Packaging

Release 24.0

unknown

Mar 10, 2024

API DOCUMENTATION

Reusable core utilities for various Python Packaging interoperability specifications.

This library provides utilities that implement the interoperability specifications which have clearly one correct behaviour (eg: **PEP 440**) or benefit greatly from having a single shared implementation (eg: **PEP 425**).

CHAPTER

ONE

INSTALLATION

You can install packaging with `pip`:

```
$ pip install packaging
```

The `packaging` library uses calendar-based versioning (`YY.N`).

1.1 Version Handling

A core requirement of dealing with packages is the ability to work with versions.

See Version Specifiers Specification for more details on the exact format implemented in this module, for use in Python Packaging tooling.

1.1.1 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'
```

(continues on next page)

(continued from previous page)

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

1.1.2 Reference

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.

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

Parse the given version string.

```
>>> parse('1.0.dev1')
<Version('1.0.dev1')>
```

Parameters
version (*str*) – The version string to parse.

Raises
InvalidVersion – When the version string is not a valid version.

Return type
Version

class packaging.version.**Version**

This class abstracts handling of a project's versions.

A *Version* instance is comparison aware and can be compared and sorted using the standard Python interfaces.

```
>>> v1 = Version("1.0a5")
>>> v2 = Version("1.0")
>>> v1
<Version('1.0a5')>
>>> v2
<Version('1.0')>
>>> v1 < v2
True
>>> v1 == v2
False
>>> v1 > v2
False
>>> v1 >= v2
False
```

(continues on next page)

(continued from previous page)

```
>>> v1 <= v2
True
```

__init__(*version*)

Initialize a Version object.

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.

Return type
None

__repr__()

A representation of the Version that shows all internal state.

```
>>> Version('1.0.0')
<Version('1.0.0')>
```

Return type
str

__str__()

A string representation of the version that can be rounded-tripped.

```
>>> str(Version("1.0a5"))
'1.0a5'
```

Return type
str

property epoch: int

The epoch of the version.

```
>>> Version("2.0.0").epoch
0
>>> Version("1!2.0.0").epoch
1
```

property release: Tuple[int, ...]

The components of the "release" segment of the version.

```
>>> Version("1.2.3").release
(1, 2, 3)
>>> Version("2.0.0").release
(2, 0, 0)
>>> Version("1!2.0.0.post0").release
(2, 0, 0)
```

Includes trailing zeroes but not the epoch or any pre-release / development / post-release suffixes.

property pre: `Tuple[str, int] | None`

The pre-release segment of the version.

```
>>> print(Version("1.2.3").pre)
None
>>> Version("1.2.3a1").pre
('a', 1)
>>> Version("1.2.3b1").pre
('b', 1)
>>> Version("1.2.3rc1").pre
('rc', 1)
```

property post: `int | None`

The post-release number of the version.

```
>>> print(Version("1.2.3").post)
None
>>> Version("1.2.3.post1").post
1
```

property dev: `int | None`

The development number of the version.

```
>>> print(Version("1.2.3").dev)
None
>>> Version("1.2.3.dev1").dev
1
```

property local: `str | None`

The local version segment of the version.

```
>>> print(Version("1.2.3").local)
None
>>> Version("1.2.3+abc").local
'abc'
```

property public: `str`

The public portion of the version.

```
>>> Version("1.2.3").public
'1.2.3'
>>> Version("1.2.3+abc").public
'1.2.3'
>>> Version("1.2.3+abc.dev1").public
'1.2.3'
```

property base_version: `str`

The "base version" of the version.

```
>>> Version("1.2.3").base_version
'1.2.3'
```

(continues on next page)

(continued from previous page)

```
>>> Version("1.2.3+abc").base_version
'1.2.3'
>>> Version("1!1.2.3+abc.dev1").base_version
'1!1.2.3'
```

The "base version" is the public version of the project without any pre or post release markers.

property is_prerelease: bool

Whether this version is a pre-release.

```
>>> Version("1.2.3").is_prerelease
False
>>> Version("1.2.3a1").is_prerelease
True
>>> Version("1.2.3b1").is_prerelease
True
>>> Version("1.2.3rc1").is_prerelease
True
>>> Version("1.2.3dev1").is_prerelease
True
```

property is_postrelease: bool

Whether this version is a post-release.

```
>>> Version("1.2.3").is_postrelease
False
>>> Version("1.2.3.post1").is_postrelease
True
```

property is_devrelease: bool

Whether this version is a development release.

```
>>> Version("1.2.3").is_devrelease
False
>>> Version("1.2.3.dev1").is_devrelease
True
```

property major: int

The first item of *release* or `0` if unavailable.

```
>>> Version("1.2.3").major
1
```

property minor: int

The second item of *release* or `0` if unavailable.

