

Packaging pour Python : organisation et reproductibilité logicielle



Fernando NIÑO
IRD/Legos

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Plan de la formation

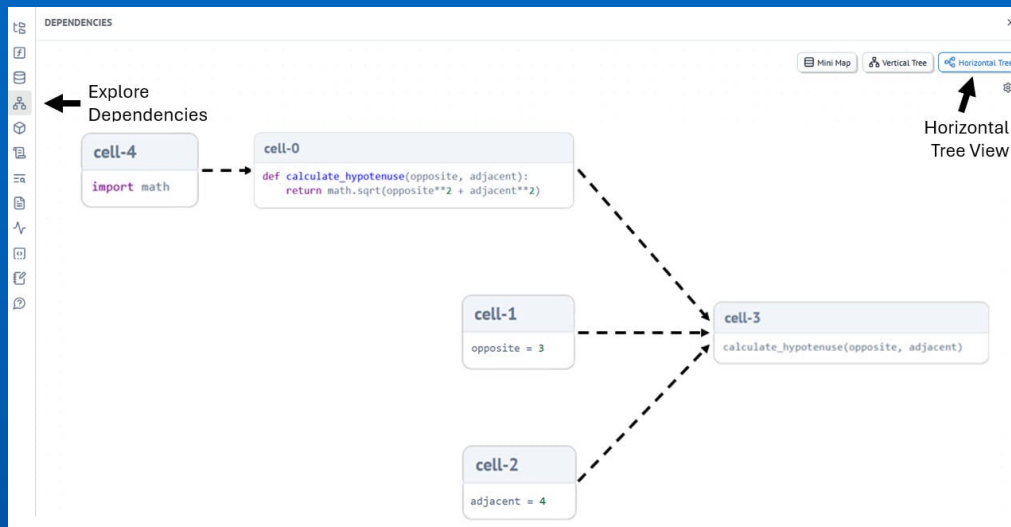
1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Motivation

- En un mot : reproductibilité
- Exemple de **jupyter notebook**
 - *DEMO: notebook.ipynb*
 - Le problème est l'exécution désordonnée de cellules

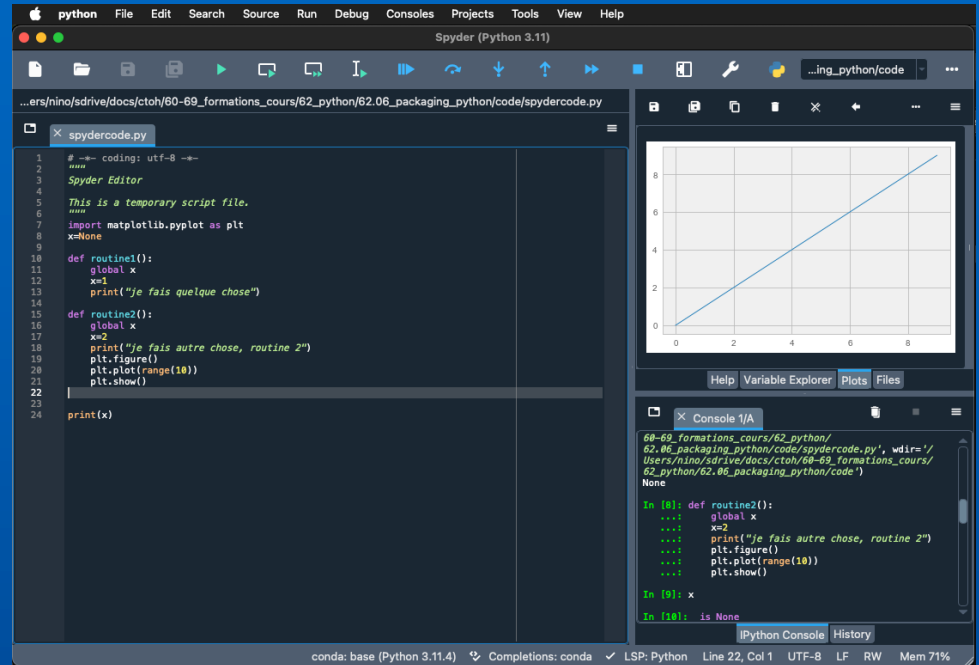
Motivation

- En un mot : reproductibilité
- Exemple de **jupyter notebook**
Une solution possible:
le notebook **marimo**
(<https://marimo.io>)



Motivation

- En un mot : reproductibilité
- Exemple de **spyder**.



