

---

# **python-semanticversion Documentation**

*Release 2.6.0*

**Raphaël Barrois**

**Sep 03, 2017**



---

## Contents

---

<b>1</b>	<b>Links</b>	<b>3</b>
<b>2</b>	<b>Getting started</b>	<b>5</b>
2.1	Versions . . . . .	5
2.2	Requirement specification . . . . .	7
<b>3</b>	<b>Using with Django</b>	<b>9</b>
<b>4</b>	<b>Contributing</b>	<b>11</b>
<b>5</b>	<b>Contents</b>	<b>13</b>
5.1	Reference . . . . .	13
5.2	Interaction with Django . . . . .	22
5.3	ChangeLog . . . . .	22
5.4	Credits . . . . .	26
<b>6</b>	<b>Indices and tables</b>	<b>27</b>
	<b>Python Module Index</b>	<b>29</b>



This small python library provides a few tools to handle [SemVer](#) in Python. It follows strictly the 2.0.0 version of the SemVer scheme.



# CHAPTER 1

---

## Links

---

- Package on PyPI: [http://pypi.python.org/pypi/semantic\\_version/](http://pypi.python.org/pypi/semantic_version/)
- Doc on ReadTheDocs: <https://python-semanticversion.readthedocs.io/>
- Source on GitHub: <http://github.com/rbarrois/python-semanticversion/>
- Build on Travis CI: <http://travis-ci.org/rbarrois/python-semanticversion/>
- Semantic Version specification: [SemVer](#)





## CHAPTER 2

---

### Getting started

---

Install the package from PyPI, using pip:

```
pip install semantic_version
```

Or from GitHub:

```
$ git clone git://github.com/rbarrois/python-semanticversion.git
```

Import it in your code:

```
import semantic_version
```

This module provides two classes to handle semantic versions:

- *Version* represents a version number (0.1.1-alpha+build.2012-05-15)
- *Spec* represents a requirement specification ( $\geq 0.1.1, < 0.3.0$ )

## Versions

Defining a *Version* is quite simple:

```
>>> import semantic_version
>>> v = semantic_version.Version('0.1.1')
>>> v.major
0
>>> v.minor
1
>>> v.patch
1
>>> v.prerelease
[]
>>> v.build
```

```
[ ]
>>> list(v)
[0, 1, 1, [], []]
```

If the provided version string is invalid, a `ValueError` will be raised:

```
>>> semantic_version.Version('0.1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/rbarrois/dev/semantic_version/src/semantic_version/base.py", line 64, in
↳in __init__
    major, minor, patch, prerelease, build = self.parse(version_string, partial)
  File "/Users/rbarrois/dev/semantic_version/src/semantic_version/base.py", line 86, in
↳in parse
    raise ValueError('Invalid version string: %r' % version_string)
ValueError: Invalid version string: '0.1'
```

In order to define “relaxed” version strings, you must pass in `partial=True`:

```
>>> v = semantic_version.Version('0.1', partial=True)
>>> list(v)
[0, 1, None, None, None]
```

Obviously, `Versions` can be compared:

```
>>> semantic_version.Version('0.1.1') < semantic_version.Version('0.1.2')
True
>>> semantic_version.Version('0.1.1') > semantic_version.Version('0.1.1-alpha')
True
>>> semantic_version.Version('0.1.1') <= semantic_version.Version('0.1.1-alpha')
False
```

You can also get a new version that represents a bump in one of the version levels:

```
>>> v = semantic_version.Version('0.1.1-pre+build')
>>> new_v = v.next_major()
>>> str(new_v)
'1.0.0'
>>> v = semantic_version.Version('1.1.1-pre+build')
>>> new_v = v.next_minor()
>>> str(new_v)
'1.2.0'
>>> v = semantic_version.Version('1.1.1-pre+build')
>>> new_v = v.next_patch()
>>> str(new_v)
'1.1.2'
```

It is also possible to check whether a given string is a proper semantic version string:

```
>>> semantic_version.validate('0.1.3')
True
>>> semantic_version.validate('0a2')
False
```

