Packaging Documentation

Release 20.1

Donald Stufft

Feb 08, 2020

Contents

Core utilities for Python packages

CHAPTER 1

Installation

You can install packaging with `pip`:

```
$ pip install packaging
```

CHAPTER 2

API

2.1 Version Handling

A core requirement of dealing with packages is the ability to work with versions. PEP 440 defines the standard version scheme for Python packages which has been implemented by this module.

2.1.1 Usage

```
>>> from packaging.version import Version, parse
>>> v1 = parse("1.0a5")
>>> v2 = Version("1.0")
>>> v1
<Version('1.0a5')>
>>> v2
<Version('1.0')>
>>> v1 < v2
True
>>> v1.epoch
0
>>> v1.release
(1, 0)
>>> v1.pre
('a', 5)
>>> v1.is_prerelease
True
>>> v2.is_prerelease
False
>>> Version("french toast")
Traceback (most recent call last):
    ...
InvalidVersion: Invalid version: 'french toast'
>>> Version("1.0").post
```

(continues on next page)

(continued from previous page)

```
>>> Version("1.0").is_postrelease
False
>>> Version("1.0.post0").post
0
>>> Version("1.0.post0").is_postrelease
True
```

2.1.2 Reference

packaging.version.**parse**(*version*)

This function takes a version string and will parse it as a *Version* if the version is a valid PEP 440 version, otherwise it will parse it as a *LegacyVersion*.

class packaging.version.**Version**(*version*)

This class abstracts handling of a project's versions. It implements the scheme defined in PEP 440. A *Version* instance is comparison aware and can be compared and sorted using the standard Python interfaces.

Parameters **version** (*str*) – The string representation of a version which will be parsed and normalized before use.

Raises ***InvalidVersion*** – If the `version` does not conform to PEP 440 in any way then this exception will be raised.

public

A string representing the public version portion of this `Version()`.

base_version

A string representing the base version of this *Version* instance. The base version is the public version of the project without any pre or post release markers.

epoch

An integer giving the version epoch of this *Version* instance

release

A tuple of integers giving the components of the release segment of this *Version* instance; that is, the `1.2.3` part of the version number, including trailing zeroes but not including the epoch or any prerelease/development/postrelease suffixes

local

A string representing the local version portion of this `Version()` if it has one, or `None` otherwise.

pre

If this *Version* instance represents a prerelease, this attribute will be a pair of the prerelease phase (the string `"a"`, `"b"`, or `"rc"`) and the prerelease number (an integer). If this instance is not a prerelease, the attribute will be *None*.

is_prerelease

A boolean value indicating whether this *Version* instance represents a prerelease and/or development release.

dev

If this *Version* instance represents a development release, this attribute will be the development release number (an integer); otherwise, it will be *None*.

is_devrelease

A boolean value indicating whether this *Version* instance represents a development release.

post
If this *Version* instance represents a postrelease, this attribute will be the postrelease number (an integer); otherwise, it will be *None*.

is_postrelease
A boolean value indicating whether this *Version* instance represents a post-release.

class packaging.version.**LegacyVersion**(*version*)
This class abstracts handling of a project's versions if they are not compatible with the scheme defined in PEP 440. It implements a similar interface to that of *Version*.

This class implements the previous de facto sorting algorithm used by setuptools, however it will always sort as less than a *Version* instance.

Parameters **version** (*str*) – The string representation of a version which will be used as is.

public
A string representing the public version portion of this *LegacyVersion*. This will always be the entire version string.

base_version
A string representing the base version portion of this *LegacyVersion* instance. This will always be the entire version string.

epoch
This will always be `-1` since without PEP 440 we do not have the concept of version epochs. The value reflects the fact that *LegacyVersion* instances always compare less than *Version* instances.

release
This will always be `None` since without PEP 440 we do not have the concept of a release segment or its components. It exists primarily to allow a *LegacyVersion* to be used as a stand in for a *Version*.

local
This will always be `None` since without PEP 440 we do not have the concept of a local version. It exists primarily to allow a *LegacyVersion* to be used as a stand in for a *Version*.

pre
This will always be `None` since without PEP 440 we do not have the concept of a prerelease. It exists primarily to allow a *LegacyVersion* to be used as a stand in for a *Version*.

is_prerelease
A boolean value indicating whether this *LegacyVersion* represents a prerelease and/or development release. Since without PEP 440 there is no concept of pre or dev releases this will always be *False* and exists for compatibility with *Version*.