The screenshot displays the Spyder Python IDE interface. The main window is titled "Spyder (Python 3.11)". The interface is divided into several panels:

- Code Editor:** Shows a Python script named "spydercode.py" with the following code:

```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7 import matplotlib.pyplot as plt
8 x=None
9
10 def routine1():
11     global x
12     x=1
13     print("je fais quelque chose")
14
15 def routine2():
16     global x
17     x=2
18     print("je fais autre chose, routine 2")
19     plt.figure()
20     plt.plot(range(10))
21     plt.show()
22
23 print(x)
24
```
- Plots Panel:** Displays a line plot with a white background and a light gray grid. The x-axis ranges from 0 to 8, and the y-axis ranges from 0 to 8. A blue line starts at (0,0) and goes up to (8,8).
- Console Panel:** Shows the output of the code execution:

```
68-69_formation_cours/62_python/
62_06_packaging_python/code/spydercode.py', vdir='/
/Users/nino/sdrive/docs/ctoh/68-69_formation_cours/
62_python/62_06_packaging_python/code')
None
In [8]: def routine2():
...:     global x
...:     x=2
...:     print("je fais autre chose, routine 2")
...:     plt.figure()
...:     plt.plot(range(10))
...:     plt.show()
In [9]: x
In [10]: is None
```

The bottom status bar shows: "conda: base (Python 3.11.4) | Completions: conda | LSP: Python | Line 22, Col 1 | UTF-8 | LF | RW | Mem 71%"

Motivation

- **En un mot : reproductibilité**
- **Gestion de versions :**
 - **gérer le développement de code avec mercurial ou git c'est bien, mais il faut aussi pouvoir gérer les versions au cours de l'exécution:**
 - **quelles versions de librairies ont été utilisées pour créer un produit ?**

```
print(monmodule.__version__)
```

- **et le lien avec git ?**
 - id changeset  __version__

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, version)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Concepts de base

- **Packaging** : mécanisme pour partager du code avec d'autres développeurs
<https://packaging.python.org>
- **module** : un fichier que l'on peut importer.
 - Un fichier est l'exemple le plus simple
 - Mais si plusieurs fichiers, ou dépendances avec des librairies non-standard, ou besoin de versions spécifiques de Python
- **packages** :
 - une **arborescence** de code (e.g. avec git)
 - **sdist** si distributions avec du code python pur
 - **wheels** si distribution du code binaire (e.g. numpy)
 - un **package** conda

Concepts de base

- **Packaging** : mécanisme pour partager du code avec d'autres développeurs
<https://packaging.python.org>
- **module** : un fichier que l'on peut importer.
 - Un fichier est l'exemple le plus simple
 - Mais si plusieurs fichiers, ou dépendances avec des librairies non-standard, ou besoin de versions spécifiques de Python
- **packages** :
 - une **arborescence** de code (e.g. avec git)
 - ~~*sdist*~~ si distributions avec du code python pur
 - **wheels** si distribution du code binaire (e.g. numpy)
 - ~~*un package conda*~~

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

modules et sys.path

- **sys.path** est une liste de chemins à chercher pour trouver des modules à importer.
- Règles dans https://docs.python.org/3.11/library/sys_path_init.html
- le premier élément de la liste est toujours le répertoire du script à exécuter
- **PYTHONPATH** peut être utilisée pour ajouter des chemins

```
PYTHONPATH=/usr/lib/toto python modulesearch.py  
['/Users/nino/sdrive/docs/ctoh/60-69_ formations_cours/62_python/62.06_packaging_python/code', '/usr/lib/toto',  
'/Users/nino/.conda/envs/fpp/lib/python3.11.zip', '/Users/nino/.conda/envs/fpp/lib/python3.11',  
'/Users/nino/.conda/envs/fpp/lib/python3.11/lib-dynload', '/Users/nino/.local/lib/python3.11/site-packages',  
'/Users/nino/.conda/envs/fpp/lib/python3.11/site-packages']
```

