

# **Rapport de stage**

## **Publication du package crisprbuilder\_tb**

Stephane Robin sous la direction de Christophe Guyeux et Jean-Claude Charr

21 juin 2020

# Remerciements

Je tiens à exprimer mes plus sincères remerciements à Messieurs Christophe Guyeux et Jean-Claude Charr, professeurs à l'université de Franche Comté pour m'avoir guidé, conseillé et soutenu durant le déroulement de ce stage. Leurs explications et leur disponibilité malgré un emploi du temps chargé ont largement contribué à ma compréhension du sujet.

Je voudrais également souligner la patience dont ils ont fait preuve pour m'expliquer les notions de bioinformatique que j'ai été amené à découvrir durant ce projet. Il en ressort que j'ai particulièrement apprécié de travailler sous leur direction.

# Table des matières

1. Présentation du projet . . . . .	4
2. Notations . . . . .	4
<b>1 Etat de l’art</b>	<b>5</b>
<b>2 Le choix des outils</b>	<b>6</b>
1. L’environnement de développement . . . . .	6
2. Composition d’un package standard . . . . .	7
2.1. Quelle licence choisir ? . . . . .	8
2.2. Choix de l’outil d’empaquetage . . . . .	9
3. Poetry . . . . .	10
3.1. Création d’un package et gestion des dépendances . . . . .	10
3.2. Construction d’un package et publication . . . . .	10
3.3. La construction du package crisprbuilder_tb . . . . .	11
4. La composition du package . . . . .	12
5. La structure d’un package sous Poetry . . . . .	12
6. Améliorer le code avec pylint . . . . .	12
7. Description du package avec le fichier crisprbuilder_tb.md . . . . .	12
8. La création de la librairie . . . . .	12
<b>3 Les tests du package</b>	<b>14</b>
<b>4 Amélioration des performances</b>	<b>15</b>
1. Les principaux éléments du code . . . . .	16

# Préambule

## 1. PRÉSENTATION DU PROJET

Christophe Guyeux, Jean-Claude Charr, ... ont créé du code "brut" leur permettant à la fois d'afficher et de stocker des données relatives à un SRA particulier. Ce code doit être rassemblé, réorganisé, nettoyé pour être conforme aux critères PEP. Il doit être rendu plus efficace si possible, et testé. Il doit s'exécuter en utilisant un CLI, et doit finalement être empaqueté pour qu'un utilisateur puisse l'utiliser quelque soit sa plateforme (Linux, MacOS ou Windows) une fois installé. Pour cela, il est nécessaire de publier ce package sur PyPI et de le documenter clairement en anglais pour en faciliter l'utilisation.

Notons qu'à terme, l'objectif du package sera de proposer une reconstitution du crispr.

Lorsque nous avons défini les objectifs du stage, il est immédiatement ressorti l'importance de lui donner un caractère pratique, qui pourrait être transposable dans le milieu professionnel. La reconstruction du programme à partir d'éléments de code, la nécessité de créer localement un environnement spécifique de travail et d'installer automatiquement cet environnement sur les ordinateurs des utilisateurs, la possibilité de faire fonctionner le package sur différentes plateformes, les différents tests effectués, la création et publication du package sont autant de compétences pratiques utiles, auxquelles il faut se confronter pour bien en comprendre les difficultés.

## 2. NOTATIONS

Définissons tout d'abord le vocabulaire que nous allons utiliser dans ce rapport de stage :

- module : un fichier contenant du code Python,
- package : un répertoire contenant des modules Python,
- distribution : une archive de modules ou packages (au format tar, whl, ...)

## Chapitre 1

### Etat de l'art

# Chapitre 2

## Le choix des outils

### 1. L'ENVIRONNEMENT DE DÉVELOPPEMENT

#### XXX -> CONSTRUCTION D'UN ENVIRONNEMENT DE TRAVAIL

Afin de tester les codes de ce projet, l'outil de création d'environnement virtuel *venv* nous permet de créer un environnement de développement. *venv* commence par constituer un dossier contenant tous les exécutables nécessaires à l'utilisation des modules d'un projet Python.

Il convient de définir la version de Python utilisée dans cet environnement de développement. Pour cela, le module *pyenv* nous permet de définir une version de Python comme version locale de travail dans l'environnement. Nous choisissons pour les besoins du test Python 3.6.5.

