versions Documentation

Release 0.10.0

Philippe Muller

May 12, 2014

Contents

Quick examples:

- Compare versions:

```
>>> from versions import Version
>>> Version.parse('2.0.0') > Version.parse('1.0.0')
True
```

- Test if constraints are satisfied by a version:

```
>>> from versions import Constraint, Constraints
>>> '2.0' in Constraint.parse('>1')
True
>>> '1.5' in Constraints.parse('>1,<2')
True
```

CHAPTER 1

Contents

1.1 Quickstart

1.1.1 Basic usage

Version comparison examples:

```
>>> from versions import Version
>>> v1 = Version.parse('1')
>>> v2 = Version.parse('2')
>>> v1 == v2
False
>>> v1 != v2
True
>>> v1 > v2
False
>>> v1 < v2
True
>>> v1 >= v2
False
>>> v1 <= v2
True
```

`Version.parse()` expects a *version expression* string and returns a corresponding `Version` object:

```
>>> from versions import Version
>>> v = Version.parse('1.2.0-dev+foo.bar')
>>> v.major, v.minor, v.patch, v.prerelease, v.build_metadata
(1, 2, 0, 'dev', set(['foo', 'bar']))
```

If it isn't a semantic version string, the parser tries to normalize it:

```
>>> v = Version.parse('1')
>>> v.major, v.minor, v.patch, v.prerelease, v.build_metadata
(1, 0, 0, None, None)
```

1.1.2 Version constraint matching

versions also implements version constraint parsing and evaluation:

```
>>> from versions import Constraint
>>> Constraint.parse('>1').match('2')
True
>>> Constraint.parse('<2').match(Version.parse('1'))
True
```

For conveniance, constraint matching can be tested using the `in` operator:

```
>>> '1.5' in Constraint.parse('<2')
True
>>> Version(2) in Constraint.parse('!=2')
False
```

Constraints can be merged using `Constraints`:

```
>>> from versions import Constraints
>>> '1.0' in Constraints.parse('>1,<2')
False
>>> '1.5' in Constraints.parse('>1,<2')
True
>>> '2.0' in Constraints.parse('>1,<2')
False
```

1.2 API

Modules:

1.2.1 version

Version expressions

Version expressions are strings representing a software version. They are defined by this EBNF grammar:

```
version_expression  ::=  main | main `-` prerelease | main `+` build_metadata | main `-` p
main                ::=  major (`.` minor (`.` patch)?)? postrelease?
major               ::=  number
minor               ::=  number
patch               ::=  number
postrelease         ::=  string
prerelease          ::=  string | number
build_metadata      ::=  string
number              ::=  [0-9]+
string              ::=  [0-9a-zA-Z.-]+
```

They can be parsed into `Version` objects using the `Version.parse()` class method.

Omitted parts in an expression use these defaults:

Part	Default value
minor	0
patch	0
postrelease	None
prerelease	None
build_metadata	None

Examples of valid version expressions:

```
>>> from versions import Version
>>> v = Version.parse('1')
>>> v == '1.0'
True
>>> v == '1.0.0'
True
>>> v.major, v.minor, v.patch, v.prerelease, v.build_metadata
(1, 0, 0, None, None)

>>> v = Version.parse('1.2.3-dev+foo')
>>> v.major, v.minor, v.patch, v.prerelease, v.build_metadata
(1, 2, 3, 'dev', 'foo')
```

When parsing fails, an `InvalidVersionExpression` exception is raised:

```
>>> Version.parse('#@!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "versions/version.py", line 119, in parse
    raise InvalidVersionExpression(version_string)
versions.version.InvalidVersionExpression: Invalid version expression: '#@!'
```

Version

class `versions.version.`**`Version`**(*major*, *minor=0*, *patch=0*, *postrelease=None*, *prerelease=None*, *build_metadata=None*)

A package version.