modules et sys.path

- **sys.path** est rempli aussi avec les contenus de **~/.local/lib/pythonX.XX**
- les fichiers ***.pth** qui s'y trouvent ajoutent aussi des chemins
- et un string vide **"** indique le répertoire courant

```
python -c "import sys;import pprint;pprint.pprint(sys.path)"
```

```
['',  
'/Users/nino/.conda/envs/fpp/lib/python311.zip',  
'/Users/nino/.conda/envs/fpp/lib/python3.11',  
'/Users/nino/.conda/envs/fpp/lib/python3.11/lib-dynload',  
'/Users/nino/.local/lib/python3.11/site-packages',  
'/Users/nino/Workspace/projects/simu_alti_abileah/src/notebooks/naahyss_lib/src',  
'/Users/nino/.local/lib/python3.11/site-packages/lenapy-0.8-py3.11.egg',  
'/Users/nino/.local/lib/python3.11/site-packages/dask-2024.7.0-py3.11.egg',  
'/Users/nino/.conda/envs/fpp/lib/python3.11/site-packages']
```

```
> ll ~/.local/lib/python3.11/site-packages/*pth
```

```
-rw-rw-r-- 1 nino staff 79 2 aoù 2024 /Users/nino/.local/lib/python3.11/site-packages/__editable__.naahyss_lib-0.1.dev1+gf5972e9.d20240802.pth  
-rw-rw-r-- 1 nino staff 75 12 jul 2024 /Users/nino/.local/lib/python3.11/site-packages/easy-install.pth
```

```
> cat ~/.local/lib/python3.11/site-packages/_editable_.naahyss_lib-0.1.dev1+gf5972e9.d20240802.pth  
/Users/nino/Workspace/projects/simu_alti_abileah/src/notebooks/naahyss_lib/src
```

```
> cat ~/.local/lib/python3.11/site-packages/easy-install.pth
```

```
./lenapy-0.8-py3.11.egg  
./dask-2024.7.0-py3.11.egg  
./lenapy-0.8-py3.11.egg
```

```
> ll ~/.local/lib/python3.11/site-packages/
```

```
total 16  
drwx----- 7 nino staff 224 2 aoù 2024 ./  
drwxrwxr-x 3 nino staff 96 12 jul 2024 ../  
-rw-rw-r-- 1 nino staff 79 2 aoù 2024 __editable__.naahyss_lib-0.1.dev1+gf5972e9.d20240802.pth  
drwxrwxr-x 4 nino staff 128 12 jul 2024 dask-2024.7.0-py3.11.egg/  
-rw-rw-r-- 1 nino staff 75 12 jul 2024 easy-install.pth  
drwxrwxr-x 4 nino staff 128 12 jul 2024 lenapy-0.8-py3.11.egg/  
drwxrwxr-x 9 nino staff 288 2 aoù 2024 naahyss_lib-0.1.dev1+gf5972e9.d20240802.dist-info/
```

modules et sys.path

- **Un bricolage usuel : modifier le sys.path dans le code lui-même**

```
import sys  
sys.path.append("/home/nom/code/lib/libpython")
```

- **et ça marche, mais il y a le problème de la maintenance des paramètres « en dur »**

