Pluggable Distributions of Python Software

==========================================

Distributions

-------------

A "Distribution" is a collection of files that represent a "Release" of a

"Project" as of a particular point in time, denoted by a

"Version"::

>>> import sys, pkg\_resources

>>> from pkg\_resources import Distribution

>>> Distribution(project\_name="Foo", version="1.2")

Foo 1.2

Distributions have a location, which can be a filename, URL, or really anything

else you care to use::

>>> dist = Distribution(

... location="http://example.com/something",

... project\_name="Bar", version="0.9"

... )

>>> dist

Bar 0.9 (http://example.com/something)

Distributions have various introspectable attributes::

>>> dist.location

'http://example.com/something'

>>> dist.project\_name

'Bar'

>>> dist.version

'0.9'

>>> dist.py\_version == '{}.{}'.format(\*sys.version\_info)

True

>>> print(dist.platform)

None

Including various computed attributes::

>>> from pkg\_resources import parse\_version

>>> dist.parsed\_version == parse\_version(dist.version)

True

>>> dist.key # case-insensitive form of the project name

'bar'

Distributions are compared (and hashed) by version first::

>>> Distribution(version='1.0') == Distribution(version='1.0')

True

>>> Distribution(version='1.0') == Distribution(version='1.1')

False

>>> Distribution(version='1.0') < Distribution(version='1.1')

True

but also by project name (case-insensitive), platform, Python version,

location, etc.::

>>> Distribution(project\_name="Foo",version="1.0") == \

... Distribution(project\_name="Foo",version="1.0")

True

>>> Distribution(project\_name="Foo",version="1.0") == \

... Distribution(project\_name="foo",version="1.0")

True

>>> Distribution(project\_name="Foo",version="1.0") == \

... Distribution(project\_name="Foo",version="1.1")

False

>>> Distribution(project\_name="Foo",py\_version="2.3",version="1.0") == \

... Distribution(project\_name="Foo",py\_version="2.4",version="1.0")

False

>>> Distribution(location="spam",version="1.0") == \

... Distribution(location="spam",version="1.0")

True

>>> Distribution(location="spam",version="1.0") == \

... Distribution(location="baz",version="1.0")

False

Hash and compare distribution by prio/plat

Get version from metadata

provider capabilities

egg\_name()

as\_requirement()

from\_location, from\_filename (w/path normalization)

Releases may have zero or more "Requirements", which indicate

what releases of another project the release requires in order to

function. A Requirement names the other project, expresses some criteria

as to what releases of that project are acceptable, and lists any "Extras"

that the requiring release may need from that project. (An Extra is an

optional feature of a Release, that can only be used if its additional

Requirements are satisfied.)

The Working Set

---------------

A collection of active distributions is called a Working Set. Note that a

Working Set can contain any importable distribution, not just pluggable ones.

For example, the Python standard library is an importable distribution that

will usually be part of the Working Set, even though it is not pluggable.

Similarly, when you are doing development work on a project, the files you are

editing are also a Distribution. (And, with a little attention to the

directory names used, and including some additional metadata, such a

"development distribution" can be made pluggable as well.)

>>> from pkg\_resources import WorkingSet

A working set's entries are the sys.path entries that correspond to the active

distributions. By default, the working set's entries are the items on

``sys.path``::

>>> ws = WorkingSet()

>>> ws.entries == sys.path

True

But you can also create an empty working set explicitly, and add distributions

to it::

>>> ws = WorkingSet([])

>>> ws.add(dist)

>>> ws.entries

['http://example.com/something']

>>> dist in ws

True

>>> Distribution('foo',version="") in ws

False

And you can iterate over its distributions::

>>> list(ws)

[Bar 0.9 (http://example.com/something)]

Adding the same distribution more than once is a no-op::

>>> ws.add(dist)

>>> list(ws)

[Bar 0.9 (http://example.com/something)]

For that matter, adding multiple distributions for the same project also does

nothing, because a working set can only hold one active distribution per

project -- the first one added to it::

>>> ws.add(

... Distribution(

... 'http://example.com/something', project\_name="Bar",

... version="7.2"

... )

... )

>>> list(ws)

[Bar 0.9 (http://example.com/something)]

You can append a path entry to a working set using ``add\_entry()``::

>>> ws.entries

['http://example.com/something']

>>> ws.add\_entry(pkg\_resources.\_\_file\_\_)

>>> ws.entries

['http://example.com/something', '...pkg\_resources...']