dev
This will always be `None` since without PEP 440 we do not have the concept of a development release. It exists primarily to allow a *LegacyVersion* to be used as a stand in for a *Version*.

is_devrelease
A boolean value indicating whether this *LegacyVersion* represents a development release. Since without PEP 440 there is no concept of dev releases this will always be *False* and exists for compatibility with *Version*.

post
This will always be `None` since without PEP 440 we do not have the concept of a postrelease. It exists primarily to allow a *LegacyVersion* to be used as a stand in for a *Version*.

is_postrelease
A boolean value indicating whether this *LegacyVersion* represents a post-release. Since without PEP

440 there is no concept of post-releases this will always be `False` and exists for compatibility with *`Version`*.

exception packaging.version.**InvalidVersion**
Raised when attempting to create a *`Version`* with a version string that does not conform to PEP 440.

packaging.version.**VERSION_PATTERN**
A string containing the regular expression used to match a valid version. The pattern is not anchored at either end, and is intended for embedding in larger expressions (for example, matching a version number as part of a file name). The regular expression should be compiled with the `re.VERBOSE` and `re.IGNORECASE` flags set.

2.2 Specifiers

A core requirement of dealing with dependency is the ability to specify what versions of a dependency are acceptable for you. PEP 440 defines the standard specifier scheme which has been implemented by this module.

2.2.1 Usage

```
>>> from packaging.specifiers import SpecifierSet
>>> from packaging.version import Version
>>> spec1 = SpecifierSet("~=1.0")
>>> spec1
<SpecifierSet('~=1.0')>
>>> spec2 = SpecifierSet(">=1.0")
>>> spec2
<SpecifierSet('>=1.0')>
>>> # We can combine specifiers
>>> combined_spec = spec1 & spec2
>>> combined_spec
<SpecifierSet('>=1.0,~=1.0')>
>>> # We can also implicitly combine a string specifier
>>> combined_spec &= "!=1.1"
>>> combined_spec
<SpecifierSet('!=1.1,>=1.0,~=1.0')>
>>> # Create a few versions to check for contains.
>>> v1 = Version("1.0a5")
>>> v2 = Version("1.0")
>>> # We can check a version object to see if it falls within a specifier
>>> v1 in combined_spec
False
>>> v2 in combined_spec
True
>>> # We can even do the same with a string based version
>>> "1.4" in combined_spec
True
>>> # Finally we can filter a list of versions to get only those which are
>>> # contained within our specifier.
>>> list(combined_spec.filter([v1, v2, "1.4"]))
[<Version('1.0')>, '1.4']
```

2.2.2 Reference

class packaging.specifiers.**SpecifierSet**(*specifiers*, *prereleases=None*)

This class abstracts handling specifying the dependencies of a project. It can be passed a single specifier (`>=3.0`), a comma-separated list of specifiers (`>=3.0,!=3.1`), or no specifier at all. Each individual specifier be attempted to be parsed as a PEP 440 specifier (*Specifier*) or as a legacy, setuptools style specifier (*LegacySpecifier*). You may combine *SpecifierSet* instances using the `&` operator (`SpecifierSet(">2") & SpecifierSet("<4")`).

Both the membership tests and the combination support using raw strings in place of already instantiated objects.

Parameters

- **specifiers** (*str*) – The string representation of a specifier or a comma-separated list of specifiers which will be parsed and normalized before use.
- **prereleases** (*bool*) – This tells the SpecifierSet if it should accept prerelease versions if applicable or not. The default of `None` will autodetect it from the given specifiers.

Raises ***InvalidSpecifier*** – If the given `specifiers` are not parseable than this exception will be raised.

prereleases

A boolean value indicating whether this *SpecifierSet* represents a specifier that includes a pre-release versions. This can be set to either `True` or `False` to explicitly enable or disable prereleases or it can be set to `None` (the default) to enable autodetection.

__contains__(*version*)

This is the more Pythonic version of *contains()*, but does not allow you to override the `prereleases` argument. If you need that, use *contains()*.

See *contains()*.

contains(*version*, *prereleases=None*)

Determines if `version`, which can be either a version string, a `Version`, or a `LegacyVersion` object, is contained within this set of specifiers.

This will either match or not match prereleases based on the `prereleases` parameter. When `prereleases` is set to `None` (the default) it will use the `Specifier().prereleases` attribute to determine if to allow them. Otherwise it will use the boolean value of the passed in value to determine if to allow them or not.

