python-semanticversion Documentation

Release 1.0.0

Raphaël Barrois

November 07, 2015

Contents

This small python library provides a few tools to handle SemVer in Python.

The first release (1.0.0) should handle the 2.0.0-rc1 version of the SemVer scheme.

CHAPTER 1

Getting started

Intall the package from PyPI, using pip:

```
pip install python-semanticversion
```

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 (`>=0.1.1`)

1.1 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'
```

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
```

1.2 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-alpha1'))
False
```

It is also possible to define ‘approximate’ version specifications:

```
>>> s = Spec('~0.1')  # Matches 0.1.*
>>> s.match(Version('0.1.0-alpha1'))
True
>>> s.match(Version('0.1.9999999999+build99'))
True
>>> s.match(Version('0.2.0'))
False
```

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

```
>>> s = Spec('~0.1.1')
>>> Version('0.1.1-alpha1') in s
True
>>> Version('0.1.2') in s
False
```

CHAPTER 2

Contents

2.1 Reference

2.1.1 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

`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`

2.1.2 Representing a version

class `semantic_version.`**`Version`**

Object representation of a SemVer-compliant version.

Constructed from a textual version string:

```
>>> Version('1.1.1')
<SemVer(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`

`list` 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`

`list` of `strings`, the build component.

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

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`*).

`__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`*, compare using the SemVer scheme, but stop at the first component undefined in the *`partial Version`*; that is, a component whose value is `None`.

`__str__`(*self*)
Returns the standard text representation of the version.

```
>>> v = Version('0.1.1-rc2+build4.4')
>>> v
<SemVer(0, 1, 1, ['rc2'], ['build4', '4'])>
>>> str(v)
'0.1.1-rc2+build4.4'
```

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)

2.1.3 Version specifications

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

class `semantic_version.`**`Spec`**
Stores a version specification, defined from a string:

```
>>> Spec('>=0.1.1')
<Spec: >= <SemVer(0, 1, 1, [], [])>>
```

This allows to test *`Version`* objects against the *`Spec`*:

```
>>> Spec('>=0.1.1').match(Version('0.1.1-rc1'))  # pre-release have lower precedence
False
>>> Version('0.1.1+build2') in Spec('>=0.1.1')    # build version have higher precedence
True
```

Attributes

`kind`
One of *`KIND_LT`*, *`KIND_LTE`*, *`KIND_EQUAL`*, *`KIND_GTE`*, *`KIND_GT`*, *`KIND_ALMOST`*.

`spec`
`Version` in the *`Spec`* description.

If *`kind`* is *`KIND_ALMOST`*, this will be 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 *`Spec`*.

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

Return type `bool`

`__contains__`(*self*, *version*)
Allows the use of the `version in spec` syntax. Simply an alias of the *`match()`* method.

Class attributes

`KIND_LT`
The kind of 'Less than' specifications

`KIND_LTE`
The kind of 'Less or equal to' specifications

`KIND_EQUAL`
The kind of 'equal to' specifications

`KIND_GTE`
The kind of 'Greater or equal to' specifications

`KIND_GT`
The kind of 'Greater than' specifications

`KIND_ALMOST`
The kind of 'Almost equal to' specifications

2.2 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

class `semantic_version.django_fields.`**`VersionField`**
Stores a *`semantic_version.Version`*.

`partial`
Boolean; whether *`partial`* versions are allowed.

class `semantic_version.django_fields.`**`SpecField`**
Stores a *`semantic_version.Spec`*.

CHAPTER 3

Links

- Package on PyPI: http://pypi.python.org/semantic_version/
- Doc on ReadTheDocs: http://readthedocs.org/docs/python-semanticversion/
- 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 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index