## Requirement specification

The *Spec* object describes a range of accepted versions:

```
>>> s = Spec('>=0.1.1') # At least 0.1.1
>>> s.match(Version('0.1.1'))
True
>>> s.match(Version('0.1.1-alpha')) # pre-release satisfy version spec
True
>>> s.match(Version('0.1.0'))
False
```

Simpler test syntax is also available using the `in` keyword:

```
>>> s = Spec('==0.1.1')
>>> Version('0.1.1-alpha') in s
True
>>> Version('0.1.2') in s
False
```

Combining specifications can be expressed in two ways:

- Components separated by commas in a single string:

```
>>> Spec('>=0.1.1,<0.3.0')
```

- Components given as different arguments:

```
>>> Spec('>=0.1.1', '<0.3.0')
```

- A mix of both versions:

```
>>> Spec('>=0.1.1', '!=0.2.4-alpha,<0.3.0')
```

## Using a specification

The *Spec.filter()* method filters an iterable of *Version*:

```
>>> s = Spec('>=0.1.0,<0.4.0')
>>> versions = (Version('0.%d.0' % i) for i in range(6))
>>> for v in s.filter(versions):
...     print v
0.1.0
0.2.0
0.3.0
```

It is also possible to select the ‘best’ version from such iterables:

```
>>> s = Spec('>=0.1.0,<0.4.0')
>>> versions = (Version('0.%d.0' % i) for i in range(6))
>>> s.select(versions)
Version('0.3.0')
```

## Coercing an arbitrary version string

Some user-supplied input might not match the semantic version scheme. For such cases, the `Version.coerce()` method will try to convert any version-like string into a valid semver version:

```
>>> Version.coerce('0')
Version('0.0.0')
>>> Version.coerce('0.1.2.3.4')
Version('0.1.2+3.4')
>>> Version.coerce('0.1.2a3')
Version('0.1.2-a3')
```

## Including pre-release identifiers in specifications

When testing a `Version` against a `Spec`, comparisons are only performed for components defined in the `Spec`; thus, a pre-release version (`1.0.0-alpha`), while not strictly equal to the non pre-release version (`1.0.0`), satisfies the `==1.0.0 Spec`.

Pre-release identifiers will only be compared if included in the `Spec` definition or (for the empty pre-release number) if a single dash is appended (`1.0.0-`):

```
>>> Version('0.1.0-alpha') in Spec('>=0.1.0') # No pre-release identifier
True
>>> Version('0.1.0-alpha') in Spec('>=0.1.0-') # Include pre-release in checks
False
```

## Including build metadata in specifications

Build metadata has no ordering; thus, the only meaningful comparison including build metadata is equality.

```
>>> Version('1.0.0+build2') in Spec('<=1.0.0') # Build metadata ignored
True
>>> Version('1.0.0+build2') in Spec('==1.0.0+build2') # Include build in checks
False
```

## CHAPTER 3

---

### Using with Django

---

The `semantic_version.django_fields` module provides django fields to store *Version* or *Spec* objects. More documentation is available in the *Interaction with Django* section.



In order to contribute to the source code:

- Open an issue on [GitHub](https://github.com/rbarrois/python-semanticversion/issues): <https://github.com/rbarrois/python-semanticversion/issues>
- Fork the [repository](#) and submit a pull request on [GitHub](#)
- Or send me a patch (<mailto:raphael.barrois+semver@polytechnique.org>)

When submitting patches or pull requests, you should respect the following rules:

- Coding conventions are based on [PEP 8](#)
- The whole test suite must pass after adding the changes
- The test coverage for a new feature must be 100%
- New features and methods should be documented in the *Reference* section and included in the *ChangeLog*
- Include your name in the *Contributors* section

---

**Note:** All files should contain the following header:

```
# -*- encoding: utf-8 -*-  
# Copyright (c) The python-semanticversion project
```

---





## Reference

### Module-level functions

`semantic_version.compare(v1, v2)`

Compare two version strings, and return a result similar to that of `cmp()`:

```
>>> compare('0.1.1', '0.1.2')
-1
>>> compare('0.1.1', '0.1.1')
0
>>> compare('0.1.1', '0.1.1-alpha')
1
```

#### Parameters

- **v1** (*str*) – The first version to compare
- **v2** (*str*) – The second version to compare