__len__()

Returns the number of specifiers in this specifier set.

__iter__()

Returns an iterator over all the underlying *Specifier* (or *LegacySpecifier*) instances in this specifier set.

filter(*iterable*, *prereleases=None*)

Takes an iterable that can contain version strings, `Version`, and `LegacyVersion` instances and will then filter it, returning an iterable that contains only items which match the rules of this specifier object.

This method is smarter than just `filter(Specifier().contains, [...])` because it implements the rule from PEP 440 where a prerelease item SHOULD be accepted if no other versions match the given specifier.

The `prereleases` parameter functions similarly to that of the same parameter in `contains`. If the value is `None` (the default) then it will intelligently decide if to allow prereleases based on the specifier, the `Specifier().prereleases` value, and the PEP 440 rules. Otherwise it will act as a boolean which will enable or disable all prerelease versions from being included.

class packaging.specifiers.**Specifier**(*specifier*, *prereleases=None*)

This class abstracts the handling of a single PEP 440 compatible specifier. It is generally not required to instantiate this manually, preferring instead to work with *SpecifierSet*.

Parameters

- **specifier** (*str*) – The string representation of a specifier which will be parsed and normalized before use.
- **prereleases** (*bool*) – This tells the specifier if it should accept prerelease versions if applicable or not. The default of `None` will autodetect it from the given specifiers.

Raises ***InvalidSpecifier*** – If the `specifier` does not conform to PEP 440 in any way then this exception will be raised.

operator

The string value of the operator part of this specifier.

version

The string version of the version part of this specifier.

prereleases

See *SpecifierSet.prereleases*.

__contains__(*version*)

See *SpecifierSet.__contains__()*.

contains(*version*, *prereleases=None*)

See *SpecifierSet.contains()*.

filter(*iterable*, *prereleases=None*)

See *SpecifierSet.filter()*.

class packaging.specifiers.**LegacySpecifier**(*specifier*, *prereleases=None*)

This class abstracts the handling of a single legacy, setuptools style specifier. It is generally not required to instantiate this manually, preferring instead to work with *SpecifierSet*.

Parameters

- **specifier** (*str*) – The string representation of a specifier which will be parsed and normalized before use.
- **prereleases** (*bool*) – This tells the specifier if it should accept prerelease versions if applicable or not. The default of `None` will autodetect it from the given specifiers.

Raises ***InvalidSpecifier*** – If the `specifier` is not parseable then this will be raised.

operator

The string value of the operator part of this specifier.

version

The string version of the version part of this specifier.

prereleases

See *SpecifierSet.prereleases*.

__contains__(*version*)

See *SpecifierSet.__contains__()*.

contains(*version*, *prereleases=None*)

See *SpecifierSet.contains()*.

filter(*iterable*, *prereleases=None*)

See *SpecifierSet.filter()*.

exception packaging.specifiers.**InvalidSpecifier**
Raised when attempting to create a *Specifier* with a specifier string that does not conform to PEP 440.

2.3 Markers

One extra requirement of dealing with dependencies is the ability to specify if it is required depending on the operating system or Python version in use. PEP 508 defines the scheme which has been implemented by this module.

2.3.1 Usage

```
>>> from packaging.markers import Marker, UndefinedEnvironmentName
>>> marker = Marker("python_version>'2'")
>>> marker
<Marker('python_version > "2"')>
>>> # We can evaluate the marker to see if it is satisfied
>>> marker.evaluate()
True
>>> # We can also override the environment
>>> env = {'python_version': '1.5.4'}
>>> marker.evaluate(environment=env)
False
>>> # Multiple markers can be ANDed
>>> and_marker = Marker("os_name=='a' and os_name=='b'")
>>> and_marker
<Marker('os_name == "a" and os_name == "b"')>
>>> # Multiple markers can be ORed
>>> or_marker = Marker("os_name=='a' or os_name=='b'")
>>> or_marker
<Marker('os_name == "a" or os_name == "b"')>
>>> # Markers can be also used with extras, to pull in dependencies if
>>> # a certain extra is being installed
>>> extra = Marker('extra == "bar"')
>>> # Evaluating an extra marker with no environment is an error
>>> try:
...     extra.evaluate()
... except UndefinedEnvironmentName:
...     pass
>>> extra_environment = {'extra': ''}
>>> extra.evaluate(environment=extra_environment)
False
>>> extra_environment['extra'] = 'bar'
>>> extra.evaluate(environment=extra_environment)
True
```

2.3.2 Reference

class packaging.markers.**Marker**(*markers*)
This class abstracts handling markers for dependencies of a project. It can be passed a single marker or multiple markers that are ANDed or ORed together. Each marker will be parsed according to PEP 508.

