Plugins & Python

Florian Strzelecki @ Python Rennes
11/01/2024

Qui suis-je ?

Florian Strzelecki

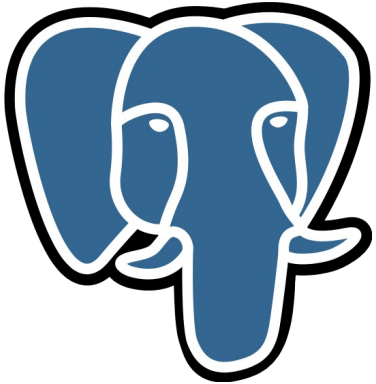
https://www.linkedin.com/in/florian-strzelecki/

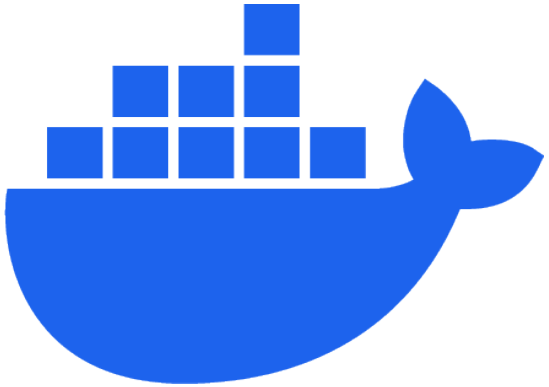
https://xrl.fr/

Software Architect, Senior Engineer, Python Expert, Lego Addict, Cinema Enthousiast, Avid Photographer, Great Dungeon Master, Casual Gamer, Passionate Stage Fencing Teacher.

I talk too much.

Qu’ont-ils en commun ?





Fonctionnel

Comment ça s’appelle ?

- Plugin
- Greffon
- Extension
- Mod
- Add-on
- Composant

- Module
- Script
- Thème
- Skin
- etc...

À quoi ça sert ?

Tout ce qui permet de **modifier**, **altérer**, **remplacer**, ou **ajouter** une fonctionnalité.

Ce n’est pas exclusif au logiciel !

- Meubles
- Outils de bricolage
- Voiture
- Engin de chantier
- Matériel médical
- Appareil photo
- Caméra cinéma
- Système de son
- Arme à feu
- Ordinateur

Que vous le vouliez ou non !

- Les utilisateurs en veulent toujours plus !
- Apporter des améliorations de la qualité de vie (QoL)
- Correction de bugs, amélioration de performance…
- … ad-blocker !

- Peuvent être des opportunités business
- Plugin d’effets audio/video payant
- Intégration avec des systèmes externes (Slack, Github, etc.)

Architecture

De quoi avez vous besoin ?

- **Identifier les points d’entrées (où intervenir)**
- **Définir les responsabilités (qui est actif)**
- **Standardiser les interactions (comment se parler)**
- **… bref, une API**

De quoi avez vous besoin ?

- Détecter
- Charger
- Initialiser
- Éxécuter
- Déinitialiser
- Désinstaller
- Découvrir
- Installer
- Lister
- Activer
- Configurer
- Désactiver

Focus du jour : le code pour…

- **Installation**
- **Détection**
- **Chargement**
- **… et un bout d’interaction !**

Développement

La méthode du fichier simple

myplugin.py

- Un **fichier** égal un **plugin**
- **Installation** : copie de fichier dans un répertoire
- **Détection** : lister les fichiers dans un répertoire
- **Chargement** : importlib (version manuelle)

importlib

`importlib` — The implementation of `import`

New in version 3.1.

Source code: Lib/importlib/__init__.py

Introduction

The purpose of the `importlib` package is three-fold.

One is to provide the implementation of the `import` statement (and thus, by extension, the `__import__()` function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an importer) to participate in the import process.

importlib

```
1   import importlib
2
3   spec = importlib.util.spec_from_file_location(
4       "myplugin",
5       "myplugin.py",
6   )
7
8   module = importlib.util.module_from_spec(spec)
9   spec.loader.exec_module(module)
10
11  module.call_something()
12
```

Avantages et inconvénients

- **Installation facile**
- **Détection simple**
- **Pas besoin de modifier le PYTHON_PATH**

- **Chargement très “à la main”, risque d’erreurs**
- **Pas de dépendances**
- **Installation manuelle**
- **Support utilisateur compliqué ! (est-ce vraiment le bon fichier ?)**

La technique des namespace packages

plugin_namespace.*

- **Déclarez un “namespace”**
- **Installation** : installation via pip (paquet python)
- **Détection** : inspection des fichiers du namespace
- **Chargement** : importlib

Namespace package ?

