Reproductibilité : RTFM, les essentiels de la doc

Violaine Louvet [1]

[1]CNRS, Laboratoire Jean Kuntzmann

Formation Reproductibilité, juillet 2025

Pour se réveiller !

- Par groupe de 3 ou 4, pendant 10 mn
- Identifier quelques mots clés sur ce que vous inspire le terme « documentation »
  - bénéfices
  - freins
  - éléments de la documentation (ce que ça comprend)
  - outil que vous utilisez ou que vous connaissez
- Compléter 4 post-its avec un mot-clé pour chacune des catégories
- Restitution avec un rapporteur par groupe

De l'importance de la documentation

Ce qui arrive régulièrement ...

- *Mais j'ai fait quoi là ??? Je ne me rappelle plus ...*
- *C'est c****t de faire de la doc, je m'en occuperai plus tard*
- *Pourquoi ça compile plus ?*
- *Pfff ça s'installe comment ce truc ?*
- *Quelle galère pour utiliser cette bibliothèque !*
- *Aie, aie, aie, mais comment je vais refaire ces calculs pour répondre au reviewer ??*

Pourquoi faire de la doc ?

- Comprendre son propre code dans 6 mois.
  6 mois après, c'est comme si c'était le code de quelqu'un d'autre ... !
- Faire en sorte que son code soit utilisé par d'autres, ou au moins faire ce qu'il faut pour
- Faciliter les contributions simples : contribuer à la documentation est souvent un premier pas vers une contribution plus technique
- Faire de la doc, c'est aussi prendre du recul sur le code et identifier potentiellement des pistes d'amélioration

→ Documenter avant tout pour soi (et s'éviter beaucoup de frustrations !)

→ Evidemment pour ses collègues, sa communauté, de futurs utilisateurs voire contributeurs

Plan

Plan

Documenter son code, pourquoi ?

- Faciliter la compréhension du code par autrui et par soi-même plus tard !
- Rendre explicites les choix de conception, d'optimisations, de structuration de données ...
- Faciliter la maintenance, la relecture, et la documentation automatique

MAIS attention : trop de commentaires = bruit inutile !!

Exemple de bruit

```
i = 0  # On initialise i à 0
```

Exemple de commentaire pertinent

```
# Compteur d'itérations pour suivre les tentatives de connexion
i = 0
```

Bien documenter son code

- Utiliser des commentaires pour expliquer ce qui n'est pas directement compréhensible dans le code lui-même
- Les commentaires ne remplacent / compensent pas un code mal écrit : il est important de bien le structurer et d'utiliser des conventions de nommage qui aide à la compréhension
- Ecrire des commentaires sous des formes qui pourront servir dans des process d'automatisation : docstrings en Python ou Doxygen en C++ par exemple
- Garder les commentaires à jour : un commentaire obsolète est pire que pas de commentaire

Ne pas utiliser les commentaires pour

- Répéter en langage naturel ce qui est bien écrit dans le code
- Remplacer le contrôle de version

```
# removed on August 5
# if() ...
```

Bonnes pratiques de nommage : les casses

- Le nommage clair améliore immédiatement la lisibilité du code
- Réduit le besoin de commentaires explicatifs
- Rend la relecture, le débogage et la collaboration plus faciles
- Adopter une convention cohérente évite les erreurs et les malentendus

« Un bon nom vaut mieux qu'un long commentaire. »

Les casses

Manières dont les noms des éléments du code sont écrits (variables, fonctions, classes ...).
afin de rendre le code plus lisible et homogène.
Les casses déterminent des conventions de nommage en définissant :

- Quand utiliser des lettres minuscules / majuscules
- Quel caractère séparateur utiliser pour une expression constituée de plusieurs mots

L'exemple de *snake_case*

- Ecrire des ensembles de mots en minuscules en les séparant par des tirets bas « _ » (underscore).
- *snake_case* s'oppose exemple au *camelCase* qui consiste à mettre en majuscule les premières lettres de chaque mot.

PEP8, Style Guide for Python Code

https://peps.python.org/pep-0008/

Python uses snake case for variable nameslowercase words separated by underscores. Python function names should also use snake case. Class names in Python use camel case, with each word starting with a capital letter.

Exemples

```
def double_distance(distance_meters):
    return distance_meters * 2
```

Règles générales pour un bon nommage

A faire

- ✓ Noms significatifs et descriptifs
- ✓ Utiliser des verbes pour les fonctions : *load_data()*, *send_request()*
- ✓ Préférer des noms anglais si le code est destiné à une communauté large
- ✓ Être cohérent dans tout le projet

A éviter

- × Abréviations obscures : tmp, nprc, cptu
- × Lettres seules sauf cas universels (*i*, *x*, *y* dans les boucles simples)
- × Mots vagues : *doStuff()*, *handleThing()*

Plan

Pourquoi documenter les dépendances ?

