Bonnes pratiques et tests en Python

1 Objectifs du TP

Ce TP a pour but de voir comment *bien* démarrer un projet de développement en Python. Le but est ici de se renseigner sur diverses *bonnes pratiques* dans ce langage, et de rendre plus concrètes les notions de qualité logicielle vues en *cours*. Ces bonnes pratiques et l'utilisation des technologies vues dans ce TP seront demandées dans votre projet.

Les points essentiels du TP sont :

- Quelles sont les *bonnes* pratiques en Python ?
- Comment *packager* et rendre distribuable son code sur PyPI ?
- Comment tester son code via pytest ?
- Comment utiliser un environnement logiciel via virtualenv ?

Tout comme le TP précédent, ce TP n'est pas une suite d'instructions précises. Il est plutôt une ouverture vers des ressources à approfondir, ainsi que des problèmes techniques auxquels on vous demande d'apporter une solution. Vous serez libre de vos choix d'implémentation mais ces choix devront être argumentés.

Ce TP est à faire **seul** et sera **évalué** (pas directement à la fin de la séance, vous avez le temps de finir le TP). Vous avez le droit de me poser des questions et de demander de l'aide pendant le TP. Le rendu est indirect, vous créerez un dépôt Git sur gitlab.com et me donnerez accès à votre dépôt (comme détaillé plus bas dans le sujet de TP).

- Rédigez vos réponses aux différentes questions dans le README du projet.
- Lorsque différents choix techniques vous sont proposés, vos choix techniques en eux-mêmes ne seront pas évalués, mais la qualité de votre argumentation le sera. Je n'attends évidemment pas de vous de preuve mathématique ni d'évaluation scientifique par l'expérimentation pour étayer vos arguments. J'attends par contre que vous citiez vos sources et commentiez la pertinence qu'elles ont selon vous.

2 Bonnes pratiques en Python

Comme de nombreux langages, Python vient avec un ensemble de *bonnes pratiques*. Suivre ces pratiques permet de se faire comprendre facilement des autres développeurs de la communauté, et *simplifie la vie* des développeurs, puisqu'elles ont pour but d'éviter les erreurs courantes. Voici une partie importante des bonnes pratiques en Python.

Comment structurer ses fichiers et comment rendre son code distribuable ? Python est assez souple là-dessus et ne vous force pas à respecter une certaine structure dans votre code, même si certaines structures sont très courantes. On va étudier une de ces structures de code dans la suite du TP et montrer qu'elle permet de créer un *package* Python rapidement.

Quel style utiliser pour le code lui-même ?
Là encore, Python ne vous force pas la main sur le style à utiliser. La référence là-dessus est la PEP 8 qui favorise la lisibilité du code, ce qui est cohérent avec l'idée qu'un code doit être lisible et modifiable par des humains pour conserver une bonne qualité logicielle. La PEP8 insiste elle-même sur sa position de **recommandation** : s'il est plus lisible de ne **pas** respecter la PEP8 sur un bloc de code, privilégiez la lisibilité. De nombreux outils Python permettent de vérifier si votre code suit les recommandations de la PEP 8, dont Pylint. Pylint est un linter, un outil qui analyse un programme pour y rechercher des erreurs ou des mauvaises pratiques. Se servir de ce type d'outil en très utile en Python comme avec d'autres langages interprétés, car l'absence de phase de compilation cache de nombreuses erreurs détectables avant que le code en question soit exécuté — ce qui peut prendre très longtemps si ce code n'est appelé que de manière conditionnelle et rare.

Comment découper son code en modules ? Quelles fonctionnalités du langage utiliser selon ce que l'on souhaite faire ?
Cette partie s'apprend surtout par l'expérience, en regardant les choix faits dans d'autres projets ou lorsque son code est *reviewé* par un expert du langage. La documentation de Python contient elle-même de nombreuses ressources intéressantes, comme la documentation officielle des modules. Il existe aussi de nombreux tutoriaux sur la question comme celui-ci qui indroduit ce qu'est un idiome de programmation et donne de nombreux exemples sur ceux recommandés en Python. Des recherches autour du *pythonic way* ou sur comment rendre son code *more pythonic* devraient vous montrer beaucoup de ressources sur le sujet.

Comment documenter son code ?
La documentation du code a pour but d'expliquer aux utilisateurs (et aux développeurs) les rôles et détails des différentes entités du code. Par exemple, la documentation d'une fonction détaille un minimum ce que fait une fonction, ce qu'elle renvoie et quels paramètres elle accepte. Être précis dans cette documentation est très important en Python puisque le langage lui-même ne vous aide pas beaucoup à spécifier de nombreuses contraintes du code, comme le

type renvoyé par une fonction ou celui de ses paramètres[1]. Une autre information cruciale concerne les contraintes d'appel des différentes entités, ce qui se manifeste souvent par des préconditions comme on le verra en programmation par contrat dans la suite du cours. Rédiger la documentation se fait directement dans le code en Python via des *docstrings*, qui sont standardisés dans la PEP 257 et d'autres documents, comme le PEP 316 pour la programmation par contrat en Python.

3 Partie évaluée

Dans cette partie, vous allez créer un *package* Python correspondant à un code jouet dont le comportement est spécifié dans le paragraphe suivant.

Nous voulons faire un script `hello.py` en Python 3 qui dit bonjour à son utilisateur. Ce script a un paramètre optionnel : le nom de la personne à saluer, sous forme de chaîne de caractères. Si la personne à saluer est `Alice`, le script doit afficher `Hello, Alice!` et terminer (en renvoyant un code d'erreur qui signifie "pas d'erreurs").