**Raises** `ValueError`, if any version string is invalid

**Return type** `int`, `-1/0/1` as for a `cmp()` comparison; `NotImplemented` if versions only differ by build metadata

**Warning:** Since build metadata has no ordering, `compare(Version('0.1.1'), Version('0.1.1+3'))` returns `NotImplemented`

`semantic_version.match(spec, version)`

Check whether a version string matches a specification string:

```
>>> match('>=0.1.1', '0.1.2')
True
>>> match('>=0.1.1', '0.1.1-alpha')
False
>>> match('~0.1.1', '0.1.1-alpha')
True
```

### Parameters

- **spec** (*str*) – The specification to use, as a string
- **version** (*str*) – The version string to test against the spec

**Raises** ValueError, if the spec or the version is invalid

**Return type** bool

`semantic_version.validate` (*version*)

Check whether a version string complies with the [SemVer](#) rules.

```
>>> semantic_version.validate('1.1.1')
True
>>> semantic_version.validate('1.2.3a4')
False
```

**Parameters** **version** (*str*) – The version string to validate

**Return type** bool

## Representing a version (the Version class)

**class** `semantic_version.Version` (*version\_string* [, *partial=False* ])

Object representation of a [SemVer](#)-compliant version.

Constructed from a textual version string:

```
>>> Version('1.1.1')
Version('1.1.1')
>>> str(Version('1.1.1'))
'1.1.1'
```

### Attributes

#### **partial**

bool, whether this is a ‘partial’ or a complete version number. Partial version number may lack *minor* or *patch* version numbers.

#### **major**

int, the major version number

#### **minor**

int, the minor version number.

May be None for a *partial* version number in a <major> format.

**patch**

int, the patch version number.

May be None for a *partial* version number in a <major> or <major>.<minor> format.

**prerelease**

tuple of strings, the prerelease component.

It contains the various dot-separated identifiers in the prerelease component.

May be None for a *partial* version number in a <major>, <major>.<minor> or <major>.<minor>.<patch> format.

**build**

tuple of strings, the build metadata.

It contains the various dot-separated identifiers in the build metadata.

May be None for a *partial* version number in a <major>, <major>.<minor>, <major>.<minor>.<patch> or <major>.<minor>.<patch>-<prerelease> format.

**Methods****`__iter__`** (*self*)

Iterates over the version components (*major, minor, patch, prerelease, build*):

```
>>> list(Version('0.1.1'))
[0, 1, 1, [], []]
```

**Note:** This may pose some subtle bugs when iterating over a single version while expecting an iterable of versions – similar to:

```
>>> list('abc')
['a', 'b', 'c']
>>> list(('abc',))
['abc']
```

**`__cmp__`** (*self, other*)

Provides comparison methods with other *Version* objects.

The rules are:

- For non-*partial* versions, compare using the *SemVer* scheme
- If any compared object is *partial*:
  - Begin comparison using the *SemVer* scheme
  - If a component (*minor, patch, prerelease* or *build*) was absent from the *partial Version* – represented with None –, consider both versions equal.

For instance, `Version('1.0', partial=True)` means “any version beginning in 1.0”.

`Version('1.0.1-alpha', partial=True)` means “The 1.0.1-alpha version or any any release differing only in build metadata”: 1.0.1-alpha+build3 matches, 1.0.1-alpha.2 doesn't.

Examples:

```
>>> Version('1.0', partial=True) == Version('1.0.1')
True
>>> Version('1.0.1-rc1.1') == Version('1.0.1-rc1', partial=True)
False
>>> Version('1.0.1-rc1+build345') == Version('1.0.1-rc1')
False
>>> Version('1.0.1-rc1+build345') == Version('1.0.1-rc1', partial=True)
True
```

**\_\_str\_\_** (*self*)

Returns the standard text representation of the version:

```
>>> v = Version('0.1.1-rc2+build4.4')
>>> v
Version('0.1.1-rc2+build4.4')
>>> str(v)
'0.1.1-rc2+build4.4'
```

**\_\_hash\_\_** (*self*)

Provides a hash based solely on the components.

Allows using a *Version* as a dictionary key.