```
>>> Version("1.2.3").minor
2
>>> Version("1").minor
0
```

property micro: int

The third item of *release* or `0` if unavailable.

```
>>> Version("1.2.3").micro
3
>>> Version("1").micro
0
```

exception packaging.version.**InvalidVersion**

Raised when a version string is not a valid version.

```
>>> Version("invalid")
Traceback (most recent call last):
    ...
packaging.version.InvalidVersion: Invalid version: 'invalid'
```

__weakref__

list of weak references to the object (if defined)

1.2 Specifiers

A core requirement of dealing with dependencies is the ability to specify what versions of a dependency are acceptable for you.

See Version Specifiers Specification for more details on the exact format implemented in this module, for use in Python Packaging tooling.

1.2.1 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')>
>>> # We can iterate over the SpecifierSet to recover the
>>> # individual specifiers
>>> sorted(combined_spec, key=str)
[<Specifier('!=1.1')>, <Specifier('>=1.0')>, <Specifier('~=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
```

(continues on next page)

(continued from previous page)

```
>>> 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']
```

1.2.2 Reference

exception packaging.specifiers.**InvalidSpecifier**

Raised when attempting to create a *Specifier* with a specifier string that is invalid.

```
>>> Specifier("lolwat")
Traceback (most recent call last):
    ...
packaging.specifiers.InvalidSpecifier: Invalid specifier: 'lolwat'
```

__weakref__

list of weak references to the object (if defined)

class packaging.specifiers.**Specifier**

This class abstracts handling of version specifiers.

Tip: It is generally not required to instantiate this manually. You should instead prefer to work with *SpecifierSet* instead, which can parse comma-separated version specifiers (which is what package metadata contains).

__init__(*spec=''*, *prereleases=None*)

Initialize a Specifier instance.

Parameters

- **spec** (*str*) – The string representation of a specifier which will be parsed and normalized before use.
- **prereleases** (*bool | None*) – 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 given specifier is invalid (i.e. bad syntax).

Return type

None

property prereleases: bool

Whether or not pre-releases as a whole are allowed.

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 use default semantics.

property operator: str

The operator of this specifier.

```
>>> Specifier("==1.2.3").operator
'=='
```

property version: str

The version of this specifier.

```
>>> Specifier("==1.2.3").version
'1.2.3'
```

__repr__()

A representation of the Specifier that shows all internal state.

```
>>> Specifier('>=1.0.0')
<Specifier('>=1.0.0')>
>>> Specifier('>=1.0.0', prereleases=False)
<Specifier('>=1.0.0', prereleases=False)>
>>> Specifier('>=1.0.0', prereleases=True)
<Specifier('>=1.0.0', prereleases=True)>
```

Return type
str

__str__()

A string representation of the Specifier that can be round-tripped.

```
>>> str(Specifier('>=1.0.0'))
'>=1.0.0'
>>> str(Specifier('>=1.0.0', prereleases=False))
'>=1.0.0'
```

Return type
str

__hash__()

Returns a hash value for this Specifier-like object.

Return type
int

__eq__(*other*)

Whether or not the two Specifier-like objects are equal.

Parameters
other (*object*) – The other object to check against.

Return type
bool

The value of *prereleases* is ignored.

```
>>> Specifier("==1.2.3") == Specifier("== 1.2.3.0")
True
>>> (Specifier("==1.2.3", prereleases=False) ==
...  Specifier("==1.2.3", prereleases=True))
True
>>> Specifier("==1.2.3") == "==1.2.3"
True
>>> Specifier("==1.2.3") == Specifier("==1.2.4")
False
>>> Specifier("==1.2.3") == Specifier("~=1.2.3")
False
```

__contains__(*item*)

Return whether or not the item is contained in this specifier.

Parameters
item (*str* | Version) – The item to check for.

Return type
bool

This is used for the `in` operator and behaves the same as *contains()* with no `prereleases` argument passed.

```
>>> "1.2.3" in Specifier(">=1.2.3")
True
>>> Version("1.2.3") in Specifier(">=1.2.3")
True
>>> "1.0.0" in Specifier(">=1.2.3")
False
>>> "1.3.0a1" in Specifier(">=1.2.3")
False
>>> "1.3.0a1" in Specifier(">=1.2.3", prereleases=True)
True
```

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

Return whether or not the item is contained in this specifier.

Parameters

- **item** (Version | *str*) – The item to check for, which can be a version string or a `Version` instance.
- **prereleases** (*bool* | *None*) – Whether or not to match prereleases with this Specifier. If set to `None` (the default), it uses *prereleases* to determine whether or not prereleases are allowed.

