

La mécanique des imports

Antoine "entwanne" Rozo



La mécanique des imports

- Comprendre ce qu'il se passe lors d'un import
- Interférer sur la découverte des modules
- Modifier le comportement de l'import
- https://github.com/entwanne/presentation_imports

Qu'est-ce qu'un import ?

Qu'est-ce qu'un import ?

- Que se passe-t-il quand on fait un `import my_module` ?

```
In [3]: import my_module  
my_module
```

```
Out[3]: <module 'my_module' from '/home/antoine/Perso/presentation_imports/generated/my_module.py'>
```

Qu'est-ce qu'un import ?

- Que se passe-t-il quand on fait un `import my_module` ?

```
In [3]: import my_module  
my_module
```

```
Out[3]: <module 'my_module' from '/home/antoine/Perso/presentation_imports/generated/my_module.py'>
```

- Cela équivaut à un appel à la fonction `__import__` avec le nom du module en argument
- Dont le retour est stocké dans le nom indiqué

```
In [4]: my_module = __import__('my_module')  
my_module
```

```
Out[4]: <module 'my_module' from '/home/antoine/Perso/presentation_imports/generated/my_module.py'>
```

importlib

- L'usage de la fonction `__import__` est cependant découragé
- `import_module` d'`importlib` offre une interface plus claire, notamment dans le cas de paquets
- On préférera alors cette fonction pour un « import programmatique »

In [5]: `import importlib`

```
my_module = importlib.import_module('my_module')
my_module
```

Out[5]: <module 'my_module' from '/home/antoine/Perso/presentation_imports/generated/my_module.py'>

Exécution du module

- L'import ne fait pas que charger le module
- Il en exécute aussi le contenu

Exécution du module

- L'import ne fait pas que charger le module
- Il en exécute aussi le contenu

```
In [6]: %%writefile my_other_module.py  
print('Coucou')
```

```
Writing my_other_module.py
```

```
In [7]: import my_other_module
```

```
Coucou
```

Import de paquets et sous-modules

- Le mécanisme d'import se charge de résoudre et d'importer les paquets parents
 - Ainsi importer `foo.spam.eggs` équivaut à importer `foo` puis `foo.spam` et enfin `foo.spam.eggs`
 - Le module `__init__` de chaque paquet est chargé et exécuté

Import de paquets et sous-modules

- Par exemple ici avec une hiérarchie sur 3 niveaux

```
In [8]: %%writefile foo/__init__.py  
print('Import foo')
```

```
Writing foo/__init__.py
```

```
In [9]: %%writefile foo/spam/__init__.py  
print('Import foo.spam')
```

```
Writing foo/spam/__init__.py
```

```
In [10]: %%writefile foo/spam/eggs.py  
print('Import foo.spam.eggs')
```

```
Writing foo/spam/eggs.py
```

```
In [11]: import foo.spam.eggs
```

```
Import foo  
Import foo.spam  
Import foo.spam.eggs
```

Import de paquets et sous-modules

- Les imports relatifs (. , . . , etc.) sont aussi résolus par ce mécanisme

Import de paquets et sous-modules

- Les imports relatifs (. , .. , etc.) sont aussi résolus par ce mécanisme

```
In [12]: %%writefile foo/spam/increment.py
def increment(x):
    return x + 1
```

Writing foo/spam/increment.py

```
In [13]: %%writefile foo/spam/relative.py
from .increment import increment

print(increment(5))
```

Writing foo/spam/relative.py

```
In [14]: import foo.spam.relative
```

```
Import foo
Import foo.spam
6
```

Étapes de l'import

- Pour résumer, l'import se déroule en plusieurs étapes :
 1. Résolution du nom
 - Pour résoudre les imports relatifs
 - `importlib.util.resolve_name`
 2. Imports récursifs des paquets parents
 3. Chargement du module
 4. Exécution du code du module