Parameters

- **major** (*int*) – Version major number
- **minor** (*int*) – Version minor number
- **patch** (*int*) – Version patch number
- **postrelease** (`str`, `int` or `None`) – Version postrelease identifier
- **prerelease** (`str`, `int` or `None`) – Version prerelease identifier
- **build_metadata** (`None` or `str`) – Version build metadata

This class constructor is usually not called directly. For version string parsing, see `Version.parse`.

`major` = None
Version major number

`minor` = None
Version minor number

`prerelease` = None
Version prerelease

build_metadata = None
Version build metadata

classmethod parse (*version_string*)
Parses a `version_string` and returns a `Version` object.

Comparison

`Version` objects are comparable with standard operators:

```
>>> from versions import Version
>>> v1 = Version(1)
>>> v2 = Version(2)
>>> v1 == v2
False
>>> v1 != v2
True
>>> v1 > v2
False
>>> v1 < v2
True
>>> v1 >= v2
False
>>> v1 <= v2
True
```

Hint: When comparing 2 versions, only the version and the pre-release are used. The build metadata are ignored.

Errors

exception `versions.version.`**InvalidVersionExpression** (*version_expression*)
Raised when failing to parse a *version expression*.

version_expression = None
The bogus version expression.

1.2.2 constraint

Constraint expressions

Constraint expressions are strings representing a constraint on a software version. They are defined by this EBNF grammar:

```
constraint_expression  ::=  operator version_expression
operator               ::=  '==' | '!=' | '>' | '<' | '<=' | '>='
version_expression     ::=  main | main '-' prerelease
main                   ::=  major ('.'  minor ('.'  patch)?)?
major                  ::=  number
minor                  ::=  number
patch                  ::=  number
prerelease             ::=  string | number
build_metadata         ::=  string
```

```
number                  ::=    [0-9]+
string                  ::=    [0-9a-zA-Z.-]+
```

They can be parsed into `Constraint` objects using the `Constraint.parse()` class method.

Examples of valid constraint expressions:

```
>>> from versions import Constraint
>>> c = Constraint.parse('==1.0')
>>> c.operator, c.version
(Operator.parse('=='), Version.parse('1.0.0'))
>>> c = Constraint.parse('>=1.2.3-dev+foo')
>>> c.operator, c.version
(Operator.parse('>='), Version.parse('1.2.3-dev+foo'))
```

When parsing fails, an `InvalidConstraintExpression` exception is raised:

```
>>> Constraint.parse('WTF')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "versions/constraint.py", line 94, in parse
    raise InvalidConstraintExpression(constraint_expression)
versions.constraint.InvalidConstraintExpression: Invalid constraint expression: 'WTF'
```

Constraint

class `versions.constraint.`**`Constraint`**(*operator*, *version*)

A constraint on a package version.

Parameters

- **operator** (`Operator`) – The constraint operator.
- **version** (`Version`) – The constraint version.

`operator` = None

The constraint `Operator`.

`version` = None

The constraint `Version`.

`match`(*version*)

Match `version` with the constraint.

Parameters **version** (*version expression* or `Version`) – Version to match against the constraint.

Return type `True` if `version` satisfies the constraint, `False` if it doesn't.

classmethod `parse`(*constraint_expression*)

Parses a *constraint_expression* and returns a `Constraint` object.

Parameters **constraint_expression** (*str*) – A *constraint expression*

Raises `InvalidConstraintExpression` when parsing fails.

Return type `Constraint`

Matching

The `Constraint.match()` method returns `True` when passed a satisfying *version expression* or `Version` object:

```
>>> from versions import Constraint, Version
>>> Constraint.parse('>1').match('2')
True
>>> Constraint.parse('<2').match(Version(1))
True
```

Matching can also be tested using the `in` operator:

```
>>> '1.5' in Constraint.parse('== 1.0')
False
>>> Version(1, 5) in Constraint.parse('> 1.0')
True
>>> Version(1) in Constraint.parse('>= 2.0.0')
False
```

Errors

exception `versions.constraint.`**`InvalidConstraintExpression`**(*constraint_expression*)

Raised when failing to parse a `constraint_expression`.