L'enfer des dépendances, ce qui arrive régulièrement ...

- *C'est galère cette install, il faut encore que je rajoute ...*
- *Pourquoi ça ne compile plus ??? Il n'y avait pas de problème le mois dernier !*
- *Mince j'ai un problème sur l'interface de cette bibliothèque depuis hier*
- *Pffff pourquoi mes résultats ne sont plus les mêmes que la semaine dernière ?*

- Pouvoir installer facilement le logiciel
- Assurer une certaine reproductibilité des résultats
- Etre capable de revenir plus tard sur le code (des mois, voire des années après)

Une dépendance non documentée peut faire échouer toute une chaîne d'exécution, même si le code est très propre

D'abord, c'est quoi une dépendance ?

- une bibliothèque ou un module externe utilisé par le logiciel
- un outil pour construire le logiciel
- un outil pour produire la documentation du logiciel
- un environnement logiciel dans lequel s'exécute le logiciel
- Cela peut aussi être des données de référence, des services externes ...

Par exemple

- Interprèteur comme *R, Python*, ...
- Bibliothèques utilisées dans le code comme *numpy, boost, ...*
- Outil de construction comme *CMake, Makefile*, ...

Les identifier

- Parcourir les fichiers sources et regarder les *imports, include* ...
- Relever les outils de compilation ou d'exécution, regarder les lignes de compilation
- Utiliser des outils automatiques : en Python *pipreqs* ou *pip freeze*, en C++ *ldd, nm, objdump*

Comment concrètement documenter les dépendances ?

En Python

- Fichier *requirements.txt*, fichier *environment.yml* (conda)
- Fixer les versions si possible

Exemple de *requirements.txt*

```
numpy==1.24.0
pandas>=1.3.0
matplotlib
scikit-learn<1.4
```

Capturer l'environnement et générer le fichier *environment.yml*

```
conda env export > environment.yml
```

En C++

- Mentionner les dépendances dans le *README*, avec les versions testées.

Plan

Le fichier *README*

- LE point d'entrée du code, c'est la première chose qu'on voit
- Fichier texte (souvent au format Markdown *README.md*) placé à la racine dun projet
- Il doit permettre de comprendre ce que fait le code, comment on l'installe et comment on l'utilise
- Il doit être bien structuré, facile à lire, efficace
- On le lit avant de lire la documentation
- Un bon *README* renforce la confiance des utilisateurs et attire les futurs contributeurs.
- Public cible :
  - Vous dans 6 mois !
  - Vos collègues
  - Des utilisateurs
  - Des reviewer (par exemple code lié à un article)

Quelques outils pour aider à faire un bon *README* :

- readme.so
- makeareadme
- readme-md-generator

Structuration du *README*

- Titre du projet
- Description du projet (objectif, contexte)
- Installation / Prérequis / Dépendances
- Instructions d'exécution
- Structure des fichiers
- Exemples d'usage
- Auteurs / Crédits
- Licence
- (Optionnel) Liens vers publications, issues, roadmap, etc.

Quelques (bons) exemples :

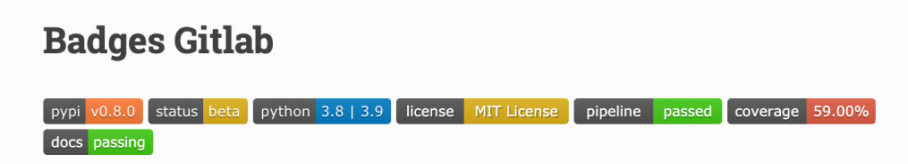
- https://github.com/scikit-learn/scikit-learn
- https://github.com/gammapy/gammapy
- https://github.com/sofa-framework/sofa

Les badges dans les *README*

- Permettent d'avoir une visibilité rapide sur certaines informations clés du projet
- Apportent un aspect plus professionnel et soigné au *README*

Exemples de badges :

- Licence
- Dernière version
- Pipeline CI, test de couverture de code ...