Système de cache

Système de cache

- Mais recharger / réexécuter le module à chaque import serait coûteux
- Python utilise alors un cache pour se souvenir des modules précédemment importés
- L'import d'un module déjà présent dans le cache peut alors court-circuiter toute la procédure d'import

Système de cache

- Mais recharger / réexécuter le module à chaque import serait coûteux
- Python utilise alors un cache pour se souvenir des modules précédemment importés
- L'import d'un module déjà présent dans le cache peut alors court-circuiter toute la procédure d'import
- `import_module` stocke aussi son résultat dans le cache
- Ce cache est accessible via `sys.modules`

```
In [15]: import sys  
        sys.modules
```

```
Out[15]: {'sys': <module 'sys' (built-in)>,  
          'builtins': <module 'builtins' (built-in)>,  
          '_frozen_importlib': <module '_frozen_importlib' (frozen)>,  
          '_imp': <module '_imp' (built-in)>,  
          '_thread': <module '_thread' (built-in)>,  
          '_warnings': <module '_warnings' (built-in)>,  
          '_weakref': <module '_weakref' (built-in)>,  
          '_io': <module '_io' (built-in)>,  
          ...}
```

Système de cache

- Changer le code d'un module à la volée ne permet alors pas de le réimporter
- À moins d'utiliser `importlib.reload`

In [16]:

```
%%writefile rewrite.py
def version():
    return 1
```

Writing rewrite.py

In [17]:

```
import rewrite
print('before', rewrite.version())

with open('rewrite.py', 'w') as f:
    print("def version():\n        return 2", file=f)

import rewrite
print('after', rewrite.version())

importlib.reload(rewrite)
print('reload', rewrite.version())
```

before 1
after 1
reload 2

Système de cache

- Ce système de cache nous permet aussi de :
 - Simplement vérifier qu'un module a déjà été importé
 - En vérifiant s'il existe dans `sys.modules`
 - Nettoyer et/ou falsifier le cache en ajoutant des modules à la volée

```
del sys.modules[...]
importlib.reload(...)
sys.modules[...] = ...
```

Système de cache

- Dans le cadre de la présentation, le cache est nettoyé après chaque bloc de code

Recherche de modules

Recherche de modules

- Pour trouver les modules à importer, Python parcourt la liste `sys.path`

```
In [18]: sys.path
```

```
Out[18]: ['/usr/lib/python312.zip',
           '/usr/lib/python3.12',
           '/usr/lib/python3.12/lib-dynload',
           '',
           '/home/antoine/Perso/presentation_imports/env/lib/python3.12/site-packages']
```

Ajouter un répertoire

- On peut ainsi ajouter des répertoires dans `sys.path` pour permettre à Python de trouver les modules qui s'y trouvent

```
In [19]: import dir_example
```

```
-----
ModuleNotFoundError
all last)
Cell In[19], line 1
----> 1 import dir_example
```

```
Traceback (most recent c
```

```
ModuleNotFoundError: No module named 'dir_example'
```

```
In [20]: sys.path.append('subdirectory')
```

```
import dir_example
dir_example.hello('PyConFR')
```

```
DIR: Hello PyConFR
```

Ajouter un répertoire

- On peut ainsi ajouter des répertoires dans `sys.path` pour permettre à Python de trouver les modules qui s'y trouvent

```
In [19]: import dir_example
```

```
-----
ModuleNotFoundError
all last)
Cell In[19], line 1
----> 1 import dir_example
```

```
Traceback (most recent c
```

```
ModuleNotFoundError: No module named 'dir_example'
```

```
In [20]: sys.path.append('subdirectory')
```

```
import dir_example
dir_example.hello('PyConFR')
```

```
DIR: Hello PyConFR
```

- Mais on préférera laisser Python gérer ça par lui-même et utiliser les répertoires d'installation pour rendre nos modules et paquets accessibles

