python-semanticversion Documentation

Release 0.0.0

Raphaël Barrois

May 26, 2022

Contents

CHAPTER 1

Introduction

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.

1.1 Links

- Package on PyPI: https://pypi.org/project/semantic-version/
- Doc on ReadTheDocs: https://python-semanticversion.readthedocs.io/
- Source on GitHub: http://github.com/rbarrois/python-semanticversion/
- Build on Github Actions: https://github.com/rbarrois/python-semanticversion/actions
- 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 classes to handle semantic versions:

- `Version` represents a version number (`0.1.1-alpha+build.2012-05-15`)
- `BaseSpec`-derived classes represent requirement specifications (`>=0.1.1,<0.3.0`):
  - `SimpleSpec` describes a natural description syntax
  - `NpmSpec` is used for NPM-style range descriptions.

2.1 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
```

(continues on next page)

(continued from previous page)

```
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 __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 parse
    raise ValueError('Invalid version string: %r' % version_string)
ValueError: Invalid version string: '0.1'
```

One may also create a `Version` with named components:

```
>>> semantic_version.Version(major=0, minor=1, patch=2)
Version('0.1.2')
```

In that case, `major`, `minor` and `patch` are mandatory, and must be integers. `prerelease` and `build`, if provided, must be tuples of strings:

```
>>> semantic_version.Version(major=0, minor=1, patch=2, prerelease=('alpha', '2'))
Version('0.1.2-alpha.2')
```

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')
```

2.1.1 Working with versions

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

2.2 Requirement specification

python-semanticversion provides a couple of ways to describe a range of accepted versions:

- The `SimpleSpec` class provides a simple, easily understood scheme – somewhat inspired from PyPI range notations;
- The `NpmSpec` class supports the whole NPM range specification scheme:

```
>>> Version('0.1.2') in NpmSpec('0.1.0-alpha.2 .. 0.2.4')
True
>>> Version('0.1.2') in NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
>>> Version('2.3.4') in NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
```

2.2.1 The `SimpleSpec` scheme

Basic usage is simply a comparator and a base version:

```
>>> s = SimpleSpec('>=0.1.1')  # At least 0.1.1
>>> s.match(Version('0.1.1'))
True
>>> s.match(Version('0.1.1-alpha1'))  # pre-release doesn't satisfy version spec
False
>>> s.match(Version('0.1.0'))
False
```

Combining specifications can be expressed as follows:

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

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

```
>>> s = SimpleSpec('==0.1.1')
>>> Version('0.1.1+git7ccc72') in s  # build variants are equivalent to full versions
True
>>> Version('0.1.1-alpha1') in s     # pre-release variants don't match the full␣
↪version.
False
```

(continues on next page)

(continued from previous page)

```
>>> Version('0.1.2') in s
False
```

Refer to the full documentation at https://python-semanticversion.readthedocs.io/en/latest/ for more details on the `SimpleSpec` scheme.

2.2.2 Using a specification

The `SimpleSpec.filter` method filters an iterable of `Version`:

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

CHAPTER 3

Contributing

In order to contribute to the source code:

- Open an issue on GitHub: 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
```

CHAPTER 4

Contents

4.1 Introduction

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.

4.1.1 Links

- Package on PyPI: https://pypi.org/project/semantic-version/
- Doc on ReadTheDocs: https://python-semanticversion.readthedocs.io/
- Source on GitHub: http://github.com/rbarrois/python-semanticversion/
- Build on Github Actions: https://github.com/rbarrois/python-semanticversion/actions
- Semantic Version specification: SemVer

4.2 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 classes to handle semantic versions:

- `Version` represents a version number (`0.1.1-alpha+build.2012-05-15`)
- `BaseSpec`-derived classes represent requirement specifications (`>=0.1.1,<0.3.0`):
  - `SimpleSpec` describes a natural description syntax
  - `NpmSpec` is used for NPM-style range descriptions.

4.2.1 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 __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 parse
    raise ValueError('Invalid version string: %r' % version_string)
ValueError: Invalid version string: '0.1'
```