Return type
bool

```
>>> Specifier(">=1.2.3").contains("1.2.3")
True
>>> Specifier(">=1.2.3").contains(Version("1.2.3"))
True
>>> Specifier(">=1.2.3").contains("1.0.0")
False
```

(continues on next page)

(continued from previous page)

```
>>> Specifier(">=1.2.3").contains("1.3.0a1")
False
>>> Specifier(">=1.2.3", prereleases=True).contains("1.3.0a1")
True
>>> Specifier(">=1.2.3").contains("1.3.0a1", prereleases=True)
True
```

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

Filter items in the given iterable, that match the specifier.

Parameters

- **iterable** (*Iterable[UnparsedVersionVar]*) – An iterable that can contain version strings and `Version` instances. The items in the iterable will be filtered according to the specifier.
- **prereleases** (*bool | None*) – Whether or not to allow prereleases in the returned iterator. If set to `None` (the default), it will be intelligently decide whether to allow prereleases or not (based on the *prereleases* attribute, and whether the only versions matching are prereleases).

Return type

Iterator[*UnparsedVersionVar*]

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

```
>>> list(Specifier(">=1.2.3").filter(["1.2", "1.3", "1.5a1"]))
['1.3']
>>> list(Specifier(">=1.2.3").filter(["1.2", "1.2.3", "1.3", Version("1.4")]))
['1.2.3', '1.3', <Version('1.4')>]
>>> list(Specifier(">=1.2.3").filter(["1.2", "1.5a1"]))
['1.5a1']
>>> list(Specifier(">=1.2.3").filter(["1.3", "1.5a1"], prereleases=True))
['1.3', '1.5a1']
>>> list(Specifier(">=1.2.3", prereleases=True).filter(["1.3", "1.5a1"]))
['1.3', '1.5a1']
```

class packaging.specifiers.**SpecifierSet**

This class abstracts handling of a set of version specifiers.

It can be passed a single specifier (`>=3.0`), a comma-separated list of specifiers (`>=3.0,!=3.1`), or no specifier at all.

__init__(*specifiers=''*, *prereleases=None*)

Initialize a SpecifierSet instance.

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 | None*) – 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.

Return type
None

property prereleases: bool | None

Whether or not pre-releases as a whole are allowed.

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 use default semantics.

__repr__()

A representation of the specifier set that shows all internal state.

Note that the ordering of the individual specifiers within the set may not match the input string.

```
>>> SpecifierSet('>=1.0.0,!=2.0.0')
<SpecifierSet('!=2.0.0,>=1.0.0')>
>>> SpecifierSet('>=1.0.0,!=2.0.0', prereleases=False)
<SpecifierSet('!=2.0.0,>=1.0.0', prereleases=False)>
>>> SpecifierSet('>=1.0.0,!=2.0.0', prereleases=True)
<SpecifierSet('!=2.0.0,>=1.0.0', prereleases=True)>
```

Return type
str

__str__()

A string representation of the specifier set that can be round-tripped.

Note that the ordering of the individual specifiers within the set may not match the input string.

```
>>> str(SpecifierSet(">=1.0.0,!=1.0.1"))
'!=1.0.1,>=1.0.0'
>>> str(SpecifierSet(">=1.0.0,!=1.0.1", prereleases=False))
'!=1.0.1,>=1.0.0'
```

Return type
str

__hash__()

Returns a hash value for this Specifier-like object.

Return type
int

__and__(*other*)

Return a SpecifierSet which is a combination of the two sets.

Parameters
other (SpecifierSet | *str*) – The other object to combine with.

Return type
SpecifierSet

```
>>> SpecifierSet(">=1.0.0,!=1.0.1") & '<=2.0.0,!=2.0.1'
<SpecifierSet('!=1.0.1,!=2.0.1,<=2.0.0,>=1.0.0')>
>>> SpecifierSet(">=1.0.0,!=1.0.1") & SpecifierSet('<=2.0.0,!=2.0.1')
<SpecifierSet('!=1.0.1,!=2.0.1,<=2.0.0,>=1.0.0')>
```

__eq__(*other*)

Whether or not the two SpecifierSet-like objects are equal.

Parameters
other (*object*) – The other object to check against.

Return type
bool

The value of *prereleases* is ignored.

```
>>> SpecifierSet(">=1.0.0,!=1.0.1") == SpecifierSet(">=1.0.0,!=1.0.1")
True
>>> (SpecifierSet(">=1.0.0,!=1.0.1", prereleases=False) ==
...  SpecifierSet(">=1.0.0,!=1.0.1", prereleases=True))
True
>>> SpecifierSet(">=1.0.0,!=1.0.1") == ">=1.0.0,!=1.0.1"
True
>>> SpecifierSet(">=1.0.0,!=1.0.1") == SpecifierSet(">=1.0.0")
False
>>> SpecifierSet(">=1.0.0,!=1.0.1") == SpecifierSet(">=1.0.0,!=1.0.2")
False
```

__len__()

Returns the number of specifiers in this specifier set.