Ajouter un fichier zip

- De la même manière, Python est en mesure d'importer des modules depuis une archive zip

```
In [21]: %%sh  
zipinfo -1 packages.zip  
zcat packages.zip
```

```
zip_example.py  
def hello(name):  
    print('ZIP:', 'Hello', name)
```

```
In [22]: sys.path.append('packages.zip')  
  
import zip_example  
zip_example.hello('PyConFR')
```

```
ZIP: Hello PyConFR
```

Ajouter un fichier zip

- Ce mécanisme permet aussi de distribuer un paquet comme un zip

In [23]:

```
%%sh  
zipinfo calc_program.zip
```

```
Archive: calc_program.zip  
Zip file size: 923 bytes, number of entries: 4  
-rw-r--r-- 3.0 unx      71 tx defN 24-Oct-31 07:31 __main__.py  
-rw-r--r-- 3.0 unx      43 tx defN 24-Oct-31 07:32 calc/__init_.  
.py  
-rw-r--r-- 3.0 unx     184 tx defN 24-Oct-31 10:51 calc/__main_.  
.py  
-rw-r--r-- 3.0 unx      43 tx stor 24-Oct-31 07:28 calc/multipli  
cation.py  
4 files, 341 bytes uncompressed, 259 bytes compressed: 24.0%
```

In [24]:

```
%%sh  
X=3 Y=4 python calc_program.zip
```

Découverte et chargement de modules

Découverte et chargement de modules

- Python utilise des *finders* pour découvrir les modules et des *loaders* pour les charger
- `sys.path_hooks` est une liste de callables créant un *finder* pour chaque entrée de `sys.path`

```
In [25]: sys.path_hooks
```

```
Out[25]: [zipimport.zipimporter,
<function _frozen_importlib_external.FileFinder.path_hook.<locals>.path_hook_for_FileFinder(path)>]
```

Découverte et chargement de modules

- Un *finder* est un objet possédant une méthode `find_spec`
 - Cette méthode prend en argument le nom complet du module
 - Elle renvoie une « spécification de module » (`ModuleSpec`), ou `None` si le module n'est pas trouvé

```
In [26]: finder = sys.path_hooks[-1]('.').# Finder sur le répertoire courant
finder.find_spec('my_module')
```

```
Out[26]: ModuleSpec(name='my_module', loader=<_frozen_importlib_external.SourceFileLoader object at 0x781d5a92fa10>, origin='/home/antoine/Perso/presentation_imports/generated/my_module.py')
```

```
In [27]: finder.find_spec('not_found')
```

Découverte et chargement de modules

- La spécification contient des attributs décrivant le module (`name`, `origin`)

```
In [28]: spec = finder.find_spec('my_module')
spec.name, spec.origin
```

```
Out[28]: ('my_module',
          '/home/antoine/Perso/presentation_imports/generated/my_module.py')
```

- et un attribut `loader` renvoyant le *loader* associé à ce type de fichier

```
In [29]: spec.loader
```

```
Out[29]: <_frozen_importlib_external.SourceFileLoader at 0x781d5a92ff80>
```

Découverte et chargement de modules

- On peut initialiser un module vide à partir de la spec
 - cela utilise la méthode `create_module` du *loader* si elle est définie

In [30]: `import importlib.util`

```
module = importlib.util.module_from_spec(spec)
module.__dict__.keys()
```

Out[30]: `dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__', '__cached__'])`

- Et charger le module via la méthode `exec_module` du *loader*

In [31]: `spec.loader.exec_module(module)`
`module.__dict__.keys()`

Out[31]: `dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__', '__cached__', '__builtins__', 'my_function'])`

In [32]: `>>> module.my_function()`

Out[32]: `True`

Découverte et chargement de modules

- Python propose des utilitaires pour gérer différents types de *finders* et *loaders*
- `PathEntryFinder` est un *finder* dédié pour les entrées de `sys.path`
- `SourceLoader` est un *loader* offrant de facilités pour importer un fichier source
 - Un *source loader* a juste à implémenter des méthodes `get_filename` et `get_data` (qui renvoie le contenu du module sous forme de *bytes*)