One may also create a `Version` with named components:

```
>>> semantic_version.Version(major=0, minor=1, patch=2)
Version('0.1.2')
```

In that case, `major`, `minor` and `patch` are mandatory, and must be integers. `prerelease` and `build`, if provided, must be tuples of strings:

```
>>> semantic_version.Version(major=0, minor=1, patch=2, prerelease=('alpha', '2'))
Version('0.1.2-alpha.2')
```

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')
```

Working with versions

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

4.2.2 Requirement specification

python-semanticversion provides a couple of ways to describe a range of accepted versions:

- The `SimpleSpec` class provides a simple, easily understood scheme – somewhat inspired from PyPI range notations;
- The `NpmSpec` class supports the whole NPM range specification scheme:

```
>>> Version('0.1.2') in NpmSpec('0.1.0-alpha.2 .. 0.2.4')
True
>>> Version('0.1.2') in NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
>>> Version('2.3.4') in NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
```

The `SimpleSpec` scheme

Basic usage is simply a comparator and a base version:

```
>>> s = SimpleSpec('>=0.1.1')  # At least 0.1.1
>>> s.match(Version('0.1.1'))
True
>>> s.match(Version('0.1.1-alpha1'))  # pre-release doesn't satisfy version spec
False
>>> s.match(Version('0.1.0'))
False
```

Combining specifications can be expressed as follows:

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

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

```
>>> s = SimpleSpec('==0.1.1')
>>> Version('0.1.1+git7ccc72') in s  # build variants are equivalent to full versions
True
>>> Version('0.1.1-alpha1') in s     # pre-release variants don't match the full␣
↪version.
False
>>> Version('0.1.2') in s
False
```

Refer to the full documentation at https://python-semanticversion.readthedocs.io/en/latest/ for more details on the `SimpleSpec` scheme.

Using a specification

The `SimpleSpec.filter` method filters an iterable of `Version`:

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

4.3 Contributing

In order to contribute to the source code:

- Open an issue on GitHub: 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
```

4.4 Guide

This module covers the 2.0.0 version of the SemVer scheme, with additional extensions:

- Coercing any version string into a SemVer version, through *`Version.coerce()`*;
- Comparing versions;
- Computing next versions;
- Modelling version range specifcations, and choosing the best match – for both its custom logic, and NPM semantics (custom range specification schemes can be added).

4.4.1 Version basics

Building `Version` instances

The core of the module is the *`Version`* class; it is usually instantiated from a version string:

```
>>> import semantic_version as semver
>>> v = semver.Version("0.1.1")
```

The version's components are available through its attributes:

- *`major`*, *`minor`*, *`patch`* are integers:

```
>>> v.major
0
>>> v.minor
1
>>> v.patch
1
```

- The *`prerelease`* and *`build`* attributes are iterables of text elements:

```
>>> v2 = semver.Version("0.1.1-dev+23.git2")
>>> v2.prerelease
["dev"]
>>> v2.build
["23", "git2"]
```

One may also build a *Version* from named components directly:

```
>>> semantic_version.Version(major=0, minor=1, patch=2)
Version('0.1.2')
```

In that case, `major`, `minor` and `patch` are mandatory, and must be integers. `prerelease` and `build`, if provided, must be tuples of strings:

```
>>> semantic_version.Version(major=0, minor=1, patch=2, prerelease=('alpha', '2'))
Version('0.1.2-alpha.2')
```

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

```
>>> semver.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 __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 parse
    raise ValueError('Invalid version string: %r' % version_string)
ValueError: Invalid version string: '0.1'
```

Working with non-SemVer version strings

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:

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

Comparing versions

Versions can be compared, following the SemVer scheme:

```
>>> semver.Version("0.1.0") < semver.Version("0.1.1")
True
>>> max(
...   semver.Version("0.1.0"),
...   semver.Version("0.2.2"),
...   semver.Version("0.1.1"),
...   semver.Version("0.2.2-rc1"),
... )
Version("0.2.2")
```

Note: As defined in SemVer, build metadata is ignored in comparisons, but not in equalities:

```
>>> semver.Version("0.1.2") <= semver.Version("0.1.2+git2")
True
>>> semver.Version("0.1.2") >= semver.Version("0.1.2+git2")
True
>>> semver.Version("0.1.2") == semver.Version("0.1.2+git2")
False
```

Iterating versions

One can get a new version that represents a bump in one of the version levels through the *Version.next_major()*, *Version.next_minor()* or *Version.next_patch()* functions:

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

Note:

- If the version includes *build* or *prerelease* metadata, that value will be empty in the next version;
- The next patch following a version with a pre-release identifier is the same version with its prerelease and build identifiers removed: `Version("0.1.1-rc1").next_patch() == Version("0.1.1")`
- Pre-release and build naming schemes are often custom and specific to a project's internal design; thus, the library can't provide a `next_xxx` method for those fields.

One may also truncate versions through the `Version.truncate()` method, removing components beyond the selected level:

```
>>> v = semver.Version("0.1.2-dev+git3")
>>> v.truncate("prerelease")
Version("0.1.2-dev")
>>> v.truncate("minor")
Version("0.1.0")
```

4.4.2 Range specifications

Comparing version numbers isn't always enough; in many situations, one needs to define a *range of acceptable versions*.

That notion is not defined in SemVer; moreover, several systems exists, with their own notations.

The `semantic_version` package provides a couple of implementations for these notions:

- *`SimpleSpec`* is a simple implementation, with reasonable expectations;
- *`NpmSpec`* sticks to the NPM specification.

Further schemes can be built in a similar manner, relying on the *`BaseSpec`* class for basics.

Core API

The core API is provided by the *`BaseSpec`* class.

Note: These examples use *`SimpleSpec`* in order to be easily reproduced by users, but only exhibit the standard parts of the interface.

It is possible to check whether a given *`Version`* matches a *`BaseSpec`* through *`match()`*:

```
>>> s = semver.SimpleSpec(">=0.1.1")
>>> s.match(Version("0.1.1"))
True
>>> s.match(Version("0.1.0"))
False
```

This feature is also available through the `in` keyword:

```
>>> s = semver.SimpleSpec(">=0.1.1")
>>> Version("0.1.1") in s
True
>>> Version("0.1.0") in s
False
```

A specification can filter compatible values from an iterable of versions with *`filter()`*:

```
>>> s = semver.SimpleSpec(">=0.2.1")
>>> versions = [
...   Version("0.1.0"),
...   Version("0.2.0"),
...   Version("0.3.0"),
...   Version("0.4.0"),
... ]
>>> list(s.filter(versions))
[Version("0.3.0"), Version("0.4.0")]
```

It can also select the "best" version from such an iterable through *`select()`*:

```
>>> s = semver.SimpleSpec(">=0.2.1")
>>> versions = [
...   Version("0.1.0"),
...   Version("0.2.0"),
...   Version("0.3.0"),
...   Version("0.4.0"),
... ]
>>> s.select(versions)
Version("0.4.0")
```

The `SimpleSpec` scheme

The *`SimpleSpec`* provides a hopefully intuitive version range specification scheme:

- A specification expression is composed of comma-separated clauses;
- Each clause can be:
  - An equality match (== or `!=`);
  - A comparison (>, >=, < , <=);
  - A compatible release clause, PyPI style (`~=2.2` for `>=2.2.0,<3.0.0`);
  - An NPM style clause:
    - `~1.2.3` for `>=1.2.3,<1.3.0`;
    - `^1.3.4` for `>=1.3.4,<2.0.0`;
- The range in each clause may include a wildcard:
  - `==0.1.*` maps to `>=0.1.0,<0.2.0`;
  - `==1.*` or `==1.*.*` map to `>=1.0.0,<2.0.0`

Special matching rules

When testing a *`Version`* against a *`SimpleSpec`*, comparisons are adjusted for common user expectations; thus, a pre-release version (`1.0.0-alpha`) will not satisfy the `==1.0.0` *`SimpleSpec`*.

