ENTWANNE

Zeste de savoir

Un usage utile du sys.meta_path en Python

jeudi 23 janvier 2025

Table des matières

Introduction

En novembre dernier je donnais une conférence à la PyConFR sur le mécanisme des imports en Python [1] et je commentais comme quoi les exemples étaient volontairement superflus car je n'avais pas d'utilisation utile à montrer.

Aujourd'hui j'en ai trouvé une au `sys.meta_path`.

1. Une histoire d'entry points

Dans le cadre d'un projet, j'exploite le mécanisme d'*entry points* de Python.

Les *entry points* (« points d'entrée ») sont des enregistrements liés à un paquet Python, qui permettent de référencer un module / une fonction du paquet pour un usage particulier.

Un cas d'usage classique est celui des *console scripts* : définir une commande `foo` sur le système qui invoque une fonction Python particulière dans notre paquet.
Ils se définissent comme suit :

```toml
[project]
name = "spam"
version = "1"

[project.entry-points.console_scripts]
foo = "spam:main"
```

Listing 1 – `pyproject.toml`[1]

1. Je prévois de le porter en article sur le site, mais c'est toujours en brouillon.
1. Dans le cas particulier des *console scripts*, on les définit habituellement plutôt dans une section `[project.scripts]` que `[project.entry-points.console_scripts]`, mais cela revient au même.

```python
def main():
    print('Hello World!')
```

Listing 2 – `spam.py`

Il suffit ensuite d'installer le paquet pour que la commande `foo` devienne disponible.

```
% pip install .
...
Successfully built spam
Installing collected packages: spam
Successfully installed spam-1
% foo
Hello World!
```

2. Un système de plugins

Dans mon cas j'utilise les *entry points* différemment : j'ai un programme principal (un paquet installé) qui se base sur des *entry points* pour gérer des plugins.
Tous les paquets installés exposant un *entry point* de type `spam.plugins` se retrouvent donc utilisables depuis le projet.
Par exemple un plugin peut être défini ainsi :

```toml
[project]
name = "foo_plugin"
version = "1"

[project.entry-points."spam.plugins"]
foo = "foo_plugin:run"
```

Listing 3 – `pyproject.toml`

```python
def run():
    print('Plugin loaded')
```

Listing 4 – `foo_plugin.py`

Une fois installé dans l'environnement courant (`pip install ...`), il est possible de trouver l'*entry point* depuis n'importe où.

```
>>> from importlib.metadata import entry_points
>>> entry_points(group='spam.plugins')
(EntryPoint(name='foo', value='foo_plugin:run',
    group='spam.plugins'),)
>>> entry_point, = entry_points(group='spam.plugins')
>>> func = entry_point.load()
>>> func
<function run at 0x7a1e2c3a8220>
>>> func()
Plugin loaded
```

3. Comment tester tout ça ?

Mais comment tester que le mécanisme de chargement de plugins est fonctionnel ? Il faudrait pour cela définir des paquets à la volée que l'on installerait ensuite à l'aide de `pip`… ça semble fastidieux et `pip` n'est pas vraiment prévu pour être invoqué depuis du code Python.
Non, il y a un autre moyen de procéder, et c'est là qu'intervient le `sys.meta_path`. Pour la faire rapidement, `sys.meta_path` définit des emplacements dynamiques qui sont interrogés par Python lorsqu'on cherche à importer un module / paquet.[1].
Mais ils ne se limitent pas à cela : ils définissent aussi comment trouver la distribution d'un paquet, c'est-à-dire toutes les métadonnées (nom, version, description, mais aussi *entry points*) qui lui sont associées.
Et donc dans mon cas je n'ai pas besoin de réellement installer de paquet depuis les tests, il me suffit qu'une entrée (un *finder*) du *meta-path* déclare que le paquet est bien installé pour qu'il le soit.
Le *finder* dispose pour cela d'une méthode `find_distributions` qui reçoit en argument un contexte qui ne nous sera pas utile ici et renvoie une liste de distributions (tous les paquets installés connus par ce *finder*).
Une distribution définit typiquement un nom et une liste d'*entry points* ; et l'*entry point* comme on l'a vu est composé d'un nom, d'un groupe (`console_scripts` / `spam.plugins`) et d'une valeur (l'action à charger).
Partant de là, je peux alors relativement facilement arriver à mes fins.

```
import math
import sys
from importlib import metadata


class FakeDistribution(metadata.Distribution):
    def __init__(self, name, *entry_points):
        self._name = name
        self._entry_points = metadata.EntryPoints(entry_points)

```

1. Mais je vous invite à consulter le support de conférence évoqué en introduction pour en savoir plus.

```
    @property
    def name(self):
        return self._name

    @property
    def entry_points(self):
        return self._entry_points


class FakeDistributionFinder(metadata.DistributionFinder):
    def __init__(self, **kwargs):
        self.distributions = [
            FakeDistribution(name, *entry_points)
            for name, entry_points in kwargs.items()
        ]

    def find_distributions(self, ctx):
        return self.distributions

    def setup(self):
        sys.meta_path.append(self)


def test_plugins():
    finder = FakeDistributionFinder(test=[metadata.EntryPoint(
        'test-math', 'math:sqrt',
        'spam.test.plugins')])
    sys.meta_path.append(finder)

    entry_point, = metadata.entry_points(group='spam.test.plugins')
    assert entry_point.name == 'test-math'
    func = entry_point.load()
    assert func is math.sqrt
    assert func(25) == 5
```

Listing 5 – `test_plugins.py`

```
% pytest test_plugins.py
test_plugins.py .
```

Conclusion

Et voilà !
Bon, je conviens que ça reste un cas d'usage à la marge, mais je suis content d'avoir trouvé à `sys.meta_path` une utilité dans un cas qui ne soit pas tordu.