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Structuration du code

- règles sont simples : un `__init__.py` indique un module
- depuis python 3.3 l'absence de `__init__.py` est possible mais indique qu'il s'agit d'un *namespace package* (un package qui regroupe de sub-package). Dans la pratique il vaut mieux toujours avoir le `__init__.py`

```
> tree 4_structuration/structuration_code | egrep -v pyc
```

```
.
├── project
│   ├── example
│   │   └── foo.py
├── project2
│   ├── example
│   │   ├── __init__.py
│   │   └── bar.py
```

```
> ipython
imPython 3.11.14 | packaged by conda-forge | (main, Oct 22 2025, 22:56:31) [Clang 19.1.7 ]
```

```
In [1]: import sys
```

```
In [2]: sys.path
```

```
Out[2]:
```

```
['/Users/nino/.conda/envs/fpp/bin',
 '/Users/nino/.conda/envs/fpp/lib/python311.zip',
 '/Users/nino/.conda/envs/fpp/lib/python3.11',
 '/Users/nino/.conda/envs/fpp/lib/python3.11/lib-dynload',
 '',
 '/Users/nino/.local/lib/python3.11/site-packages',
 '/Users/nino/Workspace/projects/simu_alti_abileah/src/notebooks/naahyss_lib/src',
 '/Users/nino/.local/lib/python3.11/site-packages/lenapy-0.8-py3.11.egg',
 '/Users/nino/.local/lib/python3.11/site-packages/dask-2024.7.0-py3.11.egg',
 '/Users/nino/.conda/envs/fpp/lib/python3.11/site-packages']
```

```
In [3]: import project
```

```
In [4]: import project.example
```

```
In [5]: from project.example import foo
```

```
In [7]: from project2.example import bar
```

```
In [8]: sys.path.append("./project")
```

```
In [9]: sys.path.append("./project2")
```

```
In [10]: import example
```

```
In [11]: from example import foo
```

```
ImportError                                Traceback (most recent call last)
```

```
Cell In[11], line 1
```

```
----> 1 from example import foo
```

```
ImportError: cannot import name 'foo' from 'example' (/Users/nino/sdrive/docs/ctoh/60-69_formation_cours/62_python/62.06_packaging_python/code/structuration_code/./project2/example/__init__.py)
```

```
In [12]: from example import bar    #>>> La deuxième entrée avec example a priorité
```

cookiecutter

- Une des nombreuses façons de copier une arborescence.
- Très facile à customiser par soi-même
- Peut utiliser des paramètres fictifs (*placeholders*)

```
> cookiecutter cookiecutter/etudes
[1/8] full_name (Prenom Nom):
[2/8] email (email@utoulouse.fr):
[3/8] project_name (Name of the project):
[4/8] project_slug (name-of-the-project): fpp
[5/8] project_short_description (A short description of the project):
[6/8] remote_url (None):
[7/8] release_date (2025-10-29):
[8/8] version (0.1.0):
```

- voir

4_structuration/cookiecutter/etudes/{{cookiecutter.project_slug}}

Structure du projet généré

```
.
├── AUTHORS.md
├── LICENSE
├── README.md
├── data
│   ├── external    <- Data from third party sources.
│   ├── interim     <- Intermediate data that has been transformed.
│   ├── processed   <- The final, canonical data sets for modeling.
│   └── raw         <- The original, immutable data dump.
├── docs            <- Documentation or scientific papers
│   ├── meetings   <- Meetings minutes
│   ├── planning   <- Work planning and scheduling
│   ├── proposal   <- Proposed work to be done in project
│   ├── deliverables <- Documents delivered by the project
│   ├── reports    <- Generated data analysis not in papers nor deliverables
│   │   └── figures <- Figures for the manuscript or reports
│   └── papers     <- Scientific papers
├── notebooks     <- Jupyter notebooks. Naming convention is a number (for ordering),
│                   the creator's initials, and a short '-' delimited description, e.g.
│                   `1.0-jqp-initial-data-exploration`.
└── src           <- Source code for this project
    ├── main.py   <- main program
    └── util.py   <- utilities
```

Voir

<https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout>

```
> cookiecutter gh:ssciwr/cookiecutter-python-package
You've downloaded /Users/nino/.cookiecutters/cookiecutter-python-package before. Is it okay to delete and re-
download it? [y/n] (y):
[1/10] project_name (My Project): myfpp
[2/10] remote_url (None):
[3/10] project_slug (myfpp):
[4/10] full_name (Your Name): fernando
[5/10] Select license
  1 - MIT
  2 - BSD-2
  3 - GPL-3.0
  4 - LGPL-3.0
  5 - None
Choose from [1/2/3/4/5] (1): 4
[6/10] Select github_actions_ci
  1 - Yes
  2 - No
Choose from [1/2] (1): 1
[7/10] Select gitlab_ci
  1 - Yes
  2 - No
Choose from [1/2] (1):
[8/10] Select notebooks
  1 - Yes
  2 - No
Choose from [1/2] (1):
[9/10] Select commandlineinterface
  1 - No
  2 - Yes
Choose from [1/2] (1): 2
[10/10] Select version_management
  1 - manually
  2 - setuptools_scm
Choose from [1/2] (1): 2
Dépôt Git vide initialisé dans /Users/nino/Downloads/fpp/myfpp/.git/
Basculement sur la nouvelle branche 'main'
[main (commit racine) 603f53b] Initial Commit
```