Multiple additions result in multiple entries, even if the entry is already in

the working set (because ``sys.path`` can contain the same entry more than

once)::

>>> ws.add\_entry(pkg\_resources.\_\_file\_\_)

>>> ws.entries

['...example.com...', '...pkg\_resources...', '...pkg\_resources...']

And you can specify the path entry a distribution was found under, using the

optional second parameter to ``add()``::

>>> ws = WorkingSet([])

>>> ws.add(dist,"foo")

>>> ws.entries

['foo']

But even if a distribution is found under multiple path entries, it still only

shows up once when iterating the working set:

>>> ws.add\_entry(ws.entries[0])

>>> list(ws)

[Bar 0.9 (http://example.com/something)]

You can ask a WorkingSet to ``find()`` a distribution matching a requirement::

>>> from pkg\_resources import Requirement

>>> print(ws.find(Requirement.parse("Foo==1.0"))) # no match, return None

None

>>> ws.find(Requirement.parse("Bar==0.9")) # match, return distribution

Bar 0.9 (http://example.com/something)

Note that asking for a conflicting version of a distribution already in a

working set triggers a ``pkg\_resources.VersionConflict`` error:

>>> try:

... ws.find(Requirement.parse("Bar==1.0"))

... except pkg\_resources.VersionConflict as exc:

... print(str(exc))

... else:

... raise AssertionError("VersionConflict was not raised")

(Bar 0.9 (http://example.com/something), Requirement.parse('Bar==1.0'))

You can subscribe a callback function to receive notifications whenever a new

distribution is added to a working set. The callback is immediately invoked

once for each existing distribution in the working set, and then is called

again for new distributions added thereafter::

>>> def added(dist): print("Added %s" % dist)

>>> ws.subscribe(added)

Added Bar 0.9

>>> foo12 = Distribution(project\_name="Foo", version="1.2", location="f12")

>>> ws.add(foo12)

Added Foo 1.2

Note, however, that only the first distribution added for a given project name

will trigger a callback, even during the initial ``subscribe()`` callback::

>>> foo14 = Distribution(project\_name="Foo", version="1.4", location="f14")

>>> ws.add(foo14) # no callback, because Foo 1.2 is already active

>>> ws = WorkingSet([])

>>> ws.add(foo12)

>>> ws.add(foo14)

>>> ws.subscribe(added)

Added Foo 1.2

And adding a callback more than once has no effect, either::

>>> ws.subscribe(added) # no callbacks

# and no double-callbacks on subsequent additions, either

>>> just\_a\_test = Distribution(project\_name="JustATest", version="0.99")

>>> ws.add(just\_a\_test)

Added JustATest 0.99

Finding Plugins

---------------

``WorkingSet`` objects can be used to figure out what plugins in an

``Environment`` can be loaded without any resolution errors::

>>> from pkg\_resources import Environment

>>> plugins = Environment([]) # normally, a list of plugin directories

>>> plugins.add(foo12)

>>> plugins.add(foo14)

>>> plugins.add(just\_a\_test)

In the simplest case, we just get the newest version of each distribution in

the plugin environment::

>>> ws = WorkingSet([])

>>> ws.find\_plugins(plugins)

([JustATest 0.99, Foo 1.4 (f14)], {})

But if there's a problem with a version conflict or missing requirements, the

method falls back to older versions, and the error info dict will contain an

exception instance for each unloadable plugin::

>>> ws.add(foo12) # this will conflict with Foo 1.4

>>> ws.find\_plugins(plugins)

