PEP 440 -- Version Identification and Dependency Specification

PEP:	440
Title:	Version Identification and Dependency Specification
Author:	Nick Coghlan <ncoghlan at gmail.com>, Donald Stufft <donald at stufft.io>
BDFL-Delegate:	Nick Coghlan <ncoghlan at gmail.com>
Discussions-To:	Distutils SIG <distutils-sig at python.org (mailto:distutils-sig@python.org?subject=PEP%20440)>
Status:	Active
Type:	Informational
Created:	18-Mar-2013
Post-History:	30 Mar 2013, 27 May 2013, 20 Jun 2013, 21 Dec 2013, 28 Jan 2014, 08 Aug 2014 22 Aug 2014
Replaces:	386 (/dev/peps/pep-0386)
Resolution:	https://mail.python.org/pipermail/distutils-sig/2014-August/024673.html (https://mail.python.org/pipermail/distutils-sig/2014-August/024673.html)

Contents

Abstract (#id19)

This PEP describes a scheme for identifying versions of Python software distributions, and declaring dependencies on particular versions.

This document addresses several limitations of the previous attempt at a standardized approach to versioning, as described in PEP 345 (/dev/peps/pep-0345) and PEP 386 (/dev/peps/pep-0386).

Definitions (#id20)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (http://tools.ietf.org/html/rfc2119.html).

"Projects" are software components that are made available for integration. Projects include Python libraries, frameworks, scripts, plugins, applications, collections of data or other resources, and various combinations thereof. Public Python projects are typically registered on the Python Package Index (https://pypi.python.org).

"Releases" are uniquely identified snapshots of a project.

"Distributions" are the packaged files which are used to publish and distribute a release.

"Build tools" are automated tools intended to run on development systems, producing source and binary distribution archives. Build tools may also be invoked by integration tools in order to build software distributed as sdists rather than prebuilt binary archives.

"Index servers" are active distribution registries which publish version and dependency metadata and place constraints on the permitted metadata.

"Publication tools" are automated tools intended to run on development systems and upload source and binary distribution archives to index servers.

"Installation tools" are integration tools specifically intended to run on deployment targets, consuming source and binary distribution archives from an index server or other designated location and deploying them to the target system.

"Automated tools" is a collective term covering build tools, index servers, publication tools, integration tools and any other software that produces or consumes distribution version and dependency metadata.

Version scheme (#id21)

Distributions are identified by a public version identifier which supports all defined version comparison operations

The version scheme is used both to describe the distribution version provided by a particular distribution archive, as well as to place constraints on the version of dependencies needed in order to build or run the software.

Public version identifiers (#id22)

The canonical public version identifiers MUST comply with the following scheme:

```
[N!]N(.N)*[{a|b|rc}N][.postN][.devN]
```

Public version identifiers MUST NOT include leading or trailing whitespace.

Public version identifiers MUST be unique within a given distribution.

Installation tools SHOULD ignore any public versions which do not comply with this scheme but MUST also include the normalizations specified below. Installation tools MAY warn the user when non-compliant or ambiguous versions are detected.

See also *Appendix B : Parsing version strings with regular expressions* which provides a regular expression to check strict conformance with the canonical format, as well as a more permissive regular expression accepting inputs that may require subsequent normalization.

Public version identifiers are separated into up to five segments:

- Epoch segment: `N!`
- Release segment: `N(.N)*`
- Pre-release segment: `{a|b|rc}N`
- Post-release segment: `.postN`
- Development release segment: `.devN`

Any given release will be a "final release", "pre-release", "post-release" or "developmental release" as defined in the following sections.

All numeric components MUST be non-negative integers represented as sequences of ASCII digits.

All numeric components MUST be interpreted and ordered according to their numeric value, not as text strings.

All numeric components MAY be zero. Except as described below for the release segment, a numeric component of zero has no special significance aside from always being the lowest possible value in the version ordering.

Note:
Some hard to read version identifiers are permitted by this scheme in order to better accommodate the wide range of versioning practices across existing public and private Python projects.
Accordingly, some of the versioning practices which are technically permitted by the PEP are strongly discouraged for new projects. Where this is the case, the relevant details are noted in the following sections.

Local version identifiers (#id23)

Local version identifiers MUST comply with the following scheme:

```
<public version identifier>[+<local version label>]
```

They consist of a normal public version identifier (as defined in the previous section), along with an arbitrary "local version label", separated from the public version identifier by a plus. Local version labels have no specific semantics assigned, but some syntactic restrictions are imposed.

Local version identifiers are used to denote fully API (and, if applicable, ABI) compatible patched versions of upstream projects. For example, these may be created by application developers and system integrators by applying specific backported bug fixes when upgrading to a new upstream release would be too disruptive to the application or other integrated system (such as a Linux distribution).

The inclusion of the local version label makes it possible to differentiate upstream releases from potentially altered rebuilds by downstream integrators. The use of a local version identifier does not affect the kind of a release but, when applied to a source distribution, does indicate that it may not contain the exact same code as the corresponding upstream release.

To ensure local version identifiers can be readily incorporated as part of filenames and URLs, and to avoid formatting inconsistencies in hexadecimal hash representations, local version labels MUST be limited to the following set of permitted characters:

- ASCII letters (`[a-zA-Z]`)
- ASCII digits (`[0-9]`)
- periods (`.`)

Local version labels MUST start and end with an ASCII letter or digit.

Comparison and ordering of local versions considers each segment of the local version (divided by a `.`) separately. If a segment consists entirely of ASCII digits then that section should be considered an integer for comparison purposes and if a segment contains any ASCII letters then that segment is compared lexicographically with case insensitivity. When comparing a numeric and lexicographic segment, the numeric section always compares as greater than the lexicographic segment. Additionally a local version with a great number of segments will always compare as greater than a local version with fewer segments, as long as the shorter local version's segments match the beginning of the longer local version's segments exactly.

An "upstream project" is a project that defines its own public versions. A "downstream project" is one which tracks and redistributes an upstream project, potentially backporting security and bug fixes from later versions of the upstream project.

Local version identifiers SHOULD NOT be used when publishing upstream projects to a public index server, but MAY be used to identify private builds created directly from the project source. Local version identifiers SHOULD be used by downstream projects when releasing a version that is API compatible with the version of the upstream project identified by the public version identifier, but contains additional changes (such as bug fixes). As the Python Package Index is intended solely for indexing and hosting upstream projects, it MUST NOT allow the use of local version identifiers.

Source distributions using a local version identifier SHOULD provide the `python.integrator` extension metadata (as defined in PEP 459 (/dev/peps/pep-0459)).

Final releases (#id24)

A version identifier that consists solely of a release segment and optionally an epoch identifier is termed a "final release".

