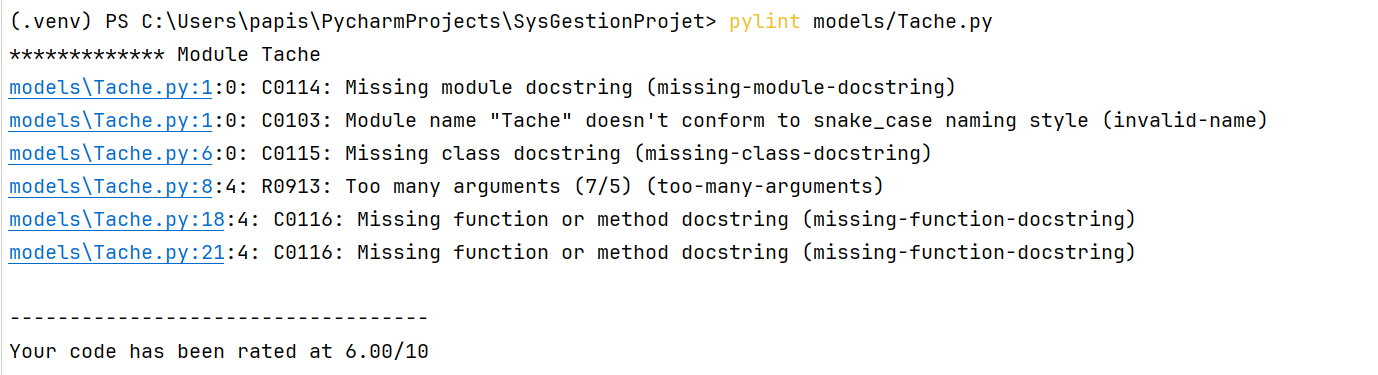
**Rapport de test, mesure et qualité du code**

***Analyse du code avec les outils de mesure de qualité.***

* Flake8

Flake8 est un outil d'analyse statique de code Python qui vérifie le respect des conventions de style (PEP 8) et d'autres bonnes pratiques de codage. Il permet d'identifier et de signaler les problèmes de style, les erreurs de syntaxe et d'autres problèmes potentiels dans le code source.

On a pris comme exemple le code de classe Projet

Avant la correction:

L'analyse de Flake8 sur le fichier `models/Projet.py` a généré plusieurs erreurs et avertissements :

1. `F401`: Ces deux erreurs signalent que les classes `EMailNotificationStrategy` et `SMSotificationStrategy` ont été importées, mais ne sont pas utilisées dans le fichier. Cela peut indiquer un code inutile ou des importations obsolètes.

2. `E302`: Cette erreur indique qu'il manque une ligne vide entre la définition de la classe et la première méthode, ce qui ne respecte pas la convention de style PEP 8.

3. `E501`: Ces trois erreurs signalent que certaines lignes de code dépassent la limite de longueur recommandée de 79 caractères, ce qui peut nuire à la lisibilité du code.

4. `W391`: Cet avertissement indique la présence d'une ligne vide à la fin du fichier, ce qui n'est généralement pas recommandé.

Après la correction:

Après avoir corrigé les problèmes signalés par Flake8, le code devrait être conforme aux conventions de style PEP 8 et aux bonnes pratiques de codage. Voici les corrections possibles:

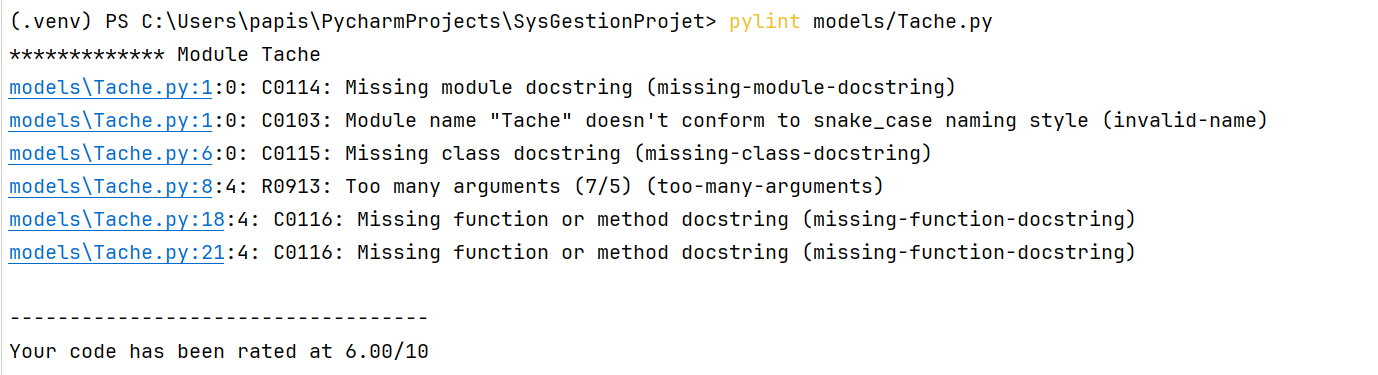
1. Supprimer les importations inutiles ou utiliser les classes importées dans le code.

2. Ajouter une ligne vide supplémentaire avant la définition de la classe pour respecter la convention de style.

3. Diviser les longues lignes de code en plusieurs lignes, en utilisant des parenthèses ou des backslashes pour envelopper les chaînes de caractères longues.

4. Supprimer la ligne vide à la fin du fichier.

En appliquant ces corrections, le code sera plus lisible, plus maintenable et conforme aux bonnes pratiques de codage.

* PYLINT

Analyse avant correction

1. C0114: Missing module docstring (missing-module-docstring)

Description: Le module n'a pas de docstring, ce qui rend difficile la compréhension de son objectif global.

1. C0103: Module name "Tache" doesn't conform to snake\_case naming style (invalid-name)

Description: Le nom du module Tache ne respecte pas le style de nommage snake\_case recommandé en Python.

1. C0115: Missing class docstring (missing-class-docstring)

Description: La classe Tache n'a pas de docstring, ce qui rend difficile la compréhension de son rôle et de ses attributs.

1. R0913: Too many arguments (7/5) (too-many-arguments)

Description: La méthode \_\_init\_\_ de la classe Tache prend trop d'arguments, ce qui peut rendre le code difficile à lire et à maintenir.

1. C0116: Missing function or method docstring (missing-function-docstring)

Description: Les méthodes ajouter\_dependance et mettre\_a\_jour\_statut n'ont pas de docstrings, rendant leur usage moins clair.

Analyse après correction :

Module docstring ajouté:

Une docstring a été ajoutée au début du module pour fournir une description générale de ce que fait le module.

Nom du module en snake\_case:

Le nom du module a été changé de Tache à tache pour suivre la convention snake\_case.

Docstring de classe ajoutée:

Une docstring détaillée a été ajoutée pour la classe Tache, décrivant son rôle et ses attributs.

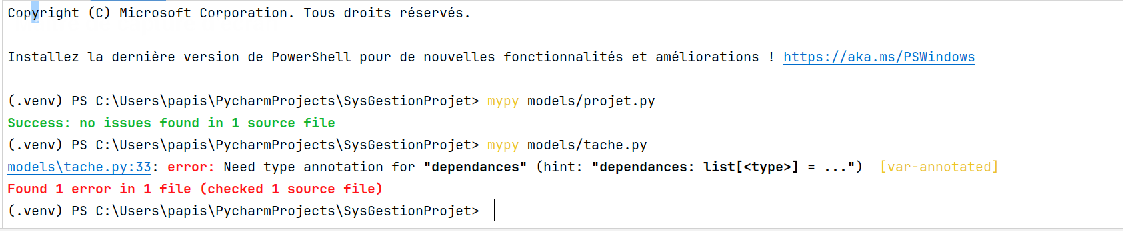
Arguments init limités:

Selon la norme pylint le nombre de paramètre doit être limitée à 5 m nous n’avons fait que suivre le modèle de classe fournit

Docstrings pour les méthodes ajoutées:

Des docstrings ont été ajoutées pour les méthodes ajouter\_dependance et mettre\_a\_jour\_statut pour décrire leurs paramètres et leurs fonctions.