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

setuptools vs pyproject.toml

- Historiquement, setuptools c'était l'outil à utiliser. Il crée des **eggs** pour distribuer. Ce ne sont plus d'actualité, ils sont *deprecated* et doivent être remplacé par des wheels.
- setuptools ne doit plus être utilisé pour l'installation, c'est maintenant un « build backend » - avant il faisait trop de choses. Le package **distutils** est *deprecated*.

~~python setup.py install~~

setuptools vs pyproject.toml

- **Malgré tout, setuptools est toujours d'actualité.**
- **Personnalisable (+)** : Setuptools est un couteau suisse.
- **Complexe (-)** : trop d'options, lent
- **Quand l'utiliser** : quand il faut des personnalisations, quand il y a du code ancien ou quand il vous faut interagir avec du code en C, par exemple.

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Autres systèmes de **build**

- **Hatchling** : cool, minimaliste, rapide. Trop récent... Mais tout n'est pas possible
- **Flit** : rapide, facile, parfait pour du code 100% python. Si le code est simple, flit est votre choix.
- **Poetry** : il fait tout. Non seulement packaging, mais dépendences, environnements, tout. Un outil unique.
- **UV** : Fait tout, mais plus rapidement et respecte les PEPs. Remplacement « drop-in » pour pip, pipx, poetry, pyenv, twine et virtualenv.

Autres systèmes de **build**

- **Hatchling** : cool, minimaliste, rapide. Trop récent... Mais tout n'est pas possible
- ~~*Flit : rapide, facile, parfait pour du code 100% python. Si le code est simple, flit est votre choix.*~~
- ~~*Poetry : il fait tout. Non seulement packaging, mais dépendances, environnements, tout. Un outil unique.*~~
- ~~*UV : Fait tout, mais plus rapidement et respecte les PEPs. Remplacement « drop-in » pour pip, pipx, poetry, pyenv, twine et virtualenv.*~~

pyproject.toml

Format TOML avec 3 sections:

- **[build-system]**
 - spécification du backend à utiliser
 - dépendences éventuelles pour le build
- **[project]**
 - métadonnées
 - dépendences à l'exécution
- **[tool]**
 - Pour des outils particuliers (e.g. intégration avec Git)

[build-system]

- **Plusieurs gestionnaires de build. Pour python basique, on préfère hatchling**

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

- **Même fonctionnalité de base avec flit, pdm, setuptools, mais utilisation avancée différente**

[project]

- **Métadonnées et dépendences. Minimum : name et version**

```
[project]
name = "fpp"
version = "0.0.1"
description = "A short description of the project"
authors=[
    {name="author1", email="email1@utoulouse.fr"},
    {name="author2", email="email2@utoulouse.fr"},
]
```

```
[project.scripts]
fpptest = "fpp.main:main"
```

Build avec pyproject.toml

```
[project]
name = "fpp"
description = "A short description of the project"
authors=[
    {name="author1", email="email1@utoulouse.fr"},
    {name="author2", email="email2@utoulouse.fr"},
]

version = "2025.10.29 »

dependencies = [
    'numpy>=1.8',
]
requires-python = ">=3.9"

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project.scripts]
fpptest = "fpp.main:main"
```



Sans cela, on peut faire `import fpp`.
Avec, `fpptest` peut être invoqué depuis un terminal

Installation

- **Installation basique**

```
cd 6_build_systems/fpp  
pip install .
```

- **Installation plus élaborée, le makefile:**

```
cd 6_build_systems/fpp  
make install
```

Installation éditable

- Pas de copie vers site-packages, mais création d'un fichier **.pth**

```
pip install -e .  
make develop
```

- Installation plus élaborée, le makefile:

```
make uninstall  
make develop
```