Return type
int

__iter__()

Returns an iterator over all the underlying *Specifier* instances in this specifier set.

```
>>> sorted(SpecifierSet(">=1.0.0,!=1.0.1"), key=str)
[<Specifier('!=1.0.1')>, <Specifier('>=1.0.0')>]
```

Return type
Iterator[Specifier]

__contains__(*item*)

Return whether or not the item is contained in this specifier.

Parameters
item (Version | *str*) – The item to check for.

Return type
bool

This is used for the `in` operator and behaves the same as *contains()* with no `prereleases` argument passed.

```
>>> "1.2.3" in SpecifierSet(">=1.0.0,!=1.0.1")
True
>>> Version("1.2.3") in SpecifierSet(">=1.0.0,!=1.0.1")
True
>>> "1.0.1" in SpecifierSet(">=1.0.0,!=1.0.1")
False
>>> "1.3.0a1" in SpecifierSet(">=1.0.0,!=1.0.1")
False
>>> "1.3.0a1" in SpecifierSet(">=1.0.0,!=1.0.1", prereleases=True)
True
```

contains(*item*, *prereleases=None*, *installed=None*)

Return whether or not the item is contained in this SpecifierSet.

Parameters

- **item** (*Version | str*) – The item to check for, which can be a version string or a `Version` instance.
- **prereleases** (*bool | None*) – Whether or not to match prereleases with this SpecifierSet. If set to `None` (the default), it uses *prereleases* to determine whether or not prereleases are allowed.
- **installed** (*bool | None*) –

Return type

bool

```
>>> SpecifierSet(">=1.0.0,!=1.0.1").contains("1.2.3")
True
>>> SpecifierSet(">=1.0.0,!=1.0.1").contains(Version("1.2.3"))
True
>>> SpecifierSet(">=1.0.0,!=1.0.1").contains("1.0.1")
False
>>> SpecifierSet(">=1.0.0,!=1.0.1").contains("1.3.0a1")
False
>>> SpecifierSet(">=1.0.0,!=1.0.1", prereleases=True).contains("1.3.0a1")
True
>>> SpecifierSet(">=1.0.0,!=1.0.1").contains("1.3.0a1", prereleases=True)
True
```

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

Filter items in the given iterable, that match the specifiers in this set.

Parameters

- **iterable** (*Iterable[UnparsedVersionVar]*) – An iterable that can contain version strings and `Version` instances. The items in the iterable will be filtered according to the specifier.
- **prereleases** (*bool | None*) – Whether or not to allow prereleases in the returned iterator. If set to `None` (the default), it will be intelligently decide whether to allow prereleases or not (based on the *prereleases* attribute, and whether the only versions matching are prereleases).

Return type

Iterator[*UnparsedVersionVar*]

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

```
>>> list(SpecifierSet(">=1.2.3").filter(["1.2", "1.3", "1.5a1"]))
['1.3']
>>> list(SpecifierSet(">=1.2.3").filter(["1.2", "1.3", Version("1.4")]))
['1.3', <Version('1.4')>]
>>> list(SpecifierSet(">=1.2.3").filter(["1.2", "1.5a1"]))
[]
>>> list(SpecifierSet(">=1.2.3").filter(["1.3", "1.5a1"], prereleases=True))
['1.3', '1.5a1']
>>> list(SpecifierSet(">=1.2.3", prereleases=True).filter(["1.3", "1.5a1"]))
['1.3', '1.5a1']
```

An "empty" SpecifierSet will filter items based on the presence of prerelease versions in the set.

```
>>> list(SpecifierSet("").filter(["1.3", "1.5a1"]))
['1.3']
>>> list(SpecifierSet("").filter(["1.5a1"]))
['1.5a1']
>>> list(SpecifierSet("", prereleases=True).filter(["1.3", "1.5a1"]))
['1.3', '1.5a1']
>>> list(SpecifierSet("").filter(["1.3", "1.5a1"], prereleases=True))
['1.3', '1.5a1']
```

1.3 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. The specification of dependency specifiers defines the scheme which has been implemented by this module.