Après ces corrections, le rapport pylint devrait afficher moins d'erreurs et obtenir un meilleur score.

* MYPY

Analyse avant correction

1. Need type annotation for "dependances":

Description: mypy indique que la variable dependances dans votre classe Tache nécessite une annotation de type. Sans cette annotation, mypy ne peut pas vérifier les types des éléments stockés dans dependances, ce qui peut entraîner des erreurs de typage difficiles à détecter.

Pour corriger cette erreur, nous devons ajouter une annotation de type pour la variable dependances.

Dans cette correction, nous avons ajouté l'annotation de type List[Tache] à la variable dependances pour indiquer qu'elle contient une liste d'objets de type Tache.

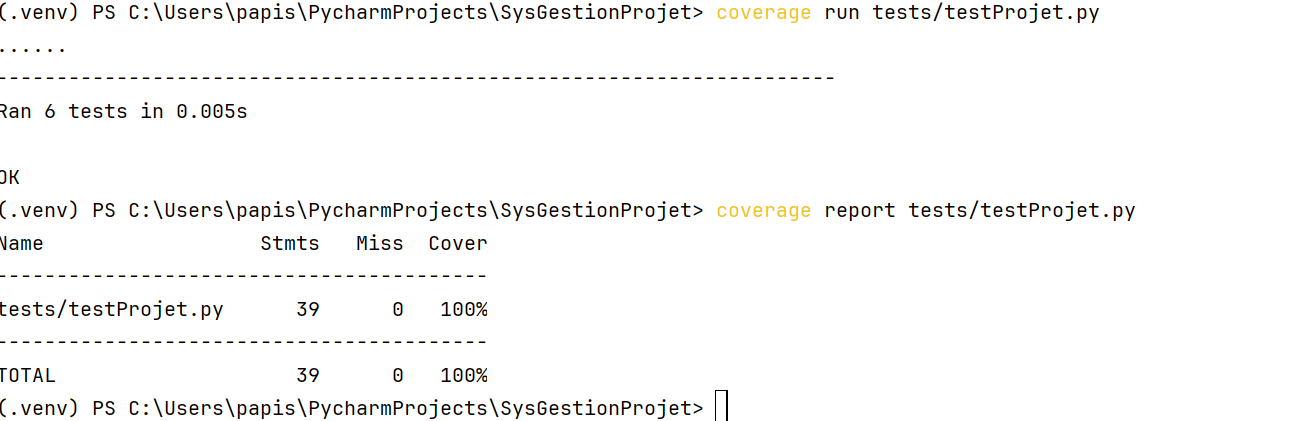
* Coverage :

Cette commande est utilisée pour exécuter les tests unitaires de votre projet avec la bibliothèque pytest et collecter des données de couverture de code avec la bibliothèque coverage.

coverage run -m models/tetProjet.py

coverage repport models/tetProjet.py

Voici le rendu :



Nous avons utilisé l'outil coverage.py pour mesurer la couverture des tests de notre fichier de test testProjet.py. La couverture des tests est essentielle pour s'assurer que notre code est bien testé et qu'il fonctionne correctement.

**Exécution des tests avec coverage** : Nous avons exécuté les tests en utilisant coverage.py pour mesurer combien de notre code est couvert par les tests. La commande utilisée est :

coverage run tests/testProjet.py

Exécution des Tests :

Commande : coverage run tests/testProjet.py

Sortie :

Ran 6 tests in 0.005s

OK

Cela indique que 6 tests ont été exécutés et qu'ils ont tous réussi.