The release segment consists of one or more non-negative integer values, separated by dots:

```
N(.N)*
```

Final releases within a project MUST be numbered in a consistently increasing fashion, otherwise automated tools will not be able to upgrade them correctly.

Comparison and ordering of release segments considers the numeric value of each component of the release segment in turn. When comparing release segments with different numbers of components, the shorter segment is padded out with additional zeros as necessary.

While any number of additional components after the first are permitted under this scheme, the most common variants are to use two components ("major.minor") or three components ("major.minor.micro").

For example:

```
0.9
0.9.1
0.9.2
...
0.9.10
0.9.11
1.0
1.0.1
1.1
2.0
2.0.1
...
```

A release series is any set of final release numbers that start with a common prefix. For example, `3.3.1`, `3.3.5` and `3.3.9.45` are all part of the `3.3` release series.

Note:
`X.Y` and `X.Y.0` are not considered distinct release numbers, as the release segment comparison rules implicit expand the two component form to `X.Y.0` when comparing it to any release segment that includes three components.

Date based release segments are also permitted. An example of a date based release scheme using the year and month of the release:

```
2012.4
2012.7
2012.10
2013.1
2013.6
...
```

Pre-releases (#id25)

Some projects use an "alpha, beta, release candidate" pre-release cycle to support testing by their users prior to a final release.

If used as part of a project's development cycle, these pre-releases are indicated by including a pre-release segment in the version identifier:

```
X.YaN   # Alpha release
X.YbN   # Beta release
X.YrcN  # Release Candidate
X.Y     # Final release
```

A version identifier that consists solely of a release segment and a pre-release segment is termed a "pre-release".

The pre-release segment consists of an alphabetical identifier for the pre-release phase, along with a non-negative integer value. Pre-releases for a given release are ordered first by phase (alpha, beta, release candidate) and then by the numerical component within that phase.

Installation tools MAY accept both `c` and `rc` releases for a common release segment in order to handle some existing legacy distributions.

Installation tools SHOULD interpret `c` versions as being equivalent to `rc` versions (that is, `c1` indicates the same version as `rc1`).

Build tools, publication tools and index servers SHOULD disallow the creation of both `rc` and `c` releases for a common release segment.

Post-releases (#id26)

Some projects use post-releases to address minor errors in a final release that do not affect the distributed software (for example, correcting an error in the release notes).

If used as part of a project's development cycle, these post-releases are indicated by including a post-release segment in the version identifier:

```
X.Y.postN    # Post-release
```

A version identifier that includes a post-release segment without a developmental release segment is termed a "post-release".

The post-release segment consists of the string `.post`, followed by a non-negative integer value. Post-releases are ordered by their numerical component, immediately following the corresponding release, and ahead of any subsequent release.

Note:
The use of post-releases to publish maintenance releases containing actual bug fixes is strongly discouraged. In general, it is better to use a longer release number and increment the final component for each maintenance release.

Post-releases are also permitted for pre-releases:

```
X.YaN.postM   # Post-release of an alpha release
X.YbN.postM   # Post-release of a beta release
X.YrcN.postM  # Post-release of a release candidate
```

Note:
Creating post-releases of pre-releases is strongly discouraged, as it makes the version identifier difficult to parse for human readers. In general, it is substantially clearer to simply create a new pre-release by incrementing the numeric component.

Developmental releases (#id27)

Some projects make regular developmental releases, and system packagers (especially for Linux distributions) may wish to create early releases directly from source control which do not conflict with later project releases.

If used as part of a project's development cycle, these developmental releases are indicated by including a developmental release segment in the version identifier:

```
X.Y.devN    # Developmental release
```

A version identifier that includes a developmental release segment is termed a "developmental release".

The developmental release segment consists of the string `.dev`, followed by a non-negative integer value. Developmental releases are ordered by their numerical component, immediately before the corresponding release (and before any pre-releases with the same release segment), and following any previous release (including any post-releases).

Developmental releases are also permitted for pre-releases and post-releases:

```
X.YaN.devM       # Developmental release of an alpha release
X.YbN.devM       # Developmental release of a beta release
X.YrcN.devM      # Developmental release of a release candidate
X.Y.postN.devM   # Developmental release of a post-release
```

Note:

While they may be useful for continuous integration purposes, publishing developmental releases of pre-releases to general purpose public index servers is strongly discouraged, as it makes the version identifier difficult to parse for human readers. If such a release needs to be published, it is substantially clearer to instead create a new pre-release by incrementing the numeric component. Developmental releases of post-releases are also strongly discouraged, but they may be appropriate for projects which use the post-release notation for full maintenance releases which may include code changes.

Version epochs (#id28)

If included in a version identifier, the epoch appears before all other components, separated from the release segment by an exclamation mark:

```
E!X.Y  # Version identifier with epoch
```

If no explicit epoch is given, the implicit epoch is `0`.

Most version identifiers will not include an epoch, as an explicit epoch is only needed if a project *changes* the way it handles version numbering in a way that means the normal version ordering rules will give the wrong answer. For example, if a project is using date based versions like `2014.04` and would like to switch to semantic versions like `1.0`, then the new releases would be identified as *older* than the date based releases when using the normal sorting scheme:

```
1.0
1.1
2.0
2013.10
2014.04
```

However, by specifying an explicit epoch, the sort order can be changed appropriately, as all versions from a later epoch are sorted after versions from an earlier epoch:

```
2013.10
2014.04
1!1.0
1!1.1
1!2.0
```

Normalization (#id29)

In order to maintain better compatibility with existing versions there are a number of "alternative" syntaxes that MUST be taken into account when parsing versions. These syntaxes MUST be considered when parsing a version, however they should be "normalized" to the standard syntax defined above.

Case sensitivity (#id30)

All ascii letters should be interpreted case insensitively within a version and the normal form is lowercase. This allows versions such as `1.1RC1` which would be normalized to `1.1rc1`.

Integer Normalization (#id31)

All integers are interpreted via the `int()` built in and normalize to the string form of the output. This means that an integer version of `00` would normalize to `0` while `09000` would normalize to `9000`. This does not hold true for integers inside of an alphanumeric segment of a local version such as `1.0+foo0100` which is already in its normalized form.

Pre-release separators (#id32)

Pre-releases should allow a `.`, `-`, or `_` separator between the release segment and the pre-release segment. The normal form for this is without a separator. This allows versions such as `1.1.a1` or `1.1-a1` which would be normalized to `1.1a1`. It should also allow a separator to be used between the pre-release signifier and the numeral. This allows versions such as `1.0a.1` which would be normalized to `1.0a1`.