Importer des .tar.gz

- On peut par exemple ajouter un *loader* pour gérer les archives .tar.gz
 - fonctionnant sur le même principe que l'import d'archives .zip

```
In [33]: %%sh  
tar -xzv0f packages.tar.gz
```

```
tar_example.py
```

```
def hello(name):  
    print('TAR:', 'Hello', name)
```

```
In [34]: sys.path.append('packages.tar.gz')  
  
import tar_example  
tar_example.hello('PyConFR')
```

```
-----  
ModuleNotFoundError  
all last)  
...  
ModuleNotFoundError: No module named 'tar_example'
```

```
Traceback (most recent c
```

Importer des .tar.gz

- Le *finder* est un `PathEntryFinder` classique

```
In [35]: import importlib.abc
import tarfile

class ArchiveFinder(importlib.abc.PathEntryFinder):
    def __init__(self, path):
        self.loader = ArchiveLoader(path)

    def find_spec(self, fullname, target=None):
        if fullname in self.loader.filenames:
            return importlib.util.spec_from_loader(fullname, self.loader)
```

Importer des .tar.gz

- Le *loaders*'occupe d'ouvrir l'archive, de localiser le module et d'en renvoyer la source

```
In [36]: class ArchiveLoader(importlib.abc.SourceLoader):  
    def __init__(self, path):  
        self.archive = tarfile.open(path, mode='r:gz')  
        self.filenames = {  
            name.removesuffix('.py'): name  
            for name in self.archive.getnames()  
            if name.endswith('.py')  
        }  
  
    def get_data(self, name):  
        member = self.archive.getmember(name)  
        fobj = self.archive.extractfile(member)  
        return fobj.read()  
  
    def get_filename(self, name):  
        return self.filenames[name]
```

Importer des .tar.gz

- Il suffit ensuite de le brancher aux `sys.path_hooks`
- Python garde en cache les *hooks* existants et il faut donc penser à nettoyer le cache

```
In [37]: def archive_path_hook(archive_path):  
    if archive_path.endswith('.tar.gz'):  
        return ArchiveFinder(archive_path)  
    raise ImportError  
  
sys.path_hooks.append(archive_path_hook)  
sys.path_importer_cache.clear()
```

```
In [38]: import tar_example  
tar_example.hello('PyConFR')
```

TAR: Hello PyConFR

Autres exemples

- On peut imaginer d'autres exemples de *path hooks*
 - Import depuis tout type d'archive, ou tout ce qui prend la forme d'une collection de fichiers
 - Import depuis le réseau (on y reviendra plus tard)

Importer de nouveaux types de fichiers

Importer de nouveaux types de fichiers

- Le *file finder* par défaut de Python gère l'import de fichiers `.py`, `.pyc` et `.so / .dll`
 - La classe `FileFinder` est pour cela instanciée en lui précisant les extensions supportées et les *loaders* associés
 - `FileFinder` permet ainsi de gérer d'autres extensions de fichiers avec d'autres *loaders*

Python++

- On peut utiliser le mécanisme des *loaders* pour étendre la syntaxe de Python
 - Par exemple en ajoutant un opérateur d'incrémentation (`++`)
 - L'idée serait que `foo++` soit transformé en `(foo := foo + 1)` au chargement du module
- `FileLoader` pourra être utilisé avec une transformation de l'entrée
 - Il ressemble à `SourceLoader` en plus minimaliste
 - On surchargera `get_source` plutôt que `get_data` (qui renvoie le contenu brut)

Python++

- Le *loaders*'occupe de lire la source et transformer les *tokens*

```
In [39]: import tokenize
```

```
class BetterPythonLoader(importlib.abc.FileLoader):  
    def get_source(self, fullname):  
        path = self.get_filename(fullname)  
        with open(path, 'rb') as f:  
            tokens = list(tokenize.tokenize(f.readline))  
        tokens = transform(tokens)  
    return tokenize.untokenize(tokens)
```