Rapport de Couverture :

Commande : coverage report tests/testProjet.py

Sortie :

Name Stmts Miss Cover

-------------------------------------------

tests/testProjet.py 39 0 100%

-------------------------------------------

TOTAL 39 0 100%

Name : Nom du fichier testé.

Stmts : Nombre de déclarations (statements) dans le fichier.

Miss : Nombre de déclarations manquantes (non couvertes par les tests).

Cover : Pourcentage de couverture des tests.

Dans ce cas, testProjet.py contient 39 déclarations, toutes couvertes par les tests, ce qui donne une couverture de 100%.

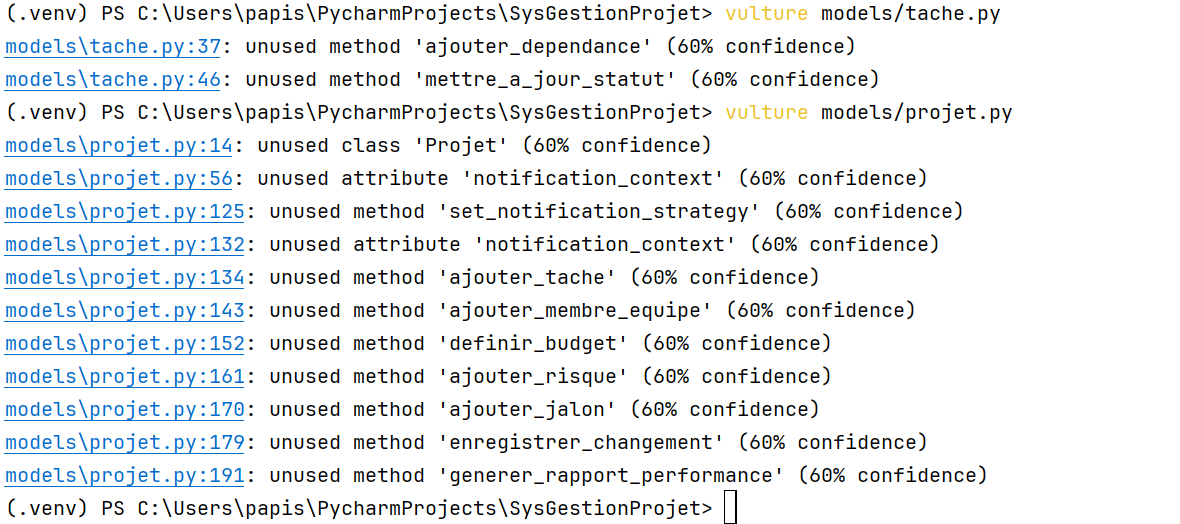
La couverture des tests pour le fichier testProjet.py est de 100%, ce qui signifie que tous les segments de code dans ce fichier ont été exécutés au moins une fois par les tests. Cela montre que notre suite de tests est complète pour ce fichier spécifique.

L'utilisation de coverage.py nous permet de garantir que notre code est bien testé et nous aide à identifier les parties du code qui pourraient nécessiter des tests supplémentaires.

* Vulture:

vulture est un outil statique d'analyse de code qui aide à trouver du code non utilisé dans les projets Python. Il peut identifier les fonctions, classes, variables, et autres éléments de code qui ne sont pas utilisés nulle part dans le projet, aidant ainsi à maintenir un code propre et sans fioritures.

