# Le juge en ligne UpyLaB

Thierry Massart, Sébastien Hoarau

# Le juge en ligne UpyLaB

Thierry Massart[1], Sébastien Hoarau[2]

[1] Département des Sciences Informatiques - Université Libre de Bruxelles
[2] Laboratoire d'Informatique et de Mathématiques – Université de la Réunion

## Résumé

L'environnement numérique UpyLaB (https://upylab2.ulb.ac.be) 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. L'utilisation d'UpyLaB par rapport à d'autres outils est justifiée par la façon dont les tests automatiques sont encodés par l'enseignant et réalisés par UpyLaB lors de l'évaluation de l'élève.
L'enseignant peut également très facilement organiser des sessions de contrôles continus sans phase chronophage de correction.


## Introduction

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 s'occuper, de façon un peu plus individualisée, des élèves en difficulté, et d'autre part proposer du contenu complémentaire aux élèves en avance. Ces environnements sont également les ingrédients 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 (https://upylab2.ulb.ac.be) comprend en particulier un exerciseur, appelé aussi juge en ligne, simple, ergonomique et efficace pour l'enseignant en programmation Python. A travers le web, UpyLaB permet :

- A l'enseignant :
  - de proposer des palettes d'exercices de codage,
  - de créer son cours structuré en chapitres ; chacun incluant une liste d'exercices soit originaux soit empruntés d'une bibliothèque d'exercices,
  - d'éventuellement intégrer les exercices de son ou ses cours dans un environnement Moodle ou EdX,
  - de mettre ce cours à disposition de ses étudiants ou éventuellement d'en faire un cours public,
  - de faire le suivi des progrès de ses étudiants pour chacun de ses cours.
  - d'organiser des sessions de contrôles continus sans phase chronophage de correction.
- A l'élève
  - de s'inscrire à un cours ouvert ou grâce à une invitation,
  - de se connecter au cours pour réaliser les exercices de façon autonome où et quand il le désire ;
  - d'avoir son exercice validé automatiquement par UpyLaB qui teste automatiquement et de façon non limitée, son code, grâce à la liste des tests spécifiés par le concepteur de l'exercice.

Des tutoriels sur UpyLaB sont disponibles via upylab2.ulb.ac.be ou sur youtube (mot clé upylab2)

## Contexte et limitations

L'objectif de cette section n'est pas de faire un cours in extenso sur le test de programme mais d'en retenir les éléments principaux dont nous avons besoin pour la validation fonctionnelle d'un petit code (petit programme, fonction, ...) En particulier, nous dégageons les aspects nécessaires à la validation automatique de codes d'élèves dans un enseignement d'initiation au codage. Pour instancier notre propos, nous supposons ici traiter des programmes en langage Python.
Nous ne parlons pas ici des bonnes pratiques et standards (respect du PEP 8 par exemple) pour écrire un programme Python "propre" (bien structuré, commenté, qui respecte les standards, ...). L'utilisation de l'application `pylint` sur un code Python permet de vérifier un certain nombre de règles de bonne écriture. Notons que `pylint` est probablement trop tatillon (par exemple dans l'utilisation des noms) ce qui le rend parfois assez antipathique !
Nous regardons donc ici comment tester un code (Python) simple. D'abord de façon assez générale, pour ensuite voir en pratique comment tester nos codes ou les codes des apprenants.

## C'est quoi un test ?

Une première définition de test (logiciel) (Glenford et al 2011) peut être donnée par "le processus d'exécution d'un programme dans le but d'en trouver des erreurs"[1]. Tester un code permet ainsi, dès lors que les problèmes identifiés sont corrigés d'en augmenter la qualité.
Le test englobe les activités permettant de donner des informations qualitatives sur un système (performance, robustesse, extensibilité, sécurité, ...). Les quatre paramètres de base d'un test sont les données, le système à tester, les résultats attendus et l'état du système après l'exécution. On distingue les tests fonctionnels sur ce que produit un système, et non fonctionnels, qui sont qualitatifs. Selon la nature de l'objet (du bout de code au système complet), il existe différents niveaux de tests (test unitaire, d'intégration, système, d'acceptation). Puisqu'ici il est question d'initiation à la programmation et de test de code très simple, nous nous intéressons uniquement à des tests unitaires. Les tests sont accompagnés de la description de leurs procédures (voir standard IEEE 829-2008 (IEEE 829-2008)
Notons que les aspects reproduction d'un test peuvent être compliqués dans le cas où le système a un comportement aléatoire, ou qui dépend du moment où il est exécuté ou encore s'il est composé de divers processus qui s'exécutent en parallèle ou en concurrence.