Parameters **markers** (*str*) – The string representation of a marker or markers.

Raises ***InvalidMarker*** – If the given markers are not parseable, then this exception will be raised.

evaluate(*environment=None*)

Evaluate the marker given the context of the current Python process.

Parameters **environment** (*dict*) – A dictionary containing keys and values to override the detected environment.

Raises UndefinedComparison: If the marker uses a PEP 440 comparison on strings which are not valid PEP 440 versions.

Raises UndefinedEnvironmentName: If the marker accesses a value that isn't present inside of the environment dictionary.

exception packaging.markers.**InvalidMarker**
Raised when attempting to create a *Marker* with a string that does not conform to PEP 508.

exception packaging.markers.**UndefinedComparison**
Raised when attempting to evaluate a *Marker* with a PEP 440 comparison operator against values that are not valid PEP 440 versions.

exception packaging.markers.**UndefinedEnvironmentName**
Raised when attempting to evaluate a *Marker* with a value that is missing from the evaluation environment.

2.4 Requirements

Parse a given requirements line for specifying dependencies of a Python project, using PEP 508 which defines the scheme that has been implemented by this module.

2.4.1 Usage

```
>>> from packaging.requirements import Requirement
>>> simple_req = Requirement("name")
>>> simple_req
<Requirement('name')>
>>> simple_req.name
'name'
>>> simple_req.url is None
True
>>> simple_req.extras
set()
>>> simple_req.specifier
<SpecifierSet('')>
>>> simple_req.marker is None
True
>>> # Requirements can be specified with extras, specifiers and markers
>>> req = Requirement('name[foo]>=2,<3; python_version>"2.0"')
>>> req.name
'name'
>>> req.extras
{'foo'}
>>> req.specifier
<SpecifierSet('<3,>=2')>
>>> req.marker
<Marker('python_version > "2.0"')>
>>> # Requirements can also be specified with a URL, but may not specify
>>> # a version.
```

(continues on next page)

(continued from previous page)

```
>>> url_req = Requirement('name @ https://github.com/pypa ;os_name=="a"')
>>> url_req.name
'name'
>>> url_req.url
'https://github.com/pypa'
>>> url_req.extras
set()
>>> url_req.marker
<Marker('os_name == "a"')>
```

2.4.2 Reference

class packaging.requirements.**Requirement**(*requirement*)

This class abstracts handling the details of a requirement for a project. Each requirement will be parsed according to PEP 508.

Parameters **requirement** (*str*) – The string representation of a requirement.

Raises ***InvalidRequirement*** – If the given `requirement` is not parseable, then this exception will be raised.

name

The name of the requirement.

url

The URL, if any where to download the requirement from. Can be None.

extras

A set of extras that the requirement specifies.

specifier

A `SpecifierSet` of the version specified by the requirement.

marker

A `Marker` of the marker for the requirement. Can be None.

exception packaging.requirements.**InvalidRequirement**

Raised when attempting to create a *Requirement* with a string that does not conform to PEP 508.

2.5 Tags

Wheels encode the Python interpreter, ABI, and platform that they support in their filenames using *'platform compatibility tags'*_. This module provides support for both parsing these tags as well as discovering what tags the running Python interpreter supports.

2.5.1 Usage

```
>>> from packaging.tags import Tag, sys_tags
>>> import sys
>>> looking_for = Tag("py{major}".format(major=sys.version_info.major), "none", "any")
>>> supported_tags = list(sys_tags())
>>> looking_for in supported_tags
True
```

(continues on next page)

(continued from previous page)

```
>>> really_old = Tag("py1", "none", "any")
>>> wheels = {really_old, looking_for}
>>> best_wheel = None
>>> for supported_tag in supported_tags:
...     for wheel_tag in wheels:
...         if supported_tag == wheel_tag:
...             best_wheel = wheel_tag
...             break
>>> best_wheel == looking_for
True
```

2.5.2 Reference

packaging.tags.**INTERPRETER_SHORT_NAMES**
: A dictionary mapping interpreter names to their abbreviation codes (e.g. `"cpython"` is `"cp"`). All interpreter names are lower-case.

class packaging.tags.**Tag**(*interpreter*, *abi*, *platform*)
: A representation of the tag triple for a wheel. Instances are considered immutable and thus are hashable. Equality checking is also supported.