`constraint_expression` = None

The bogus constraint expression.

1.2.3 constraints

Constraints expressions

Constraints expressions are strings representing multiple constraints on a software version. They are defined by this EBNF grammar:

```
constraints_expression  ::=  constraint_expression (',' constraint_expression)*
constraint_expression   ::=  operator version_expression
operator                ::=  '==' | '!=' | '>' | '<' | '<=' | '>='
version_expression      ::=  main | main '-' prerelease
main                    ::=  major ('.'  minor ('.'  patch)?)?
major                   ::=  number
minor                   ::=  number
patch                   ::=  number
prerelease              ::=  string | number
build_metadata          ::=  string
number                  ::=  [0-9]+
string                  ::=  [0-9a-zA-Z.-]+
```

They can be parsed into `Constraints` objects using the `Constraints.parse()` class method.

Examples of valid constraint expressions:

```
>>> from versions import Constraints

>>> c = Constraints.parse('==1.0')
>>> c.constraints
[Constraint.parse('==1.0.0')]

>>> c = Constraints.parse('>=1.0,<2,!=1.5')
```

```
>>> c.constraints
[Constraint.parse('!=1.5.0'), Constraint.parse('>=1.0.0'), Constraint.parse('<2.0.0')]
```

Constraints

class versions.constraints.**Constraints**(*constraints=None*)

A collection of Constraint objects.

constraints = None

List of Constraint.

match(*version*)

Match version with this collection of constraints.

Parameters **version** (*version expression* or Version) – Version to match against the constraint.

Return type True if version satisfies the constraint, False if it doesn't.

classmethod parse(*constraints_expression*)

Parses a *constraints_expression* and returns a Constraints object.

Merging

Constraint objects can be merged using a Constraints object and the + operator:

```
>>> from versions import Constraints, Constraint
>>> Constraints() + Constraint.parse('<2') + Constraint.parse('!=1.5')
Constraints.parse('<2.0.0,!=1.5.0')
```

Note: The Constraints object must be on the left side of the + operator. The Constraint object must be on right side.

If the constraint is a *constraints expression*, it is automatically parsed into a Constraints object.

The previous example can therefore be shortened as:

```
>>> Constraints() + '<2' + '!=1.5'
Constraints.parse('!=1.5.0,<2.0.0')
```

Or:

```
>>> Constraints() + '<2,!=1.5'
Constraints.parse('!=1.5.0,<2.0.0')
```

Matching

Constraints objects work like Constraint objects: they have a Constraints.match() method which returns True when passed a *version expression* or Version matching all constraints:

```
>>> Constraints.parse('>=1,<2').match('1.4')
True
>>> '1.4' in Constraints.parse('>=1.2,<2,!=1.4')
False
```

Conflicts

When merging conflicting constraints, an `ExclusiveConstraints` exception is raised:

```
>>> Constraints.parse('<1') + '>1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "versions/constraints.py", line 96, in __add__
    return Constraints(self._merge(constraint))
  File "versions/constraints.py", line 89, in _merge
    return merge(self.constraints + constraints)
  File "versions/constraints.py", line 203, in merge
    raise ExclusiveConstraints(g_constraint, [l_constraint])
versions.constraints.ExclusiveConstraints: Constraint >1.0.0 conflicts with constraints <1.0.0

>>> Constraints.parse('<1') + '==1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "versions/constraints.py", line 96, in __add__
    return Constraints(self._merge(constraint))
  File "versions/constraints.py", line 89, in _merge
    return merge(self.constraints + constraints)
  File "versions/constraints.py", line 217, in merge
    raise ExclusiveConstraints(eq_constraint, conflict_list)
versions.constraints.ExclusiveConstraints: Constraint ==1.0.0 conflicts with constraints <1.0.0

>>> Constraints.parse('>=1') + '!=1' + '<=1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "versions/constraints.py", line 96, in __add__
    return Constraints(self._merge(constraint))
  File "versions/constraints.py", line 89, in _merge
    return merge(self.constraints + constraints)
  File "versions/constraints.py", line 217, in merge
    raise ExclusiveConstraints(eq_constraint, conflict_list)
versions.constraints.ExclusiveConstraints: Constraint ==1.0.0 conflicts with constraints !=1.0.0
```

exception `versions.constraints.`**`ExclusiveConstraints`**(*constraint*, *constraints*)