---

**Note:** A fully qualified *partial Version*

(up to the *build* component) will hash the same as the equally qualified, non-*partial Version*:

```
>>> hash(Version('1.0.1+build4')) == hash(Version('1.0.1+build4',
↳partial=True))
True
```

---

## Class methods

**classmethod parse** (*cls*, *version\_string* [, *partial=False* ])

Parse a version string into a (major, minor, patch, prerelease, build) tuple.

**Parameters**

- **version\_string** (*str*) – The version string to parse
- **partial** (*bool*) – Whether this should be considered a *partial* version

**Raises** ValueError, if the *version\_string* is invalid.

**Return type** (*major*, *minor*, *patch*, *prerelease*, *build*)

**classmethod coerce** (*cls*, *version\_string* [, *partial=False* ])

Try to convert an arbitrary version string into a *Version* instance.

Rules are:

- If no minor or patch component, and *partial* is False, replace them with zeroes
- Any character outside of a-zA-Z0-9.+– is replaced with a –
- If more than 3 dot-separated numerical components, everything from the fourth component belongs to the *build* part

- Any extra + in the *build* part will be replaced with dots

Examples:

```
>>> Version.coerce('02')
Version('2.0.0')
>>> Version.coerce('1.2.3.4')
Version('1.2.3+4')
>>> Version.coerce('1.2.3.4beta2')
Version('1.2.3+4beta2')
>>> Version.coerce('1.2.3.4.5_6/7+8+9+10')
Version('1.2.3+4.5-6-7.8.9.10')
```

### Parameters

- **version\_string** (*str*) – The version string to coerce
- **partial** (*bool*) – Whether to allow generating a *partial* version

**Raises** ValueError, if the `version_string` is invalid.

**Return type** *Version*

## Version specifications (the Spec class)

Version specifications describe a ‘range’ of accepted versions: older than, equal, similar to, ...

The main issue with representing version specifications is that the usual syntax does not map well onto *SemVer* precedence rules:

- A specification of `<1.3.4` is not expected to allow `1.3.4-rc2`, but strict *SemVer* comparisons allow it ; prereleases has the issue of excluding `1.3.3+build3` ;
- It may be necessary to exclude either all variations on a patch-level release (`!=1.3.3`) or specifically one build-level release (`1.3.3-build.434`).

In order to have version specification behave naturally, the rules are the following:

- If no pre-release number was included in the specification, pre-release numbers are ignored when deciding whether a version satisfies a specification.
- If no build metadata was included in the specification, build metadata is ignored when deciding whether a version satisfies a specification.

This means that:

```
>>> Version('1.1.1-rc1') in Spec('<1.1.1')
False
>>> Version('1.1.1-rc1') in Spec('<1.1.1-rc4')
True
>>> Version('1.1.1-rc1+build4') in Spec('<=1.1.1-rc1')
True
>>> Version('1.1.1-rc1+build4') in Spec('==1.1.1-rc1+build2')
False
```

---

**Note:** python-semanticversion also accepts "\*" as a version spec, that matches all (valid) version strings.

---



---

**Note:** python-semanticversion supports PyPI-style [compatible release clauses](#):

---

- `~=2.2` means “Any release between 2.2.0 and 3.0.0”
  - `~=1.4.5` means “Any release between 1.4.5 and 1.5.0”
- 

**Note:** python-semanticversion includes support for NPM-style specs:

- `~1.2.3` means “Any release between 1.2.3 and 1.3.0”
  - `^1.3.4` means “Any release between 1.3.4 and 2.0.0”
- 

In order to force matches to *strictly* compare version numbers, these additional rules apply:

- Setting a pre-release separator without a pre-release identifier (`<=1.1.1-`) forces match to take into account pre-release version:

```
>>> Version('1.1.1-rc1') in Spec('<1.1.1')
False
>>> Version('1.1.1-rc1') in Spec('<1.1.1-')
True
```

- Setting a build metadata separator without build metadata (`<=1.1.1+`) forces matches “up to the build metadata”; use this to include/exclude a release lacking build metadata while excluding/including all other builds of that release

```
>>> Version('1.1.1') in Spec('==1.1.1+')
True
>>> Version('1.1.1+2') in Spec('==1.1.1+')
False
```

**Warning:** As stated in the [SemVer](#) specification, the ordering of build metadata is *undefined*. Thus, a *Spec* string can only mention build metadata to include or exclude a specific version:

- `==1.1.1+b1234` includes this specific build
- `!=1.1.1+b1234` excludes it (but would match `1.1.1+b1235`)
- `<1.1.1+b1` is invalid

**class** `semantic_version.Spec` (`spec_string`[, `spec_string`[, ... ]])  
Stores a list of *SpecItem* and matches any *Version* against all contained *specs*.