Pre-release spelling (#id33)

Pre-releases allow the additional spellings of `alpha`, `beta`, `c`, `pre`, and `preview` for `a`, `b`, `rc`, `rc`, and `rc` respectively. This allows versions such as `1.1alpha1`, `1.1beta2`, or `1.1c3` which normalize to `1.1a1`, `1.1b2`, and `1.1rc3`. In every case the additional spelling should be considered equivalent to their normal forms.

Implicit pre-release number (#id34)

Pre releases allow omitting the numeral in which case it is implicitly assumed to be `0`. The normal form for this is to include the `0` explicitly. This allows versions such as `1.2a` which is normalized to `1.2a0`.

Post release separators (#id35)

Post releases allow a `.`, `-`, or `_` separator as well as omitting the separator all together. The normal form of this is with the `.` separator. This allows versions such as `1.2-post2` or `1.2post2` which normalize to `1.2.post2`. Like the pre-release separator this also allows an optional separator between the post release signifier and the numeral. This allows versions like `1.2.post-2` which would normalize to `1.2.post2`.

Post release spelling (#id36)

Post-releases allow the additional spellings of `rev` and `r`. This allows versions such as `1.0-r4` which normalizes to `1.0.post4`. As with the pre-releases the additional spellings should be considered equivalent to their normal forms.

Implicit post release number (#id37)

Post releases allow omitting the numeral in which case it is implicitly assumed to be `0`. The normal form for this is to include the `0` explicitly. This allows versions such as `1.2.post` which is normalized to `1.2.post0`.

Implicit post releases (#id38)

Post releases allow omitting the `post` signifier all together. When using this form the separator MUST be `-` and no other form is allowed. This allows versions such as `1.0-1` to be normalized to `1.0.post1`. This particular normalization MUST NOT be used in conjunction with the implicit post release number rule. In other words, `1.0-` is *not* a valid version and it does *not* normalize to `1.0.post0`.

Development release separators (#id39)

Development releases allow a `.`, `-`, or a `_` separator as well as omitting the separator all together. The normal form of this is with the `.` separator. This allows versions such as `1.2-dev2` or `1.2dev2` which normalize to `1.2.dev2`.

Implicit development release number (#id40)

Development releases allow omitting the numeral in which case it is implicitly assumed to be `0`. The normal form for this is to include the `0` explicitly. This allows versions such as `1.2.dev` which is normalized to `1.2.dev0`.

Local version segments (#id41)

With a local version, in addition to the use of `.` as a separator of segments, the use of `-` and `_` is also acceptable. The normal form is using the `.` character. This allows versions such as `1.0+ubuntu-1` to be normalized to `1.0+ubuntu.1`.

Preceding v character (#id42)

In order to support the common version notation of `v1.0` versions may be preceded by a single literal `v` character. This character MUST be ignored for all purposes and should be omitted from all normalized forms of the version. The same version with and without the `v` is considered equivalent.

Leading and Trailing Whitespace (#id43)

Leading and trailing whitespace must be silently ignored and removed from all normalized forms of a version. This includes `" "`, `\t`, `\n`, `\r`, `\f`, and `\v`. This allows accidental whitespace to be handled sensibly, such as a version like `1.0\n` which normalizes to `1.0`.

Examples of compliant version schemes (#id44)

The standard version scheme is designed to encompass a wide range of identification practices across public and private Python projects. In practice, a single project attempting to use the full flexibility offered by the scheme would create a situation where human users had difficulty figuring out the relative order of versions, even though the rules above ensure all compliant tools will order them consistently.

The following examples illustrate a small selection of the different approaches projects may choose to identify their releases, while still ensuring that the "latest release" and the "latest stable release" can be easily determined, both by human users and automated tools.

Simple "major.minor" versioning:

```
0.1
0.2
0.3
1.0
1.1
...
```

Simple "major.minor.micro" versioning:

```
1.1.0
1.1.1
1.1.2
1.2.0
...
```

"major.minor" versioning with alpha, beta and candidate pre-releases:

```
0.9
1.0a1
1.0a2
1.0b1
1.0rc1
1.0
1.1a1
...
```

"major.minor" versioning with developmental releases, release candidates and post-releases for minor corrections:

```
0.9
1.0.dev1
1.0.dev2
1.0.dev3
1.0.dev4
1.0c1
1.0c2
1.0
1.0.post1
1.1.dev1
...
```

Date based releases, using an incrementing serial within each year, skipping zero:

```
2012.1
2012.2
2012.3
...
2012.15
2013.1
2013.2
...
```

Summary of permitted suffixes and relative ordering (#id45)

Note:
This section is intended primarily for authors of tools that automatically process distribution metadata, rather than developers of Python distributions deciding on a versioning scheme.

The epoch segment of version identifiers MUST be sorted according to the numeric value of the given epoch. If no epoch segment is present, the implicit numeric value is `0`.

The release segment of version identifiers MUST be sorted in the same order as Python's tuple sorting when the normalized release segment is parsed as follows:

```
tuple(map(int, release_segment.split(".")))
```

All release segments involved in the comparison MUST be converted to a consistent length by padding shorter segments with zeros as needed.

Within a numeric release (`1.0`, `2.7.3`), the following suffixes are permitted and MUST be ordered as shown:

```
.devN, aN, bN, rcN, <no suffix>, .postN
```

Note that *c* is considered to be semantically equivalent to *rc* and must be sorted as if it were *rc*. Tools MAY reject the case of having the same `N` for both a `c` and a `rc` in the same release segment as ambiguous and remain in compliance with the PEP.

Within an alpha (`1.0a1`), beta (`1.0b1`), or release candidate (`1.0rc1`, `1.0c1`), the following suffixes are permitted and MUST be ordered as shown:

```
.devN, <no suffix>, .postN
```

Within a post-release (`1.0.post1`), the following suffixes are permitted and MUST be ordered as shown:

```
.devN, <no suffix>
```

Note that `devN` and `postN` MUST always be preceded by a dot, even when used immediately following a numeric version (e.g. `1.0.dev456`, `1.0.post1`).

Within a pre-release, post-release or development release segment with a shared prefix, ordering MUST be by the value of the numeric component.

The following example covers many of the possible combinations:

```
1.dev0
1.0.dev456
1.0a1
1.0a2.dev456
1.0a12.dev456
1.0a12
1.0b1.dev456
1.0b2
1.0b2.post345.dev456
1.0b2.post345
1.0rc1.dev456
1.0rc1
1.0
1.0+abc.5
1.0+abc.7
1.0+5
1.0.post456.dev34
1.0.post456
1.0.15
1.1.dev1
```

Version ordering across different metadata versions (#id46)

Metadata v1.0 (PEP 241 (/dev/peps/pep-0241)) and metadata v1.1 (PEP 314 (/dev/peps/pep-0314)) do not specify a standard version identification or ordering scheme. However metadata v1.2 (PEP 345 (/dev/peps/pep-0345)) does specify a scheme which is defined in PEP 386 (/dev/peps/pep-0386).