[1] Testing is the process of executing a program with the intent of finding errors

# Test d'un programme ou d'une partie de programme "simple"

Nous nous limitons ici aux aspects fonctionnels qui sont souvent les premières exigences données pour son bon fonctionnement et aux codes séquentiels reproductibles, qui donnent le même résultat si on l'exécute plusieurs fois sur la même plateforme. Nous supposons également que nous travaillons avec une architecture de von Neumann.
Dans ce cas, le test d'un programme simple (sans utilisation du réseau, ....) ou d'une fonction doit spécifier2 :

- l'état du programme à tester avant exécution du test (par exemple si des variables sont initialisées avant le début du programme, ...) ;
- les valeurs données en entrée ou en paramètres de la fonction à tester ;
- les valeurs ou affichages attendus après l'exécution du code ou de la fonction ;
- l'état, après l'exécution[2], Du programme ou des arguments de la fonction appelée.

On parle donc d'une part d'entrées / sorties correspondant à du texte donné au code, en entrée ou via un fichier, ou affiché sur l'écran, et d'autre part d'états qui correspondent à des "valeurs" de variables[3] ou de fichiers, c'est-à-dire d'objets si l'on regarde du code Python.

# Tests par boîte blanche ou noir

Selon que l'on dispose ou non des sources du code, on parle de test par boîte blanche (white box) - où l'on voit le code et analyse sa structure - ou par boîte noire (black box) - où on ne peut voir le code mais observer ses effets.
Si l'on doit tester son propre code, libre à nous de faire le type de test que l'on désire. Le test par boîte blanche va être plus utile pour montrer que certaines parties de code ne font pas n'importe quoi : on va par exemple essayer d'avoir un ensemble de tests qui assurent une couverture, si possible, complète du code en terme de point ou de chemin du programme : chaque instruction est exécutée au moins une fois par exemple.
L'approche boîte noire, même avec un code accessible, peut s'avérer plus intéressante puisque l'idée est de tester, si possible, l'ensemble des fonctionnalités. Par exemple, si on veut tester qu'une fonction fun fait correctement son travail, on essaye de tester un échantillon de données par "cas d'utilisation" possibles en veillant à tester les cas limites (par exemple liste vide, si un des paramètres est une liste , ...).

## Comment tester si le résultat est correct

La question semble simple. Si le code affiche des résultats, ou s'il renvoie 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 escompté.
La notion de conformité doit être précisée.

### Pour l'affichage de résultats

On peut tester les textes (chaînes de caractères) produites (parfois avoir une tolérance sur le format). Ce type de tests est souvent suffisant pour des valeurs simples.
Il faut quand même faire attention aux faux semblants (affichage semblables d'objets différents). Ceci est d'autant plus vrai que l'objet affiché est structuré.
L'affichage de la valeur d'objets structurés plus complexes (par exemple un x binaire) ne permet pourtant en général pas de déterminer si la valeur est correcte ; l'affichage ne donnant qu'une image souvent imprécise de la valeur et surtout de sa structure.

### Pour l'égalité de valeurs en Python

Un test plus précis consiste donc à déterminer si une valeur est égale à la "valeur correcte". Plusieurs soucis existent dans la réalisation d'un tel test :
Il peut être difficile d'avoir la "valeur correcte" 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 qu'un arbre binaire construit a la bonne hauteur).
Si l'on suppose que l'on a la `valeur_correcte` et la `valeur_a_tester`, le test que l'affichage est le même et même le test en Python `valeur_correcte == valeur_a_tester` n'est soit pas pertinent soit pas suffisant pour tester que les deux objets testés, vus comme des structures de données sont "égales".
Par exemple, si l'on veut montrer que la fonction renvoie une matrice nulle de dimension 3x3 sous forme d'une liste avec trois sous-listes de 3 valeurs 0, ayant :
`valeur_correcte=[[0]*3 for _ in range(3)]`
et `valeur_a_tester=[0]*3]*3 #` qui donne une liste pointant 3 fois vers la même sous-liste
`print(valeur_a_tester)`
renvoie
`[[0, 0, 0], [0, 0, 0], [0, 0, 0]]`
et, étant donné la façon dont l'égalité (==) de listes est réalisé (En effet, ici chaque structure est une liste de 3 sous-listes contenant 3 zéros; le souci est que `valeur_a_tester` pointe trois fois vers la même sous-liste),
`valeur_correcte == valeur_a_tester`
renvoie
`True` puisque Python teste que dans les deux cas on a une liste avec trois éléments qui sont chacun une sous-liste de 3 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. Pourtant les deux objets n'ont pas du tout la même structure.
Notons que le mot-clé `is`, teste que les 2 variables pointent vers le même objet : il ne peut donc aider ici puisque l'on parle de deux objets distincts.