It is built from a comma-separated list of version specifications:

```
>>> Spec('>=1.0.0,<1.2.0,!1.1.4')
<Spec: (
  <SpecItem: >= Version('1.0.0', partial=True)>,
  <SpecItem: < Version('1.2.0', partial=True)>,
  <SpecItem: != Version('1.1.4', partial=True)>
)>
```

Version specifications may also be passed in separated arguments:

```
>>> Spec('>=1.0.0', '<1.2.0', '!1.1.4,!1.1.13')
<Spec: (
  <SpecItem: >= Version('1.0.0', partial=True)>,
  <SpecItem: < Version('1.2.0', partial=True)>,
  <SpecItem: != Version('1.1.4', partial=True)>,
  <SpecItem: != Version('1.1.13', partial=True)>
)>
```

```
<SpecItem: != Version('1.1.4', partial=True)>,
<SpecItem: != Version('1.1.13', partial=True)>,
)>
```

## Attributes

### specs

Tuple of *SpecItem*, the included specifications.

## Methods

### match (*self*, *version*)

Test whether a given *Version* matches all included *SpecItem*:

```
>>> Spec('>=1.1.0,<1.1.2').match(Version('1.1.1'))
True
```

**Parameters** *version* (*Version*) – The version to test against the specs

**Return type** bool

### filter (*self*, *versions*)

Extract all compatible *versions* from an iterable of *Version* objects.

**Parameters** *versions* (iterable of *Version*) – The versions to filter

**Yield** *Version*

### select (*self*, *versions*)

Select the highest compatible version from an iterable of *Version* objects.

```
>>> s = Spec('>=0.1.0')
>>> s.select([])
None
>>> s.select([Version('0.1.0'), Version('0.1.3'), Version('0.1.1')])
Version('0.1.3')
```

**Parameters** *versions* (iterable of *Version*) – The versions to filter

**Return type** The highest compatible *Version* if at least one of the given versions is compatible; None otherwise.

### \_\_contains\_\_ (*self*, *version*)

Alias of the *match()* method; allows the use of the version in *speclist* syntax:

```
>>> Version('1.1.1-alpha') in Spec('>=1.1.0,<1.1.1')
True
```

### \_\_str\_\_ (*self*)

Converting a *Spec* returns the initial description string:

```
>>> str(Spec('>=0.1.1,!<0.1.2'))
'>=0.1.1,!<0.1.2'
```

`__iter__(self)`

Returns an iterator over the contained specs:

```
>>> for spec in Spec('>=0.1.1,!=0.1.2'):  
...     print spec  
>=0.1.1  
!=0.1.2
```

`__hash__(self)`

Provides a hash based solely on the hash of contained specs.

Allows using a *Spec* as a dictionary key.

## Class methods

**classmethod** `parse(self, specs_string)`

Retrieve a (\*specs) tuple from a string.

**Parameters** `requirement_string` (*str*) – The textual description of the specifications

**Raises** `ValueError`: if the `requirement_string` is invalid.

**Return type** (\*spec) tuple

**class** `semantic_version.SpecItem(spec_string)`

---

**Note:** This class belong to the private python-semanticversion API.

---

Stores a version specification, defined from a string:

```
>>> SpecItem('>=0.1.1')  
<SpecItem: >= Version('0.1.1', partial=True)>
```

This allows to test *Version* objects against the *SpecItem*:

```
>>> SpecItem('>=0.1.1').match(Version('0.1.1-rc1')) # pre-release satisfy_  
↳conditions  
True  
>>> Version('0.1.1+build2') in SpecItem('>=0.1.1') # build metadata is ignored_  
↳when checking for precedence  
True  
>>>  
>>> # Use the '-' marker to include the pre-release component in checks  
>>> SpecItem('>=0.1.1-').match(Version('0.1.1-rc1'))  
False  
>>> # Use the '+' marker to include the build metadata in checks  
>>> SpecItem('==0.1.1+').match(Version('0.1.1+b1234'))  
False  
>>>
```

## Attributes

**kind**

One of `KIND_LT`, `KIND_LTE`, `KIND_EQUAL`, `KIND_GTE`, `KIND_GT` and `KIND_NEQ`.