Due to the nature of the simple installer API it is not possible for an installer to be aware of which metadata version a particular distribution was using. Additionally installers required the ability to create a reasonably prioritized list that includes all, or as many as possible, versions of a project to determine which versions it should install. These requirements necessitate a standardization across one parsing mechanism to be used for all versions of a project.

Due to the above, this PEP MUST be used for all versions of metadata and supersedes PEP 386 (/dev/peps/pep-0386) even for metadata v1.2. Tools SHOULD ignore any versions which cannot be parsed by the rules in this PEP, but MAY fall back to implementation defined version parsing and ordering schemes if no versions complying with this PEP are available.

Distribution users may wish to explicitly remove non-compliant versions from any private package indexes they control.

Compatibility with other version schemes (#id47)

Some projects may choose to use a version scheme which requires translation in order to comply with the public version scheme defined in this PEP. In such cases, the project specific version can be stored in the metadata while the translated public version is published in the version field.

This allows automated distribution tools to provide consistently correct ordering of published releases, while still allowing developers to use the internal versioning scheme they prefer for their projects.

Semantic versioning (#id48)

Semantic versioning (http://semver.org/) is a popular version identification scheme that is more prescriptive than this PEP regarding the significance of different elements of a release number. Even if a project chooses not to abide by the details of semantic versioning, the scheme is worth understanding as it covers many of the issues that can arise when depending on other distributions, and when publishing a distribution that others rely on.

The "Major.Minor.Patch" (described in this PEP as "major.minor.micro") aspects of semantic versioning (clauses 1-8 in the 2.0.0 specification) are fully compatible with the version scheme defined in this PEP, and abiding by these aspects is encouraged.

Semantic versions containing a hyphen (pre-releases - clause 10) or a plus sign (builds - clause 11) are *not* compatible with this PEP and are not permitted in the public version field.

One possible mechanism to translate such semantic versioning based source labels to compatible public versions is to use the `.devN` suffix to specify the appropriate version order.

Specific build information may also be included in local version labels.

DVCS based version labels (#id49)

Many build tools integrate with distributed version control systems like Git and Mercurial in order to add an identifying hash to the version identifier. As hashes cannot be ordered reliably such versions are not permitted in the public version field.

As with semantic versioning, the public `.devN` suffix may be used to uniquely identify such releases for publication, while the original DVCS based label can be stored in the project metadata.

Identifying hash information may also be included in local version labels.

Olson database versioning (#id50)

The `pytz` project inherits its versioning scheme from the corresponding Olson timezone database versioning scheme: the year followed by a lowercase character indicating the version of the database within that year.

This can be translated to a compliant public version identifier as `<year>.<serial>`, where the serial starts at zero or one (for the '<year>a' release) and is incremented with each subsequent database update within the year.

As with other translated version identifiers, the corresponding Olson database version could be recorded in the project metadata.

Version specifiers (#id51)

A version specifier consists of a series of version clauses, separated by commas. For example:

```
~= 0.9, >= 1.0, != 1.3.4.*, < 2.0
```

The comparison operator determines the kind of version clause:

- ~=: Compatible release (#compatible-release) clause
- ==: Version matching (#version-matching) clause
- !=: Version exclusion (#version-exclusion) clause
- <=, >=: Inclusive ordered comparison (#inclusive-ordered-comparison) clause
- <, >: Exclusive ordered comparison (#exclusive-ordered-comparison) clause
- ===: Arbitrary equality (#arbitrary-equality) clause.

The comma (",") is equivalent to a logical **and** operator: a candidate version must match all given version clauses in order to match the specifier as a whole.

Whitespace between a conditional operator and the following version identifier is optional, as is the whitespace around the commas.

When multiple candidate versions match a version specifier, the preferred version SHOULD be the latest version as determined by the consistent ordering defined by the standard Version scheme (#version-scheme). Whether or not pre-releases are considered as candidate versions SHOULD be handled as described in Handling of pre-releases (#handling-of-pre-releases).

Except where specifically noted below, local version identifiers MUST NOT be permitted in version specifiers, and local version labels MUST be ignored entirely when checking if candidate versions match a given version specifier.

Compatible release (#id52)

A compatible release clause consists of the compatible release operator `~=` and a version identifier. It matches any candidate version that is expected to be compatible with the specified version.

The specified version identifier must be in the standard format described in Version scheme (#version-scheme). Local version identifiers are NOT permitted in this version specifier.

For a given release identifier `V.N`, the compatible release clause is approximately equivalent to the pair of comparison clauses:

```
>= V.N, == V.*
```

This operator MUST NOT be used with a single segment version number such as `~=1`.

For example, the following groups of version clauses are equivalent:

```
~= 2.2
>= 2.2, == 2.*

~= 1.4.5
>= 1.4.5, == 1.4.*
```

If a pre-release, post-release or developmental release is named in a compatible release clause as `V.N.suffix`, then the suffix is ignored when determining the required prefix match:

```
~= 2.2.post3
>= 2.2.post3, == 2.*

~= 1.4.5a4
>= 1.4.5a4, == 1.4.*
```

The padding rules for release segment comparisons means that the assumed degree of forward compatibility in a compatible release clause can be controlled by appending additional zeros to the version specifier:

```
~= 2.2.0
>= 2.2.0, == 2.2.*

~= 1.4.5.0
>= 1.4.5.0, == 1.4.5.*
```

Version matching (#id53)

A version matching clause includes the version matching operator == and a version identifier.

The specified version identifier must be in the standard format described in Version scheme (#version-scheme), but a trailing `.*` is permitted on public version identifiers as described below.

By default, the version matching operator is based on a strict equality comparison: the specified version must be exactly the same as the requested version. The *only* substitution performed is the zero padding of the release segment to ensure the release segments are compared with the same length.

Whether or not strict version matching is appropriate depends on the specific use case for the version specifier. Automated tools SHOULD at least issue warnings and MAY reject them entirely when strict version matches are used inappropriately.

Prefix matching may be requested instead of strict comparison, by appending a trailing `.*` to the version identifier in the version matching clause. This means that additional trailing segments will be ignored when determining whether or not a version identifier matches the clause. If the specified version includes only a release segment, than trailing components (or the lack thereof) in the re-

lease segment are also ignored.