Commande : vulture models/

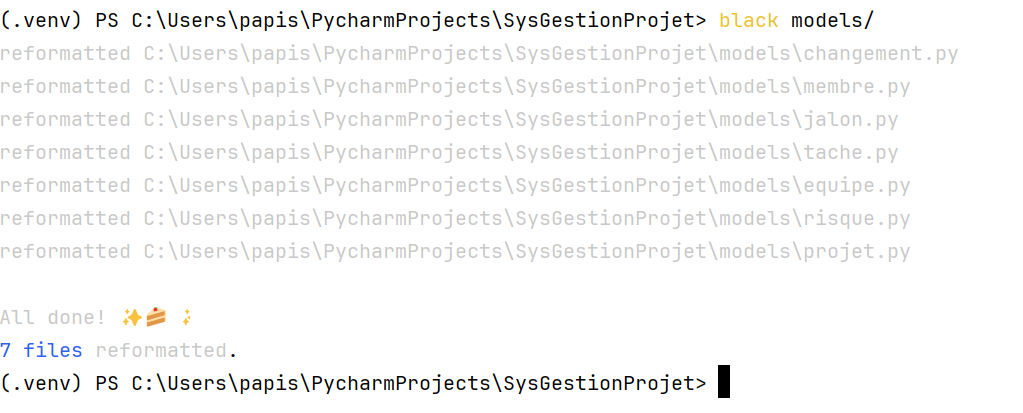


Après avoir exécuté Vulture sur les fichiers tache.py et projet.py, plusieurs éléments inutilisés ont été identifiés avec un niveau de confiance de 60%. Parmi eux, on trouve des méthodes comme ajouter\_dependance(), mettre\_a\_jour\_statut(), ajouter\_tache(), ajouter\_membre\_equipe(), definir\_budget(), ajouter\_risque(), ajouter\_jalon(), enregistrer\_changement() et generer\_rapport\_performance(). Des attributs comme notification\_context et la classe Projet ont également été signalés comme inutilisés.

En éliminant ces éléments inutiles, nous pourrons réduire la complexité du code, faciliter sa maintenance et potentiellement améliorer ses performances. Une revue approfondie du code sera nécessaire pour confirmer les résultats de Vulture et procéder aux suppressions appropriées.

* Black:

black est un formateur de code Python qui reformate le code pour qu'il soit conforme aux conventions de style de code. Il permet de s'assurer que tout le code dans un projet respecte les mêmes normes de style, améliorant ainsi la lisibilité et la maintenabilité du code.

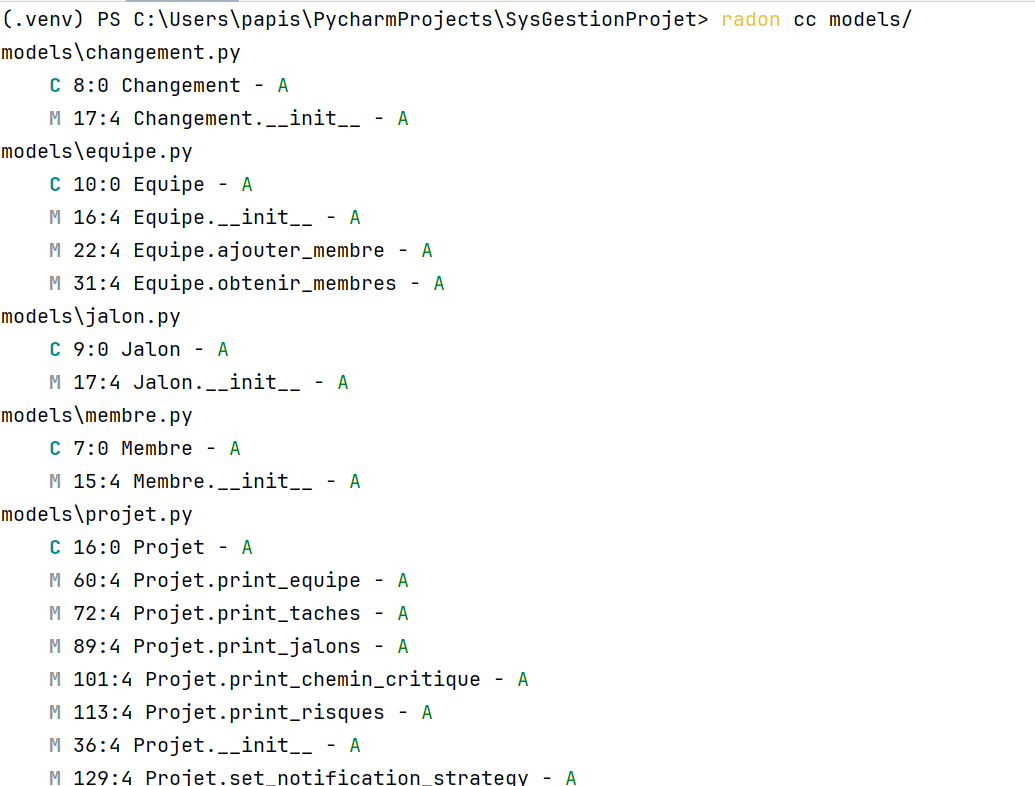
Commande : black .pour formater tous le code, 