  Parameters

  - **interpreter** (*str*) – The interpreter name, e.g. `"py"` (see *INTERPRETER_SHORT_NAMES* for mapping well-known interpreter names to their short names).
  - **abi** (*str*) – The ABI that a wheel supports, e.g. `"cp37m"`.
  - **platform** (*str*) – The OS/platform the wheel supports, e.g. `"win_amd64"`.

  interpreter
  : The interpreter name.

  abi
  : The supported ABI.

  platform
  : The OS/platform.

packaging.tags.**parse_tag**(*tag*)
: Parses the provided `tag` into a set of *Tag* instances.

  Returning a set is required due to the possibility that the tag is a compressed tag set, e.g. `"py2.py3-none-any"` which supports both Python 2 and Python 3.

  Parameters **tag** (*str*) – The tag to parse, e.g. `"py3-none-any"`.

packaging.tags.**sys_tags**(*, *warn=False*)
: Yields the tags that the running interpreter supports.

  The iterable is ordered so that the best-matching tag is first in the sequence. The exact preferential order to tags is interpreter-specific, but in general the tag importance is in the order of:

  1. Interpreter
  2. Platform
  3. ABI

This order is due to the fact that an ABI is inherently tied to the platform, but platform-specific code is not necessarily tied to the ABI. The interpreter is the most important tag as it dictates basic support for any wheel.

The function returns an iterable in order to allow for the possible short-circuiting of tag generation if the entire sequence is not necessary and tag calculation happens to be expensive.

Parameters **warn** (*bool*) – Whether warnings should be logged. Defaults to `False`.

`packaging.tags.`**interpreter_name**`()`
Returns the running interpreter's name.

This typically acts as the prefix to the *interpreter* tag.

`packaging.tags.`**interpreter_version**`(*, warn=False)`
Returns the running interpreter's version.

This typically acts as the suffix to the *interpreter* tag.

`packaging.tags.`**mac_platforms**`(version=None, arch=None)`
Yields the *platform* tags for macOS.

Parameters

- **version** (*tuple*) – A two-item tuple presenting the version of macOS. Defaults to the current system's version.
- **arch** (*str*) – The CPU architecture. Defaults to the architecture of the current system, e.g. `"x86_64"`.

Note: Equivalent support for the other major platforms is purposefully not provided:

- On Windows, platform compatibility is statically specified
- On Linux, code must be run on the system itself to determine compatibility

---

`packaging.tags.`**compatible_tags**`(python_version=None, interpreter=None, platforms=None)`
Yields the tags for an interpreter compatible with the Python version specified by `python_version`.

The specific tags generated are:

- `py*-none-<platform>`
- `<interpreter>-none-any` if `interpreter` is provided
- `py*-none-any`

Parameters

- **python_version** (*Sequence*) – A one- or two-item sequence representing the compatible version of Python. Defaults to `sys.version_info[:2]`.
- **interpreter** (*str*) – The name of the interpreter (if known), e.g. `"cp38"`. Defaults to the current interpreter.
- **platforms** (*Iterable*) – Iterable of compatible platforms. Defaults to the platforms compatible with the current system.

`packaging.tags.`**cpython_tags**`(python_version=None, abis=None, platforms=None, *, warn=False)`
Yields the tags for the CPython interpreter.

The specific tags generated are:

- `cp<python_version>-<abi>-<platform>`

- cp<python_version>-abi3-<platform>
- cp<python_version>-none-<platform>
- cp<older version>-abi3-<platform> where "older version" is all older minor versions down to Python 3.2 (when abi3 was introduced)

If python_version only provides a major-only version then only user-provided ABIs via abis and the none ABI will be used.

Parameters

- **python_version** (*Sequence*) – A one- or two-item sequence representing the targetted Python version. Defaults to sys.version_info[:2].
- **abis** (*Iterable*) – Iterable of compatible ABIs. Defaults to the ABIs compatible with the current system.
- **platforms** (*Iterable*) – Iterable of compatible platforms. Defaults to the platforms compatible with the current system.
- **warn** (*bool*) – Whether warnings should be logged. Defaults to False.

packaging.tags.**generic_tags**(*interpreter=None*, *abis=None*, *platforms=None*, *, *warn=False*)

Yields the tags for an interpreter which requires no specialization.