For example, given the version `1.1.post1`, the following clauses would match or not as shown:

```
== 1.1        # Not equal, so 1.1.post1 does not match clause
== 1.1.post1  # Equal, so 1.1.post1 matches clause
== 1.1.*      # Same prefix, so 1.1.post1 matches clause
```

For purposes of prefix matching, the pre-release segment is considered to have an implied preceding `.`, so given the version `1.1a1`, the following clauses would match or not as shown:

```
== 1.1        # Not equal, so 1.1a1 does not match clause
== 1.1a1      # Equal, so 1.1a1 matches clause
== 1.1.*      # Same prefix, so 1.1a1 matches clause
```

An exact match is also considered a prefix match (this interpretation is implied by the usual zero padding rules for the release segment of version identifiers). Given the version `1.1`, the following clauses would match or not as shown:

```
== 1.1        # Equal, so 1.1 matches clause
== 1.1.0      # Zero padding expands 1.1 to 1.1.0, so it matches clause
== 1.1.dev1   # Not equal (dev-release), so 1.1 does not match clause
== 1.1a1      # Not equal (pre-release), so 1.1 does not match clause
== 1.1.post1  # Not equal (post-release), so 1.1 does not match clause
== 1.1.*      # Same prefix, so 1.1 matches clause
```

It is invalid to have a prefix match containing a development or local release such as `1.0.dev1.*` or `1.0+foo1.*`. If present, the development release segment is always the final segment in the public version, and the local version is ignored for comparison purposes, so using either in a prefix match wouldn't make any sense.

The use of == (without at least the wildcard suffix) when defining dependencies for published distributions is strongly discouraged as it greatly complicates the deployment of security fixes. The strict version comparison operator is intended primarily for use when defining dependencies for repeatable *deployments of applications* while using a shared distribution index.

If the specified version identifier is a public version identifier (no local version label), then the local version label of any candidate versions MUST be ignored when matching versions.

If the specified version identifier is a local version identifier, then the local version labels of candidate versions MUST be considered when matching versions, with the public version identifier being matched as described above, and the local version label being checked for equivalence using a strict string equality comparison.

Version exclusion (#id54)

A version exclusion clause includes the version exclusion operator `!=` and a version identifier.

The allowed version identifiers and comparison semantics are the same as those of the Version matching (#version-matching) operator, except that the sense of any match is inverted.

For example, given the version `1.1.post1`, the following clauses would match or not as shown:

```
!= 1.1        # Not equal, so 1.1.post1 matches clause
!= 1.1.post1  # Equal, so 1.1.post1 does not match clause
!= 1.1.*      # Same prefix, so 1.1.post1 does not match clause
```

Inclusive ordered comparison (#id55)

An inclusive ordered comparison clause includes a comparison operator and a version identifier, and will match any version where the comparison is correct based on the relative position of the candidate version and the specified version given the consistent ordering defined by the standard Version scheme (#version-scheme).

The inclusive ordered comparison operators are <= and >=.

As with version matching, the release segment is zero padded as necessary to ensure the release segments are compared with the same length.

Local version identifiers are NOT permitted in this version specifier.

Exclusive ordered comparison (#id56)

The exclusive ordered comparisons > and < are similar to the inclusive ordered comparisons in that they rely on the relative position of the candidate version and the specified version given the consistent ordering defined by the standard Version scheme (#version-scheme). However, they specifically exclude pre-releases, post-releases, and local versions of the specified version.

The exclusive ordered comparison >V **MUST NOT** allow a post-release of the given version unless V itself is a post release. You may mandate that releases are later than a particular post release, including additional post releases, by using `>V.postN`. For example, `>1.7` will allow `1.7.1` but not `1.7.0.post1` and `>1.7.post2` will allow `1.7.1` and `1.7.0.post3` but not `1.7.0`.

The exclusive ordered comparison >V **MUST NOT** match a local version of the specified version.

The exclusive ordered comparison <V **MUST NOT** allow a pre-release of the specified version unless the specified version is itself a pre-release. Allowing pre-releases that are earlier than, but not equal to a specific pre-release may be accomplished by using `<V.rc1` or similar.

As with version matching, the release segment is zero padded as necessary to ensure the release segments are compared with the same length.

Local version identifiers are NOT permitted in this version specifier.

Arbitrary equality (#id57)

Arbitrary equality comparisons are simple string equality operations which do not take into account any of the semantic information such as zero padding or local versions. This operator also does not support prefix matching as the == operator does.

The primary use case for arbitrary equality is to allow for specifying a version which cannot otherwise be represented by this PEP. This operator is special and acts as an escape hatch to allow someone using a tool which implements this PEP to still install a legacy version which is otherwise incompatible with this PEP.

An example would be `===foobar` which would match a version of `foobar`.

This operator may also be used to explicitly require an unpatched version of a project such as `===1.0` which would not match for a version `1.0+downstream1`.

Use of this operator is heavily discouraged and tooling MAY display a warning when it is used.

Handling of pre-releases (#id58)

Pre-releases of any kind, including developmental releases, are implicitly excluded from all version specifiers, *unless* they are already present on the system, explicitly requested by the user, or if the only available version that satisfies the version specifier is a pre-release.

By default, dependency resolution tools SHOULD:

- accept already installed pre-releases for all version specifiers
- accept remotely available pre-releases for version specifiers where there is no final or post release that satisfies the version specifier
- exclude all other pre-releases from consideration

Dependency resolution tools MAY issue a warning if a pre-release is needed to satisfy a version specifier.

Dependency resolution tools SHOULD also allow users to request the following alternative behaviours:

- accepting pre-releases for all version specifiers
- excluding pre-releases for all version specifiers (reporting an error or warning if a pre-release is already installed locally, or if a pre-release is the only way to satisfy a particular specifier)

Dependency resolution tools MAY also allow the above behaviour to be controlled on a per-distribution basis.

Post-releases and final releases receive no special treatment in version specifiers - they are always included unless explicitly excluded.

Examples (#id59)

- `~=3.1`: version 3.1 or later, but not version 4.0 or later.
- `~=3.1.2`: version 3.1.2 or later, but not version 3.2.0 or later.
- `~=3.1a1`: version 3.1a1 or later, but not version 4.0 or later.
- `== 3.1`: specifically version 3.1 (or 3.1.0), excludes all pre-releases, post releases, developmental releases and any 3.1.x maintenance releases.
- `== 3.1.*`: any version that starts with 3.1. Equivalent to the `~=3.1.0` compatible release clause.
- `~=3.1.0, != 3.1.3`: version 3.1.0 or later, but not version 3.1.3 and not version 3.2.0 or later.

Direct references (#id60)

Some automated tools may permit the use of a direct reference as an alternative to a normal version specifier. A direct reference consists of the specifier @ and an explicit URL.

Whether or not direct references are appropriate depends on the specific use case for the version specifier. Automated tools SHOULD at least issue warnings and MAY reject them entirely when direct references are used inappropriately.

Public index servers SHOULD NOT allow the use of direct references in uploaded distributions. Direct references are intended as a tool for software integrators rather than publishers.

Depending on the use case, some appropriate targets for a direct URL reference may be an sdist or a wheel binary archive. The exact URLs and targets supported will be tool dependent.