1.3.1 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
```

(continues on next page)

(continued from previous page)

```
>>> 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"')
>>> # You can do simple comparisons between marker objects:
>>> Marker("python_version > '3.6'") == Marker("python_version > '3.6'")
True
>>> # You can also perform simple comparisons between sets of markers:
>>> markers1 = {Marker("python_version > '3.6'"), Marker('os_name == "unix"')}
>>> markers2 = {Marker('os_name == "unix"'), Marker("python_version > '3.6'")}
>>> markers1 == markers2
True
```

1.3.2 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 the specification.

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 comparison on strings which are not valid versions per the specification of version specifiers.

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 the specification.

exception packaging.markers.**UndefinedComparison**

Raised when attempting to evaluate a *Marker* with a comparison operator against values that are not valid versions per the specification of version specifiers.

exception packaging.markers.**UndefinedEnvironmentName**

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

1.4 Requirements

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

1.4.1 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.
>>> 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"')>
>>> # You can do simple comparisons between requirement objects:
>>> Requirement("packaging") == Requirement("packaging")
True
>>> # You can also perform simple comparisons between sets of requirements:
>>> requirements1 = {Requirement("packaging"), Requirement("pip")}
>>> requirements2 = {Requirement("pip"), Requirement("packaging")}
>>> requirements1 == requirements2
True
```

Changed in version 23.2: When a requirement is specified with a URL, the *Requirement* class used to check the

URL and reject values containing invalid scheme and netloc combinations. This is no longer performed since the specification does not have such rules, and the check incorrectly disallows valid requirement strings from being parsed.

1.4.2 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 the specification.

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 the specification.

1.5 Metadata

Both source distributions and *binary distributions* (_sdists_ and _wheels_, respectively) contain files recording the core metadata for the distribution. This information is used for everything from recording the name of the distribution to the installation dependencies.

1.5.1 Usage

```
>>> from packaging.metadata import parse_email
>>> metadata = "Metadata-Version: 2.3\nName: packaging\nVersion: 24.0"
>>> raw, unparsed = parse_email(metadata)
>>> raw["metadata_version"]
'2.3'
>>> raw["name"]
'packaging'
>>> raw["version"]
'24.0'
>>> from packaging.metadata import Metadata
```

(continues on next page)

(continued from previous page)

```
>>> parsed = Metadata.from_raw(raw)
>>> parsed.name
'packaging'
>>> parsed.version
<Version('24.0')>
```

1.5.2 Reference

High Level Interface

class packaging.metadata.**Metadata**

Representation of distribution metadata.

Compared to *RawMetadata*, this class provides objects representing metadata fields instead of only using built-in types. Any invalid metadata will cause *InvalidMetadata* to be raised (with a `__cause__` attribute as appropriate).

classmethod from_raw(*data*, *, *validate=True*)

Create an instance from *RawMetadata*.

If *validate* is true, all metadata will be validated. All exceptions related to validation will be gathered and raised as an *ExceptionGroup*.

Parameters

- **data** (RawMetadata) –
- **validate** (*bool*) –

Return type
Metadata

classmethod from_email(*data*, *, *validate=True*)

Parse metadata from email headers.

If *validate* is true, the metadata will be validated. All exceptions related to validation will be gathered and raised as an *ExceptionGroup*.

Parameters

- **data** (*bytes | str*) –
- **validate** (*bool*) –

Return type
Metadata

metadata_version: _Validator[Literal['1.0', '1.1', '1.2', '2.1', '2.2', '2.3']]

Metadata-Version (required; validated to be a valid metadata version)

name: _Validator[str]

Name (required; validated using *canonicalize_name()* and its *validate* parameter)

version: _Validator[Version]

Version (required)

dynamic: _Validator[List[str] | None]

Dynamic (multiple use) (validated against core metadata field names and lowercased)

platforms: _Validator[List[str] | None]
Platform (multiple use)

supported_platforms: _Validator[List[str] | None]
Supported-Platform (multiple use)

summary: _Validator[str | None]
Summary (validated to contain no newlines)

description: _Validator[str | None]
Description

description_content_type: _Validator[str | None]
Description-Content-Type (validated)

keywords: _Validator[List[str] | None]
Keywords

home_page: _Validator[str | None]
Home-page

download_url: _Validator[str | None]
Download-URL

author: _Validator[str | None]
Author

author_email: _Validator[str | None]
Author-email

maintainer: _Validator[str | None]
Maintainer

maintainer_email: _Validator[str | None]
Maintainer-email

license: _Validator[str | None]
License

classifiers: _Validator[List[str] | None]
Classifier (multiple use)

requires_dist: _Validator[List[*Requirement*] | None]
Requires-Dist (multiple use)

requires_python: _Validator[*SpecifierSet* | None]
Requires-Python

requires_external: _Validator[List[str] | None]
Requires-External (multiple use)

project_urls: _Validator[Dict[str, str] | None]
Project-URL (multiple-use)

provides_extra: _Validator[List[*NormalizedName*] | None]
Provides-Extra (multiple use)

provides_dist: _Validator[List[str] | None]

Provides-Dist (multiple use)

obsoletes_dist: _Validator[List[str] | None]

Obsoletes-Dist (multiple use)

requires: _Validator[List[str] | None]

`Requires` (deprecated)

provides: _Validator[List[str] | None]

`Provides` (deprecated)

obsoletes: _Validator[List[str] | None]

`Obsoletes` (deprecated)

Low Level Interface

class packaging.metadata.**RawMetadata**

A dictionary of raw core metadata.