[2] On suppose que ce programme peut être la première partie d'un programme plus long ; son état à la fin peut donc être important.

[3] y compris la valeur d'arguments après exécution, si l'on teste une fonction

Le problème de l'opérateur Python d'égalité == est qu'il est définit par son concepteur au moment où il définit la classe : pour une nouvelle classe définies 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 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 expliqué plus loin.

## 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 un 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, 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[4] 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[5] 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. Pour les attributs de données simples, le test d'égalité (==) est appliqué (à une précision epsilon prêt pour les float). 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 ; dans ce cas, aucun test n'est appliqué.

### Tests automatisés

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 grand avantage de cette approche est que chaque nouveau test peut être produit avec des données **aléatoires** (sur les valeurs, la taille des séquences, la séquences des opérations sur les structures en construction, ...) 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.

# UpyLaB

L'environnement et juge en ligne UpyLaB (upylab2.ulb.ac.be) 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. UpyLaB permet de faire des tests unitaires sur des codes Python simples. 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, qui ne permettent souvent pas de bien tester les codes, comme présenté plus haut.
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 (pour des étudiants en Sciences informatiques et mathématiques et Sciences de l'ingénieur de l'Ecole Polytechnique de l'Université Libre de Bruxelles et pour des étudiants de l'Université de la Réunion); il est également utilisé dans les MOOCs Apprendre à coder avec Python[6] qui comptabilise fin 2022, plus de 130,000 inscrits depuis son début en 2019 et Numerique et sciences informatiques : les fondamentaux.
L'interface d'UpyLaB peut être visualisé en utilisant l'outil, soit comme étudiant, soit si c'est le cas, comme

[4] Par exemple dans deux bacs à sable, sécurisés pour ne pas avoir de soucis dans le cas de code d'apprenant ou même de référence, malveillant.

[5] Avec la convention habituelle Python sur les attributs dits privés

[6] https://www.fun-mooc.fr/fr/cours/apprendre-a-coder-avec-python/

enseignant; certaines captures d'images sont données en annexe.

## Promouvoir les ressources d'enseignement de qualité et libres

L'environnement et outil UpyLaB est une pierre à l'édifice de la promotion des valeurs de création ou 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 montré l'approche adoptée pour concevoir l'outil UpyLaB et surtout comment il réalise les tests validant les codes des apprenants ; en justifiant en particulier pourquoi l'utilisation de l'opérateur d'égalité en Python, utilisé fréquemment dans des outils de test, n'est pas satisfaisante pour valider des codes.

## Références


- Glenford J. Myers, Corey Sandler, Tom Badgett, The Art of Software Testing, 3rd Edition, , Wiley, 2011
- IEEE-829-2008 IEEE Standard for Software and System Test Documentation, IEEE Computer Society, 2008. https://standards.ieee.org/ieee/829/3787/
- 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