- **Permet de déclarer des “namespaces” utilisable par différents paquets Python**
- **À lire :**
  - PEP 420 : Implicit Namespace Package
  - https://packaging.python.org/en/latest/guides/packaging-namespace-packages/

```
mynamespace-subpackage-a/
    pyproject.toml
    src/
        mynamespace/
            subpackage_a/
                __init__.py

mynamespace-subpackage-b/
    pyproject.toml
    src/
        mynamespace/
            subpackage_b/
                __init__.py
            module_b.py
```

Importlib + pkgutil

```
import importlib
import pkgutil

import myapp.plugins

def iter_namespace(ns_pkg):
    # Specifying the second argument (prefix) to iter_modules makes the
    # returned name an absolute name instead of a relative one. This allows
    # import_module to work without having to do additional modification to
    # the name.
    return pkgutil.iter_modules(ns_pkg.__path__, ns_pkg.__name__ + ".")

discovered_plugins = {
    name: importlib.import_module(name)
    for finder, name, ispkg
    in iter_namespace(myapp.plugins)
}
```

Avantages et inconvénients

- **Installation via pip**
- **Compatibilité des versions & deps via pip**
- **Import assez simple, moins de risques d’erreur**
- **Moins de risque de mauvaises manipulations par les utilisateurs**

- **Nécessite des connaissances en packaging pour les auteurs de plugins**

Grand Master Level : Entry Point

Entry Point

- **Déclarez un “entry point”**
- **Installation** : installation via pip (paquet python)
- **Détection** : importlib.metadata
- **Chargement** : EntryPoint.load

Entry Point ?

- **Déclaration d'extension par groupes nommés**
- **À l'origine : setuptools**
- **Fonctionne avec setup.cfg & pyproject.toml**
- **À Lire :**
  - https://docs.python.org/3/library/importlib.metadata.html#entry-points
  - https://packaging.python.org/en/latest/guides/creating-and-discovering-plugins/#using-package-metadata

```
[project.entry-points.'myapp.plugins']
a = 'myapp_plugin_a'
```

Importlib.metadata.entry_points

```python
import importlib

group = 'myapp.plugin_group'
entry_points = importlib.metadata.entry_points(group=group)

modules = []
for entry_point in entry_points:
    modules.append(entry_point.load())

```

Avantages et inconvénients

- Installation via pip
- Compatibilité des versions & deps via pip
- Chargement très simple
- Moins de risque de mauvaises manipulations par les utilisateurs
- Grande flexibilité pour les auteurs de plugins

- Nécessite des connaissances en packaging pour les auteurs de plugins
- Conflit de nom !

Interaction et autres considérations

Disclaimer

- **Il existe plusieurs façons de faire.**
- **Ce sont uniquement des exemples et des idées.**
- **À vous d’adapter à vos besoins & contextes.**
- **J’ai mes opinions, et parfois elles changent !**
- **Je ne suis pas la source de vérité.**

Structure initiale

```python
class Event:
    ...

class App:
    EVENT_A: Event

    def register_event(
        self, event: Event, handler: callable
    ):
        ...

```

```python
class PluginHandler:
    def load(self):
        ...

    def setup(self, app: App):
        ...

    def on_event_a(self, app: App, event: Event):
        ...

    def on_event_b(self, app: App, event: Event):
        ...

```

App register vs Plugin register

```python
class App:

    def register_plugin_handlers(self, plugin: PluginHandler):
        for attr in dir(plugin):
            if attr == 'on_event_a':
                self.register_event(self.EVENT_A, plugin.on_event_a)
            elif attr == 'on_event_b':
                self.register_event(self.EVENT_B, plugin.on_event_b)
```

```python
class PluginHandler:

    def register(self, app: App):
        app.register_event(app.EVENT_A, self.on_event_a)
        app.register_event(app.EVENT_B, self.on_event_b)
```

Simplifier le développement de plugins

- Un framework de configuration
- Un framework de tests pour les plugins
- Typage statique des données traitées
- Donner le maximum de liberté tout en cadrant fortement les possibilités
- Une documentation claire et extensive (un tutoriel + des exemples + une doc de réference)
- Un cycle de vie clair pour les nouvelles interfaces et l’obsolescence des anciennes

L’application devient un FRAMEWORK

Pensez comme un éditeur de Framework
et pas seulement un éditeur de logiciel


Des questions ?