Finalement, nous installons dans cet environnement les dépendances nécessaires à l'aide de l'outil *pip* :

- `pkg-resources==0.0.0` (installé automatiquement)

- ```
pip install xmltodict
```

le module `xmltodict` permet de lire du code XML comme si il s'agissait de code JSON. Il permet donc une lecture plus rapide des fichiers.

- ```
pip install openpyxl
```

le package `openpyxl` permet de lire et d'écrire dans des fichiers Excel de type `xlsx`, `xlsm`, `xltx`, `xltn`. Il comprend les modules `et-xmlfile` et `jdcal`.

- ```
pip install xlrd
```

le module `xlrd` extrait des données d'un tableur Excel à partir de la version 2.0 et avant de les formate.

- ```
pip install biopython
```

le package `biopython` regroupe un ensemble d'outils Python pour le traitement informatique de la biologie moléculaire et comprend le module `numpy`.

- `DateTime==4.3`

Le module `datetime` permet de manipuler des dates et heures en gérant des objets de type `DateTime`.

- `pytz==2019.3` (installé avec le module `datetime`)
- `zope.interface==5.0.1` (installé avec le module `datetime`)

Notons que certains modules utilisés dans ce package sont nativement présents dans Python3. C'est le cas par exemple de :

- le module `os` fournit une manière portable d'utiliser les fonctionnalités dépendantes du système d'exploitation,
- le module `pickle` permet de sérialiser et désérialiser une structure d'objet Python. Il remplace le module primitif `marshal`. `pickle` se trouve déjà dans la librairie standard Python3,
- le module `csv` implémente des classes pour lire et écrire des données liées à des feuilles de calcul ou des bases de données au format `csv`,
- le module `shutil` propose des opérations sur les fichiers et collections de fichiers, notamment la copie et suppression de fichiers. Le module `subprocess.run` permet de gérer de nouveaux processus, de se connecter à leurs flux d'input/output/erreurs. Il remplace plusieurs modules dépréciés : `os.system`, `os.spawn*`, `os.popen*`, `popen2.*`, `commands.*`

Le package, une fois installé par l'utilisateur, devra fournir automatiquement un environnement de travail similaire, afin que son exécution soit rendue possible. On devra donc retrouver les modules installés via `pip` dans un fichier tel que `requirements.txt`.

## 2. COMPOSITION D'UN PACKAGE STANDARD

Un package standard comporte obligatoirement un fichier `__init__.py` qui va définir la version du projet et le nom du module de lancement du programme (souvent `__main__.py`). Il est toutefois acceptable et parfois même recommandé de conserver un fichier `__init__.py` vide.

Les fichiers `setup.py`, `requirements.txt`, `LICENSE`, `README.md` ET `MANIFEST.in` sont également nécessaires.

- `setup.py` est le script de construction et configuration destiné au `setuptools`. Il définit notamment le nom et la version du package, ainsi que les fichiers qu'il contient. Il sert également d'interface en ligne de commande relative aux différentes fonctionnalités du package. `setup.cfg` est un fichier d'initialisation qui contient les options par défaut des commandes du `setup.py`.
- `requirements.txt` permet l'installation des dépendances à l'aide d'un unique fichier contenant un module à installer par ligne. Il nécessite l'instruction

```
pip install -r requirements.txt
```

pour commencer ces installations.

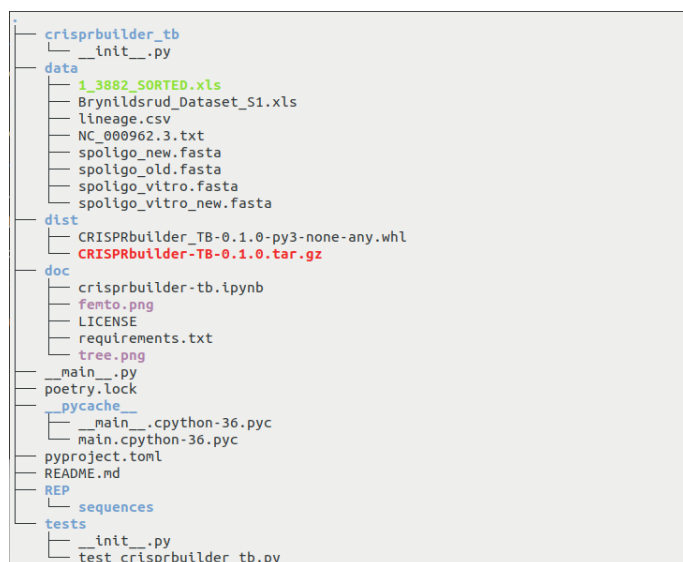
- `LICENSE` définit les termes légaux de la distribution. De nombreux pays n'autorisent pas l'utilisation ou la distribution de packages qui ne disposent pas de licence.
- `README.md` décrit l'objectif du package, son installation, la nature de ses dépendances et les principales fonctionnalités.

