Le juge en ligne UpyLaB

Sébastien Hoarau

Laboratoire d'Informatique et de Mathématiques – Université de la Réunion

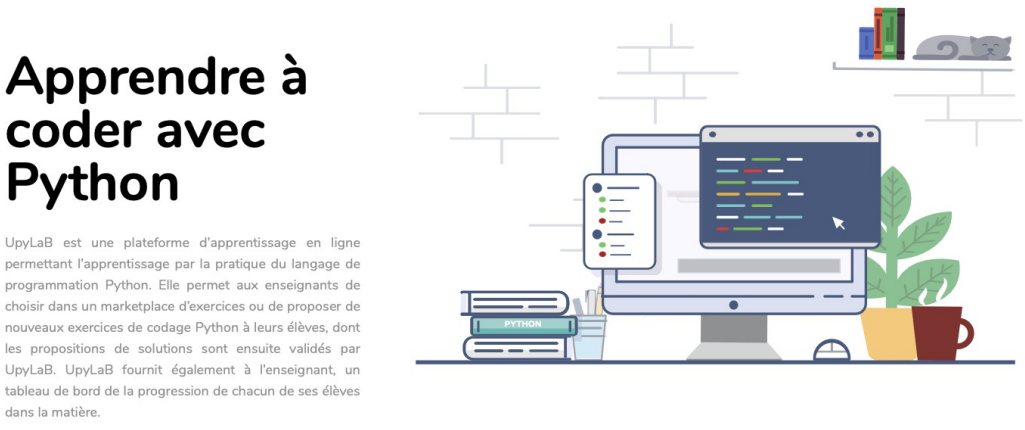
Thierry Massart

Département des Sciences Informatiques - Université Libre de Bruxelles

Numéro thématique 3 / 2023-T3

RÉSUMÉ *L'environnement numérique UpyLaB (UpyLaB (2021)) permet à l'enseignant qui initie à la programmation Python, de créer son cours en proposant une palette d'exercices de codage, soit originaux, soit empruntés d'une bibliothèque d'exercices, que les élèves connectés peuvent réaliser de façon autonome où et quand ils le désirent. Chaque exercice de codage est accompagné d'une liste de tests à réaliser. Lorsque chaque étudiant s'évalue sur un exercice, UpyLaB réalise ces tests automatiquement et de façon non limitée, pour valider le code, ce qui permet des tests plus fins ou plus simples que ceux basés sur l'affichage ou des assertions. Ces tests automatiques sont encodés par l'enseignant via une interface dédiée et réalisés par UpyLaB lors de l'évaluation de l'élève. L'enseignant peut organiser des sessions de contrôles continus et profiter de l'automatisation pour se décharger de la phase chronophage de corrections.*

MOTS-CLÉS *• Juge en ligne • Exerciseur • Python*



Objectifs

L'utilisation d'outils numériques d'aide à l'enseignement de l'informatique fait partie des besoins concrets de l'enseignant, qui est confronté à des élèves de niveau très varié et doit jongler pour d'une part dégager du temps pour réaliser un suivi plus individualisé des élèves en difficulté, et d'autre part proposer du contenu complémentaire aux élèves en avance. Ces environnements sont également les supports qui permettent l'entraide entre enseignants à travers l'échange et le partage d'expériences et de ressources pédagogiques. Parmi celles-ci, l'environnement UpyLaB (UpyLaB (2021)) comprend en particulier un exerciseur, également appelé juge en ligne, adapté à l'enseignement de la programmation en Python et offrant une interface ergonomique pour la création d'exercices.

Présentation de l'activité : UpyLaB

A travers le web, l'environnement et juge en ligne UpyLaB permet à chaque professeur de créer son enseignement de programmation avec le langage Python grâce à des mécanismes de création, d'emprunt et de mise à disposition des ressources créées. Il est accessible à tout enseignant pour lui permettre de créer ses propres exercices de codage Python ou d'emprunter des exercices publics pour créer ses cours (avec comptes enseignants et comptes élèves). L'enseignant pourra ensuite proposer son cours à ses étudiants et faire le suivi de sa classe ; UpyLaB faisant son travail de juge en ligne en validant chaque essai de chaque étudiant à travers des tests unitaires sur des codes Python simples. La partie gestion de classe est facultative dans l'environnement. En effet, le protocole LTI permet d'intégrer les exercices dans une plateforme EdX ou Moodle par exemple. Les élèves connectés pourront valider leurs exercices à souhait depuis la plateforme upylab2.ulb.ac.be ou celle de l'enseignant via l'intégration LTI. L'outil peut réaliser différents tests sur des codes simples ou orientés objet et ne se limite pas à des tests sur les affichages produits, tests qui, souvent, ne permettent pas de tester correctement les codes.