For example, a local source archive may be referenced directly:

```
pip @ file:///localbuilds/pip-1.3.1.zip
```

Alternatively, a prebuilt archive may also be referenced:

```
pip @ file:///localbuilds/pip-1.3.1-py33-none-any.whl
```

All direct references that do not refer to a local file URL SHOULD specify a secure transport mechanism (such as `https`) AND include an expected hash value in the URL for verification purposes. If a direct reference is specified without any hash information, with hash information that the tool doesn't understand, or with a selected hash algorithm that the tool considers too weak to trust, automated tools SHOULD at least emit a warning and MAY refuse to rely on the URL. If such a direct reference also uses an insecure transport, automated tools SHOULD NOT rely on the URL.

It is RECOMMENDED that only hashes which are unconditionally provided by the latest version of the standard library's `hashlib` module be used for source archive hashes. At time of writing, that list consists of `'md5'`, `'sha1'`, `'sha224'`, `'sha256'`, `'sha384'`, and `'sha512'`.

For source archive and wheel references, an expected hash value may be specified by including a `<hash-algorithm>=<expected-hash>` entry as part of the URL fragment.

For version control references, the `VCS+protocol` scheme SHOULD be used to identify both the version control system and the secure transport, and a version control system with hash based commit identifiers SHOULD be used. Automated tools MAY omit warnings about missing hashes for version control systems that do not provide hash based commit identifiers.

To handle version control systems that do not support including commit or tag references directly in the URL, that information may be appended to the end of the URL using the `@<commit-hash>` or the `@<tag>#<commit-hash>` notation.

Note:

This isn't *quite* the same as the existing VCS reference notation supported by pip. Firstly, the distribution name is moved in front rather than embedded as part of the URL. Secondly, the commit hash is included even when retrieving based on a tag, in order to meet the requirement above that *every* link should include a hash to make things harder to forge (creating a malicious repo with a particular tag is easy, creating one with a specific *hash*, less so).

Remote URL examples:

```
pip @ https://github.com/pypa/pip/archive/1.3.1.zip#sha1=da9234ee9982d4bbb3c72346a6de940a148ea686
pip @ git+https://github.com/pypa/pip.git@7921be1537eac1e97bc40179a57f0349c2aee67d
pip @ git+https://github.com/pypa/pip.git@1.3.1#7921be1537eac1e97bc40179a57f0349c2aee67d
```

File URLs (#id61)

File URLs take the form of `file://<host>/<path>`. If the `<host>` is omitted it is assumed to be `localhost` and even if the `<host>` is omitted the third slash MUST still exist. The `<path>` defines what the file path on the filesystem that is to be accessed.

On the various *nix operating systems the only allowed values for `<host>` is for it to be omitted, `localhost`, or another FQDN that the current machine believes matches its own host. In other words, on *nix the `file://` scheme can only be used to access paths on the local machine.

On Windows the file format should include the drive letter if applicable as part of the `<path>` (e.g. `file:///c:/path/to/a/file`). Unlike *nix on Windows the `<host>` parameter may be used to specify a file residing on a network share. In other words, in order to translate `\\machine\volume\file` to a `file://` url, it would end up as `file://machine/volume/file`. For more information on `file://` URLs on Windows see MSDN [4] (#id12).

Updating the versioning specification (#id62)

The versioning specification may be updated with clarifications without requiring a new PEP or a change to the metadata version.

Any technical changes that impact the version identification and comparison syntax and semantics would require an updated versioning scheme to be defined in a new PEP.

Summary of differences from pkg_resources.parse_version (#id63)

- Local versions sort differently, this PEP requires that they sort as greater than the same version without a local version, whereas `pkg_resources.parse_version` considers it a pre-release marker.
- This PEP purposely restricts the syntax which constitutes a valid version while `pkg_resources.parse_version` attempts to provide some meaning from *any* arbitrary string.
- `pkg_resources.parse_version` allows arbitrarily deeply nested version signifiers like `1.0.dev1.post1.dev5`. This PEP however allows only a single use of each type and they must exist in a certain order.

Summary of differences from PEP 386 (/dev/peps/pep-0386)

- Moved the description of version specifiers into the versioning PEP
- Added the "direct reference" concept as a standard notation for direct references to resources (rather than each tool needing to invent its own)

- Added the "local version identifier" and "local version label" concepts to allow system integrators to indicate patched builds in a way that is supported by the upstream tools, as well as to allow the incorporation of build tags into the versioning of binary distributions.
- Added the "compatible release" clause
- Added the trailing wildcard syntax for prefix based version matching and exclusion
- Changed the top level sort position of the `.devN` suffix
- Allowed single value version numbers
- Explicit exclusion of leading or trailing whitespace
- Explicit support for date based versions
- Explicit normalisation rules to improve compatibility with existing version metadata on PyPI where it doesn't introduce ambiguity
- Implicitly exclude pre-releases unless they're already present or needed to satisfy a dependency
- Treat post releases the same way as unqualified releases
- Discuss ordering and dependencies across metadata versions
- Switch from preferring `c` to `rc`.

The rationale for major changes is given in the following sections.

Changing the version scheme (#id65)

One key change in the version scheme in this PEP relative to that in PEP 386 (/dev/peps/pep-0386) is to sort top level developmental releases like `X.Y.devN` ahead of alpha releases like `X.Ya1`. This is a far more logical sort order, as projects already using both development releases and alphas/betas/release candidates do not want their developmental releases sorted in between their release candidates and their final releases. There is no rationale for using `dev` releases in that position rather than merely creating additional release candidates.

The updated sort order also means the sorting of `dev` versions is now consistent between the metadata standard and the pre-existing behaviour of `pkg_resources` (and hence the behaviour of current installation tools).

Making this change should make it easier for affected existing projects to migrate to the latest version of the metadata standard.

Another change to the version scheme is to allow single number versions, similar to those used by non-Python projects like Mozilla Firefox, Google Chrome and the Fedora Linux distribution. This is actually expected to be more useful for version specifiers, but it is easier to allow it for both version specifiers and release numbers, rather than splitting the two definitions.

The exclusion of leading and trailing whitespace was made explicit after a couple of projects with version identifiers differing only in a trailing `\n` character were found on PyPI.

Various other normalisation rules were also added as described in the separate section on version normalisation below.

Appendix A shows detailed results of an analysis of PyPI distribution version information, as collected on 8th August, 2014. This analysis compares the behavior of the explicitly ordered version scheme defined in this PEP with the de facto standard defined by the behavior of setuptools. These metrics are useful, as the intent of this PEP is to follow existing setuptools behavior as closely as is feasible, while still throwing exceptions for unorderable versions (rather than trying to guess an appropriate order as setuptools does).