L'utilisation de black nous a permet de garantir un style de code uniforme dans tout le projet, ce qui améliore la lisibilité et facilite la collaboration.

* Black:

radon est un outil d'analyse statique pour Python qui permet de mesurer la complexité cyclomatique, la complexité des lignes, et de détecter les fragments de code malodorants (code smells). Il est utile pour identifier les parties du code qui pourraient être difficiles à comprendre, à tester ou à maintenir.

**Analyse de la complexité cyclomatique des classes/méthodes**

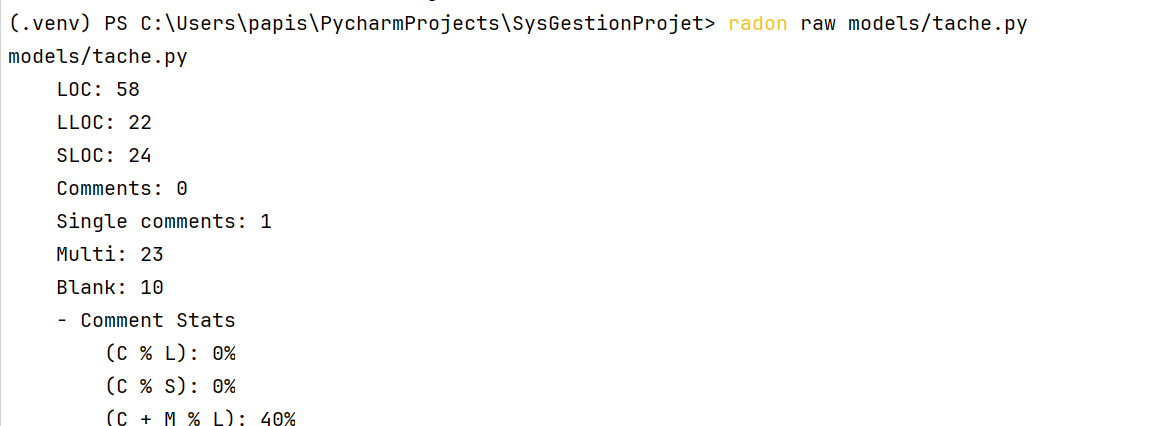
Commande :radon cc models/ pour mesurer la complexité cyclomatique du répertoire models

Cette sortie montre la complexité cyclomatique (notée A pour simple) des différentes classes et méthodes de nos modèles Python. Par exemple, pour la classe Projet, des méthodes comme print\_equipe, print\_taches, print\_chemins\_critique, etc. ont une complexité A (simple).

Une complexité cyclomatique élevée peut indiquer un code plus difficile à comprendre et à maintenir. Dans notre cas, toutes les classes et méthodes analysées ont une complexité A, ce qui est un bon signe de simplicité du code.