- *MANIFEST.in* permet d'inclure dans le package certains fichiers qui ne sont pas automatiquement intégrés.

L'ensemble des modules nécessaires au fonctionnement du package sont regroupés dans un répertoire portant le même nom que le package avec le fichier `__init__.py`.

Avec la PEP-518, le PyPA a proposé un nouveau standard au format `package.toml`, qui remplace les fichiers `setup.py`, `requirements.txt`, `setup.cfg`, `MANIFEST.in` et `Pipfile`. C'est ce nouveau standard qui est utilisé lors de la création d'un package avec Poetry.

La commande `poetry new nomPaquet` permet de générer le squelette de l'application, comprenant les tests unitaires, le fichier `pyproject.toml`, le fichier `README.rst` que nous changeons au format markdown `README.md`, le répertoire du projet et le fichier `init.py`. Nous rajoutons un fichier `LICENSE`, les composants principaux de la librairie, le présent rapport de stage et un fichier `.gitignore` pour la gestion des versions.



La plupart des systèmes d'exploitation incorporent Python2.7 par défaut. L'environnement de travail devra donc expressément définir Python3 comme version pour le projet. Nous avons choisi la version 3.6.5 de Python dans le fichier `package_mymtc.toml`.

Les paquets construits pour des systèmes Unix (Linux et MacOS) nécessitent l'incorporation de fichiers `build.sh` et `meta.yaml`. Les paquets construits pour les systèmes Windows nécessitent l'incorporation des fichiers `bld.bat` et `meta.yaml`. A VERIFIER DANS LE CAS DE POETRY

## 2.1. Quelle licence choisir ?

Trois licences retiennent notre attention. En voici les principales caractéristiques :

- la licence MIT, courte et permissive, préserve exclusivement le copyright et les avis de licence. Toute modification ultérieure peut être distribuée suivant une licence différente et notamment utilisée à des fins personnelles ou commerciales, sans obligation de publication des codes source,
- la licence Apache (2.0) est également permissive et sensiblement similaire dans ses conditions à la licence MIT. Toutefois, elle requiert de préciser les modifications effectuées lors de nouvelles distributions,



- la licence GNU (GPL v3.0) préserve également le copyright et les avis de licence. Elle peut être utilisée à des fins personnelles et commerciales. Elle impose en outre, en cas de modification, la publication complète des codes et l'utilisation de la licence GNU pour les nouvelles distributions.

Dans le cadre de ce projet, aucune spécification restrictive n'étant requise, nous avons choisi la licence MIT qui est simple et peu contraignante.

## 2.2. Choix de l'outil d'empaquetage

Le PyPA *Python Packaging User Guide* recommande l'utilisation de :

- *setuptools* pour définir des projets et créer des sources de distribution,
- *pipenv* pour la gestion des dépendances de packages lors du développement d'applications,
- *venv* pour isoler les dépendances particulières d'une application et créer un environnement de travail,
- *conda* permettant de fournir un environnement de travail favorable aux projets scientifiques, avec notamment tous les modules essentiels pré-installés,
- *buildout* pour les projets de développement Web,
- *poetry* pour un besoin particulier non couvert par *pipenv*,
- *pip* pour l'installation de librairies à partir de PyPI *Python Package Index*.

Au regard de ces recommandations, nous avons testé les outils suivants dans le but de définir un environnement de développement et de construire un package incorporant les dépendances requises.

*pipenv* est un gestionnaire de haut niveau pour les environnements, les dépendances et les packages Python. Contrairement à *virtualenv*, *pipenv* distingue les dépendances du projet et les dépendances des dépendances du projet. Par ailleurs, *pipenv* différencie le mode développement du mode production. Il offre l'avantage de bien fonctionner sur Windows. Toutefois, la communauté Python l'a peu mis à jour depuis 2018.