- Si le nom de la personne à saluer n'est pas spécifié, le nom d'utilisateur (au sens système d'exploitation) de l'utilisateur doit être utilisé à la place.
- Si le nom de la personne à saluer dépasse 16 caractères, le script doit afficher `Hello, Long-named folk!` au lieu du nom d'utilisateur.

3.1 Instructions

1. Créez un dépôt Git pour ce projet `hello` sur gitlab.com et ajoutez-moi (`mpoquet`) en tant que développeur sur votre projet. Clonez ce dépôt sur votre machine locale. Faites ensuite un commit pour chacune des instructions suivantes si nécessaire, et poussez régulièrement vos commits sur le serveur.

2. Écrivez une première version de `hello.py` sous forme de script appelable. Le paramètre optionnel doit pouvoir être donné en tant que paramètre positionnel **optionnel** du script — votre script doit être appelable via `python3 hello.py Alice`, en supposant que `python3` soit un interpréteur Python 3 dans votre `PATH`.

    - Ce tutoriel montre différentes bibliothèques courantes pour définir une interface en ligne de commande (CLI) en Python. Vous pouvez également vous servir de toute autre bibliothèque du même genre, ou choisir de vous passer de telles bibliothèques. **Argumentez** (concisément) **votre choix de bibliothèque dans le README de votre**

[1] On peut spécifier des *type hints* optionnels depuis quelques années en Python. Les utiliser est conseillé car ils devraient à terme permettre de vérifier automatiquement ces informations de type lors de l'exécution du code. Les raisons qui ont poussé à introduire les *type hints* sont argumentées dans la PEP 483, qui en montre également des exemples de syntaxe.

projet. En particulier, quels sont les avantages/défauts que vous imaginez a priori dans les différentes alternatives que vous avez étudié ? Quels sont les impacts positifs et négatifs sur la qualité logicielle de votre programme si vous choisissez d'utiliser une bibliothèque de CLI ou de vous en passer ?

- Comment obtenez-vous le nom d'utilisateur lorsque le nom de la personne à saluer n'est pas donné ? Pourquoi avez-vous opté pour ce choix technique ?

3. Faites en sorte que votre script soit exécutable directement, par exemple via `./hello.py Bob`.

   - Argumentez votre choix de shebang dans votre README. Quelles ressources vous ont permis de croire que vous faisiez le bon choix ? À quel point faites-vous confiance à ces ressources et pourquoi ?

4. On souhaite désormais que `hello.py` soit, en plus d'être un script appelable directement, un module Python utilisable depuis un autre code Python. On va pour cela créer une fonction `hello` dans un module lui-aussi nommé `hello`, qui **renvoie** la chaîne de caractères affichée par le script précédent. La gestion de la ligne de commande doit elle être placée dans une fonction dédiée appelée `cli`. Vous devez faire en sorte que votre fonction `hello` soit exécutable depuis un autre module Python tout en conservant le comportement précédent lorsque le script `hello.py` est appelé (dans ce cas, la fonction `hello` doit également être appelée, et ce qu'elle retourne doit être affiché). Note : quand le module `hello` est importé, aucun n'affichage intempestif ne doit être produit.

   - Quelle est la manière *pythonique* de gérer le fait que que la personne à saluer est optionnelle ?
   - Quelle est la manière *pythonique* d'appeler un bout de code uniquement lorsqu'un module python est appelé en tant que script ?

5. Utilisez Pylint sur votre script et corrigez les erreurs et avertissements éventuels, de sorte que votre code respecte la PEP 8 (sauf pour des soucis de documentation manquante). Si vous pensez qu'appliquer certaines recommandations de la PEP 8 serait nocif à la lisibilité de votre code, listez-les et dites pourquoi.

6. Documentez votre script grâce à Pylint, à la PEP 257 ou à d'autres outils/ressources que vous pouvez trouver en ligne.

7. Grâce à cet exemple presque minimal de projet Python packagé et la documentation officielle du packaging en Python, écrivez un `setup.py` qui définit un paquet Python qui contient à la fois un module `hello` importable et un script exécutable `hello.py`. Note : utiliser une approche descriptive (en listant les différents fichiers à *packager*) comme celle de l'exemple presque minimal fourni est fortement recommandé. Si vous préférez utiliser une autre approche, argumentez ;). Note : si votre code a des dépendances

qui ne sont pas dans la bibliothèque standard Python, vous aurez besoin de les indiquer dans votre `setup.py`.

- Grâce à virtualenv, placez-vous dans un environnement virtuel Python isolé vide. Dans cet environnement, essayez d'installer votre projet grâce à `pip install .` (en supposant que vous êtes à la racine de votre projet Git et que votre `setup.py` se trouve à la racine de votre projet). Cet environnement vous permet de voir si votre `setup.py` fonctionne puis de vérifier qu'à la fois votre script et votre module fonctionnent correctement. Après un appel réussi à `pip install .`, appeler `hello.py` doit avoir le résultat escompté, tout comme le fait d'importer votre module et d'appeler votre fonction depuis un interpréteur Python lancé depuis l'intérieur de l'environnement virtuel.

8. Écrivez des tests unitaires pytest pour votre fonction `hello` dans un fichier `test_hello.py`. Vérifiez que votre code passe vos tests en appelant `pytest`.

9. Générez un rapport de *coverage* grâce à pytest. On souhaite ici savoir quelles instructions sont appelées ou non par nos tests. Si toutes les instructions de votre fonction `hello` ne sont pas couvertes par vos tests, faites en sorte que toute votre fonction soit couverte (soit en ajoutant des cas de test manquants, soit en supprimant du code inutile).

    - Avez-vous eu besoin d'installer des paquets supplémentaires pour avoir un rapport de *coverage* avec pytest ? Si oui lesquels ? Quelle commande lancez-vous pour obtenir un rapport de *coverage* utile ici ?