This function should be used if one of the other interpreter-specific functions provided by this module is not appropriate (i.e. not calculating tags for a CPython interpreter).

The specific tags generated are:

- <interpreter>-<abi>-<platform>

The "none" ABI will be added if it was not explicitly provided.

Parameters

- **interpreter** (*str*) – The name of the interpreter. Defaults to being calculated.
- **abis** (*Iterable*) – Iterable of compatible ABIs. Defaults to the ABIs compatible with the current system.
- **platforms** (*Iterable*) – Iterable of compatible platforms. Defaults to the platforms compatible with the current system.
- **warn** (*bool*) – Whether warnings should be logged. Defaults to False.

2.6 Utilities

A set of small, helper utilities for dealing with Python packages.

2.6.1 Reference

packaging.utils.**canonicalize_name**(*name*)

This function takes a valid Python package name, and returns the normalized form of it.

Parameters **name** (*str*) – The name to normalize.

```
>>> from packaging.utils import canonicalize_name
>>> canonicalize_name("Django")
'django'
>>> canonicalize_name("oslo.concurrency")
'oslo-concurrency'
>>> canonicalize_name("requests")
'requests'
```

packaging.utils.**canonicalize_version**(*version*)

This function takes a string representing a package version (or a `Version` instance), and returns the normalized form of it.

Parameters **version** (*str*) – The version to normalize.

```
>>> from packaging.utils import canonicalize_version
>>> canonicalize_version('1.4.0.0.0')
'1.4'
```

CHAPTER 3

Project

3.1 Development

As an open source project, packaging welcomes contributions of all forms. The sections below will help you get started.

File bugs and feature requests on our issue tracker on GitHub. If it is a bug check out what to put in your bug report.

3.1.1 Getting started

Working on packaging requires the installation of a small number of development dependencies. To see what dependencies are required to run the tests manually, please look at the `noxfile.py` file.

Running tests

The packaging unit tests are found in the `tests/` directory and are designed to be run using pytest. pytest will discover the tests automatically, so all you have to do is:

```
$ python -m pytest
...
29204 passed, 4 skipped, 1 xfailed in 83.98 seconds
```

This runs the tests with the default Python interpreter. This also allows you to run select tests instead of the entire test suite.

You can also verify that the tests pass on other supported Python interpreters. For this we use nox, which will automatically create a virtualenv for each supported Python version and run the tests. For example:

```
$ nox -s tests
...
nox > Ran multiple sessions:
nox > * tests-2.7: success
```

(continues on next page)

(continued from previous page)

```
nox > * tests-3.4: skipped
nox > * tests-3.5: success
nox > * tests-3.6: success
nox > * tests-3.7: success
nox > * tests-3.8: success
nox > * tests-pypy: skipped
nox > * tests-pypy3: skipped
```

You may not have all the required Python versions installed, in which case you will see one or more `InterpreterNotFound` errors.

Running linters

If you wish to run the linting rules, you may use pre-commit or run `nox -s lint`.

```
$ nox -s lint
...
nox > Session lint was successful.
```

Building documentation

packaging documentation is stored in the `docs/` directory. It is written in reStructured Text and rendered using Sphinx.

Use nox to build the documentation. For example:

```
$ nox -s docs
...
nox > Session docs was successful.
```

The HTML documentation index can now be found at `docs/_build/html/index.html`.

3.1.2 Submitting patches

- Always make a new branch for your work.
- Patches should be small to facilitate easier review. Studies have shown that review quality falls off as patch size grows. Sometimes this will result in many small PRs to land a single large feature.
- Larger changes should be discussed in a ticket before submission.
- New features and significant bug fixes should be documented in the *Changelog*.
- You must have legal permission to distribute any code you contribute and it must be available under both the BSD and Apache Software License Version 2.0 licenses.

If you believe you've identified a security issue in packaging, please follow the directions on the *security page*.

Code

This project's source is auto-formatted with `black`. You can check if your code meets our requirements by running our linters against it with `nox -s lint` or `pre-commit run --all-files`.

Write comments as complete sentences.