**spec**

*Version* in the *SpecItem* description.

It is always a *partial Version*.

**Class methods****classmethod parse** (*cls*, *requirement\_string*)

Retrieve a (kind, version) tuple from a string.

**Parameters** **requirement\_string** (*str*) – The textual description of the specification

**Raises** ValueError: if the requirement\_string is invalid.

**Return type** (kind, version) tuple

**Methods****match** (*self*, *version*)

Test whether a given *Version* matches this *SpecItem*:

```
>>> SpecItem('>=0.1.1').match(Version('0.1.1-alpha'))
True
>>> SpecItem('>=0.1.1-').match(Version('0.1.1-alpha'))
False
```

**Parameters** **version** (*Version*) – The version to test against the spec

**Return type** bool

**\_\_str\_\_** (*self*)

Converting a *SpecItem* to a string returns the initial description string:

```
>>> str(SpecItem('>=0.1.1'))
'>=0.1.1'
```

**\_\_hash\_\_** (*self*)

Provides a hash based solely on the current kind and the specified version.

Allows using a *SpecItem* as a dictionary key.

**Class attributes****KIND\_LT**

The kind of ‘Less than’ specifications:

```
>>> Version('1.0.0-alpha') in Spec('<1.0.0')
False
```

**KIND\_LTE**

The kind of ‘Less or equal to’ specifications:

```
>>> Version('1.0.0-alpha1+build999') in Spec('<=1.0.0-alpha1')
True
```

#### **KIND\_EQUAL**

The kind of 'equal to' specifications:

```
>>> Version('1.0.0+build3.3') in Spec('==1.0.0')
True
```

#### **KIND\_GTE**

The kind of 'Greater or equal to' specifications:

```
>>> Version('1.0.0') in Spec('>=1.0.0')
True
```

#### **KIND\_GT**

The kind of 'Greater than' specifications:

```
>>> Version('1.0.0+build667') in Spec('>1.0.1')
False
```

#### **KIND\_NEQ**

The kind of 'Not equal to' specifications:

```
>>> Version('1.0.1') in Spec('!=1.0.1')
False
```

The kind of 'Almost equal to' specifications

## Interaction with Django

The `python-semanticversion` package provides two custom fields for Django:

- `VersionField`: stores a `semantic_version.Version` object
- `SpecField`: stores a `semantic_version.Spec` object

Those fields are `django.db.models.CharField` subclasses, with their `max_length` defaulting to 200.

**class** `semantic_version.django_fields.VersionField`

Stores a `semantic_version.Version` as its string representation.

**partial**

Boolean; whether *partial* versions are allowed.

**coerce**

Boolean; whether passed in values should be coerced into a semver string before storing.

**class** `semantic_version.django_fields.SpecField`

Stores a `semantic_version.Spec` as its comma-separated string representation.

## ChangeLog

### 2.6.0 (2016-09-25)

*New:*

- #43: Add support for Django up to 1.10.

*Removed:*

- Remove support for Django<1.7

*Bugfix:*

- #35: Properly handle ^0.X.Y in a NPM-compatible way

## 2.5.0 (2016-02-12)

*Bugfix:*

#18: According to SemVer 2.0.0, build numbers aren't ordered.

- Remove specs of the Spec ('<1.1.3+') form
- Comparing Version('0.1.0') to Version('0.1.0+bcd') has new rules:

```
>>> Version('0.1.0+1') == Version('0.1.0+bcd')
False
>>> Version('0.1.0+1') != Version('0.1.0+bcd')
True
>>> Version('0.1.0+1') < Version('0.1.0+bcd')
False
>>> Version('0.1.0+1') > Version('0.1.0+bcd')
False
>>> Version('0.1.0+1') <= Version('0.1.0+bcd')
False
>>> Version('0.1.0+1') >= Version('0.1.0+bcd')
False
>>> compare(Version('0.1.0+1'), Version('0.1.0+bcd'))
NotImplemented
```

- `semantic_version.compare()` returns NotImplemented when its parameters differ only by build metadata
- Spec ('<=1.3.0') now matches Version('1.3.0+abde24fe883')
- #24: Fix handling of bumping pre-release versions, thanks to @minchinweb.
- #30: Add support for NPM-style ^1.2.3 and ~2.3.4 specs, thanks to @skwashd

## 2.4.2 (2015-07-02)

*Bugfix:*

- Fix tests for Django 1.7+, thanks to @mhrivnak.