([JustATest 0.99, Foo 1.2 (f12)], {Foo 1.4 (f14): VersionConflict(...)})

But if you disallow fallbacks, the failed plugin will be skipped instead of

trying older versions::

>>> ws.find\_plugins(plugins, fallback=False)

([JustATest 0.99], {Foo 1.4 (f14): VersionConflict(...)})

Platform Compatibility Rules

----------------------------

On the Mac, there are potential compatibility issues for modules compiled

on newer versions of macOS than what the user is running. Additionally,

macOS will soon have two platforms to contend with: Intel and PowerPC.

Basic equality works as on other platforms::

>>> from pkg\_resources import compatible\_platforms as cp

>>> reqd = 'macosx-10.4-ppc'

>>> cp(reqd, reqd)

True

>>> cp("win32", reqd)

False

Distributions made on other machine types are not compatible::

>>> cp("macosx-10.4-i386", reqd)

False

Distributions made on earlier versions of the OS are compatible, as

long as they are from the same top-level version. The patchlevel version

number does not matter::

>>> cp("macosx-10.4-ppc", reqd)

True

>>> cp("macosx-10.3-ppc", reqd)

True

>>> cp("macosx-10.5-ppc", reqd)

False

>>> cp("macosx-9.5-ppc", reqd)

False

Backwards compatibility for packages made via earlier versions of

setuptools is provided as well::

>>> cp("darwin-8.2.0-Power\_Macintosh", reqd)

True

>>> cp("darwin-7.2.0-Power\_Macintosh", reqd)

True

>>> cp("darwin-8.2.0-Power\_Macintosh", "macosx-10.3-ppc")

False

Environment Markers

-------------------

>>> from pkg\_resources import invalid\_marker as im, evaluate\_marker as em

>>> import os

>>> print(im("sys\_platform"))

Invalid marker: 'sys\_platform', parse error at ''

>>> print(im("sys\_platform=="))

Invalid marker: 'sys\_platform==', parse error at ''

>>> print(im("sys\_platform=='win32'"))

False

>>> print(im("sys=='x'"))

Invalid marker: "sys=='x'", parse error at "sys=='x'"

>>> print(im("(extra)"))

Invalid marker: '(extra)', parse error at ')'

>>> print(im("(extra"))

Invalid marker: '(extra', parse error at ''

>>> print(im("os.open('foo')=='y'"))

Invalid marker: "os.open('foo')=='y'", parse error at 'os.open('

>>> print(im("'x'=='y' and os.open('foo')=='y'")) # no short-circuit!

Invalid marker: "'x'=='y' and os.open('foo')=='y'", parse error at 'and os.o'

>>> print(im("'x'=='x' or os.open('foo')=='y'")) # no short-circuit!

Invalid marker: "'x'=='x' or os.open('foo')=='y'", parse error at 'or os.op'

>>> print(im("'x' < 'y' < 'z'"))

Invalid marker: "'x' < 'y' < 'z'", parse error at "< 'z'"

>>> print(im("r'x'=='x'"))

Invalid marker: "r'x'=='x'", parse error at "r'x'=='x"

>>> print(im("'''x'''=='x'"))

Invalid marker: "'''x'''=='x'", parse error at "'x'''=='"

>>> print(im('"""x"""=="x"'))

Invalid marker: '"""x"""=="x"', parse error at '"x"""=="'

>>> print(im(r"x\n=='x'"))

Invalid marker: "x\\n=='x'", parse error at "x\\n=='x'"

>>> print(im("os.open=='y'"))

Invalid marker: "os.open=='y'", parse error at 'os.open='

>>> em("sys\_platform=='win32'") == (sys.platform=='win32')

True

>>> em("python\_version >= '2.7'")

True

>>> em("python\_version > '2.6'")

True

>>> im("implementation\_name=='cpython'")

False

>>> im("platform\_python\_implementation=='CPython'")

False

>>> im("implementation\_version=='3.5.1'")

False