Every code file must start with the boilerplate licensing notice:

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.
```

Additionally, every Python code file must contain

```
from __future__ import absolute_import, division, print_function
```

Tests

All code changes must be accompanied by unit tests with 100% code coverage (as measured by the combined metrics across our build matrix).

Documentation

All features should be documented with prose in the `docs` section.

When referring to a hypothetical individual (such as "a person receiving an encrypted message") use gender neutral pronouns (they/them/their).

Docstrings are typically only used when writing abstract classes, but should be written like this if required:

```
def some_function(some_arg):
    """
    Does some things.

    :param some_arg: Some argument.
    """
```

So, specifically:

- Always use three double quotes.
- Put the three double quotes on their own line.
- No blank line at the end.
- Use Sphinx parameter/attribute documentation syntax.

3.1.3 Reviewing and merging patches

Everyone is encouraged to review open pull requests. We only ask that you try and think carefully, ask questions and are excellent to one another. Code review is our opportunity to share knowledge, design ideas and make friends.

When reviewing a patch try to keep each of these concepts in mind:

Architecture

- Is the proposed change being made in the correct place?

Intent

- What is the change being proposed?
- Do we want this feature or is the bug they're fixing really a bug?

Implementation

- Does the change do what the author claims?
- Are there sufficient tests?
- Has it been documented?
- Will this change introduce new bugs?

Grammar and style

These are small things that are not caught by the automated style checkers.

- Does a variable need a better name?
- Should this be a keyword argument?

3.1.4 Release Process

1. Checkout the current `master` branch.
2. Install the latest `nox`:

```
$ pip install nox
```

3. Modify the `CHANGELOG.rst` to include changes made since the last release and update the section header for the new release.
4. Run the release automation with the required version number (YY.N):

