fix some typos in Towncrier configuration
- #388: For pytest, switch to the more modern `importlib` approach as it doesn't require to modify `sys.path`: <https://docs.pytest.org/en/7.2.x/explanation/pythonpath.html>
- PR #389: Add public class variable `Version.NAMES`.

This class variable contains a tuple of strings that contains the names of all attributes of a `Version` (like "major", "minor" etc).

In cases we need to have dynamical values, this makes it easier to iterate.

1.10 Changelog semver3 development

This site contains all the changes during the development phase.

1.10.1 Version 3.0.0-dev.4

Released

2022-12-18

Maintainer

Bug Fixes

- [#374](#): Correct Towncrier’s config entries in the `pyproject.toml` file. The old entries `[[tool.towncrier.type]]` are deprecated and need to be replaced by `[tool.towncrier.fragment.<TYPE>]`.

Deprecations

- [#372](#): Deprecate support for Python 3.6.

Python 3.6 reached its end of life and isn’t supported anymore. At the time of writing (Dec 2022), the lowest version is 3.7.

Although the [poll](#) didn’t cast many votes, the majority agree to remove support for Python 3.6.

Improved Documentation

- [#335](#): Add new section “Converting versions between PyPI and semver” the limitations and possible use cases to convert from one into the other versioning scheme.
- [#340](#): Describe how to get version from a file
- [#343](#): Describe combining Pydantic with semver in the “Advanced topic” section.
- [#350](#): Restructure usage section. Create subdirectory “usage/” and splitted all section into different files.
- [#351](#): Introduce new topics for:
 - “Migration to semver3”
 - “Advanced topics”

Features

- [PR #359](#): Add optional parameter `optional_minor_and_patch` in `Version.parse()` to allow optional minor and patch parts.
- [PR #362](#): Make `Version.match()` accept a bare version string as match expression, defaulting to equality testing.
- [#364](#): Enhance `pyproject.toml` to make it possible to use the **pyproject-build** command from the build module. For more information, see [Building semver](#).
- [#365](#): Improve `pyproject.toml`.
 - Use `setuptools`, add metadata. Taken approach from [A Practical Guide to Setuptools and Pyproject.toml](#).
 - Doc: Describe building of semver
 - Remove `.travis.yml` in `MANIFEST.in` (not needed anymore)
 - Distinguish between Python 3.6 and others in `tox.ini`
 - Add `skip_missing_interpreters` option for `tox.ini`
 - GH Action: Upgrade `setuptools` and `setuptools-scm` and test against 3.11.0-rc.2

Trivial/Internal Changes

- #378: Fix some typos in Towncrier configuration
-

1.10.2 Version 3.0.0-dev.3

Released

2022-01-19

Maintainer

Tom Schraitle

Bug Fixes

- #310: Rework API documentation. Follow a more “semi-manual” attempt and add auto directives into docs/api.rst.

Improved Documentation

- #312: Rework “Usage” section.
 - Mention the rename of *VersionInfo* to *Version* class
 - Remove `semver.` prefix in doctests to make examples shorter
 - Correct some references to dunder methods like `__getitem__()`, `__gt__()` etc.
 - Remove inconsistencies and mention module level function as deprecated and discouraged from using
 - Make empty `super()` call in `semverwithvprefix.py` example
- #315: Improve release procedure text

Trivial/Internal Changes

- #309: Some (private) functions from the `semver.version` module has been changed.

The following functions got renamed:

- function `semver.version.comparator` got renamed to `semver.version._comparator()` as it is only useful inside the *Version* class.
- function `semver.version.cmp` got renamed to `semver.version._cmp()` as it is only useful inside the *Version* class.

The following functions got integrated into the *Version* class:

- function `semver.version._nat_cmd` as a classmethod
- function `semver.version.ensure_str`

- #313: Correct `tox.ini` for changelog entry to skip installation for `semver`. This should speed up the execution of towncrier.

- [#316](#): Comparisons of `Version` class and other types return now a `NotImplemented` constant instead of a `TypeError` exception.

The `NotImplemented` section of the Python documentation recommends returning this constant when comparing with `__gt__`, `__lt__`, and other comparison operators to “to indicate that the operation is not implemented with respect to the other type”.

- [#319](#): Introduce stages in `.travis.yml` The config file contains now two stages: check and test. If check fails, the test stage won't be executed. This could speed up things when some checks fails.
 - [#322](#): Switch from Travis CI to GitHub Actions.
 - [#347](#): Support Python 3.10 in GitHub Action and other config files.
-

1.10.3 Version 3.0.0-dev.2

Released

2020-11-01

Maintainer

Tom Schraitle

Deprecations

- [#169](#): Deprecate CLI functions not imported from `semver.cli`.

Features

- [#169](#): Create `semver` package and split code among different modules in the packages.
 - Remove `semver.py`
 - Create `src/semver/__init__.py`
 - Create `src/semver/cli.py` for all CLI methods
 - Create `src/semver/_deprecated.py` for the `deprecated` decorator and other deprecated functions
 - Create `src/semver/__main__.py` to allow calling the CLI using **`python -m semver`**
 - Create `src/semver/_types.py` to hold type aliases
 - Create `src/semver/version.py` to hold the `Version` class (old name `VersionInfo`) and its utility functions
 - Create `src/semver/__about__.py` for all the metadata variables
- [#305](#): Rename `VersionInfo` to `Version` but keep an alias for compatibility

Improved Documentation

- [#304](#): Several improvements in documentation:
 - Reorganize API documentation.
 - Add migration chapter from semver2 to semver3.
 - Distinguish between changelog for version 2 and 3
- [#305](#): Add note about `Version` rename.