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

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

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

```
>>> Version('1.0.0+build2') in SimpleSpec('<=1.0.0')   # Build metadata ignored
True
>>> Version('1.0.0+build1') in SimpleSpec('==1.0.0+build2')  # Include build in checks
False
```

Note: The full documentation is provided in the reference section for the *`SimpleSpec`* class.

The `NpmSpec` scheme

The *`NpmSpec`* class implements the full NPM specification (from https://github.com/npm/node-semver#ranges):

```
>>> semver.Version("0.1.2") in semver.NpmSpec("0.1.0-alpha.2 .. 0.2.4")
True
>>> semver.Version('0.1.2') in semver.NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
>>> semver.Version('2.3.4') in semver.NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
```

4.4.3 Using with Django

The *semantic_version.django_fields* module provides django fields to store *Version* or *BaseSpec* objects.

More documentation is available in the *Interaction with Django* section.

4.5 Reference

4.5.1 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

4.5.2 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'
```

class semantic_version.**Version**(*major: int*, *minor: int*, *patch: int*, *prereleases: tuple*, *build: tuple*[, *partial=False*])

Constructed from named components:

```
>>> Version(major=1, minor=2, patch=3)
Version('1.2.3')
```

Attributes

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.

precedence_key
Read-only attribute; suited for use in `sort(versions, key=lambda v: v.precedence_key)`. The actual value of the attribute is considered an implementation detail; the only guarantee is that ordering versions by their precedence_key will comply with semver precedence rules.

> **Warning:** Changed in version 2.10.0.
>
> The *build* is included in the precedence_key computation, but only for ordering stability. The only guarantee is that, for a given release of python-semanticversion, two versions' *precedence_key* will always compare in the same direction if they include build metadata; that ordering is an implementation detail and shouldn't be relied upon.

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

Deprecated since version 2.7: The ability to define a partial version will be removed in version 3.0. Use *SimpleSpec* instead: `SimpleSpec('1.x.x')`.

Methods

next_major(*self*)
Return the next major version, i.e the smallest version strictly greater than the current one with minor and patch set to 0 and no prerelease/build.

```
>>> Version('1.0.2').next_major()
Version('2.0.0')
>>> Version('1.0.0+b3').next_major()
Version('2.0.0')
>>> Version('1.0.0-alpha').next_major()
Version('1.0.0')
```

next_minor(*self*)
Return the next minor version, i.e the smallest version strictly greater than the current one, with a patch level of `0`.

```
>>> Version('1.0.2').next_minor()
Version('1.1.0')
>>> Version('1.0.0+b3').next_minor()
Version('1.1.0')
>>> Version('1.1.2-alpha').next_minor()
Version('1.2.0')
>>> Version('1.1.0-alpha').next_minor()
Version('1.1.0')
```

next_patch(*self*)
Return the next patch version, i.e the smallest version strictly greater than the current one with empty *prerelease* and *build*.

```
>>> Version('1.0.2').next_patch()
Version('1.0.3')
>>> Version('1.0.2+b3').next_patch()
Version('1.0.3')
>>> Version('1.0.2-alpha').next_patch()
Version('1.0.2')
```

Warning: The next patch version of a version with a non-empty *prerelease* is the version without that *prerelease* component: it's the smallest "pure" patch version strictly greater than that version.

truncate(self, level='patch'):
Returns a similar level, but truncated at the provided level.