Les badges dans les *README* : exemple

Exemple d'en-tête de fichier *README* sur une forge gitlab

```
# Mon projet

![License](https://img.shields.io/badge/license-MIT-blue.svg)
![Last commit](https://img.shields.io/gitlab/last-commit/<namespace>/<project>.svg?
   gitlab_base_url=https://gitlab.mon-serveur.fr)
![pipeline status]
   (https://gitlab.mon-serveur.fr/<namespace>/<projet>/badges/<branche>/pipeline.svg)]
   (https://gitlab.mon-serveur.fr/<namespace>/<projet>/pipelines)

Description du projet ...
```

- A noter que Gitlab peut auto-générer ce dernier badge.
- Github fournit le même type de badge

La structuration du dépôt : les répertoires

La structuration du repo selon les attendus habituels permet de comprendre facilement l'organisation du code. Pour la plupart des langages :

- *src/* Code source principal
- *test/* Tests unitaires ou d'intégration
- *docs/* Documentation utilisateur ou développeur
- *examples/* Exemples d'utilisation du code

Pour les langages compilés :

- *build/* Répertoire de compilation
- *include/* Fichiers d'en-tête
- *lib/* Bibliothèques issues de la compilation ou dépendances du code
- *bin/* Fichiers exécutables

La structuration du dépôt : les fichiers

README Fichier principal pour le projet

LICENSE ou *COPYING* Informations sur la licence du projet

AUTHORS Liste des auteurs

CHANGELOG Changements notables pour chaque version du projet

CONTRIBUTORS Liste des contributeurs

CONTRIBUTING Guide pour les contributeurs

INSTALL Instructions d'installation

CODE_OF_CONDUCT Code de conduite pour la communauté du projet, définissant les attentes pour un comportement respectueux et professionnel

La structuration du dépôt : *AUTHORS* et *CONTRIBUTORS*

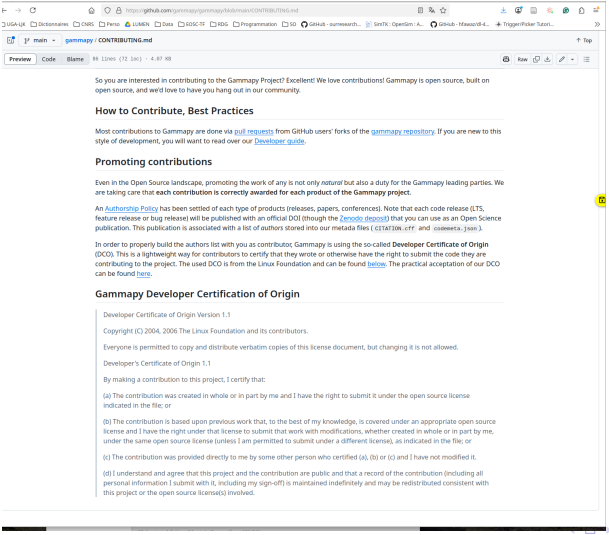
Différence entre auteurs et contributeurs

- Auteur = personne physique qui crée une œuvre de l'esprit
  $\longrightarrow$ la ou les personnes qui ont contribué de manière originale et significative à sa conception ou son développement
- Contributeur = personne qui participe au projet (correctifs, documentation, idées...) sans nécessairement créer une partie originale du logiciel
  Un contributeur peut devenir un auteur si sa contribution est jugée créative et substantielle
  $\longrightarrow$ Ce qui distingue un auteur d'un simple contributeur est l'originalité de la contribution
- Beaucoup de projets incluent un « Developer Certificate of Origin » (DCO) ou utilisent un « Contributor License Agreement » (CLA) pour clarifier les cessions de droits

Developer Certificate of Origin déclaration par laquelle un contributeur certifie qu'il est bien l'auteur de sa contribution et qu'il a le droit de la soumettre (chez nous, implique la hiérarchie du contributeur)

Contributor License Agreement contrat formel par lequel le contributeur cède ou accorde une licence sur ses droits à l'organisation qui maintient le projet (chez nous, implique le service juridique)

Exemple de DCO



So you are interested in contributing to the Gammapy Project? Excellent! We love contributions! Gammapy is open source, built on open source, and we'd love to have you hang out in our community.

How to Contribute, Best Practices

Most contributions to Gammapy are done via pull requests from GitHub users' forks of the gammapy repository. If you are new to this style of development, you will want to read over our Developer guide.

Promoting contributions

Even in the Open Source landscape, promoting the work of any is not only *natural* but also a duty for the Gammapy leading parties. We are taking care that **each contribution is correctly awarded for each product of the Gammapy project**.

An Authorship Policy has been settled of each type of products (releases, papers, conferences). Note that each code release (LTS, feature release or bug release) will be published with an official DOI (though the Zenodo deposit) that you can use as an Open Science publication. This publication is associated with a list of authors stored into our metada files (`CITATION.cff` and `codemeta.json`).

In order to properly build the authors list with you as contributor, Gammapy is using the so-called **Developer Certificate of Origin** (DCO). This is a lightweight way for contributors to certify that they wrote or otherwise have the right to submit the code they are contributing to the project. The used DCO is from the Linux Foundation and can be found below. The practical acceptation of our DCO can be found here.