Trivial/Internal Changes

- [#169](#): Adapted infrastructure code to the new project layout.
 - Replace `setup.py` with `setup.cfg` because the `setup.cfg` is easier to use
 - Adapt documentation code snippets where needed
 - Adapt tests
 - Changed the `deprecated` to hardcode the `semver` package name in the warning.

Increase coverage to 100% for all non-deprecated APIs

- [#304](#): Support PEP-561 `py.typed`.

According to the mentioned PEP:

“Package maintainers who wish to support type checking of their code **MUST** add a marker file named `py.typed` to their package supporting typing.”

Add `package_data` to `setup.cfg` to include this marker in dist and whl file.

1.10.4 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
- [#213](#): Add typing information

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**.

INDICES AND TABLES

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

PYTHON MODULE INDEX

S

`semver.__about__`, [32](#)
`semver.__main__`, [39](#)
`semver._deprecated`, [32](#)
`semver.cli`, [38](#)
`semver.version`, [39](#)

Symbols

[__author__](#) (in module `semver.__about__`), 32
[__description__](#) (in module `semver.__about__`), 32
[__eq__\(\)](#) (`semver.version.Version` method), 39
[__ge__\(\)](#) (`semver.version.Version` method), 40
[__getitem__\(\)](#) (`semver.version.Version` method), 40
[__gt__\(\)](#) (`semver.version.Version` method), 40
[__hash__\(\)](#) (`semver.version.Version` method), 40
[__iter__\(\)](#) (`semver.version.Version` method), 40
[__le__\(\)](#) (`semver.version.Version` method), 40
[__lt__\(\)](#) (`semver.version.Version` method), 40
[__maintainer__](#) (in module `semver.__about__`), 32
[__ne__\(\)](#) (`semver.version.Version` method), 41
[__repr__\(\)](#) (`semver.version.Version` method), 41
[__str__\(\)](#) (`semver.version.Version` method), 41
[__version__](#) (in module `semver.__about__`), 32
[--help](#)
 pysemver command line option, 46
[--version](#)
 pysemver command line option, 46
[-h](#)
 pysemver command line option, 46
[<PART>](#)
 pysemver command line option, 46
[<VERSION1>](#)
 pysemver command line option, 47
[<VERSION2>](#)
 pysemver command line option, 47
[<VERSION>](#)
 pysemver command line option, 47

B

[build](#) (`semver.version.Version` property), 41
[bump_build\(\)](#) (in module `semver._deprecated`), 32
[bump_build\(\)](#) (`semver.version.Version` method), 41
[bump_major\(\)](#) (in module `semver._deprecated`), 33
[bump_major\(\)](#) (`semver.version.Version` method), 41
[bump_minor\(\)](#) (in module `semver._deprecated`), 33
[bump_minor\(\)](#) (`semver.version.Version` method), 41
[bump_patch\(\)](#) (in module `semver._deprecated`), 33
[bump_patch\(\)](#) (`semver.version.Version` method), 42

[bump_prerelease\(\)](#) (in module `semver._deprecated`), 34
[bump_prerelease\(\)](#) (`semver.version.Version` method), 42

C

[cmd_bump\(\)](#) (in module `semver.cli`), 38
[cmd_check\(\)](#) (in module `semver.cli`), 38
[cmd_compare\(\)](#) (in module `semver.cli`), 38
[compare\(\)](#) (in module `semver._deprecated`), 32
[compare\(\)](#) (`semver.version.Version` method), 42
[createparser\(\)](#) (in module `semver.cli`), 38

D

[deprecated\(\)](#) (in module `semver._deprecated`), 34

F

[finalize_version\(\)](#) (in module `semver._deprecated`), 34
[finalize_version\(\)](#) (`semver.version.Version` method), 43
[format_version\(\)](#) (in module `semver._deprecated`), 35

I

[is_compatible\(\)](#) (`semver.version.Version` method), 43
[is_valid\(\)](#) (`semver.version.Version` class method), 43

M

[main\(\)](#) (in module `semver.cli`), 38
[major](#) (`semver.version.Version` property), 44
[match\(\)](#) (in module `semver._deprecated`), 35
[match\(\)](#) (`semver.version.Version` method), 44
[max_ver\(\)](#) (in module `semver._deprecated`), 35
[min_ver\(\)](#) (in module `semver._deprecated`), 36
[minor](#) (`semver.version.Version` property), 44
[module](#)
 [semver.__about__](#), 32
 [semver.__main__](#), 39
 [semver._deprecated](#), 32
 [semver.cli](#), 38
 [semver.version](#), 39

N

NAMES (*semver.version.Version* attribute), 39

next_version() (*semver.version.Version* method), 44

P

parse() (*in module semver._deprecated*), 36

parse() (*semver.version.Version* class method), 44

parse_version_info() (*in module semver._deprecated*), 36

patch (*semver.version.Version* property), 45

prerelease (*semver.version.Version* property), 45

process() (*in module semver.cli*), 39

pysemver command line option

 --help, 46

 --version, 46

 -h, 46

 <PART>, 46

 <VERSION1>, 47

 <VERSION2>, 47

 <VERSION>, 47

R

replace() (*in module semver._deprecated*), 37

replace() (*semver.version.Version* method), 45

S

semver.__about__
 module, 32

semver.__main__
 module, 39

semver._deprecated
 module, 32

semver.cli
 module, 38

semver.version
 module, 39

SEMPER_SPEC_VERSION (*in module semver.__about__*),
 32

T

to_dict() (*semver.version.Version* method), 45

to_tuple() (*semver.version.Version* method), 45

V

Version (*class in semver.version*), 39

VersionInfo (*in module semver.version*), 39