A more opinionated description of the versioning scheme (#id66)

As in PEP 386 (/dev/peps/pep-0386), the primary focus is on codifying existing practices to make them more amenable to automation, rather than demanding that existing projects make non-trivial changes to their workflow. However, the standard scheme allows significantly more flexibility than is needed for the vast majority of simple Python packages (which often don't even need maintenance releases - many users are happy with needing to upgrade to a new feature release to get bug fixes).

For the benefit of novice developers, and for experienced developers wishing to better understand the various use cases, the specification now goes into much greater detail on the components of the defined version scheme, including examples of how each component may be used in practice.

The PEP also explicitly guides developers in the direction of semantic versioning (without requiring it), and discourages the use of several aspects of the full versioning scheme that have largely been included in order to cover esoteric corner cases in the practices of existing projects and in repackaging software for Linux distributions.

Describing version specifiers alongside the versioning scheme (#id67)

The main reason to even have a standardised version scheme in the first place is to make it easier to do reliable automated dependency analysis. It makes more sense to describe the primary use case for version identifiers alongside their definition.

Changing the interpretation of version specifiers (#id68)

The previous interpretation of version specifiers made it very easy to accidentally download a pre-release version of a dependency. This in turn made it difficult for developers to publish pre-release versions of software to the Python Package Index, as even marking the package as hidden wasn't enough to keep automated tools from downloading it, and also made it harder for users to obtain the test release manually through the main PyPI web interface.

The previous interpretation also excluded post-releases from some version specifiers for no adequately justified reason.

The updated interpretation is intended to make it difficult to accidentally accept a pre-release version as satisfying a dependency, while still allowing pre-release versions to be retrieved automatically when that's the only way to satisfy a dependency.

The "some forward compatibility assumed" version constraint is derived from the Ruby community's "pessimistic version constraint" operator [2] (#id10) to allow projects to take a cautious approach to forward compatibility promises, while still easily setting a minimum required version for their dependencies. The spelling of the compatible release clause (~=) is inspired by the Ruby (~>) and PHP (~) equivalents.

Further improvements are also planned to the handling of parallel installation of multiple versions of the same library, but these will depend on updates to the installation database definition along with improved tools for dynamic path manipulation.

The trailing wildcard syntax to request prefix based version matching was added to make it possible to sensibly define compatible release clauses.

Support for date based version identifiers (#id69)

Excluding date based versions caused significant problems in migrating `pytz` to the new metadata standards. It also caused concerns for the OpenStack developers, as they use a date based versioning scheme and would like to be able to migrate to the new metadata standards without changing it.

Adding version epochs (#id70)

Version epochs are added for the same reason they are part of other versioning schemes, such as those of the Fedora and Debian Linux distributions: to allow projects to gracefully change their approach to numbering releases, without having a new release appear to have a lower version number than previous releases and without having to change the name of the project.

In particular, supporting version epochs allows a project that was previously using date based versioning to switch to semantic versioning by specifying a new version epoch.

The `!` character was chosen to delimit an epoch version rather than the `:` character, which is commonly used in other systems, due to the fact that `:` is not a valid character in a Windows directory name.

Adding direct references (#id71)

Direct references are added as an "escape clause" to handle messy real world situations that don't map neatly to the standard distribution model. This includes dependencies on unpublished software for internal use, as well as handling the more complex compatibility issues that may arise when wrapping third party libraries as C extensions (this is of especial concern to the scientific community).

Index servers are deliberately given a lot of freedom to disallow direct references, since they're intended primarily as a tool for integrators rather than publishers. PyPI in particular is currently going through the process of *eliminating* dependencies on external references, as unreliable external services have the effect of slowing down installation operations, as well as reducing PyPI's own apparent reliability.

Adding arbitrary equality (#id72)

Arbitrary equality is added as an "escape clause" to handle the case where someone needs to install a project which uses a non compliant version. Although this PEP is able to attain ~97% compatibility with the versions that are already on PyPI there are still ~3% of versions which cannot be parsed. This operator gives a simple and effective way to still depend on them without having to "guess" at the semantics of what they mean (which would be required if anything other than strict string based equality was supported).

Adding local version identifiers (#id73)

It's a fact of life that downstream integrators often need to backport upstream bug fixes to older versions. It's one of the services that gets Linux distro vendors paid, and application developers may also apply patches they need to bundled dependencies.

Historically, this practice has been invisible to cross-platform language specific distribution tools - the reported "version" in the upstream metadata is the same as for the unmodified code. This inaccuracy can then cause problems when attempting to work with a mixture of integrator provided code and unmodified upstream code, or even just attempting to identify exactly which version of the software is installed.

The introduction of local version identifiers and "local version labels" into the versioning scheme, with the corresponding `python.integrator` metadata extension allows this kind of activity to be represented accurately, which should improve interoperability between the upstream tools and various integrated platforms.

The exact scheme chosen is largely modeled on the existing behavior of `pkg_resources.parse_version` and `pkg_resources.parse_requirements`, with the main distinction being that where `pkg_resources` currently always takes the suffix into account when comparing versions for exact matches, the PEP requires that the local version label of the candidate version be ignored when no local version label is present in the version specifier clause. Furthermore, the PEP does not attempt to impose any structure on the local version labels (aside from limiting the set of permitted characters and defining their ordering).

This change is designed to ensure that an integrator provided version like `pip 1.5+1` or `pip 1.5+1.git.abc123de` will still satisfy a version specifier like `pip>=1.5`.

The plus is chosen primarily for readability of local version identifiers. It was chosen instead of the hyphen to prevent `pkg_resources.parse_version` from parsing it as a prerelease, which is important for enabling a successful migration to the new, more structured, versioning scheme. The plus was chosen instead of a tilde because of the significance of the tilde in Debian's version ordering algorithm.

Providing explicit version normalization rules (#id74)

Historically, the de facto standard for parsing versions in Python has been the `pkg_resources.parse_version` command from the setuptools project. This does not attempt to reject *any* version and instead tries to make something meaningful, with varying levels of success, out of whatever it is given. It has a few simple rules but otherwise it more or less relies largely on string comparison.

The normalization rules provided in this PEP exist primarily to either increase the compatibility with `pkg_resources.parse_version`, particularly in documented use cases such as `rev`, `r`, `pre`, etc or to do something more reasonable with versions that already exist on PyPI.

All possible normalization rules were weighed against whether or not they were *likely* to cause any ambiguity (e.g. while someone might devise a scheme where `v1.0` and `1.0` are considered distinct releases, the likelihood of anyone actually doing that, much less on any scale that is noticeable, is fairly low). They were also weighed against how `pkg_resources.parse_version` treated a particular version string, especially with regards to how it was sorted. Finally each rule was weighed against the kinds of additional versions it allowed, how "ugly" those versions looked, how hard there were to parse (both mentally and mechanically) and how much additional compatibility it would bring.

