La méta-programmation logique sur du code source Python pour rechercher des bonnes ou mauvaises pratiques dans le code des étudiants

Nathan Corbisier

UCLouvain

Résumé. La capacité à interroger et extraire des informations à partir de code source de programmes rédigés par les étudiants revêt une importance significative, permettant ainsi d'automatiser la réponse aux questions suivantes : "Ce code contient-il des erreurs?", "Est-il conforme aux conventions d'écriture en Python?", "Présente-t-il de mauvaises pratiques?", "L'étudiant a-t-il substitué un 'return' par un 'print'?", etc. Nous proposons d'explorer la technique de la méta-programmation logique pour écrire des requêtes dédiées à l'analyse de pratiques spécifiques dans le code des étudiants. L'idée sous-jacente est de traduire du code Python en une base de données logique, et ensuite de pouvoir interroger cette base de données au moyen de différentes requêtes écrites dans un langage de programmation logique.

Introduction. Les cours d'introduction à la programmation utilisent régulièrement des correcteurs automatiques pour aider les étudiants en leur fournissant des retours automatisés sur leurs exercices de programmation. Ces retours, conçus pour aider les étudiants à tirer des enseignements de leurs erreurs et à les rectifier, devraient être de haute qualité. Toutefois, ces correcteurs ont généralement comme objectif principal de vérifier la validité du code par rapport à la question posée. Cependant, tenir compte et corriger les pratiques moins souhaitables, telles que l'utilisation de réponses hardcodées, est tout aussi important.

Notre motivation est d'utiliser la programmation logique comme moyen expressif et déclaratif de décrire et identifier certains motifs dans le code source des étudiants. De cette manière, notre objectif est de mettre à disposition des enseignants un langage déclaratif de haut niveau permettant la détection des mauvaises ou bonnes pratiques dans le code source des étudiants. À terme, nous aimerions pouvoir intégrer cette méthode dans les systèmes de correction automatisée afin d'améliorer la qualité de retours fournis aux étudiants.

Méthodologie. Pour pouvoir effectuer des requêtes logiques sur du code source Python, nous avons opté pour l'utilisation du langage Prolog. Cependant, avant de pouvoir exécuter des requêtes logiques, il est impératif de convertir le code source en faits logiques. Cette transformation s'opère via les étapes suivantes. Tout d'abord, un parser intégré à Python permet la création d'un Abstract Syntax Tree (AST). Ensuite, cet AST est converti en format JSON. SWI-Prolog propose une librairie officielle pour convertir un JSON en données manipulables

par un programme logique, ce qui justifie notre choix de cette implémentation du langage Prolog. Une fois l'AST Python transmis au programme SWI-Prolog sous forme de JSON, nous traduisons de manière automatique chaque noeud de l'arbre en faits logique. C'est à partir de ces faits logiques que nous pouvons analyser et écrire des premières requêtes pour trouver certaines pratiques au sein du code source d'étudiants.

Premiers résultats. Après avoir converti un premier programme simple en faits logiques, nous avons constaté la nécessité de garder un lien entre différents faits logiques séparés pour pouvoir traverser notre AST aussi bien dans le sens descendant qu'ascendant. En suivant cette idée nous avons décidé de simplifier nos faits en y ajoutant un identifiant unique. À partir de là et en exploitant les propriétés de la programmation logique nous avons créé des premières fonctions nous facilitant la navigation dans l'arbre ainsi que la collecte d'informations.

Prédicat	Signification
id_type(I, T)	identifiant I lié au type T
child_parent(C, P, L)	C est un enfant de P à un niveau de profondeur L
child_parent(C, C_T, P, P_T, L)	Idem avec le type de l'enfant C_T et du parent P_T
take_info(I, N, X)	X est l'information N du noeud avec identifiant I
count_parentid_childrentype(S, P, C_T)	Parent P a au moins S enfant de type C_T

En utilisant ces premiers prédicats on peut créer des prédicats plus complexes.

```
% trouve les fonctions sans return
function_no_return(Name) :-
    id_type(Id, 'FunctionDef'),
    not(child_parent(_,'Return', Id, _, _)
    take_info(Id, 'name', Name).
```

Ce prédicat nous permet d'identifier dans n'importe quel programme Python, le nom des fonctions n'ayant pas de 'return', ce qui peut dire que l'étudiant n'a pas vraiment compris le concept d'une fonction qui retourne une valeur. Un des avantages d'utiliser un langage logique est qu'il est plus déclaratif et que les prédicats de base son multi-directionnel, permettant d'explorer l'AST dans différentes directions.

Conclusion. Nous avons fait part dans ce document de nos premiers résultats dans la transformation de code source Python en une base de données de faits logiques, ainsi que des premières possibilités d'exécution de requêtes sur ces données. Combinées avec une détection automatique des patterns positifs ou négatifs au sein des codes sources étudiants, nous pouvons envisager à terme de proposer un outil supplémentaire qui se veut déclaratif afin d'identifier automatiquement les erreurs récurrentes des étudiants, améliorant ainsi les feedback proposés aux étudiants lors de l'utilisation d'un correcteur automatique.

Nous tenons également à souligner que, si les exemples ont été réalisés sur des codes sources Python, notre capacité à traduire automatiquement ces codes en faits logiques signifie que notre démarche est facilement adaptable à d'autres langages de programmation présentant des structures d'AST similaires ou légèrement différentes.