OUTILS D'ANALYSE ET DE MESURE DE LA QUALITÉ DU CODE

OUTILS D'ANALYSE ET DE MESURE DE LA QUALITÉ DU CODE

Intérêt de ces outils / métriques

- Approche complémentaire aux tests
- Détection d'anomalies différentes de celle des tests

Plusieurs types d'analyse peuvent se compléter

- Analyse syntaxique / normative
- Analyse statique et dynamique
- Approche orientée indicateurs : couverture de code, calcul de la complexité
- Détection des problèmes de sécurité

PREMIÈRE APPROCHE : LES OUTILS D'ANALYSE SYNTAXIQUE

L'ANALYSE SYNTAXIQUE

- Dans tous les langages, des normes/conventions syntaxiques existent et sont intégrées dans les IDE ou dans des outils dédiés
- Pour Python, les "recommandations" PEP8 et PEP257 sont intégrées dans quasiment tous les IDE
- PEP 8 fournit des conventions de codage pour le code Python.
- PEP 257 décrit les conventions des docstrings de Python, qui sont des chaînes de caractères destinées à documenter les modules, classes, fonctions et méthodes.

PEP 8 : PYTHON EXTENSION PROPOSAL 8

PEP 8 : MISE EN PAGE

- Une ligne doit contenir 80 caractères maximum.
- L'indentation doit être de 4 espaces.
- Ajoutez deux lignes vides entre deux éléments de haut niveau, des classes par exemple, pour des questions d'ergonomie.
- Séparez chaque fonction par une ligne vide.
- Les noms (variable, fonction, classe, ...) ne doivent pas contenir d'accent. Que des lettres ou des chiffres !

PEP 8 : DOCUMENTATION

Selon la PEP 8, chaque partie de votre code devrait contenir une Docstring - voir PEP 257

- tous les modules publics
- toutes les fonctions
- toutes les classes
- toutes les méthodes de ces classes

PEP 8 : LES IMPORTS

- L'import d'une librairie doit être rapide à déceler
- Il est également important de bien différencier la source des librairies : *standard, externe ou locale*
- Les imports sont à placer au début d'un script.
- Ils précèdent les Docstrings.
- Une ligne par librairie.
  Exemple : `import os`
- Une ligne peut néanmoins inclure plusieurs composantes.
  Exemple : `from subprocess import Popen, PIPE`
- L'import doit suivre l'ordre suivant : *Bibliothèques standard, Bibliothèques tierces et imports locaux.* Sautez une ligne entre chacun de ces blocs.

PEP 8 : LES ESPACES DANS LES INSTRUCTIONS

Les espaces suivent la syntaxe anglosaxone et non française. De manière plus générale, elle s'axe sur la lisibilité tout en supprimant les espaces surperflus.

- Pas d'espace avant `:` mais un après.
  Exemple : `{oeufs: 2}`
- **Opérateurs**: un espace avant et un après.
  Exemple : `i = 1 + 1`
- Aucun espace avant et après un signe `=` lorsque vous assignez la valeur par défaut du paramètre d'une fonction.
  Exemple : `def elephant(trompe=True, pattes=4)`
- Une instruction par ligne.

PEP 8 : LES COMMENTAIRES

- Ecrivez des phrases complètes, ponctuées et compréhensibles.
- Le commentaire doit être cohérent avec le code.
- Il doit suivre la même indentation que le code qu'il commente.
- Evitez d'enfoncer des portes ouvertes : ne décrivez pas le code, expliquez plutôt à quoi il sert.
- Il doit être en anglais.

PEP 8 : LES CONVENTIONS DE NOMMAGE

- **Modules** : nom court, tout en minuscules, tiret du bas si nécessaire. `great_module`
- **paquets** : nom court, tout en minuscules, tirets du bas très déconseillés. `paquet`
- **classes** : lettres majuscules en début de mot. `MyGreatClass`
- **fonctions** : minuscules et tiret du bas : `my_function()`
- **méthodes** : minuscules, tiret du bas et `self` en premier paramètre : `my_method(self)`
- **arguments des méthodes et fonctions** : identique aux fonctions. `my_function(param=False)`
- **variables** : identique aux fonctions.
- **constantes** : tout en majuscules avec des tirets si nécessaire. `I_WILL_NEVER_CHANGE`
- **privé** : précédé de deux tirets du bas : `__i_am_private`
- **protégé** : précédé d'un tiret du bas : `_i_am_protected`

PEP 257

- Une docstring est un ensemble de mots qui documente un bout de code.
- Elle commence par trois guillemets ouvrants, le commentaire que vous souhaitez apporter puis trois guillements fermants.
- Les doctrings peuvent être utilisés pour générer de la doc automatiquement

```
def maFonction(a, b):
    """
    Cette fonction est une fonction de test
    Elle sert a calculer a + b
    :param a : Valeur 1
    :param b : Valeur 2
    :return : Somme des Valeur 1 et Valeur 2
    """
    return(a + b)

print(maFonction(1, 2))
```

ANALYSE STATIQUE ET DYNAMIQUE

DÉFINITION

- L'**analyse statique** de code est une analyse ne nécessitant pas d'exécution du code et permettant de détecter des erreurs de conception et d'implémentation dans le code
- L'**analyse dynamique** de code a le même objectif mais nécessite une exécution du code
- Ces deux types peuvent regrouper différentes techniques d'analyse ayant chacune leurs atouts, par exemple :
  - l'analyse statique peut permettre de réaliser de la preuve formelle
  - l'analyse dynamique peut permettre de détecter empiriquement les problèmes de gestion de la mémoire
- Le premier objectif de ces outils est d'augmenter la fiabilité du code

VÉRIFICATIONS COURANTES

- Division par zéro
- Débordement mémoire
- Erreur de typage
- Variable non-initialisées
- Branches inatteignables/boucles infinies
- Index hors limites
- Fuites mémoires
- ...