Each field in core metadata maps to a key of this dictionary (when data is provided). The key is lower-case and underscores are used instead of dashes compared to the equivalent core metadata field. Any core metadata field that can be specified multiple times or can hold multiple values in a single field have a key with a plural name. See *Metadata* whose attributes match the keys of this dictionary.

Core metadata fields that can be specified multiple times are stored as a list or dict depending on which is appropriate for the field. Any fields which hold multiple values in a single field are stored as a list.

static __new__(*cls*, */*, **args*, ***kwargs*)

packaging.metadata.**parse_email**(*data*)

Parse a distribution's metadata stored as email headers (e.g. from `METADATA`).

This function returns a two-item tuple of dicts. The first dict is of recognized fields from the core metadata specification. Fields that can be parsed and translated into Python's built-in types are converted appropriately. All other fields are left as-is. Fields that are allowed to appear multiple times are stored as lists.

The second dict contains all other fields from the metadata. This includes any unrecognized fields. It also includes any fields which are expected to be parsed into a built-in type but were not formatted appropriately. Finally, any fields that are expected to appear only once but are repeated are included in this dict.

Parameters
data (*bytes | str*) –

Return type
Tuple[RawMetadata, *Dict*[str, *List*[str]]]

Exceptions

class packaging.metadata.**InvalidMetadata**

A metadata field contains invalid data.

__init__(*field*, *message*)

Parameters

- **field** (*str*) –

- **message** (*str*) –

Return type
None

field: `str`

The name of the field that contains invalid data.

class packaging.metadata.**ExceptionGroup**

A minimal implementation of `ExceptionGroup` from Python 3.11.

If `ExceptionGroup` is already defined by Python itself, that version is used instead.

__init__(*message*, *exceptions*)

Parameters

- **message** (*str*) –
- **exceptions** (*List[Exception]*) –

Return type
None

1.6 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.

1.6.1 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
>>> 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
```

1.6.2 Reference

High Level Interface

The following functions are the main interface to the library, and are typically the only items that applications should need to reference, in order to parse and check tags.

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.

Changed in version 21.3: Added the *pp3-none-any* tag (#311).

Low Level Interface

The following functions are low-level implementation details. They should typically not be needed in application code, unless the application has specialised requirements (for example, constructing sets of supported tags for environments other than the running interpreter).

These functions capture the precise details of which environments support which tags. That information is not defined in the compatibility tag standards but is noted as being up to the implementation to provide.

`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.

`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.platform_tags(version=None, arch=None)`

Yields the *platform* tags for the running interpreter.

`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 targeted 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.

1.7 Utilities

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

1.7.1 Reference

class packaging.utils.**NormalizedName**

A `typing.NewType` of `str`, representing a normalized name.

packaging.utils.**canonicalize_name**(*name*, *validate=False*)

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

The return type is typed as *NormalizedName*. This allows type checkers to help require that a string has passed through this function before use.

If **validate** is true, then the function will check if **name** is a valid distribution name before normalizing.

Parameters

- **name** (*str*) – The name to normalize.
- **validate** (*bool*) – Check whether the name is a valid distribution name.

Raises

InvalidName – If **validate** is true and the name is not an acceptable distribution name.

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

packaging.utils.**is_normalized_name**(*name*)

Check if a name is already normalized (i.e. *canonicalize_name()* would roundtrip to the same value).

Parameters

name (*str*) – The name to check.

```
>>> from packaging.utils import is_normalized_name
>>> is_normalized_name("requests")
True
>>> is_normalized_name("Django")
False
```

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'
```

packaging.utils.**parse_wheel_filename**(*filename*)

This function takes the filename of a wheel file, and parses it, returning a tuple of name, version, build number, and tags.

The name part of the tuple is normalized and typed as *NormalizedName*. The version portion is an instance of *Version*. The build number is `()` if there is no build number in the wheel filename, otherwise a two-item tuple of an integer for the leading digits and a string for the rest of the build number. The tags portion is an instance of *Tag*.

Parameters
filename (*str*) – The name of the wheel file.

Raises
InvalidWheelFilename – If the filename in question does not follow the wheel specification.

```
>>> from packaging.utils import parse_wheel_filename
>>> from packaging.tags import Tag
>>> from packaging.version import Version
>>> name, ver, build, tags = parse_wheel_filename("foo-1.0-py3-none-any.whl")
>>> name
'foo'
>>> ver == Version('1.0')
True
>>> tags == {Tag("py3", "none", "any")}
True
>>> not build
True
```

packaging.utils.**parse_sdist_filename**(*filename*)

This function takes the filename of a sdist file (as specified in the Source distribution format documentation), and parses it, returning a tuple of the normalized name and version as represented by an instance of *Version*.