Python++

- La transformation consiste à détecter les + enchaînés après un nom et à les remplacer par une expression d'incrémentation

```
In [40]: def transform(tokens):  
    stack = []  
    for token in tokens:  
        match token.type:  
            case tokenize.NAME if not stack:  
                stack.append(token)  
            case tokenize.OP if stack and token.string == '+':  
                if len(stack) < 2:  
                    stack.append(token)  
                else:  
                    yield from increment_token(token, stack)  
            case _:  
                yield from stack  
                stack.clear()  
                yield token
```

Python++

- On produit alors les *tokens* correspondant à cette expression

```
In [41]: def increment_token(token, stack):  
    name_token = stack.pop(0)  
    stack.clear()  
  
    start = name_token.start  
    end = token.end  
    line = name_token.line  
  
    yield tokenize.TokenInfo(type=tokenize.OP, string='(', start=start, end=end)  
    yield tokenize.TokenInfo(type=tokenize.NAME, string=name_token.string, start=start, end=end)  
    yield tokenize.TokenInfo(type=tokenize.OP, string=':=', start=start, end=end)  
    yield tokenize.TokenInfo(type=tokenize.NAME, string=name_token.string, start=start, end=end)  
    yield tokenize.TokenInfo(type=tokenize.OP, string='+', start=start, end=end)  
    yield tokenize.TokenInfo(type=tokenize.NUMBER, string='1', start=start, end=end)  
    yield tokenize.TokenInfo(type=tokenize.OP, string=')', start=start, end=end)
```

Python++

- Il suffit ensuite de configurer un *finder* lié à ce *loader*

```
In [42]: path_hook = importlib.machinery.FileFinder.path_hook(  
    (importlib.machinery.SourceFileLoader, ['.py']),  
    (BetterPythonLoader, ['.pycc']),  
)  
sys.path_hooks.insert(0, path_hook)  
sys.path_importer_cache.clear()
```

Python++

- Et de tester !

```
In [43]: %%writefile increment.pycc
def test(x=0):
    for _ in range(10):
        print(x++)
```

Writing increment.pycc

```
In [44]: import increment
increment.test(4)
```

```
5
6
7
8
9
10
11
12
13
14
```

Transformer le texte lu en entrée

- On peut aussi imaginer vouloir lire (et décoder) des fichiers Python chiffrés
 - En guise de chiffrement j'utiliserais ici du rot-13 ☺
- On pourra là encore faire appel à un `FileLoader`

Transformer le texte lu en entrée

- Idem, le *loader* transforme la source et est branché à un *finder*

```
In [45]: import codecs
import importlib.machinery

class Rot13Loader(importlib.abc.FileLoader):
    def get_source(self, fullname):
        data = self.get_data(self.get_filename(fullname))
        return codecs.encode(data.decode(), 'rot_13')

path_hook = importlib.machinery.FileFinder.path_hook(
    (importlib.machinery.SourceFileLoader, ['.py']),
    (Rot13Loader, ['.pyr']),
)
sys.path_hooks.insert(0, path_hook)
sys.path_importer_cache.clear()
```

Transformer le texte lu en entrée

- Qui permet d'importer des fichiers .pyr

```
In [46]: %%writefile secret.pyr  
qrs gbgb():  
    erghea 4
```

Writing secret.pyr

```
In [47]: import secret  
secret.toto()
```

Out[47]: 4

```
In [48]: %%writefile secret2.pyr  
qrs gbgb():  
    erghea 42
```

Writing secret2.pyr

```
In [49]: import secret2  
secret2.toto()
```

Out[49]: 42

Import brainfuck

- Enfin on peut étendre le mécanisme d'imports pour gérer d'autres langages que Python
- Par exemple un interpréteur brainfuck sous forme de *loader*

In [50]:

```
import ast
import pathlib

# définition des opérateurs
OPS = {
    '>': ast.parse('cur += 1').body,
    '<': ast.parse('cur -= 1').body,
    '+': ast.parse('mem[cur] = mem.get(cur, 0) + 1').body,
    '-': ast.parse('mem[cur] = mem.get(cur, 0) - 1').body,
    '.': ast.parse('print(chr(mem.get(cur, 0)), end="")').body,
    'init': ast.parse('mem, cur = {}, 0').body,
    'test': ast.parse('mem.get(cur, 0)').body[0].value,
}
```