```
$ nox -s release -- YY.N
```

5. Modify the `CHANGELOG.rst` to reflect the development version does not have any changes since the last release.
6. Notify the other project owners of the release.

Note: Access needed for making the release are:

- PyPI maintainer (or owner) access to *packaging*
- push directly to the *master* branch on the source repository
- push tags directly to the source repository

3.2 Security

We take the security of packaging seriously. If you believe you've identified a security issue in it, DO NOT report the issue in any public forum, including (but not limited to):

- GitHub issue tracker
- Official or unofficial chat channels
- Official or unofficial mailing lists

Please report your issue to `security@python.org`. Messages may be optionally encrypted with GPG using key fingerprints available at the Python Security page.

Once you've submitted an issue via email, you should receive an acknowledgment within 48 hours, and depending on the action to be taken, you may receive further follow-up emails.

3.3 Changelog

3.3.1 *unreleased*

- Fix a bug that caused a 32-bit OS that runs on a 64-bit ARM CPU (e.g. ARM-v8, aarch64), to report the wrong bitness.

3.3.2 20.1 - 2020-01-24

- Fix a bug caused by reuse of an exhausted iterator. (#257)

3.3.3 20.0 - 2020-01-06

- Add type hints (#191)
- Add proper trove classifiers for PyPy support (#198)
- Scale back depending on `ctypes` for manylinux support detection (#171)
- Use `sys.implementation.name` where appropriate for `packaging.tags` (#193)
- Expand upon the API provded by `packaging.tags`: `interpreter_name()`, `mac_platforms()`, `compatible_tags()`, `cpython_tags()`, `generic_tags()` (#187)
- Officially support Python 3.8 (#232)
- Add `major`, `minor`, and `micro` aliases to `packaging.version.Version` (#226)
- Properly mark `packaging` has being fully typed by adding a *py.typed* file (#226)

3.3.4 19.2 - 2019-09-18

- Remove dependency on `attrs` (#178, #179)
- Use appropriate fallbacks for CPython ABI tag (#181, #185)
- Add manylinux2014 support (#186)
- Improve ABI detection (#181)

- Properly handle debug wheels for Python 3.8 (#172)
- Improve detection of debug builds on Windows (#194)

3.3.5 19.1 - 2019-07-30

- Add the `packaging.tags` module. (#156)
- Correctly handle two-digit versions in `python_version` (#119)

3.3.6 19.0 - 2019-01-20

- Fix string representation of PEP 508 direct URL requirements with markers.
- Better handling of file URLs

  This allows for using `file:///absolute/path`, which was previously prevented due to the missing `netloc`.

  This allows for all file URLs that `urlunparse` turns back into the original URL to be valid.

3.3.7 18.0 - 2018-09-26

- Improve error messages when invalid requirements are given. (#129)

3.3.8 17.1 - 2017-02-28

- Fix `utils.canonicalize_version` when supplying non PEP 440 versions.

3.3.9 17.0 - 2017-02-28

- Drop support for python 2.6, 3.2, and 3.3.
- Define minimal pyparsing version to 2.0.2 (#91).
- Add `epoch`, `release`, `pre`, `dev`, and `post` attributes to `Version` and `LegacyVersion` (#34).
- Add `Version().is_devrelease` and `LegacyVersion().is_devrelease` to make it easy to determine if a release is a development release.
- Add `utils.canonicalize_version` to canonicalize version strings or `Version` instances (#121).

3.3.10 16.8 - 2016-10-29

- Fix markers that utilize `in` so that they render correctly.
- Fix an erroneous test on Python RC releases.

3.3.11 16.7 - 2016-04-23

- Add support for the deprecated `python_implementation` marker which was an undocumented setuptools marker in addition to the newer markers.

3.3.12 16.6 - 2016-03-29

- Add support for the deprecated, PEP 345 environment markers in addition to the newer markers.

3.3.13 16.5 - 2016-02-26

- Fix a regression in parsing requirements with whitespaces between the comma separators.

3.3.14 16.4 - 2016-02-22

- Fix a regression in parsing requirements like `foo (==4)`.

3.3.15 16.3 - 2016-02-21

- Fix a bug where `packaging.requirements:Requirement` was overly strict when matching legacy requirements.

3.3.16 16.2 - 2016-02-09

- Add a function that implements the name canonicalization from PEP 503.

3.3.17 16.1 - 2016-02-07

- Implement requirement specifiers from PEP 508.

3.3.18 16.0 - 2016-01-19

- Relicense so that packaging is available under *either* the Apache License, Version 2.0 or a 2 Clause BSD license.
- Support installation of packaging when only distutils is available.
- Fix == comparison when there is a prefix and a local version in play. (#41).
- Implement environment markers from PEP 508.

3.3.19 15.3 - 2015-08-01

- Normalize post-release spellings for rev/r prefixes. #35

3.3.20 15.2 - 2015-05-13

- Fix an error where the arbitary specifier (===) was not correctly allowing pre-releases when it was being used.
- Expose the specifier and version parts through properties on the `Specifier` classes.
- Allow iterating over the `SpecifierSet` to get access to all of the `Specifier` instances.
- Allow testing if a version is contained within a specifier via the `in` operator.

3.3.21 15.1 - 2015-04-13

- Fix a logic error that was causing inconsistent answers about whether or not a pre-release was contained within a `SpecifierSet` or not.

3.3.22 15.0 - 2015-01-02

- Add `Version().is_postrelease` and `LegacyVersion().is_postrelease` to make it easy to determine if a release is a post release.
- Add `Version().base_version` and `LegacyVersion().base_version` to make it easy to get the public version without any pre or post release markers.
- Support the update to PEP 440 which removed the implied `!=V.*` when using either `>V` or `<V` and which instead special cased the handling of pre-releases, post-releases, and local versions when using `>V` or `<V`.

3.3.23 14.5 - 2014-12-17

- Normalize release candidates as `rc` instead of `c`.
- Expose the `VERSION_PATTERN` constant, a regular expression matching a valid version.

3.3.24 14.4 - 2014-12-15

- Ensure that versions are normalized before comparison when used in a specifier with a less than (<) or greater than (>) operator.

3.3.25 14.3 - 2014-11-19

- **BACKWARDS INCOMPATIBLE** Refactor specifier support so that it can sanely handle legacy specifiers as well as PEP 440 specifiers.
- **BACKWARDS INCOMPATIBLE** Move the specifier support out of `packaging.version` into `packaging.specifiers`.

3.3.26 14.2 - 2014-09-10

- Add prerelease support to `Specifier`.
- Remove the ability to do `item in Specifier()` and replace it with `Specifier().contains(item)` in order to allow flags that signal if a prerelease should be accepted or not.
- Add a method `Specifier().filter()` which will take an iterable and returns an iterable with items that do not match the specifier filtered out.

3.3.27 14.1 - 2014-09-08

- Allow `LegacyVersion` and `Version` to be sorted together.
- Add `packaging.version.parse()` to enable easily parsing a version string as either a `Version` or a `LegacyVersion` depending on it's PEP 440 validity.

3.3.28 14.0 - 2014-09-05

- Initial release.

Index

Symbols

A

B

C

D

E

F

G

I