UpyLaB permet de mettre en place de nombreuses pédagogies qui favorisent l'autonomie et fait une part belle à la pratique. Il allie simplicité et richesse pour l'enseignant, qui peut ainsi se concentrer sur les exercices Python à mettre à la disposition des étudiants avec la liste des tests à appliquer aux codes des élèves. La démarche essai-erreur qu'il propose, a été intégrée et validée avec son utilisation, depuis une douzaine d'années, dans le cadre de cours d'initiation au codage Python dans des enseignements de niveaux secondaire et universitaire (à l'Université Libre de Bruxelles et l'Université de la Réunion) ; il est également utilisé dans les MOOCs - Apprendre à coder avec Python (Hoarau S., Massart T., Poirier I. (2019)) qui comptabilise à la mi 2023, plus de 140,000 inscrits depuis son début en 2019, et - Numérique et sciences informatiques : les fondamentaux ouvert depuis 2022.

UpyLaB peut être utilisé directement via le serveur et à l'aide d'un compte dédié (étudiant ou enseignant) ; certaines captures d'images sont données dans la version étendue de cet article (Massart T., Hoarau S. (2023)). Des tutoriels sur UpyLaB sont également disponibles via upylab2.ulb.ac.be ou sur youtube (mot clé upylab2) La version longue de cet article explique de façon détaillée la démarche qui a abouti à la conception des tests réalisés par UpyLaB ; ceux-ci font sa spécificité. Dans cette version courte, nous illustrons avec deux exemples, cette spécificité d'UpyLaB.

Exemple 1 : Initialisation d'une matrice

Supposons que l'enseignant désire proposer un simple exercice sur les listes Python, et demande l'écriture d'une fonction `initialisation_matrice` qui reçoit une valeur entière `n` et renvoie une matrice nulle de dimension `n x n` sous forme d'une liste avec n sous-listes de n valeurs `0`. Par exemple : si `n` vaut `3`, la fonction devra renvoyer la structure : `[[0, 0, 0], [0, 0, 0], [0, 0, 0]]`

Nous n'abordons pas ici la notion de *programme Python bien écrit* au sens des bonnes pratiques et standards (bien structuré, commenté, respectant les standards comme PEP8, ...). L'utilisation de l'application *pylint* sur un code Python permet de vérifier un certain nombre de règles de bonne écriture. Le but ici est plutôt de tester un échantillon de données par "cas d'utilisation" possibles en veillant à tester les cas limites (par exemple une matrice vide ou de dimension 1x1).

Notons d'abord que pour les exercices les plus simples, tester l'affichage des résultats par rapport à ce qui est escompté suffit. Si par contre l'exercice consiste à produire des objets ou renvoyer une ou des valeurs (`return` ou des paramètres après l'exécution d'une fonction, ou valeur de variables à la fin de l'exécution du programme) , il faut vérifier qu'ils sont "conformes" à ce qui est attendu. Dans ce cas, il faut déterminer si chaque valeur produite ou renvoyée est "valide". Plusieurs difficultés existent dans la réalisation d'un tel test.

Il peut être difficile d'avoir la "valeur valide" de référence que doit produire le code à tester : par exemple, si l'on doit tester qu'une fonction renvoie une structure de données complexe (par exemple un arbre binaire), il peut être difficile de construire cette valeur pour pouvoir réaliser le test. Dans ce cas, généralement on se contente de montrer que la valeur à tester a de bonnes propriétés (par exemple que l'arbre binaire construit est bien de la bonne hauteur).

Notons que si l'on a construit la `valeur_correcte` et ayant la `valeur_a_tester`, le test que l'affichage de valeur_a_tester est le même que celui de `valeur_correcte` ne donne pas suffisamment d'information pour tester la validité. De même le test en Python `valeur_correcte == valeur_a_tester` peut lui aussi soit être non pertinent, soit non suffisant pour tester que la `valeur_a_tester`, vue comme une structure de données, est valide par rapport à la valeur de référence `valeur_correcte`.