APPROCHE ORIENTÉE INDICATEURS

LA COUVERTURE DE CODE

La couverture de code est une mesure utilisée pour déterminer le taux de code source exécuté lorsqu'une suite de tests est lancée. Pour essayer de limiter les bugs, les tests doivent couvrir une large proportion de code. La couverture du code par les tests est un indicateur de la qualité des tests et non du code. Différentes méthodes existent :

- **Couverture des fonctions (ou méthodes)** : est-ce que toutes les méthodes du code ont été appelées par les tests ?
- **Couverture des instructions** : est-ce que les tests sont passés sur chaque ligne de code ?
- **Couverture des chemins d'exécution (décisions)** : est-ce que l'on est passé dans toutes les branches de notre code ? Par exemple, l'instruction `if` génère deux branches de code : une dans laquelle la condition évaluée est vraie, une autre où la condition est fausse.
- **Couverture des points de tests (MCDC)** : est-ce que chaque condition sur le test d'une variable a été couverte ?

L'ANALYSE DE LA COMPLEXITÉ

- La complexité cyclomatique d'une méthode est définie par le nombre de chemins linéairement indépendants qu'il est possible d'emprunter dans cette méthode.
- Plus simplement, il s'agit du nombre de points de décision de la méthode (if, case, while, ...) + 1 (le chemin principal).
- La complexité cyclomatique d'une méthode vaut au minimum 1, puisqu'il y a toujours au moins un chemin.
- Une complexité cyclomatique trop élevée (supérieure à 30) indique qu'il faut refactoriser la méthode. Une complexité cyclomatique inférieure à 30 peut être acceptable si la méthode est suffisament testée.
- La complexité cyclomatique est liée à la notion de "code coverage", c'est à dire la couverture du code par les tests. Dans l'idéal, une méthode devrait avoir un nombre de tests unitaires égal à sa complexité cyclomatique pour avoir un "code coverage" de 100%. Cela signifie que chaque chemin de la méthode a été testé.

L'ANALYSE DE LA DUPLICATION DE CODE

- La duplication de code est une suite d'instructions similaires dans des endroits distincts du code
- Le code dupliqué pose des problèmes de maintenance dont l'importance augmente avec la quantité de code dupliqué. Plus le code est dupliqué, plus il y a de code à maintenir.
- Il est nécessaire de factoriser le code pour éviter ces duplications

ANALYSE DU NIVEAU DE SÉCURITÉ

- La notion de qualité d'une application repose aussi sur la notion de sécurité . Différentes métriques peuvent être mises en place pour :
  - détecter les vulnérabilités de composants tiers (CVE : Common Vulnerabilities and Exposures)
  - détecter des problèmes potentiels d'injection de dépendances, d'injection SQL ...
  - détecter des mauvaises configurations, des fichiers non sécurisés ...

LES OUTILS

OUTILS PAR LANGAGE

- **Python :** pylint, SonarQube, ...
- **Java :** SonarQube, Infer, SpotBugs, ...
- **C/C++ :** Cppcheck, Infer, Frama-C, PolySpace, Understand, ...

OUTIL PYTHON : Pylint

- Vérification des standards : PEP8
- Erreurs de code
- Problème de syntaxe, d'indentation
- Modèles de codes dangereux
- Détection des mauvaises pratiques de programmation
- Duplication de code
- Code non conforme aux conventions définies

- **Outils similaires :** Flake8, Pylama
- **Outils complémentaires:** Bandit pour la partie sécurité, radon pour la complexité
- Ces outils sont disponibles pour la plupart en ligne de commande ou sous forme de plug-ins dans les IDE

OUTILS GÉNÉRIQUES

SonarQube est un logiciel libre* permettant de mesurer la qualité du code source en continu sur de nombreux langages

- identification des duplications de code
- mesure du niveau de documentation
- respect des règles de programmation
- détection des bugs potentiels
- évaluation de la couverture de code par les tests unitaires
- analyse de la répartition de la complexité
- analyse du design et de l'architecture d'une application
- mesure d'une dette technique

BONNES PRATIQUES

- Les outils présentés sont en général utilisés dans la partie CI/CD en complèment des tests unitaires
- Les résultats / rapports peuvent être utilisés lors de la revue de code
- Les indicateurs ne sont pas forcément tous pertinents à l'instant T
- Il convient de les utiliser en fonction du contexte du projet et de la maturité de celui-ci