Raised when cannot merge a new constraint with pre-existing constraints.

`constraint` = None

The conflicting constraint.

`constraints` = None

The constraints with which it conflicts.

1.2.4 requirements

Requirement expressions

Requirement expressions are strings representing a required software package. They are defined by this EBNF grammar:

```
requirement_expression  ::=  name build_options?  constraints_expression?
name                    ::=  [A-Za-z0-9][-A-Za-z0-9]*
build_options           ::=  '[' name (',' name)* ']'
```

```
constraints_expression  ::=  constraint_expression (',' constraint_expression)*
constraint_expression   ::=  operator version_expression
operator                ::=  '==' | '!=' | '>' | '<' | '<=' | '>='
version_expression      ::=  main | main '-' prerelease | main '+' build_metadata | main
main                    ::=  major ('.'  minor ('.'  patch)?)?
major                   ::=  number
minor                   ::=  number
patch                   ::=  number
prerelease              ::=  string | number
build_metadata          ::=  string
number                  ::=  [0-9]+
string                  ::=  [0-9a-zA-Z.-]+
```

The `Requirement.parse()` class method parses requirement expressions into `Requirement` objects:

```
>>> from versions import Requirement
>>> r = Requirement.parse('foo')
>>> r.name
'foo'
>>> r = Requirement.parse('foo >1.0, <2.0')
>>> r.version_constraints
Constraints.parse('>1.0.0,<2.0.0')
>>> r = Requirement.parse('vim [python, perl] >7')
>>> r.build_options
set(['python', 'perl'])
```

Matching

The `Requirement.match()` method returns `True` when passed a package which satisfies the requirement:

```
>>> from versions import Requirement, Package, Version
>>> Requirement('foo').match(Package('foo', Version.parse('1.0')))
True
```

If passed a `str`, it is automatically parsed using `Package.parse()`:

```
>>> Requirement.parse('foo [baz, bar] >0.9').match('foo-1.0+bar.baz')
True
```

Matching can also be tested using the `in` operator:

```
>>> 'foo-0.2' in Requirement.parse('foo [bar] >0.9')
False
```

Requirement

class `versions.requirements.`**`Requirement`**(*name*, *version_constraints=None*, *build_options=None*)

Package requirements are used to define a dependency from a `Package` to another.

Parameters

- **name** (*str*) – The required package name.
- **version_constraints** (`Version` or `None`) – Constraints on the package version.
- **build_options** (`set` of `str` or `None`) – Required build options.

name = None
Name of the required package.

version_constraints = None
`Constraints` on the required package version.

build_options = None
set of required build options

match (*package*)
Match `package` with the requirement.

Parameters **package** (package expression string or `Package`) – Package to test with the requirement.

Returns `True` if `package` satisfies the requirement.

Return type bool

classmethod parse (*requirement_expression*)
Parses a *requirement_expression* into a `Requirement` object.

Parameters **requirement_expression** (*requirement_expression*) – A package requirement expression.