Installation éditable

```
> ls -l ~/.local/lib/python3.11/site-packages
total 24
drwx-----  9 nino  staff  288 29 oct 18:58 ./
drwxrwxr-x   3 nino  staff   96 12 jul  2024 ../
-rw-rw-r--   1 nino  staff   79  2 août  2024 __editable__.naahyss_lib-0.1.dev1+gf5972e9.d20240802.pth
-rw-rw-r--   1 nino  staff  113 29 oct 18:58 _fpp.pth
drwxrwxr-x   4 nino  staff  128 12 jul  2024 dask-2024.7.0-py3.11.egg/
-rw-rw-r--   1 nino  staff   75 12 jul  2024 easy-install.pth
drwxrwxr-x  10 nino  staff  320 29 oct 18:58 fpp-2025.10.29.dist-info/
drwxrwxr-x   4 nino  staff  128 12 jul  2024 lenapy-0.8-py3.11.egg/
drwxrwxr-x   9 nino  staff  288  2 août  2024 naahyss_lib-0.1.dev1+gf5972e9.d20240802.dist-info/

~/docs/ctoh/60-69_formationen_cours/62_python/62.06_packaging_python/code/6_build_systems/fpp main*
> cat ~/.local/lib/python3.11/site-packages/_fpp.pth
/Users/nino/sdrive/docs/ctoh/60-69_formationen_cours/62_python/62.06_packaging_python/code/6_build_systems/fpp/src%
```

Le code du répertoire de travail est accessible telquel – les changements sont immédiats, sans refaire l’installation. Pratique, mais dangereux.

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Motivation

- **En un mot : reproductibilité**
- **Gestion de versions :**
 - **gérer le développement de code avec mercurial ou git c'est bien, mais il faut aussi pouvoir gérer les versions au cours de l'exécution:**
 - **quelles versions de librairies ont été utilisées pour créer un produit ?**

```
print(monmodule.__version__)
```

- **et le lien avec git ?**
 - id changeset  __version__

Modification du pyproject.toml

```
[project]
name = "fpp"
description = "A short description of the project"
dynamic = ["version"]
authors=[
    {name="author1", email="email1@utoulouse.fr"},
    {name="author2", email="email2@utoulouse.fr"},
]
```

Indique que la version n'est pas à chercher dans le projet

```
dependencies = [
    'numpy>=1.8',
]
requires-python = ">=3.9"
```

```
[build-system]
requires = ["hatchling", "hatch-vcs"]
build-backend = "hatchling.build"
```

On ajoute hatch-vcs

```
[tool.hatch.version]
source = "vcs"
```

on utilise hatch-vcs via le vcs

```
# file will be generated in version-file
```

```
[tool.hatch.build.hooks.vcs]
version-file = "src/fpp/_version.py"
```

on indique le fichier de destination

```
[project.scripts]
fptest = "fpp.main:main"
```

Intégration avec VCS/SCM

- L'idée est qu'à chaque invocation du build, on utilise l'API du SCM pour récupérer un identifiant et générer une variable python avec cette information. Vieille idée, beaucoup d'implementations (e.g. vcversioner).

```
make info
```

```
make infohatch
```

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**

Reproductibilité

Il faut que les environnements d'exécution soient reproductibles. Plusieurs environnements virtuels :

- **virtualenv**
- **conda**
- **uv / poetry / ...**

En plus de cela, il faut que les versions de code/données utilisées soient les mêmes. Avec **hatchling-vcs/setuptools-scm** on traite seulement une partie du problème.

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**
9. **Autres sujets**

Plan de la formation

1. **Motivation (scripts vs notebooks vs modules)**
2. **Concepts de base (module vs package, structure, `__init__.py`)**
3. **module search path**
4. **Structuration du code**
5. **setuptools vs pyproject.toml**
6. **Build systems: setuptools / hatchling**
7. **Intégration avec les SCM**
8. **Reproductibilité**
9. **Distribution avec wheels**
10. **Autres sujets**