Par exemple, pour notre fonction qui renvoie une matrice nulle (avec n = 3) : `valeur_correcte = [[0]*3 for _ in range(3)]` et `valeur_a_tester = [0]*3]*3` qui donne une liste pointant trois fois vers la même sous-liste `print(valeur_a_tester)` et `print(valeur_correcte)` renvoient dans les deux cas `[[0, 0, 0], [0, 0, 0], [0, 0, 0]]` et, étant donné la façon dont l'égalité (==) de listes est réalisée, `valeur_correcte == valeur_a_tester` renvoie `True`, puisque l'interpréteur Python teste que dans les deux cas on a une liste avec trois éléments qui sont chacun une sous-liste de trois valeurs entières nulles, sans se préoccuper du fait que, pour valeur_a_tester on parle chaque fois de la même sous-liste, ce qui ne devrait pas être le cas ; et donc `assert valeur_correcte == valeur_a_tester` passe sans soucis malgré la différence entre les valeurs : les deux objets n'ont pas du tout la même structure.

Notons que le mot-clé `is`, teste que les deux variables pointent vers le même objet : il ne peut donc aider ici puisque l'on parle de deux objets distincts.

Le problème de l'opérateur Python d'égalité == est qu'il est défini par le programmeur au moment où il définit la classe : pour une nouvelle classe définie dans le programme, l'égalité est

par défaut, donnée par la valeur de l'opérateur `is`; et sinon peut-être définie par le programmeur à sa guise. Sa définition peut donc être fort éloignée de ce que l'on pourrait avoir envie de définir en tant qu'égalité de structures de données.

Notons également que si l'on teste `valeur_a_tester` comme étant un objet d'une classe par rapport à une `valeur_correcte` qui peut être d'une autre classe, et étant donné qu'il existe aussi des méthodes qui ne peuvent être testées comme de simples valeurs, les choses se compliquent encore.

De façon générale, l'utilisation de l'opérateur relationnel d'égalité (==) dépend de sa définition (définie dans la classe), et est donc non fiable pour tester que deux valeurs sont équivalentes. Un test plus pertinent semble donc être un test sur la *structure* des deux objets qui doivent être **isomorphes** comme précisé dans la version longue de cet article.

Cas de l'enseignant qui doit tester le code d'un apprenant

Le test, par un enseignant ou automatisé, du code d'un étudiant, correspondant à un exercice donné, est un cas particulier plus favorable. En effet, l'enseignant possède un code de référence correct de l'exercice donné à l'apprenant. Dans ce cas, il peut comparer les résultats entre le code de référence et le code de l'étudiant (affichage, valeurs de retour d'une fonction, valeur de variables à la fin de l'exécution ou au retour d'une fonction). Le cas intéressant ici est celui de la comparaison de valeurs : `valeur_correcte` est "égale" à `valeur_a_tester`. On a vu qu'en général l'opération `valeur_correcte == valeur_a_tester` n'est pas suffisante ou correcte, par exemple dans les cas de listes non simples, d'objets de classes définies par l'utilisateur, etc.

Égalité entre deux objets

Supposons que l'on ait exécuté (par exemple dans deux bacs à sable, sécurisés) le code de référence et le code de l'étudiant, et que l'on ait les deux variables valeur_reference et valeur_etudiant. Les classes utilisées pour les objets sont soit des classes prédéfinies, soit des classes définies dans un espace de nom différent. Notons `X_R` et `X_A`, la classe `X` respectivement dans l'espace `R` de référence et `A` de l'apprenant.

Dans ce cas, si l'on suppose, en utilisant la convention Python (avec la convention habituelle Python sur les attributs dits privés) que l'on ne tient compte que des **attributs de données publics** des objets, tester l'égalité des deux valeurs correspond, à un test d'isomorphisme de graphes orientés, colorés et étiquetés (voir par exemple Hsieh et al 2006), qui teste que les deux diagrammes d'état correspondants sont isomorphes au nom de l'espace de nom prêt. Une exception doit être faite si l'attribut de données est une fonction ; l'égalité de deux fonctions étant considérée ici comme insoluble sans faire de la preuve d'équivalence de code.