Return type `Requirement`

1.2.5 packages

Package expressions

Package expressions are strings representing a software package. They are defined by this EBNF grammar:

```
package_expression      ::=  name '-' version_expression dependency*
name                    ::=  [A-Za-z0-9][-A-Za-z0-9]*
dependency              ::=  ';' 'depends' requirement_expression
requirement_expression  ::=  name build_options?  constraints_expression?
build_options           ::=  '[' name (',' name)* ']'
constraints_expression  ::=  constraint_expression (',' constraint_expression)*
constraint_expression   ::=  operator version_expression
operator                ::=  '==' | '!=' | '>' | '<' | '<=' | '>='
version_expression      ::=  main | main '-' prerelease | main '+' build_metadata | main
main                    ::=  major ('.'  minor ('.'  patch)?)?
major                   ::=  number
minor                   ::=  number
patch                   ::=  number
prerelease              ::=  string | number
build_metadata          ::=  string
number                  ::=  [0-9]+
string                  ::=  [0-9a-zA-Z.-]+
```

The `Package.parse()` class method parses package expressions into corresponding `Package` objects:

```
>>> from versions import Package
>>> p = Package.parse('foo-1.0')
>>> p.name
'foo'
```

```
>>> p.version
Version.parse('1.0.0')
```

Dependencies can also be specified in a package expression:

```
>>> package = Package.parse('foo-1.0; depends bar; depends baz >1, <2')
>>> package.dependencies
set([Requirement.parse('baz>1.0.0,<2.0.0'), Requirement.parse('bar')])
```

class versions.packages.**Package**(*name*, *version*, *dependencies=None*)

A package.

Parameters

- **name** (*str*) – Package name.
- **version** (Version) – Package version.

name = None

Package name.

version = None

Package version.

dependencies = None

set of Requirement objects

build_options

The package build options.

Returns set() of build options strings.

classmethod parse(*package_expression*)

Parse a package_expression into a Package object.

1.2.6 repositories

class versions.repositories.**Repository**(*packages=None*)

A package repository.

Parameters **packages** (set() of Package or *None*) – Repository packages.

packages = None

set() of Package objects.

get(*requirement*)

Find packages matching requirement.

Parameters **requirement** (*str* or Requirement) – Requirement to match against repository packages.

Returns list() of matching Package objects.

class versions.repositories.**Pool**(*repositories=None*)

A package repository pool.

When querying a repository pool, it queries all repositories, and merges their results.

Parameters **repositories** (list() of Repository or None) – Underlying package repositories.

repositories = None

list() of Repository

get (*requirement*)
Find packages matching `requirement`.

Parameters **requirement** (*str* or `Requirement`) – Requirement to get from all underlying repositories.

Returns `list()` of matching `Package` objects.

1.2.7 operators

class `versions.operators.`**Operator** (*func*, *string*)
A package version constraint operator.

Parameters

- **func** (*callable*) – The operator callable.
- **string** (*str*) – The operator string representation.

func = None
Operator callable

string = None
Operator string representation

classmethod parse (*string*)
Parses *string* and returns an `Operator` object.

Raises `InvalidOperatorExpression` If *string* is not a valid operator.

Valid operators are ==, !=, <, >, <=, and >=.

exception `versions.operators.`**InvalidOperatorExpression** (*operator*)
Raised when failing to parse an operator.

operator = None
The bogus operator.

1.2.8 errors

Error

exception `versions.errors.`**Error**
An error occurred in `versions`.

1.3 Changelog

- : Improved ascii/unicode strings handling across Python versions.
- : Allow for post-release identifiers using a 4th number identifier (eg. `1.0.0.42`).
- : Allow for post-release identifiers in versions (eg. `1.0.1f`).
- : Added `Repository`.
- : Added `Pool`.
- : Improved all docs!

- : Renamed exceptions.
- : More documentation for `packages`.
- : `Constraint` now supports merging with `Constraint` or `Constraints` objects using the + operator.
- : Completed implementation of `Requirement`.
- : Base implementation of `Requirement`.
- : Added `Package`.
- : Wrote more docs on constraints.
- : Simplified *versions.version.Version.__cmp__* for readability.
- : Base implementation of `Version`, `Constraint` and `Constraints`.

CHAPTER 2

Indices and tables

- *genindex*
- *search*

V