Création de **wheels** pour distribution

- `pip install build`
- `python -m build --wheel --no-isolation --outdir artifacts ./fpp`

```
> ll artifacts
total 8
drwxrwxr-x  3 nino  staff   96 29 oct 21:16 ./
drwxrwxr-x  5 nino  staff  160 29 oct 21:16 ../
-rw-r--r--  1 nino  staff 3852 29 oct 21:16 fpp-0.0.1-py3-none-any.whl

scp artifacts/fpp* summit2-legos:
```

```
nino@summit2-legos:~
> pip install fpp-0.0.1-py3-none-any.whl
Defaulting to user installation because normal site-packages is not writeable
Processing ./fpp-0.0.1-py3-none-any.whl
ERROR: Package 'fpp' requires a different Python: 3.6.15 not in '>=3.9'

nino@summit2-legos:~
> module load conda python

nino@summit2-legos:~
> pip install fpp-0.0.1-py3-none-any.whl
Processing ./fpp-0.0.1-py3-none-any.whl
Requirement already satisfied: numpy>=1.8 in /ctoh/data/shared/envs/py311/lib/python3.11/site-packages (from fpp==0.0.1) (2.0.2)
Installing collected packages: fpp
Successfully installed fpp-0.0.1
```

Build avec extensions en C

- Nécessite un **setup.py** pour construire les extension et un **pyproject.toml** pour la suite. Donc, **setuptools** obligatoire.

```
[project]
name = "ctoh_hydro"
description = "CTOH's hydrological time series from altimetry"
dynamic = ["version"]
authors=[
    {name="Denis", email="denis.blumstein@cnes.fr"},
    {name="Fernando Nino", email="fernando.nino@ird.fr"}
]
dependencies = [
    'numpy>=1.8',
]
requires-python = ">=3.9"

[build-system]
requires = ["numpy", "setuptools>=61.0", "setuptools-scm>=8.0"]
build-backend = "setuptools.build_meta"

[tool.setuptools.packages.find]
where = ['src']

[tool.setuptools_scm]
version_file = "src/volodia/_version.py"

[project.scripts]
build_watermarks="mask:build_watermarks.main"
```

```
from setuptools import setup, Extension
import numpy
import platform

compile_args = "-O3 -DNDEBUG -std=c99 -mcmmodel=medium -mtune=native -march=native "

mod_kmask = Extension('_kmask',
    sources=['src/volodia_util/virtualraster/kernel/kmask.c',
            'src/volodia_util/virtualraster/kernel/patterns.c',
            'src/volodia_util/virtualraster/kernel/kmask_python.c'],
    include_dirs=[numpy.get_include()],
    extra_compile_args=compile_args.split())

mod_kdem = Extension('_kdem',
    sources=['src/volodia_util/virtualraster/kernel/kdem.c',
            'src/volodia_util/virtualraster/kernel/kdem_python.c'],
    include_dirs=[numpy.get_include()],
    extra_compile_args=compile_args.split())

modules = [mod_kmask, mod_kdem]
module_names=[n.name[1:] for n in modules]
print("\n\nBuilding modules:\n\n", module_names)

setup(
    ext_modules=modules
)
```

Utilisation avec modules

```
> diff 10_autres/fpp/Makefile 10_autres/fpp/Makefile.modules
22,23c22,23
< #PIPINSTALL_OPTIONS=--no-index --no-build-isolation
< #PIPINSTALL_TARGET=--prefix=${TMPDIR}/module_build
---
> PIPINSTALL_OPTIONS=--no-index --no-build-isolation
> PIPINSTALL_TARGET=--prefix=${TMPDIR}/module_build
25,26c25
< PIPINSTALL_OPTIONS=
< PIPINSTALL_TARGET=
---
> #PIPINSTALL_OPTIONS=""
54c53
< # @echo;echo "*** Modules installed in ${TMPDIR}/module_build ***";echo
---
> @echo;echo "*** Modules installed in ${TMPDIR}/module_build ***";echo
```

package resources

- La question est comment accéder à des ressources associées au package. Des templates, des fichiers de configuration par défaut, des images ...

`pkg_resources` is deprecated, use **`importlib.resources`**

```
fpp
├── __init__.py
├── __version__.py
├── etc
│   └── template.yml
├── main.py
└── util.py
```

- les ressources doivent être associés à un module.

```
from importlib_resources import files
with open(files(fpp).joinpath("etc","template.yml")) as f:
    print(f.read())
```