Import brainfuck

- On fournit un *loader* basique qui implémente juste `exec_module`

```
In [51]: class BrainfuckLoader(importlib.abc.Loader):
```

```
    def __init__(self, fullname, path):  
        self.path = pathlib.Path(path)  
  
    def exec_module(self, module):  
        content = self.path.read_text()  
        body = parse_body(content)  
        tree = parse_tree(body)  
        code = compile(tree, self.path, 'exec')  
        exec(code, module.__dict__)
```

Import brainfuck

- Et une fonction qui transforme les *tokens* brainfuck en nœuds AST Python

In [52]:

```
def parse_body(content):
    body = [*OPS['init']]
    stack = [body]
    for char in content:
        current = stack[-1]
        match char:
            case '[':
                loop = ast.While(
                    test=OPS['test'],
                    body=[ast.Pass()],
                    orelse=[],
                )
                current.append(loop)
                stack.append(loop.body)
            case ']':
                stack.pop()
            case c if c in OPS:
                current.extend(OPS[c])
            case ' ' | '\n':
                pass
            case _:
                raise SyntaxError
    return body
```

Import brainfuck

- Que l'on intègre à un AST de module contenant une fonction (`run`), ensuite compilé

```
In [53]: def parse_tree(body):
    tree = ast.Module(
        body=[

            ast.FunctionDef(
                name='run',
                args=ast.arguments(posonlyargs=[], args=[], kwonlyargs=[],
                decorator_list=[],
                body=body,
            ),
        ],
        type_ignores=[],
    )

    ast.fix_missing_locations(tree)
    return tree
```

Import brainfuck

- À nouveau le *loader* est configuré dans les *path hooks*

```
In [54]: path_hook = importlib.machinery.FileFinder.path_hook(  
    (importlib.machinery.SourceFileLoader, ['.py']),  
    (BrainfuckLoader, ['.bf']),  
)  
sys.path_hooks.insert(0, path_hook)  
sys.path_importer_cache.clear()
```

Import brainfuck

- Et permet d'importer notre fichier markdown et d'en exposer une fonction `run`

```
In [55]: %%writefile hello.bf  
+++++[>++++++>++++++>+++>+<<<-]>++.>+,+++++. .+++.>++.<<+++++
```

Writing hello.bf

```
In [56]: import hello  
hello.run()
```

Hello World!

Découvrir des modules
ailleurs que dans les
fichiers

Découvrir des modules ailleurs que dans les fichiers

- On a jusqu'ici utilisé `FileFinder` pour découvrir nos modules
- Celui-ci s'appuie sur des répertoires (ou apparentés) sur le système de fichiers pour les localiser
 - Ils reposent pour cela sur `PathEntryFinder`

Découvrir des modules ailleurs que dans les fichiers

- On a jusqu'ici utilisé `FileFinder` pour découvrir nos modules
- Celui-ci s'appuie sur des répertoires (ou apparentés) sur le système de fichiers pour les localiser
 - Ils reposent pour cela sur `PathEntryFinder`
- Mais il est possible d'imaginer d'autres manières de découvrir des modules

Meta-path

- `sys.meta_path` liste les *meta finders* utilisés par Python pour rechercher un module
 - Ceux-ci implémentent l'interface de `MetaPathFinder`
 - Très proche de `PathEntryFinder`, elle demande une méthode `find_spec` recevant le nom du module et son chemin

```
In [57]: sys.meta_path
```

```
Out[57]: [_frozen_importlib.BuiltinImporter,  
          _frozen_importlib.FrozenImporter,  
          _frozen_importlib_external.PathFinder,  
          <six._SixMetaPathImporter at 0x781d5c08f260>]
```