*Anaconda* est une distribution de logiciels multiplateformes (Windows, Linux, MacOS) qui facilite l'installation des librairies scientifiques *Numpy* et *Scipy*, ce qui est particulièrement intéressant dans le cas des plateformes Windows où ce processus est plus complexe. Elle incorpore une librairie open-source appelée *conda* permettant la gestion des dépendances, de l'environnement de travail ainsi que la création de packages. *Anaconda* semble être approprié au projet, mais c'est une distribution trop lourde pour être intégrée à notre package et *Miniconda*, qui ne comporte que Python, *conda* et *pip*, ne répond pas aux besoins du projet.

Nous avons tout d'abord cherché à construire le package manuellement, à partir de *pipenv* associé à *pip*, puis de *conda* qui dispose d'une riche bibliothèque. Cet effort s'est avéré laborieux et a révélé des incompatibilités lors du *build* qui n'ont pas permis de valider les exigences de la plateforme *testPyPI*.

Nous avons donc décidé de construire notre package en utilisant *Poetry*, qui est un outil complet multiplateformes autour duquel la communauté Python reste très active. Il propose à la fois la gestion des dépendances, l'empaquetage (création d'une structure pour un projet et la génération de fichiers de configuration et de manifestes) et la publication. *Poetry* automatise ces différents procédés et facilite grandement le travail. Nous étudions en détail l'utilisation de cet outil dans la section suivante.

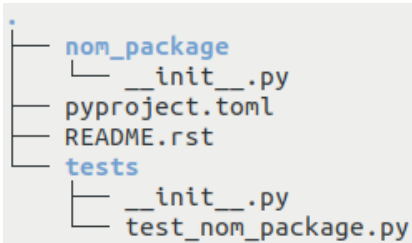
### 3. POETRY

#### 3.1. Création d'un package et gestion des dépendances

La création d'un projet se fait à l'aide de la commande

```
poetry new nom_package
```

qui va créer un répertoire *nom\_package* contenant les éléments suivants :



```
.
├── nom_package
│   ├── __init__.py
│   ├── pyproject.toml
│   ├── README.rst
│   └── tests
│       ├── __init__.py
│       └── test_nom_package.py
```

Le fichier *nom\_package.toml* remplace les anciens standards de définition de packages : *setup.exe* et *requirements.txt*. Il précise notamment le nom du package, sa version, sa description, l'emplacement de son dépôt (par exemple sur GitHub), l'adresse email de l'auteur du package, et la version des dépendances.

En ce qui concerne la version des dépendances, il faut tout d'abord se placer dans le répertoire *nom\_package* puis utiliser l'instruction

```
poetry add nom_dependance
```

qui assure la compatibilité de la dépendance *nom\_dependance* avec le package *nom\_package*. Il est également possible d'imposer certaines contraintes sur les versions des dépendances ou encore de rentrer manuellement les dépendances dans le fichier *nom\_package.toml*, mais l'instruction

```
poetry add nom_dependance
```

offre l'avantage de chercher automatiquement une version compatible de la dépendance, puis de l'inscrire dans *nom\_package.toml*.

Pour installer ensuite les dépendances du projet, il est nécessaire d'utiliser l'instruction

```
poetry install
```

qui crée le fichier *mon\_package.lock*. Ce fichier empêche les dépendances de télécharger la dernière version au moment de l'installation, en fixant la version utilisable par le package.

#### 3.2. Construction d'un package et publication

Pour emballer le projet, il faut utiliser l'instruction

```
poetry build
```

qui va permettre de créer un fichier source au format *sdist* et une distribution compilée au format *wheel*.

On peut vérifier la conformité du package avec l'instruction

```
poetry check
```

qui renvoie

```
All set !
```

si le package ne comporte aucune discordance et peut être publié.

Avant de publier le package, il peut être bon de tester son comportement à l'installation. Pour cela, il est possible de le publier officiellement sur la plateforme de test *testPyPI*. Il faut alors se placer à la racine du package et lancer en ligne de commande l'instruction

```
twine upload -repository testpypi dist/*
```

L'installation du package à partir de *testPyPI* se fait à l'aide de l'instruction

```
pip3 install -i https://test.pypi.org/simple/ nom_package
```

ou si on précise la version du package

```
pip3 install -i https://test.pypi.org/simple/ nom_package==nmr_version
```

A noter qu'il ne faut utiliser de guillemets autour de la version du package.