Gammapy Developer Certification of Origin

> Developer Certificate of Origin Version 1.1
>
> Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
>
> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.
>
> Developer's Certificate of Origin 1.1
>
> By making a contribution to this project, I certify that:
>
> (a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
>
> (b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
>
> (c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
>
> (d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Les fichiers de métadonnées

A quoi ça sert ?

Décrire formellement le projet, ses auteurs, sa citation, ses dépendances ou son usage, pour :

- Améliorer l'interopérabilité
- Favoriser la citation
- Faciliter l'archivage
- Assurer la traçabilité scientifique

Exemples de fichiers

codemeta.json Métadonnées structurées interopérables et standards (`https://codemeta.github.io/`)

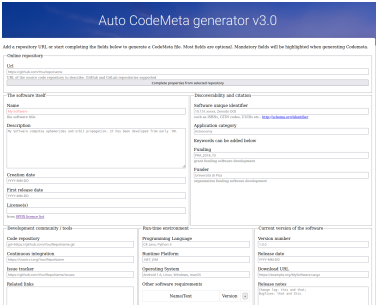
CITATION.cff Permet de citer correctement le logiciel (lu par GitHub) (`https://citation-file-format.github.io/`)

pyproject.toml / *setup.py* Métadonnées de packaging : nom du package, version, description, auteurs, licence

Exemple de génération d'un fichier *codemeta*

Des outils pour faciliter la génération :

- CodeMeta generator, formulaire en ligne
- Codemetapy, pour Python à partir des fichiers de packaging
- Codemetar, pour R à partir des fichiers de packaging
- Auto CodeMeta generator, à partir des éléments d'un repo GitHub



Plan

Générer la doc automatiquement

- Gain de temps !
- Extraction d'informations du code sous des formes visuelles pertinentes (par exemple les hiérarchies de classes)
- Documentation plutôt orientée développeurs (en particulier explicite les interfaces)
- Doc toujours à jour avec le code (en particulier si les commentaires sont à jour ...)
- Génération de fichiers de documentation sous différents formats (html, pdf, ...)
- Automatise la mise en place d'index, ...

Deux exemples principaux

- Doxygen pour C, C++, Java, Python, Php et autres langages, il permet de générer la doc à partir des commentaires et du code lui-même. Logiciel libre sous GPL V2.
- Sphinx développé pour la communauté Python, logiciel libre sous BSD. Il s'est élargi à d'autres langages de programmation comme C, C++ ...

Commençons par Doxygen

1. Installer *Doxygen* (packagé pour différentes distributions, binaires Windows, ...)
   Si on veut des diagrammes et des graphes, il faut aussi installer Graphviz
2. Vérifier que le langage de programmation est bien supporté : C, C++, Lex, C#, Objective-C, IDL, Java, PHP, Python, Fortran et D par défaut
3. Créer un fichier de configuration :

```
doxygen -g <config-file>
```

   Il est possible d'utiliser une interface graphique :

```
doxywizard &
```

   Et le personnaliser en fonction de ses besoins en particulier
   - *INPUT = src/*
   - *RECURSIVE = YES*
4. Exécuter *Doxygen*

```
doxygen <config-file>
```

5. Par défaut la doc est accessible via *html/index.html*

Le rôle des balises

Sans mettre de balises dans le code

Extraction automatique de :

- noms de classes, de fonctions, de variables globales
- signatures (paramètres, types, retour)
- arborescence des fichiers, des modules, des dépendances
- relations entre classes (héritage, composition ...)

$\longrightarrow$ documentation technique mais peu informative

Seul les commentaires « publics » sont extraits :

```
// Commentaire privé

/**
 * Commentaire public
 */
```

Les principales balises

Balises pour fonctions, classes

- *@brief* : résumé court
- *@param* : description dun paramètre dentrée (*@param nom description*)
- *@return* : description de la valeur retournée

Balises de structuration

- *@file* : utilisée en haut dun fichier source pour décrire son but
- *@class* : pour documenter une classe
- *@struct* : pour documenter une struct

Balises de navigation

- *@see* : référence vers une autre fonction, classe, fichier...
- *@ref* : crée un lien interne vers une étiquette ou un élément nommé

Balise d'informations

- *@todo* : tâche à faire
- *@author* : nom de l'auteur
- *@date* : date de création / modification
- *@warning* : mise en garde à l'utilisateur

L'utilisation de Doxygen est à intégrer dès le début du développement sinon cela peut être fastidieux !

Utiliser Sphinx

- Générateur de documentation pour projets Python (ou tout projet texte + docstrings)
- Très utilisé !