Meta-path

- `sys.meta_path` liste les *meta finders* utilisés par Python pour rechercher un module
 - Ceux-ci implémentent l'interface de `MetaPathFinder`
 - Très proche de `PathEntryFinder`, elle demande une méthode `find_spec` recevant le nom du module et son chemin

```
In [57]: sys.meta_path
```

```
Out[57]: [_frozen_importlib.BuiltinImporter,  
          _frozen_importlib.FrozenImporter,  
          _frozen_importlib_external.PathFinder,  
          <six._SixMetaPathImporter at 0x781d5c08f260>]
```

- On remarque que `PathFinder` (et donc les mécanismes liés à `sys.path` et `sys.meta_path`) est lui aussi une entrée *meta path*

Imports installables

- On peut concevoir un mécanisme d'import s'assurant qu'un paquet est installé
- Pour cela le *finder* peut faire appel à `pip` afin d'installer un paquet manquant

Imports installables

- On peut concevoir un mécanisme d'import s'assurant qu'un paquet est installé
- Pour cela le *finder* peut faire appel à `pip` afin d'installer un paquet manquant

```
In [58]: import subprocess
```

```
class PipFinder(importlib.abc.MetaPathFinder):  
    def __init__(self, *allowed_modules):  
        self.allowed_modules = set(allowed_modules)  
  
    def find_spec(self, fullname, path, target=None):  
        if fullname not in self.allowed_modules:  
            return None  
  
        print('Installing', fullname)  
        subprocess.run(['pip', 'install', fullname])  
  
    return importlib.util.find_spec(fullname)
```

Imports installables

- On ajoute ensuite le *finder* au *meta-path* (en dernière position) pour le rendre accessible

```
In [59]: sys.meta_path.append(PipFinder('requests'))
```

```
import requests
print(requests.get('https://pycon.fr'))
```

```
Installing requests
Collecting requests
...
Successfully installed requests-2.32.3
<Response [200]>
```

Imports réseau

- Si on s'abstient du système de fichiers, on peut aussi envisager des imports via le réseau
- En disposant par exemple d'un serveur HTTP exposant des modules

```
In [60]: import http.server  
import threading
```

```
class ServerHandler(http.server.BaseHTTPRequestHandler):  
    files = {  
        'remote.py': b'def test():\n            print("Hello")'  
    }  
  
    def do_GET(self):  
        filename = self.path[1:]  
        content = self.files.get(filename)  
        if content is None:  
            self.send_error(404)  
        else:  
            self.send_response(200)  
            self.end_headers()  
            self.wfile.write(content)  
  
    def do_HEAD(self):  
        filename = self.path[1:]  
        if filename in self.files:  
            self.send_response(200)  
            self.end_headers()  
        else:  
            self.send_error(404)
```

Imports réseau

- Que l'on lancerait ici dans un *thread* dédié, mais qu'on pourrait imaginer tourner sur un serveur distant (RPC)

```
In [61]: server = http.server.HTTPServer(('', 8080), ServerHandler)
thr = threading.Thread(target=server.serve_forever)
thr.start()
```

```
127.0.0.1 - - [03/Nov/2024 08:04:07] "HEAD /remote.py HTTP/1.1" 200 -
127.0.0.1 - - [03/Nov/2024 08:04:07] "GET /remote.py HTTP/1.1" 200 -
127.0.0.1 - - [03/Nov/2024 08:04:10] code 404, message Not Found
127.0.0.1 - - [03/Nov/2024 08:04:10] "HEAD /dynamic__foo_bar__toto_tata.py HTTP/1.1" 404 -
```

Imports réseau

- On utilise alors un *finder* simple s'appuyant sur un *loader* pour la partie réseau

In [62]:

```
class NetworkFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self.loader = NetworkLoader(baseurl)

    def find_spec(self, fullname, path, target=None):
        if self.loader.exists(fullname):
            return importlib.util.spec_from_loader(fullname, self.loader)
```