Lorsque le package est finalement prêt pour être publié sur PyPI, on utilise l'instruction

```
poetry publish
```

L'auteur du package doit pour cela être enregistré sur PyPI avec un identifiant et un mot de passe. A partir de ce moment-là, le package est rendu disponible publiquement.

L'installation du package par un utilisateur quelconque se fait maintenant grâce à l'instruction

```
python3 -m pip install nom_package
```

Les explications relatives à l'utilisation pratique de *Poetry* sont reprises dans le tutoriel suivant que nous venons de publier sur YouTube : adresse YuouTube

### 3.3. La construction du package *crisprbuilder\_tb*

Par exemple pour construire le package *crisprbuilder\_tb* à l'aide de *Poetry*, les instructions suivantes sont nécessaires en ligne de commande :

```
poetry new crisprbuilder_tb cd crisprbuilder_tb
poetry add python3
poetry add xlrd
poetry add xmltodict
poetry add biopython
poetry install
poetry build
poetry check
poetry publish
```

#### 4. LA COMPOSITION DU PACKAGE

Le package est décrit en détail dans le document `crisprbuilder_tb.md`

Ainsi, une recherche effectuée à partir de ERR2704808 permet d'obtenir le résultat suivant :

#### 5. LA STRUCTURE D'UN PACKAGE SOUS POETRY

Lors de l'exécution de l'instruction `poetry new crisprbuilder_tb` s'est créé le package `crisprbuilder_tb` disposant de la structure suivante :

photo structure

Le répertoire `crisprbuilder_tb` contient les fichiers `__init__.py` et `__main__.py`

#### 6. AMÉLIORER LE CODE AVEC PYLINT

*pylint* fournit une note entre 0 et 10 qui reflète le respect des règles du PEP8, notamment en ce qui concerne la lisibilité du code. Il reprend également certaines exigences relatives à ...

Le fichier `__main__.py` est noté 9, / 10 avec les commentaires suivants : XXX

et le fichier `support.py` est noté 9, / 10 avec les commentaires suivants : XXX

S'agissant d'un test statique, le code n'est pas exécuté et la performance du code n'est pas prise en compte lors de l'évaluation faite par *pylint*. Ainsi, *pylint* peut reprocher l'utilisation injustifiée de listes compréhensives, sans tenir compte de l'efficacité d'un tel traitement.

#### 7. DESCRIPTION DU PACKAGE AVEC LE FICHIER CRISPRBUILDER\_TB.MD

#### 8. LA CRÉATION DE LA LIBRAIRIE

=== A CHANGER === Pour créer une librairie à partir de conda, il est nécessaire d'installer conda-build puis de construire un recipe composé de :

- un fichier `meta.yaml` contenant toutes les métadonnées du recipe
- un script `build.sh` qui installe les fichiers de la librairie sur Linux et macOS, exécuté avec une commande `bash`

- un script `bld.bat` qui installe les fichiers de la librairie sur Windows, exécuté avec une commande `cmd`
- un fichier optionnel `run_test.py`, qui s'exécute automatiquement pour effectuer des tests
- un fichier `readme` et des icônes si nécessaire.

Les trois premiers fichiers se créent avec la commande `conda skeleton`. ===FIN CHANGEMENT===

## Chapitre 3

# Les tests du package

## Chapitre 4

# Amélioration des performances

# Conclusion

Ce stage m'a ouvert l'esprit vers le domaine de la bioinformatique et m'a permis de réfléchir à

## 1. LES PRINCIPAUX ÉLÉMENTS DU CODE

dico est composé de la façon suivante :

Un fichier pkl tel que dico\_africanum.pkl est créé par pickle et contient un flux d'octets représentant les objets à sérialiser.

pickle permet aux objets d'être sérialisés en fichiers sur disque et désérialisés dans le programme au moment de l'exécution.



# Références

<https://packaging.python.org/guides/>  
<https://realpython.com/pypi-publish-python-package/>

# Index

pip, 6  
pyenv, 6

venv, 6