1. Installation

```
pip install sphinx
```

2. Initialisation (voir Démarrer avec Sphinx)

```
sphinx-quickstart docs/
```

   Créé le répertoire source avec *index.rst* (page d'accueil) et *conf.py*

3. Activer *extensions* dans *conf.py*. En particulier *autodoc* pour les docstrings et *napoleon* pour les formats numpy et google.

```
extensions = ["sphinx.ext.autodoc", "sphinx.ext.napoleon"]
```

4. Générer la documentation dans le répertoire *docs*

```
make html
```

5. Par défaut la doc est accessible via *docs/_build/html/index.html*

Les docstrings

- Des règles simples, la docstring est toujours :
  - Placée immédiatement après la déclaration de la fonction (ou de la classe/module) avant le code
  - Entourée de triple guillemets (""" ... """) ou de triples guillemets simples (''' ... ''')
  - Phrase courte impérative
  - L'utilisation de # définit un commentaire, pas une docstring
- Utiliser un format standardisé homogène pour les docstrings sur tout le code (NumPy-style, Google-style, reST)

Exemple du format NumPy-style

```
def moyenne(a: float, b: float) -> float:
    """
    Calcule la moyenne de deux nombres.

    Parameters
    ----------
    a : float
        Premier nombre.
    b : float
        Deuxième nombre.

    Returns
    -------
    float
        Moyenne de a et b.
    """
    return (a + b) / 2
```

Plan

Documentations utilisateur et développeur

Objectifs de la documentation utilisateur

Aider l'utilisateur à installer, comprendre et utiliser un logiciel sans connaissance préalable du code.
Répondre aux questions :

- Que puis-je faire ?
- Comment le faire ?
- Comment ça marche ?
- Et si ça ne marche pas ?

Objectifs de la documentation développeur

Fournir aux développeurs toutes les informations nécessaires pour :

- Comprendre l'architecture du code
- Contribuer efficacement et proprement
- Maintenir et faire évoluer le projet

Contenu possible pour la documentation utilisateur

1. Page d'accueil (en général le fichier *index*) : présentation du projet, logo, badges, lien vers Git, contact
2. Introduction : à quoi sert le logiciel ? Pour qui ? Contexte / objectifs
3. Installation : prérequis, installation (conda/pip/docker/autres), tests rapides
4. Quick start : exemples simples, commandes de base, tutoriel minimal
5. Guide d'utilisation : explications détaillées des fonctionnalités principales
6. Cas d'usage : exemples concrets, workflows, intégration avec d'autres outils
7. FAQ : problèmes courants, erreurs possibles, questions fréquentes
8. Contribution : comment contribuer, ouvrir une issue, signaler un bug
9. Licence / Crédits : informations légales, auteurs, citations

Contenu possible pour la documentation développeur

1. Vue d'ensemble : architecture générale, composants principaux, schéma ou diagramme
2. Configuration du projet : environnement de dev (dépendances, Docker, venv, etc.)
3. Arborescence du code : structure des dossiers, logique de séparation des modules
4. Convention de codage : linting, style guide (PEP8, black, clang-format...), nommage, typage
5. Points d'entrée du code : fichiers clés, fonctions principales, boucle principale
6. API interne / modules : référence détaillée des modules/classes/fonctions (via Sphinx/Doxygen par exemple)
7. Tests : comment exécuter/écrire les tests, stratégie de test, outils utilisés
8. Contribution : règles de contribution, format des commits, branches, review
9. Intégration continue : pipeline CI/CD, couverture, badges, tests automatisés
10. Changelog / historique : évolution des versions, journal des modifications (CHANGELOG.md)

Bonnes pratiques

Pour la documentation utilisateur

- Une navigation claire avec une table des matières / barre latérale
- Un langage simple et direct sans jargon technique inutile
- Des exemples concrets avec commandes copiables
- Des captures d'écran / schémas pour illustrer
- Des liens internes pour guider l'utilisateur

Pour la documentation développeur

- Séparer documentation utilisateur et développeur
- Documenter les choix d'architecture technique
- Utiliser des schémas (par exemple UML) pour illustrer
- Faciliter la prise en main par de nouveaux développeurs

Faire une documentation spécifique pour les contributeurs

Objectifs

Permettre à toute personne extérieure de comprendre comment contribuer efficacement et dans le respect des règles du projet.