## 2.4.1 (2015-04-01)

*Bugfix:*

- Fix packaging metadata (advertise Python 3.4 support)

## 2.4.0 (2015-04-01)

*New:*

- #16: Add an API for bumping versions, by @RickEyre.

## 2.3.1 (2014-09-24)

*Bugfix:*

- #13: Fix handling of files encoding in `setup.py`.

## 2.3.0 (2014-03-16)

*New:*

- Handle the full `semver-2.0.0` specifications (instead of the `2.0.0-rc2` of previous releases)
- #8: Allow '\*' as a valid version spec

## 2.2.2 (2013-12-23)

*Bugfix:*

- #5: Fix packaging (broken symlinks, old-style distutils, etc.)

## 2.2.1 (2013-10-29)

*Bugfix:*

- #2: Properly expose `validate()` as a top-level module function.

## 2.2.0 (2013-03-22)

*Bugfix:*

- #1: Allow partial versions without minor or patch level

*New:*

- Add the `Version.coerce` class method to `Version` class for mapping arbitrary version strings to semver.
- Add the `validate()` method to validate a version string against the SemVer rules.
- Full Python3 support

## 2.1.2 (2012-05-22)

*Bugfix:*

- Properly validate `VersionField` and `SpecField`.

### 2.1.1 (2012-05-22)

*New:*

- Add introspection rules for south

### 2.1.0 (2012-05-22)

*New:*

- Add `semantic_version.Spec.filter()` (filter a list of `Version`)
- Add `semantic_version.Spec.select()` (select the highest `Version` from a list)
- Update `semantic_version.Version.__repr__()`

### 2.0.0 (2012-05-22)

*Backwards incompatible changes:*

- Removed “loose” specification support
- Cleanup `Spec` to be more intuitive.
- Merge `Spec` and `SpecList` into `Spec`.
- Remove `SpecListField`

### 1.2.0 (2012-05-18)

*New:*

- Allow split specifications when instantiating a `SpecList`:

```
>>> SpecList('>=0.1.1', '!=0.1.3') == SpecList('>=0.1.1,!=0.1.3')
True
```

### 1.1.0 (2012-05-18)

*New:*

- Improved “loose” specification support (`>~`, `<~`, `!~`)
- Introduced “not equal” specifications (`!=`, `!~`)
- `SpecList` class combining many `Spec`
- Add `SpecListField` to store a `SpecList`.

### 1.0.0 (2012-05-17)

First public release.

*New:*

- `Version` and `Spec` classes
- Related django fields: `VersionField` and `SpecField`

## Credits

### Maintainers

The `python-semanticversion` project is operated and maintained by:

- Raphaël Barrois <raphael.barrois+semver@polytechnique.org> (<https://github.com/rbarrois>)

### Contributors

The project has received contributions from (in alphabetical order):

- Raphaël Barrois <raphael.barrois+semver@polytechnique.org> (<https://github.com/rbarrois>)
- Rick Eyre <rick.eyre@outlook.com> (<https://github.com/rickeyre>)
- Hugo Rodger-Brown <hugo@yunojuno.com> (<https://github.com/yunojuno>)
- Michael Hrivnak <mhrivnak@hrivnak.org> (<https://github.com/mhrivnak>)
- William Minchin <w\_minchin@hotmail.com> (<https://github.com/minchinweb>)
- Dave Hall <skwadhd@gmail.com> (<https://github.com/skwashd>)
- Martin Ek <mail@ekmartin.com> (<https://github.com/ekmartin>)

### Contributor license agreement

---

**Note:** This agreement is required to allow redistribution of submitted contributions. See <http://oss-watch.ac.uk/resources/cla> for an explanation.