The breadth of possible normalizations were kept to things that could easily be implemented as part of the parsing of the version and not pre-parsing transformations applied to the versions. This was done to limit the side effects of each transformation as simple search and replace style transforms increase the likelihood of ambiguous or "junk" versions.

For an extended discussion on the various types of normalizations that were considered, please see the proof of concept for PEP 440 (/dev/peps/pep-0440) within pip [5] (#id13).

Allowing Underscore in Normalization (#id75)

There are not a lot of projects on PyPI which utilize a `_` in the version string. However this PEP allows its use anywhere that `-` is acceptable. The reason for this is that the Wheel normalization scheme specifies that `-` gets normalized to a `_` to enable easier parsing of the filename.

Summary of changes to PEP 440 (/dev/peps/pep-0440)

The following changes were made to this PEP based on feedback received after the initial reference implementation was released in setuptools 8.0 and pip 6.0:

- The exclusive ordered comparisons were updated to no longer imply a `!=V.*` which was deemed to be surprising behavior which was too hard to accurately describe. Instead the exclusive ordered comparisons will simply disallow matching pre-releases, post-releases, and local versions of the specified version (unless the specified version is itself a pre-release, post-release or local version). For an extended discussion see the threads on distutils-sig [6] (#id14) [7] (#id15).
- The normalized form for release candidates was updated from 'c' to 'rc'. This change was based on user feedback received when setuptools 8.0 started applying normalisation to the release metadata generated when preparing packages for publication on PyPI [8] (#id16).
- The PEP text and the `is_canonical` regex were updated to be explicit that numeric components are specifically required to be represented as sequences of ASCII digits, not arbitrary Unicode [Nd] code points. This was previously implied by the version parsing regex in Appendix B, but not stated explicitly [10] (#id18).

References (#id77)

The initial attempt at a standardised version scheme, along with the justifications for needing such a standard can be found in PEP 386 (/dev/peps/pep-0386).

[1] Reference Implementation of PEP 440 (/dev/peps/pep-0440) Versions and Specifiers https://github.com/pypa/packaging/pull/1 (https://github.com/pypa/packaging/pull/1)

[2] (#id3) Version compatibility analysis script: https://github.com/pypa/packaging/blob/master/tasks/check.py (https://github.com/pypa/packaging/blob/master/tasks/check.py)

[3] Pessimistic version constraint http://docs.rubygems.org/read/chapter/16 (http://docs.rubygems.org/read/chapter/16)

[4] (#id2) File URIs in Windows http://blogs.msdn.com/b/ie/archive/2006/12/06/file-uris-in-windows.aspx (http://blogs.msdn.com/b/ie/archive/2006/12/06/file-uris-in-windows.aspx)

[5] (#id4) Proof of Concept: PEP 440 (/dev/peps/pep-0440) within pip https://github.com/pypa/pip/pull/1894 (https://github.com/pypa/pip/pull/1894)

[6] (#id5) PEP440: foo-X.Y.Z does not satisfy "foo>X.Y" https://mail.python.org/pipermail/distutils-sig/2014-December/025451.html (https://mail.python.org/pipermail/distutils-sig/2014-December/025451.html)

[7] (#id6) PEP440: >1.7 vs >=1.7 https://mail.python.org/pipermail/distutils-sig/2014-December/025507.html (https://mail.python.org/pipermail/distutils-sig/2014-December/025507.html)

[8] (#id7) Amend PEP 440 (/dev/peps/pep-0440) with Wider Feedback on Release Candidates https://mail.python.org/pipermail/distutils-sig/2014-December/025409.html (https://mail.python.org/pipermail/distutils-sig/2014-December/025409.html)

[9] Changing the status of PEP 440 (/dev/peps/pep-0440) to Provisional https://mail.python.org/pipermail/distutils-sig/2014-December/025412.html (https://mail.python.org/pipermail/distutils-sig/2014-December/025412.html)

[10] PEP 440 (/dev/peps/pep-0440): regex should not permit Unicode [Nd] characters https://github.com/python/peps/pull/966 (#id8) (https://github.com/python/peps/pull/966)

Appendix A (#id78)

Metadata v2.0 guidelines versus setuptools:

```
$ invoke check.pep440
Total Version Compatibility:              245806/250521 (98.12%)
Total Sorting Compatibility (Unfiltered): 45441/47114 (96.45%)
Total Sorting Compatibility (Filtered):   47057/47114 (99.88%)
Projects with No Compatible Versions:     498/47114 (1.06%)
Projects with Differing Latest Version:   688/47114 (1.46%)
```

Appendix B : Parsing version strings with regular expressions (#id79)

As noted earlier in the *Public version identifiers* section, published version identifiers SHOULD use the canonical format. This section provides regular expressions that can be used to test whether a version is already in that form, and if it's not, extract the various components for subsequent normalization.

To test whether a version identifier is in the canonical format, you can use the following function:

```
import re
def is_canonical(version):
    return re.match(r'^([1-9][0-9]*!)?(0|[1-9][0-9]*)(\.(0|[1-9][0-9]*))*((a|b|rc)(0|[1-9][0-
9]*))?(\.post(0|[1-9][0-9]*))?(\.dev(0|[1-9][0-9]*))?$', version) is not None
```

To extract the components of a version identifier, use the following regular expression (as defined by the packaging (https://github.com/pypa/packaging) project):

```
VERSION_PATTERN = r"""
    v?
    (?:
        (?:(?P<epoch>[0-9]+)!)?                           # epoch
        (?P<release>[0-9]+(?:\.[0-9]+)*)                  # release segment
        (?P<pre>                                          # pre-release
            [-_\.]?
            (?P<pre_l>(a|b|c|rc|alpha|beta|pre|preview))
            [-_\.]?
            (?P<pre_n>[0-9]+)?
        )?
        (?P<post>                                         # post release
            (?:-(?P<post_n1>[0-9]+))
            |
            (?:
                [-_\.]?
                (?P<post_l>post|rev|r)
                [-_\.]?
                (?P<post_n2>[0-9]+)?
            )
        )?
        (?P<dev>                                          # dev release
            [-_\.]?
            (?P<dev_l>dev)
            [-_\.]?
            (?P<dev_n>[0-9]+)?
        )?
    )
    (?:\+(?P<local>[a-z0-9]+(?:[-_\.][a-z0-9]+)*))?       # local version
"""

_regex = re.compile(
    r"^\s*" + VERSION_PATTERN + r"\s*$",
    re.VERBOSE | re.IGNORECASE,
)
```

Copyright (#id80)

Source: https://github.com/python/peps/blob/master/pep-0440.txt (https://github.com/python/peps/blob/master/pep-0440.txt)