Structure type possible (dans un fichier *CONTRIBUTING.md* ou en lien avec ce fichier) :

1. Avant de commencer : lien vers code de conduite, types de contributions attendues
2. Cloner et installer : instructions pour forker, cloner, installer les dépendances
3. Structure du projet : brève présentation de l'architecture (pointer vers la doc dev)
4. Règles de contribution : convention de commit, format PR, branches
5. Workflow git : fork, branche feature, pull request / merge request
6. Tests : comment exécuter les tests, en écrire, les outils utilisés
7. Relecture & intégration : processus de review, CI/CD, fusion, réponse aux commentaires
8. Checklist avant PR : formatée en liste : lint, tests, changelog, documentation mise à jour

Bonnes pratiques pour la documentation contributeurs

- Inclure un lien vers *CODE_OF_CONDUCT.md*
- Utiliser un/des modèles de pull / merge request (*.github/PULL_REQUEST_TEMPLATE.md* ou *.gitlab/merge_request_templates*)
- Clarifier la gouvernance : qui valide ? quels droits ?
- Documenter les outils internes utiles (ex : pre-commit, ...)
- Si ils existent, documenter les DCO et CLA

Documentation développeur $\neq$ documentation contributeur

Documentation développeur : éléments techniques détaillés pour faciliter la compréhension du fonctionnement interne du logiciel

Documentation contributeur : éléments organisationnels et sociaux sur le processus de contribution

Le code de conduite du projet : *CODE_OF_CONDUCT.md*

Objectifs

Définir les règles de comportement attendues dans la communauté du projet et promouvoir un environnement accueillant, respectueux et inclusif pour tous, quel que soit le niveau d'expertise ou l'origine.

En général, il inclut :

- Comportements encouragés : bienveillance, écoute, collaboration, ouverture d'esprit
- Comportements inacceptables : propos discriminatoires, attaques personnelles, harcèlement, langage offensant
- Procédure de signalement : à qui signaler un problème, comment, et quelles sont les garanties de confidentialité
- Responsabilités de léquipe de modération : enquêter, agir, protéger la communauté

Voir par exemple `https://www.contributor-covenant.org/` pour un modèle reconnu

Notebooks et documentation

Intérêt des notebooks pour la documentation

- Si le code s'y prête !
- Mélange de codes, textes, visualisations et résultats dans un seul document
- Permet d'expliquer pas à pas une analyse, un algorithme, ou un usage du logiciel

- Peuvent être un bon complément à la doc classique par des exemples concrets et exécutables
- Par exemple pour faire des tutoriels interactifs, expliciter des cas d'usage ...
- BinderHub permet d'exécuter les notebooks en ligne depuis un dépôt Git, sans rien installer (nécessite d'ajouter un fichier d'environnement *environment.yml*, *requirements.txt* ...)
- Sphinx propose une extension pour les inclure dans la doc (*nbsphinx*)

Plan

Héberger la documentation : les sites statiques

- Les générateurs de documentation proposent différents formats, et en particulier du html
- Les autres types de documentation peuvent être rédigés de différentes manières mais :
  - il faut des formats ouverts
  - facile à parcourir via un navigateur pour les utilisateurs
  - Et surtout simple à faire évoluer pour les rédacteurs
  - Idéalement au même endroit que le code pour être versionné

$\longrightarrow$ utilisation intensive du markdown et des sites statiques

Quelques exemples

- Sphinx et Doxygen génèrent des sites statiques
- MkDocs : à partir de markdown, outil simple et très utilisé pour la documentation
- Hugo, Jekyll (intégré au GitHub Pages) ...

Gitlab / GitHub Pages

- Permettent de publier un site web statique, construit par exemple avec un générateur de site statique, sur une url avec le nom du projet
  - En particulier publier automatiquement de la documentation de code
- Mettre en ligne des pdf (articles de livres, support de cours, etc.) construits à partir de sources (Tex, markdown, etc.)
- Permet de faire de l'édition collaborative
- Permet de versionner le contenu
- Permet reconstruire le site à chaque commit et d'avoir de la documentation à jour grâce à l'intégration continue
- Pas de problèmes de sécurité grâce à l'approche statique

Les limites

- Pas de contenu web dynamique
- Une taille maximale de contenu publiable

Plan

Utiliser un LLM pour générer de la documentation

Pourquoi utiliser un LLM ?

- La documentation est souvent un point faible, une tâche rébarbative
- C'est aussi quelque chose qui peut être en grande partie automatisée

$\longrightarrow$ Les LLM peuvent offrir une aide efficace

Les cas d'usage les plus adaptés

- Générer des commentaires pour les fonctions et les classes, pour des sections de code
- Rédiger des fichiers README ou les docstring
- Traduire la documentation en plusieurs langues
- Générer les fichiers de métadonnées (*codemeta.json*, *CITATION.cff* ...)

Les outils basés sur les LLM

- Outils intégrés dans des IDE :
  - GitHub Copilot (intégré dans VS Code, PyCharm ...)
  - Amazon CodeWhisperer
- Interfaces en ligne :
  - ChatGPT / Claude / Gemini
  - Mistral / Codestral

Exemples de prompt

Veiller à fournir du contexte : objectifs du code, cas d'usage ..., et rédiger des instructions claires

- « Explique ce que fait cette fonction »
- « Crée un README minimal pour ce script »
- « Liste les dépendances implicites utilisées ici »

Bonnes pratiques et limites

Bonnes pratiques

- Relire systématiquement les sorties
- Plus le prompt est contextualisé et précis, plus la sortie du LLM sera pertinente
- Préférer l'usage local (on peut installer localement des modèles de taille raisonnable) en particulier pour les données sensibles.