## Remerciements


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

# Annexe: Interfaces d'UpyLaB

## Interface Professeur

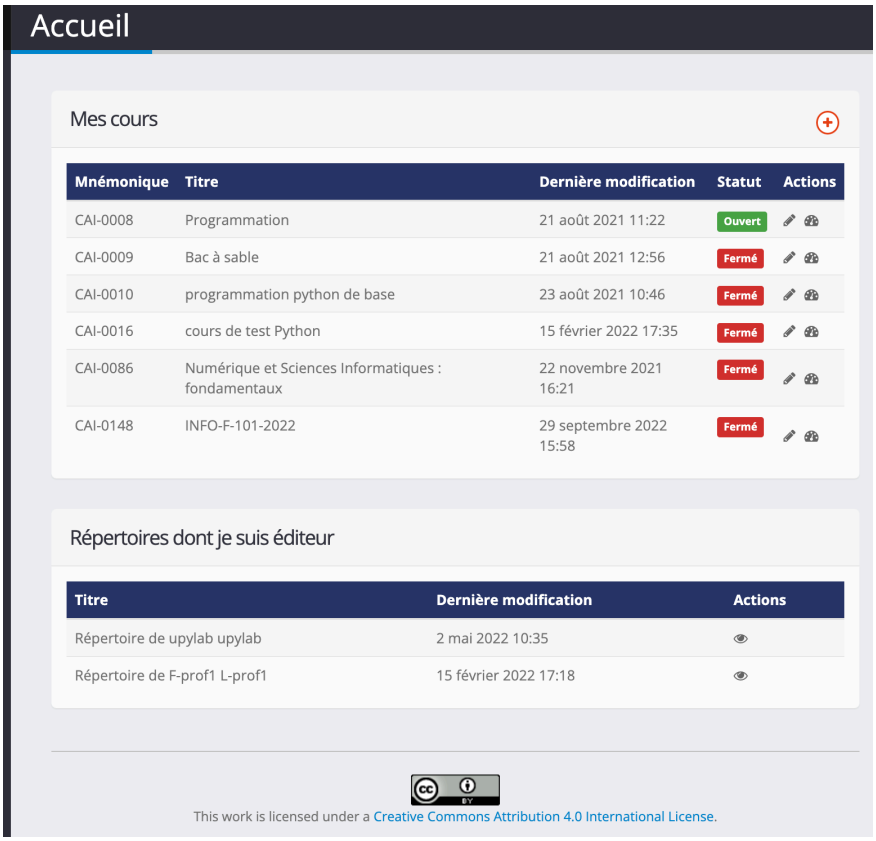



Figure 1: Accueil


Figure 2: Gestion des exercices d'un cours

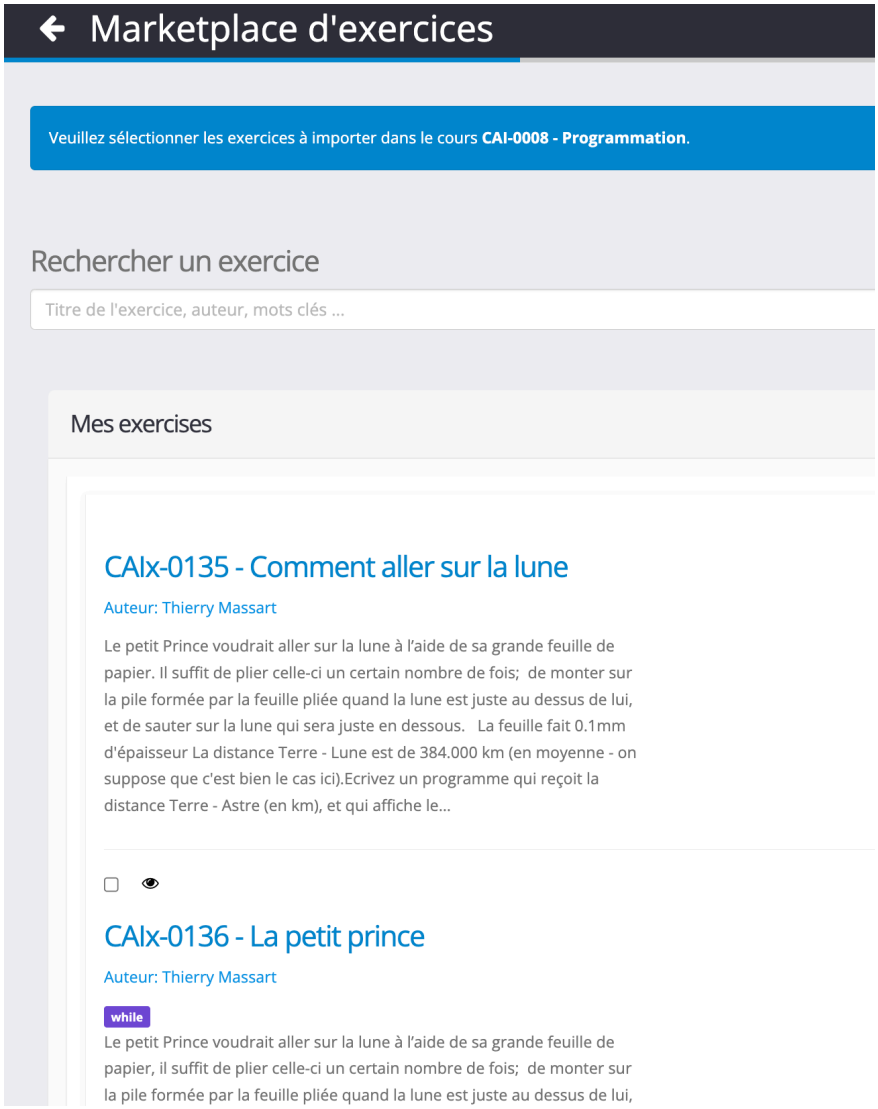
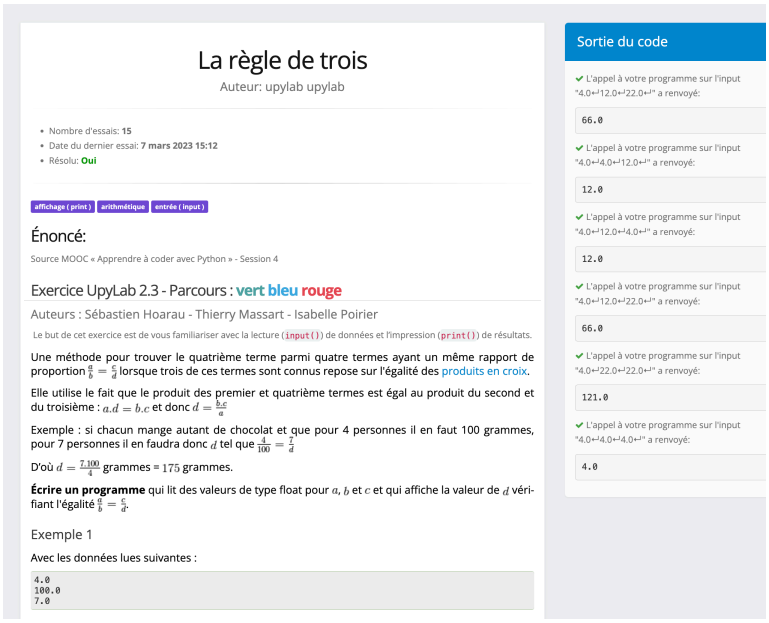