Cette approche peut être automatisée : ayant 1. le code de l'étudiant, 2. le code de référence, 3. les jeux de test que l'on désire appliquer, 4. le type de test (ensemble des variables ou fonction ou l'affichage à valider), l'outil peut, sans dévoiler le code de référence, proposer à l'apprenant cette validation. Un autre avantage de cette approche est que chaque nouveau test peut être produit avec des données aléatoires (sur les valeurs, la taille des séquences, la séquence des opérations sur les structures en construction, ...) appliquées d'une part au code de l'apprenant et d'autre part au code de référence, ce qui, pour l'apprenant, rend beaucoup plus difficile l'écriture de codes ad hoc qui renvoient les résultats corrects uniquement pour valider les tests proposés.

Exemple 2 : Arbre binaire
Supposons que l'on demande de créer une classe Arbre binaire avec des méthodes `__init__` et `__repr__`, et de tester en fin de code les valeurs des variables `x`, `y`, `res` et `res2` en fournissant déjà la fonction const_arbre et le code principal qui suit. Une solution pourrait être :

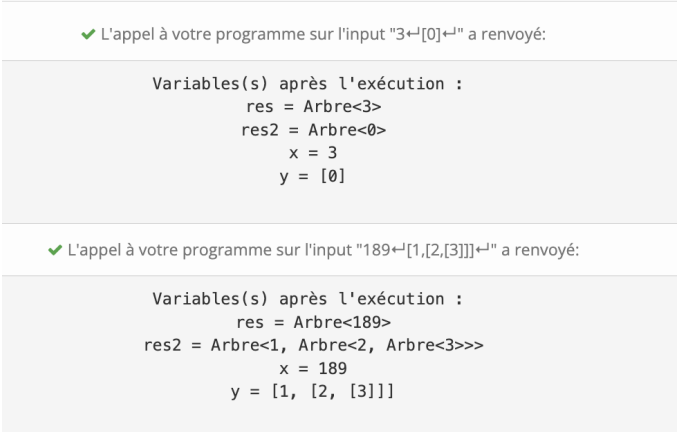
```
class Arbre(object):
    """definition de la classe arbre"""
    def __init__(self, val=0, fg=None, fd=None):
        self.val = val
        self.fg = fg
        self.fd = fd

    def __repr__(self):
        return 'Arbre<'+ repr(self.val) +\
             (', ' + repr(self.fg) if type(self.fg) is Arbre else "") +\
             (', ' + repr(self.fd) if type(self.fd) is Arbre else "") + '>'

def const_arbre(a):
    return Arbre(*[const_arbre(e) if type(e) is list else e for e in a])

# programme principal
x = int(input())
res = const_arbre([x])
y = eval(input())
res2 = const_arbre(y)
```

Le test UpyLaB pourrait donner :



Promouvoir des ressources d'enseignement de qualité et libres

L'environnement et outil UpyLaB co-réalisé dans le cadre du projet cai.community est une pierre à l'édifice de la promotion des valeurs de co-création de ressources pédagogiques et de partage pour les autres enseignants, dans une démarche de diffusion libre et gratuite de la connaissance.

Nous avons proposé un environnement incluant des autotests de code Python pour des débutants avec version élève et version enseignant et montré son efficacité dans la résolution de quelques difficultés. En particulier, nous avons montré que ni le test d'égalité d'affichage ni l'utilisation de l'opérateur d'égalité en Python, utilisé fréquemment dans des outils de test, ne sont satisfaisants pour valider les structures de données (dont ceux avec des listes, dictionnaires, classes, ...) construites dans les codes.

Contact : Sébastien Hoarau (seb.hoarau@univ-reunion.fr) et Thierry Massart (thierry.massart@ulb.be)

Références

Hsieh Shu-Ming, Hsu Chiun-Chieh, Hsu Li-Fu, Efficient Method to Perform Isomorphism Testing of Labeled Graphs in: Computational Science and Its Applications - ICCSA 2006, LNCS 3984, pp 422–431, 2006

Hoarau S., Massart T., Poirier I. (2019), Apprendre à coder avec Python, https://www.fun-mooc.fr/fr/cours/apprendre-a-coder-avec-python/

Massart T., Hoarau S. (2023), Le juge en ligne UpyLaB, rapport hal-04023112 https://hal.science/hal-04023112

UpyLaB (2021). Site de l'outil UpyLaB (2021), https/upylab2.ulb.ac.be

Remerciements

Les développements d'UpyLaB ont été co-financés grâce au projet ERASMUS+ Communauté d'Apprentissage de l'Informatique (cai.community) 2019-1-BE01-KA201-050429