Limites

- La compréhension du code peut être partielle, complexe ou erronnée
- Il peut y avoir des biais vers des formulations génériques ou carrément mauvaises
- Les LLM propriétaires peuvent intégrer les données pour leur apprentissage donc en avoir conscience et ne pas les utiliser quand vous avez des données personnelles dans vos codes / prompts
- Le coût environnemental de ces outils est énorme : les utiliser à bon escient !

Plan

Archivage de code

- Les forges ne sont pas des plateformes d'archivage : elles peuvent disparaître
- Archiver un code permet de pérenniser son accès sur le long terme
- Software Heritage archive les codes (et tout leur contexte de développement) en particulier en moissonnant automatiquement un grand nombre de forges
- Initiative d'INRIA soutenue par de nombreux établissements et entreprises

Software Heritage

Diffusion et référencement de code

- La diffusion du code peut se faire via l'url du repo (c'est très souvent le cas dans les publis)
- Mais la forge n'étant pas forcément pérenne, il est préférable de pointer sur l'archive correspondante sur Software Heritage
- Ce qui ne sera pas satisfaisant d'un point de vue référencement car on n'aura pas facilement accès à des informations synthétiques et controlées
- Le référencement permet de répondre à cette problématique en utilisant des dépôts comme HAL pour assurer la qualité des métadonnées associées au code

Lien avec la documentation

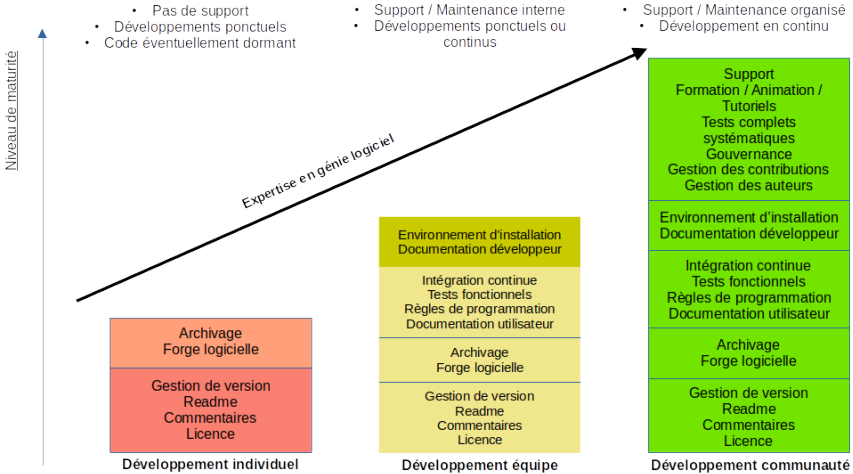
- C'est une autre forme de documentation qui vise à rendre visible le logiciel de façon différente
- Outre la description, les métadonnées de référencement permettent :
  - d'associer le logiciel à des métadonnées auteur et affiliation liées à des référentiels controlés
  - d'inscrire le logiciel dans les mêmes écosystèmes que les autres produits de recherche, données et publications
  - de faire des liens simples entre toutes ces productions
- La citation est aussi facilitée :
  - soit via les exports proposés par HAL (ou d'autres archives ouvertes)
  - soit directement avec Software Heritage si il y a un fichier *codemeta.json* dans le dépôt renseignant les métadonnées
  - L'utilisation de l'identifiant de Software Heritage, le SWHID, pour pointer le code sur l'archive, permet de désigner précisément un commit, un fichier voir même quelques lignes de codes selon le besoin

Plan

Conclusions

- Il y a pléthore de types de documentation qui ont chacun un rôle complémentaire
- Les cibles de ces différentes formes ont aussi des fonctions diverses dans la vie du code et donc des attentes variées en terme de documentation
- Il existent pas mal d'outils pour aider, structurer, déployer, faciliter la mise en oeuvre d'une documentation pertinente par les auteurs et contributeurs de codes
- Evidemment tout n'est pas à mettre en place pour tous les codes !