Imports réseau

- Le *loader* interroge le serveur configuré pour obtenir le code source des modules

```
In [63]: import urllib

class NetworkLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self.baseurl = baseurl
    def get_url(self, fullname):
        return f'{self.baseurl}/{fullname}.py'
    def get_data(self, url):
        with urllib.request.urlopen(url) as f:
            return f.read()
    def get_filename(self, name):
        return f'{self.get_url(name)}'
    def exists(self, name):
        req = urllib.request.Request(self.get_url(name), method='HEAD')
        try:
            with urllib.request.urlopen(req) as f:
                pass
        except:
            return False
        return f.status == 200
```

Imports réseau

- Il suffit alors de créer une entrée pour notre serveur précédemment instancié

```
In [64]: sys.meta_path.append(NetworkFinder('http://localhost:8080'))
```

```
import remote  
remote.test()
```

```
Hello
```

Imports dynamiques

- Enfin on peut exploiter le mécanisme des *loaders* pour charger le code du module à la volée
- Par exemple un module qui définirait ses attributs en fonction de son nom

```
In [65]: class DynamicFinder(importlib.abc.MetaPathFinder):  
    def find_spec(self, fullname, path, target=None):  
        if fullname.startswith('dynamic_'):  
            parts = fullname.split('_')[1:]  
            attributes = dict(part.split('_') for part in parts)  
            return importlib.util.spec_from_loader(  
                fullname,  
                DynamicLoader(attributes)  
            )
```

Imports dynamiques

- Avec le *loader* associé

```
In [66]: class DynamicLoader(importlib.abc.Loader):  
    def __init__(self, attributes):  
        self.attributes = attributes  
  
    def exec_module(self, module):  
        module.__dict__.update(self.attributes)
```

```
In [67]: sys.meta_path.append(DynamicFinder())  
  
import dynamic_foo_bar_toto_tata as mod  
print(mod)  
print(mod.foo)  
print(mod.toto)
```

```
<module 'dynamic_foo_bar_toto_tata' (<__main__.DynamicLoader object at 0x781d5aa62cc0>)>  
bar  
tata
```

Autres exemples

- Les exemples des précédents chapitres (*path hooks*, extensions particulières) peuvent être réécrits à l'aide de *meta finders*
 - Mais ils nécessitent alors que chaque *finder* se charge de parcourir `sys.path` pour itérer sur les répertoires

Autres exemples

- Les exemples des précédents chapitres (*path hooks*, extensions particulières) peuvent être réécrits à l'aide de *meta finders*
 - Mais ils nécessitent alors que chaque *finder* se charge de parcourir `sys.path` pour itérer sur les répertoires
- On peut aussi imaginer d'autres manières de générer du code à la volée
 - Import *copilot*: <https://pypi.org/project/copilot-import/>

Conclusion

L'import en bref

- Vue d'ensemble des étapes lors d'un import :
 1. Résolution du nom du module
 2. Recherche du module dans le cache (court-circuit si trouvé)
 3. Résolution des modules parents dans le cas d'un paquet
 4. Identification de la spécification du module (*finder*)
 5. Chargement du module (*loader*)
 6. Stockage dans le cache
 7. Exécution du code du module (*loader*)
- <https://docs.python.org/3/library/importlib.html#approximating-importlib-import-module>

Conclusion

- Le mécanisme d'imports est paramétrable à de multiples niveaux
- Et permet de tordre Python comme on le veut

Liens utiles

- Quelques liens utiles
 - <https://peps.python.org/pep-0302/>
 - <https://peps.python.org/pep-0451/>
 - <https://docs.python.org/3/reference/import.html>
 - <https://docs.python.org/3/library/importlib.html>

Liens utiles

- Quelques liens utiles
 - <https://peps.python.org/pep-0302/>
 - <https://peps.python.org/pep-0451/>
 - <https://docs.python.org/3/reference/import.html>
 - <https://docs.python.org/3/library/importlib.html>
- Et retrouvez les sources de cette présentation
 - https://github.com/entwanne/presentation_imports

Questions ?