---

Any contributor proposing updates to the code or documentation of this project *MUST* add its name to the list in the *Contributors* section, thereby “signing” the following contributor license agreement:

They accept and agree to the following terms for their present and future contributions submitted to the `python-semanticversion` project:

- They represent that they are legally entitled to grant this license, and that their contributions are their original creation
- They grant the `python-semanticversion` project a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare derivative works of, publicly display, sublicense and distribute their contributions and such derivative works.
- They are not expected to provide support for their contributions, except to the extent they desire to provide support.

---

**Note:** The above agreement is inspired by the Apache Contributor License Agreement.

---

## CHAPTER 6

---

### Indices and tables

---

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





**S**

`semantic_version`, [13](#)

`semantic_version.django_fields`, [22](#)



## Symbols

[\\_\\_cmp\\_\\_\(\)](#) (semantic\_version.Version method), 15  
[\\_\\_contains\\_\\_\(\)](#) (semantic\_version.Spec method), 19  
[\\_\\_hash\\_\\_\(\)](#) (semantic\_version.Spec method), 20  
[\\_\\_hash\\_\\_\(\)](#) (semantic\_version.SpecItem method), 21  
[\\_\\_hash\\_\\_\(\)](#) (semantic\_version.Version method), 16  
[\\_\\_iter\\_\\_\(\)](#) (semantic\_version.Spec method), 19  
[\\_\\_iter\\_\\_\(\)](#) (semantic\_version.Version method), 15  
[\\_\\_str\\_\\_\(\)](#) (semantic\_version.Spec method), 19  
[\\_\\_str\\_\\_\(\)](#) (semantic\_version.SpecItem method), 21  
[\\_\\_str\\_\\_\(\)](#) (semantic\_version.Version method), 16

## B

[build](#) (semantic\_version.Version attribute), 15

## C

[coerce](#) (semantic\_version.django\_fields.VersionField attribute), 22  
[coerce\(\)](#) (semantic\_version.Version class method), 16  
[compare\(\)](#) (in module semantic\_version), 13

## F

[filter\(\)](#) (semantic\_version.Spec method), 19

## K

[kind](#) (semantic\_version.SpecItem attribute), 20

## M

[major](#) (semantic\_version.Version attribute), 14  
[match\(\)](#) (in module semantic\_version), 13  
[match\(\)](#) (semantic\_version.Spec method), 19  
[match\(\)](#) (semantic\_version.SpecItem method), 21  
[minor](#) (semantic\_version.Version attribute), 14

## P

[parse\(\)](#) (semantic\_version.Spec class method), 20  
[parse\(\)](#) (semantic\_version.SpecItem class method), 21  
[parse\(\)](#) (semantic\_version.Version class method), 16

[partial](#) (semantic\_version.django\_fields.VersionField attribute), 22

[partial](#) (semantic\_version.Version attribute), 14

[patch](#) (semantic\_version.Version attribute), 14

[prerelease](#) (semantic\_version.Version attribute), 15

Python Enhancement Proposals

PEP 8, 11

## S

[select\(\)](#) (semantic\_version.Spec method), 19

[semantic\\_version](#) (module), 13

[semantic\\_version.django\\_fields](#) (module), 22

[Spec](#) (class in semantic\_version), 18

[spec](#) (semantic\_version.SpecItem attribute), 20

[SpecField](#) (class in semantic\_version.django\_fields), 22

[SpecItem](#) (class in semantic\_version), 20

[SpecItem.KIND\\_EQUAL](#) (in module semantic\_version), 21

[SpecItem.KIND\\_GT](#) (in module semantic\_version), 22

[SpecItem.KIND\\_GTE](#) (in module semantic\_version), 22

[SpecItem.KIND\\_LT](#) (in module semantic\_version), 21

[SpecItem.KIND\\_LTE](#) (in module semantic\_version), 21

[SpecItem.KIND\\_NEQ](#) (in module semantic\_version), 22

[specs](#) (semantic\_version.Spec attribute), 19

## V

[validate\(\)](#) (in module semantic\_version), 14

[Version](#) (class in semantic\_version), 14

[VersionField](#) (class in semantic\_version.django\_fields), 22