Différents contextes de développement pour les codes de recherche



Checklist pour un code perso

Créé par Gemini

Checklist pour un code perso

✓ README

- Avec au minimum la description des dépendances
- et la procédure d'installation

Créé par Gemini

Checklist pour un code perso

- ✓ README
  - Avec au minimum la description des dépendances
  - et la procédure d'installation
- ✓ Commentaires dans le code

Créé par Gemini

Checklist pour un code perso

- ✓ README
  - Avec au minimum la description des dépendances
  - et la procédure d'installation
- ✓ Commentaires dans le code
- ✓ Licence

Créé par Gemini

Checklist pour un code perso

- ✓ README
  - Avec au minimum la description des dépendances
  - et la procédure d'installation
- ✓ Commentaires dans le code
- ✓ Licence
- ✓ Notice HAL avec lien sur Software Heritage

Créé par Gemini

Checklist pour un code d'équipe

Créé par Gemini

Checklist pour un code d'équipe

✓ Casse et règles de nommage communes

Créé par Gemini

Checklist pour un code d'équipe

- ✓ Casse et règles de nommage communes
- ✓ Fichier *AUTHORS*

Créé par Gemini

Checklist pour un code d'équipe

- ✓ Casse et règles de nommage communes
- ✓ Fichier *AUTHORS*
- ✓ Automatisation via Doxygen / Sphinx

Créé par Gemini

Checklist pour un code d'équipe

- ✓ Casse et règles de nommage communes
- ✓ Fichier *AUTHORS*
- ✓ Automatisation via Doxygen / Sphinx
- ✓ Documentation utilisateur

Créé par Gemini

Checklist pour un code d'équipe

- ✓ Casse et règles de nommage communes
- ✓ Fichier *AUTHORS*
- ✓ Automatisation via Doxygen / Sphinx
- ✓ Documentation utilisateur
- ✓ Documentation développeur

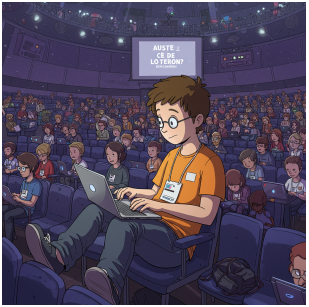
Créé par Gemini

Checklist pour un code d'équipe

- ✓ Casse et règles de nommage communes
- ✓ Fichier *AUTHORS*
- ✓ Automatisation via Doxygen / Sphinx
- ✓ Documentation utilisateur
- ✓ Documentation développeur
- ✓ Déploiement sur les Pages ou site pour la documentation

Créé par Gemini

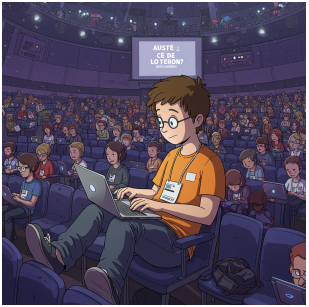
Checklist pour un code communautaire



Créé par Gemini

Checklist pour un code communautaire

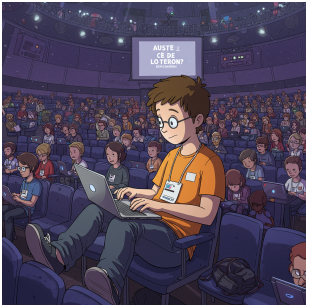
✓ Formalisation des dépendances (via *requirements.txt* par exemple)



Créé par Gemini

Checklist pour un code communautaire

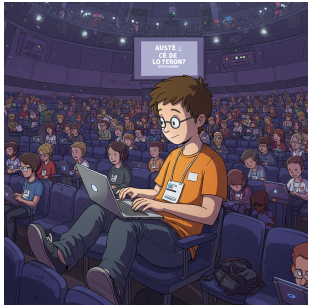
- ✓ Formalisation des dépendances (via *requirements.txt* par exemple)
- ✓ Fichier *CHANGELOG*



Créé par Gemini

Checklist pour un code communautaire

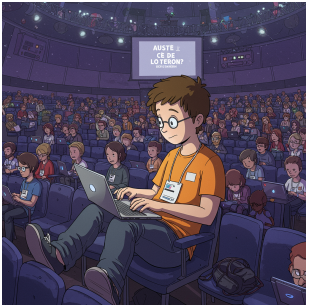
- ✓ Formalisation des dépendances (via *requirements.txt* par exemple)
- ✓ Fichier *CHANGELOG*
- ✓ Fichier *CONTRIBUTING* ou documentation contributeur
  - Avec si possible un fichier *CODE_OF_CONDUCT*



Créé par Gemini

Checklist pour un code communautaire

- ✓ Formalisation des dépendances (via *requirements.txt* par exemple)
- ✓ Fichier *CHANGELOG*
- ✓ Fichier *CONTRIBUTING* ou documentation contributeur
  - Avec si possible un fichier *CODE_OF_CONDUCT*
- ✓ Fichier de métadonnées (par exemple *codemeta.json*)



Créé par Gemini