**Analyse de la complexité des lignes de la classe /méthodes/tache.py**

Commande :radon raw models/tache.py



LOC (Lines of Code) : 58 lignes de code

LLOC (Logical Lines of Code) : 22 lignes de code logiques

SLOC (Source Lines of Code) : 24 lignes de code source

Aucun commentaire dans le code

23 lignes de commentaires multi-lignes

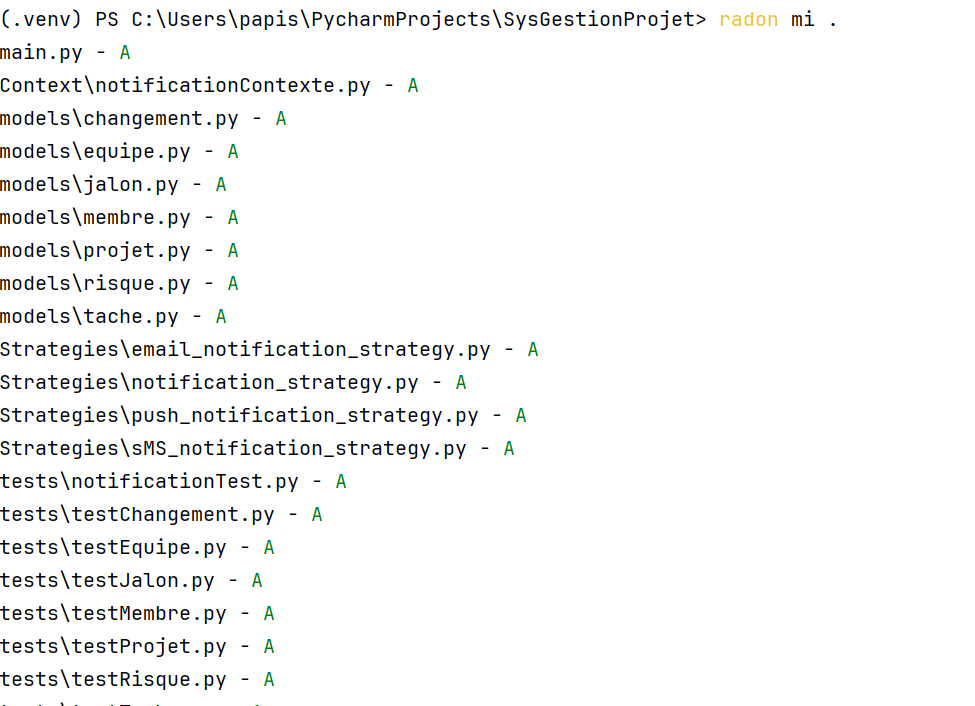
10 lignes blanches

40% du code est composé de commentaires et lignes blanches

Cette analyse suggère que le fichier tache.py contient un code relativement concis, avec peu de commentaires et de lignes blanches. Cependant, une proportion assez élevée (40%) du fichier est consacrée à la documentation sous forme de commentaires multi-lignes.

**Analyse des Fragments de Code Malodorants pour tout le projet**

Commande : radon mi . , pour détecter les mauvaises pratiques de codage



La sortie montre que tous les fichiers analysés ont une complexité cyclomatique notée A, ce qui est le niveau le plus bas et indique un code relativement simple et facile à comprendre.

Voici un résumé des différents fichiers/modules analysés :

main.py : Probablement le point d'entrée principal de votre application.

Context/notificationContexte.py : Module lié à la gestion des contextes de notification.

models/\* : Les différents modèles de données de votre application (changement, équipe, jalon, membre, projet, risque, tâche).

Strategies/\* : Les stratégies de notification différentes (email, push, SMS, etc.).

tests/\* : Les fichiers de tests unitaires pour vos différents modèles.

Le fait que tous ces fichiers aient une complexité cyclomatique A est un bon signe, indiquant que votre code est probablement bien structuré, modulaire et facile à maintenir.

* PYFlakes:

pyflakes est un outil d'analyse statique de code pour Python qui se concentre sur la détection des erreurs de programmation telles que les importations inutilisées, les variables non définies et d'autres erreurs courantes sans imposer de style de codage. Il est léger et rapide, ce qui en fait un excellent choix pour intégrer dans les pipelines d'intégration continue ou pour l'utiliser localement avant de soumettre du code.

Commande : pyflakes .

Dans notre cas la sortie est nulle car il considère que l’ensemble de notre code est correct, Ce qui peut s’expliquer par l’utilisation des outils précédemment présentés vu qu’ils nous on permit de corriger l’ensemble des défauts de notre code.