Parameters
filename (*str*) – The name of the sdist file.

Raises
InvalidSdistFilename – If the filename does not end with an sdist extension (`.zip` or `.tar.gz`), or if it does not contain a dash separating the name and the version of the distribution.

```
>>> from packaging.utils import parse_sdist_filename
>>> from packaging.version import Version
>>> name, ver = parse_sdist_filename("foo-1.0.tar.gz")
>>> name
'foo'
>>> ver == Version('1.0')
True
```

exception packaging.utils.**InvalidName**

Raised when a distribution name is invalid.

exception packaging.utils.**InvalidWheelFilename**

Raised when a file name for a wheel is invalid.

exception packaging.utils.**InvalidSdistFilename**

Raised when a source distribution file name is considered invalid.

1.8 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.

1.8.1 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-3.7: success
nox > * tests-3.8: success
nox > * tests-3.9: success
nox > * tests-3.10: success
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`.

1.8.2 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.
```

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.

1.8.3 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?

1.8.4 Release Process

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

   ```
   $ pip install nox
   ```

3. Run the release automation with the required version number (YY.N):

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

   You will need the password for your GPG key as well as an API token for PyPI.
4. Add a release on GitHub.
5. 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 `main` branch on the source repository
- push tags directly to the source repository

1.9 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.

1.10 Changelog

1.10.1 24.0 - 2024-03-10

- Do specifier matching correctly when the specifier contains an epoch number and has more components than the version (#683)
- Support the experimental `--disable-gil` builds in packaging.tags (#727)
- BREAKING: Make optional `metadata.Metadata` attributes default to `None` (#733)
- Fix errors when trying to access the `description_content_type`, `keywords`, and `requires_python` attributes on `metadata.Metadata` when those values have not been provided (#733)
- Fix a bug preventing the use of the built in `ExceptionGroup` on versions of Python that support it (#725)

1.10.2 23.2 - 2023-10-01

- Document calendar-based versioning scheme (#716)
- Enforce that the entire marker string is parsed (#687)
- Requirement parsing no longer automatically validates the URL (#120)
- Canonicalize names for requirements comparison (#644)
- Introduce `metadata.Metadata` (along with `metadata.ExceptionGroup` and `metadata.InvalidMetadata`; #570)
- Introduce the `validate` keyword parameter to `utils.normalize_name()` (#570)
- Introduce `utils.is_normalized_name()` (#570)
- Make `utils.parse_sdist_filename()` and `utils.parse_wheel_filename()` raise `InvalidSdistFilename` and `InvalidWheelFilename`, respectively, when the version component of the name is invalid

1.10.3 23.1 - 2023-04-12

- Parse raw metadata (#671)
- Import underlying parser functions as an underscored variable (#663)
- Improve error for local version label with unsupported operators (#675)
- Add dedicated error for specifiers with incorrect .* suffix
- Replace spaces in platform names with underscores (#620)
- Relax typing of `_key` on `_BaseVersion` (#669)
- Handle prefix match with zeros at end of prefix correctly (#674)

1.10.4 23.0 - 2023-01-08

- Allow `"extra"` to be `None` in the marker environment (#650)
- Refactor `tags._generic_api` to use `EXT_SUFFIX` (#607)
- Correctly handle trailing whitespace on URL requirements (#642)
- Fix typing for `specifiers.BaseSpecifier.filter()` (#643)
- Use stable Python 3.11 in tests (#641)
- Correctly handle non-normalised specifiers in requirements (#634)
- Move to `src/` layout (#626)
- Remove `__about__` file, in favour of keeping constants in `__init__` (#626)

1.10.5 22.0 - 2022-12-07

- Explicitly declare support for Python 3.11 (#587)
- Remove support for Python 3.6 (#500)
- Remove `LegacySpecifier` and `LegacyVersion` (#407)
- Add `__hash__` and `__eq__` to `Requirement` (#499)
- Add a `cpNNN-none-any` tag (#541)
- Adhere to **PEP 685** when evaluating markers with extras (#545)
- Allow accepting locally installed prereleases with `SpecifierSet` (#515)
- Allow pre-release versions in marker evaluation (#523)
- Correctly parse ELF for musllinux on Big Endian (#538)
- Document `packaging.utils.NormalizedName` (#565)
- Document exceptions raised by functions in `packaging.utils` (#544)
- Fix compatible version specifier incorrectly strip trailing `0` (#493)
- Fix macOS platform tags with old macOS SDK (#513)
- Forbid prefix version matching on pre-release/post-release segments (#563)
- Normalize specifier version for prefix matching (#561)
- Improve documentation for `packaging.specifiers` and `packaging.version`. (#572)
- `Marker.evaluate` will now assume evaluation environment with empty `extra`. Evaluating markers like `"extra == 'xyz'"` without passing any extra in the `environment` will no longer raise an exception (#550)