```
>>> Version('1.0.2-rc1+b43.24').truncate()
Version('1.0.2')
>>> Version('1.0.2-rc1+b43.24').truncate('minor')
Version('1.0.0')
>>> Version('1.0.2-rc1+b43.24').truncate('prerelease')
Version('1.0.2-rc1')
```

__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*

4.5.3 Version specifications (the Spec class)

The SemVer specification doesn't provide a standard description of version ranges. And simply using a naive implementation leads to unexpected situations: `>=1.2.0,<1.3.0` isn't expected to match version `1.3.0-rc.1`, yet a strict application of SemVer precedence rules would include it.

In order to solve this problem, each SemVer-based package management platform has designed its own rules. python-semanticversion provides a couple of implementations of those range definition syntaxes:

- `'simple'` (through *SimpleSpec*): A python-semanticversion specific syntax, which supports simple / intuitive patterns, and some NPM-inspired extensions;
- `'npm'` (through *NpmSpec*): The NPM syntax, based on https://github.com/npm/node-semver#ranges
- More might be added in the future.

Each of those `Spec` classes provides a shared set of methods to work with versions:

class semantic_version.**BaseSpec**(*spec_string*)

Converts an expression describing a range of versions into a set of clauses, and matches any *Version* against those clauses.

Attributes

This class has no public attributes.

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'
```

__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*, *expression*, *syntax='simple'*)
Retrieve a *BaseSpec* object tuple from a string.

Parameters

- **requirement_string** (*str*) – The textual description of the specifications
- **syntax** (*str*) – The identifier of the syntax to use for parsing

Raises ValueError: if the requirement_string is invalid.

Return type *BaseSpec* subclass

Changed in version 2.7: This method used to return a tuple of *SpecItem* objects.

class semantic_version.**SimpleSpec**(*spec_string*)
New in version 2.7: Previously reachable through *Spec*.

Applies the python-semanticversion range specification:

- A specification of `<1.3.4` is not expected to allow `1.3.4-rc2`, but strict SemVer comparisons allow it ;
- 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`).

Specification structure:

In order to have version specification behave naturally, the *`SimpleSpec`* syntax uses the following rules:

- A specification expression is a list of clauses separated by a comma (`,`);
- A version is matched by an expression if, and only if, it matches every clause in the expression;
- A clause of `*` matches every valid version;

Equality clauses

- A clause of `==0.1.2` will match version `0.1.2` and any version differing only through its build number (`0.1.2+b42` matches);
- A clause of `==0.1.2+b42` will only match that specific version: `0.1.2+b43` and `0.1.2` are excluded;
- A clause of `==0.1.2+` will only match that specific version: `0.1.2+b42` is excluded;
- A clause of `!=0.1.2` will prevent all versions with the same major/minor/patch combination: `0.1.2-rc.1` and `0.1.2+b42` are excluded'
- A clause of `!=0.1.2-` will only prevent build variations of that version: `0.1.2-rc.1` is included, but not `0.1.2+b42`;
- A clause of `!=0.1.2+` will exclude only that exact version: `0.1.2-rc.1` and `0.1.2+b42` are included;
- Only a `==` or `!=` clause may contain build-level metadata: `==1.2.3+b42` is valid, `>=1.2.3+b42` isn't.

Comparison clauses

- A clause of `<0.1.2` will match versions strictly below `0.1.2`, excluding prereleases of `0.1.2`: `0.1.2-rc.1` is excluded;
- A clause of `<0.1.2-` will match versions strictly below `0.1.2`, including prereleases of `0.1.2`: `0.1.2-rc.1` is included;
- A clause of `<0.1.2-rc.3` will match versions strictly below `0.1.2-rc.3`, including prereleases: `0.1.2-rc.2` is included;
- A clause of `<=XXX` will match versions that match `<XXX` or `==XXX`
- A clause of `>0.1.2` will match versions strictly above `0.1.2`, including all prereleases of `0.1.3`.
- A clause of `>0.1.2-rc.3` will match versions strictly above `0.1.2-rc.3`, including matching prereleases of `0.1.2`: `0.1.2-rc.10` is included;
- A clause of `>=XXX` will match versions that match `>XXX` or `==XXX`