Figure 3: Emprunt d'exercices

# Interface Elève



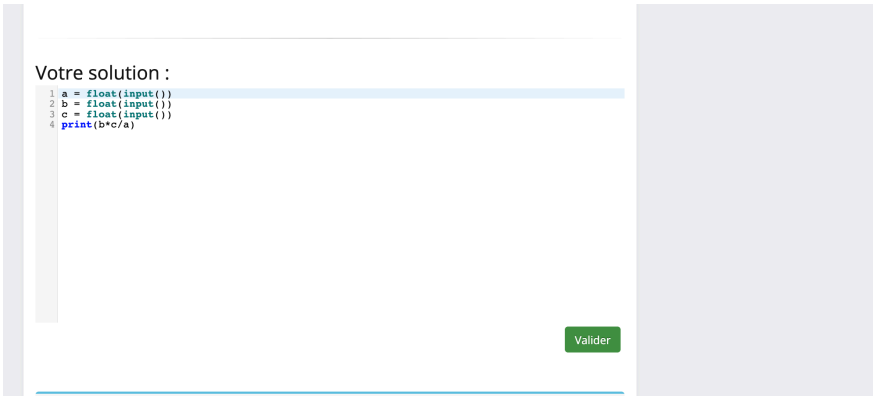




Figure 4: Exercice


Figure 5: Liste des exercices d'un cours

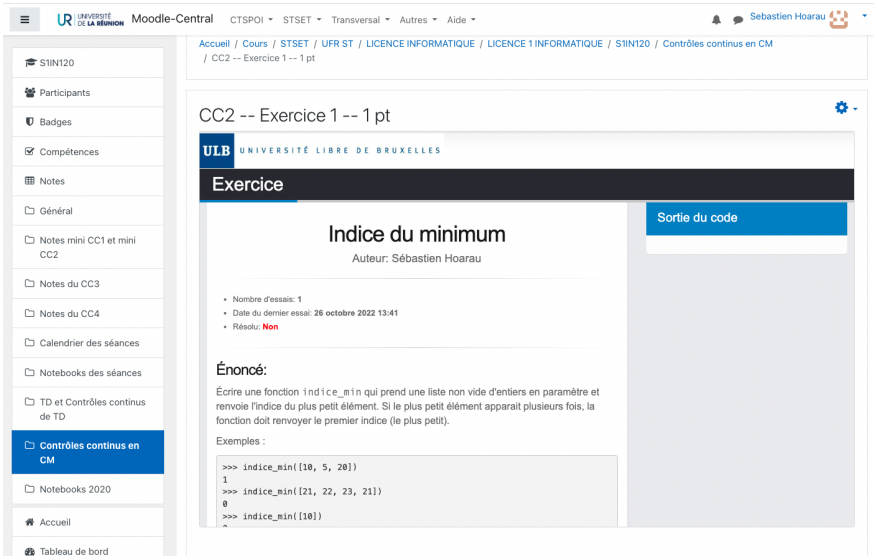



Figure 6: Intégration d'un exercice dans un environnement Moodle grâce au protocole LTI

# Addresses

## Surface mail

Université Libre de Bruxelles CP 212
Département d'Informatique
Boulevard du Triomphe
B-1050 Brussels
Belgium

## Web

http://www.ulb.ac.be
http://www.ulb.ac.be/di