- Remove dependency on `pyparsing`, by replacing it with a hand-written parser. This package now has no runtime dependencies (#468)
- Update return type hint for `Specifier.filter` and `SpecifierSet.filter` to use `Iterator` instead of `Iterable` (#584)

1.10.6 21.3 - 2021-11-17

- Add a `pp3-none-any` tag (#311)
- Replace the blank pyparsing 3 exclusion with a 3.0.5 exclusion (#481, #486)
- Fix a spelling mistake (#479)

1.10.7 21.2 - 2021-10-29

- Update documentation entry for 21.1.

1.10.8 21.1 - 2021-10-29

- Update pin to pyparsing to exclude 3.0.0.

1.10.9 21.0 - 2021-07-03

- PEP 656: musllinux support (#411)
- Drop support for Python 2.7, Python 3.4 and Python 3.5.
- Replace distutils usage with sysconfig (#396)
- Add support for zip files in `parse_sdist_filename` (#429)
- Use cached `_hash` attribute to short-circuit tag equality comparisons (#417)
- Specify the default value for the `specifier` argument to `SpecifierSet` (#437)
- Proper keyword-only "warn" argument in packaging.tags (#403)
- Correctly remove prerelease suffixes from ~= check (#366)
- Fix type hints for `Version.post` and `Version.dev` (#393)
- Use typing alias `UnparsedVersion` (#398)
- Improve type inference for `packaging.specifiers.filter()` (#430)
- Tighten the return type of `canonicalize_version()` (#402)

1.10.10 20.9 - 2021-01-29

- Run isort over the code base (#377)
- Add support for the `macosx_10_*_universal2` platform tags (#379)
- Introduce `packaging.utils.parse_wheel_filename()` and `parse_sdist_filename()` (#387 and #389)

1.10.11 20.8 - 2020-12-11

- Revert back to setuptools for compatibility purposes for some Linux distros (#363)
- Do not insert an underscore in wheel tags when the interpreter version number is more than 2 digits (#372)

1.10.12 20.7 - 2020-11-28

No unreleased changes.

1.10.13 20.6 - 2020-11-28

Note: This release was subsequently yanked, and these changes were included in 20.7.

- Fix flit configuration, to include LICENSE files (#357)
- Make *intel* a recognized CPU architecture for the *universal* macOS platform tag (#361)
- Add some missing type hints to *packaging.requirements* (issue:*350*)

1.10.14 20.5 - 2020-11-27

- Officially support Python 3.9 (#343)
- Deprecate the `LegacyVersion` and `LegacySpecifier` classes (#321)
- Handle `OSError` on non-dynamic executables when attempting to resolve the glibc version string.

1.10.15 20.4 - 2020-05-19

- Canonicalize version before comparing specifiers. (#282)
- Change type hint for `canonicalize_name` to return `packaging.utils.NormalizedName`. This enables the use of static typing tools (like mypy) to detect mixing of normalized and un-normalized names.

1.10.16 20.3 - 2020-03-05

- Fix changelog for 20.2.

1.10.17 20.2 - 2020-03-05

- 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.

1.10.18 20.1 - 2020-01-24

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

1.10.19 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 provided 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` (#225)
- Properly mark `packaging` has being fully typed by adding a *py.typed* file (#226)

1.10.20 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)

1.10.21 19.1 - 2019-07-30

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

1.10.22 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.

1.10.23 18.0 - 2018-09-26

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

1.10.24 17.1 - 2017-02-28

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

1.10.25 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).

1.10.26 16.8 - 2016-10-29

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

1.10.27 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.

1.10.28 16.6 - 2016-03-29

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

1.10.29 16.5 - 2016-02-26

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

1.10.30 16.4 - 2016-02-22

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

1.10.31 16.3 - 2016-02-21

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

1.10.32 16.2 - 2016-02-09

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

1.10.33 16.1 - 2016-02-07

- Implement requirement specifiers from PEP 508.

1.10.34 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.

1.10.35 15.3 - 2015-08-01

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

1.10.36 15.2 - 2015-05-13

- Fix an error where the arbitrary 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.

1.10.37 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.

1.10.38 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`.

1.10.39 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.

1.10.40 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.

1.10.41 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`.

1.10.42 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.

1.10.43 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.

1.10.44 14.0 - 2014-09-05

- Initial release.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

S

T

U

V