Wildcards

- A clause of `==0.1.*` is equivalent to `>=0.1.0,<0.2.0`
- A clause of `>=0.1.*` is equivalent to `>=0.1.0`
- A clause of `==1.*` or `==1.*.*` is equivalent to `>=1.0.0,<2.0.0`
- A clause of `>=1.*` or `>=1.*.*` is equivalent to `>=1.0.0`
- A clause of `==*` maps to `>=0.0.0`
- A clause of `>=*` maps to `>=0.0.0`

Extensions

Additionnally, python-semanticversion supports extensions from specific packaging platforms:

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"

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"

Some examples:

```
>>> Version('0.1.2-rc.1') in SimpleSpec('*')
True
>>> SimpleSpec('<0.1.2').filter([Version('0.1.2-rc.1'), Version('0.1.1'), Version(
↪'0.1.2+b42')])
[Version('0.1.1')]
>>> SimpleSpec('<0.1.2-').filter([Version('0.1.2-rc.1'), Version('0.1.1'),␣
↪Version('0.1.2+b42')])
[Version('0.1.2-rc.1'), Version('0.1.1')]
>>> SimpleSpec('>=0.1.2,!=0.1.3,!=0.1.4-rc.1',!=0.1.5+b42).filter([
        Version('0.1.2'), Version('0.1.3'), Version('0.1.3-beta'),
        Version('0.1.4'), Version('0.1.5'), Version('0.1.5+b42'),
        Version('2.0.1-rc.1'),
    ])
[Version('0.1.2'), Version('0.1.4'), Version('0.1.5'), Version('2.0.1-rc.1')]
```

class semantic_version.**NpmSpec**(*spec_string*)

New in version 2.7.

A NPM-compliant version matching engine, based on the https://github.com/npm/node-semver#ranges specification.

```
>>> Version('0.1.2') in NpmSpec('0.1.0-alpha.2 .. 0.2.4')
True
>>> Version('0.1.2') in NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
>>> Version('2.3.4') in NpmSpec('>=0.1.1 <0.1.3 || 2.x')
True
```

class semantic_version.**Spec**(*spec_string*)

Deprecated since version 2.7: The alias from *Spec* to *SimpleSpec* will be removed in 3.1.

Alias to *LegacySpec*, for backwards compatibility.

class semantic_version.**LegacySpec**(*spec_string*)

Deprecated since version 2.7: The *LegacySpec* class will be removed in 3.0; use *SimpleSpec* instead.

A *LegacySpec* class has the exact same behaviour as *SimpleSpec*, with backwards-compatible features:

It accepts version specifications passed in as 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)>,
)>
```

It keeps a list of *SpecItem* objects, based on the initial expression components.

__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
```

Attributes

specs

Tuple of *SpecItem*, the included specifications.

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

Deprecated since version 2.7: This class will be removed in 3.0.

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
```

(continues on next page)

(continued from previous page)

```
>>> # 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 alway 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
```

KIND_COMPATIBLE
The kind of compatible release clauses specifications:

```
>>> Version('1.1.2') in Spec('~=1.1.0')
True
```

4.6 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.BaseSpec`* 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
Deprecated since version 2.7: Support for partial versions will be removed in 3.0.

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.BaseSpec* as its textual representation.

syntax
The syntax to use for the field; defaults to `'simple'`.

New in version 2.7.

4.7 ChangeLog

4.7.1 2.10.0 (2022-05-26)

New:

- 132: Ensure sorting a collection of versions is always stable, even with build metadata.

4.7.2 2.9.0 (2022-02-06)

New:

- Add support for Django 3.1, 3.2, 4.0
- Add support for Python 3.7 / 3.8 / 3.9 / 3.10

4.7.3 2.8.5 (2020-04-29)

Bugfix:

- 98: Properly handle wildcards in `SimpleSpec` (e.g. `==1.2.*`).

4.7.4 2.8.4 (2019-12-21)

Bugfix:

- #89: Properly coerce versions with leading zeroes in components (e.g. `1.01.007`)

4.7.5 2.8.3 (2019-11-21)

New:

- Add *Clause.prettyprint()* for debugging

Bugfix:

- #86: Fix handling of prerelease ranges within *NpmSpec*

4.7.6 2.8.2 (2019-09-06)

Bugfix:

- #82: Restore computation of `Spec.specs` for single-term expressions (`>=0.1.2`)

4.7.7 2.8.1 (2019-08-29)

Bugfix:

- Restored attribute `Spec.specs`, removed by mistake during the refactor.

4.7.8 2.8.0 (2019-08-29)

New:

- Restore support for Python 2.

4.7.9 2.7.1 (2019-08-28)

Bugfix:

- Fix parsing of npm-based caret expressions.

4.7.10 2.7.0 (2019-08-28)

This release brings a couple of significant changes:

- Allow to define several range description syntaxes (`SimpleSpec`, `NpmSpec`, ...)
- Fix bugs and unexpected behaviours in the `SimpleSpec` implementation.

Backwards compatibility has been kept, but users should adjust their code for the new features:

- Use `SimpleSpec` instead of `Spec`
- Replace calls to `Version('1.2', partial=True)` with `SimpleSpec('~1.2')`
- `iter(some_spec)` is deprecated.

New:

- Allow creation of a `Version` directly from parsed components, as keyword arguments (`Version(major=1, minor=2, patch=3)`)
- Add `Version.truncate()` to build a truncated copy of a `Version`
- Add `NpmSpec(...)`, following strict NPM matching rules (https://github.com/npm/node-semver#ranges)
- Add `Spec.parse('xxx', syntax='<syntax>')` for simpler multi-syntax support
- Add `Version().precedence_key`, for use in `sort(versions, key=lambda v: v.precedence_key)` calls. The contents of this attribute is an implementation detail.

Bugfix:

- Fix inconsistent behaviour regarding versions with a prerelease specification.

Deprecated:

- Deprecate the `Spec` class (Removed in 3.1); use the `SimpleSpec` class instead
- Deprecate the internal `SpecItem` class (Removed in 3.0).
- Deprecate the `partial=True` form of `Version`; use `SimpleSpec` instead.

Removed:

- Remove support for Python2 (End of life 4 months after this release)

Refactor:

- Switch spec computation to a two-step process: convert the spec to a combination of simple comparisons with clear semantics, then use those.

4.7.11 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

4.7.12 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

4.7.13 2.4.2 (2015-07-02)

Bugfix:

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

4.7.14 2.4.1 (2015-04-01)

Bugfix:

- Fix packaging metadata (advertise Python 3.4 support)

4.7.15 2.4.0 (2015-04-01)

New:

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

4.7.16 2.3.1 (2014-09-24)

Bugfix:

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

4.7.17 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

4.7.18 2.2.2 (2013-12-23)

Bugfix:

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

4.7.19 2.2.1 (2013-10-29)

Bugfix:

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

4.7.20 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

4.7.21 2.1.2 (2012-05-22)

Bugfix:

- Properly validate *VersionField* and *SpecField*.

4.7.22 2.1.1 (2012-05-22)

New:

- Add introspection rules for south

4.7.23 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__()`

4.7.24 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`

4.7.25 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
```

4.7.26 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`.

4.7.27 1.0.0 (2012-05-17)

First public release.

New:

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

4.8 Credits

4.8.1 Maintainers

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

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

4.8.2 Contributors

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

- Kyle Baird <kylegbaird@gmail.com> (https://github.com/kgbaird)
- 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)

4.8.3 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 end 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 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

B

C

F

K

L

M

N

P

S

V