



Le juge en ligne UpyLaB

Thierry Massart, Sébastien Hoarau

► To cite this version:

Thierry Massart, Sébastien Hoarau. Le juge en ligne UpyLaB. Université Libre de Bruxelles. 2023. hal-04023112v2

HAL Id: hal-04023112

<https://hal.science/hal-04023112v2>

Submitted on 6 Jul 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

Le juge en ligne UpyLaB

Thierry Massart¹, Sébastien Hoarau²

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 (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.

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 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 exerciceur, également appelé juge en ligne, adapté à l'enseignement de la programmation en Python et offrant une interface ergonomique pour la création d'exercices.

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 upy-lab2.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 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. 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 (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.

¹Testing 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'exécution³, 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 variables³ 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),

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

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

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 la validité de valeurs en Python

Un test plus précis consiste donc à déterminer si une valeur est valide par rapport 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.

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 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 exécuté⁴ 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 Python⁵ 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.

Code de test de la validité entre deux objets

Précisons dans cette section, comment UpyLaB teste la validité entre deux objets `r1` et `r2`

Le code python de la fonction `test_validity` est donné en annexe.

Il satisfait par exemple, aux assertions suivantes :

```
# test la fonction test_validity
from copy import deepcopy
x = [[0] * 3] * 3
y = [[0] * 3 for _ in range(3)]
z = [[0] * 3, [0] * 3, [0] * 3]
assert not test_validity(x, y)
assert not test_validity(x, z)
assert test_validity(y, z)
g1 = [1, 2, 3]
g1[1] = g1
g1 = [g1[:], g1, g1[1:]]
g2 = deepcopy(g1)
assert test_validity(g1, g2)
g2[1] = deepcopy(g1)
assert not test_validity(g1, g2)

# ne teste pas le contenu des fonctions
assert test_validity(lambda x: x + 1, lambda x: x + 2)
```

Objet Python vu comme un graphe

Un objet python peut être vu comme un graphe avec un sommet principal et un sommet pour chaque objet ou sous objet qui le compose. Chaque sommet du graphe a comme valeur, les attributs de données de l'objet qu'il représente ainsi que le nom de sa classe, et comme voisins, les sous-objets qui le compose.

La structure générale d'un objet, est bien illustrée par exemple avec les diagrammes d'état donnés par l'outil bien connu pythontutor (Pythontutor (2023)).

Par exemple l'objet `[1, 2, [3, 4]]` sera vu, comme illustré en figure 1, comme un objet (graphe avec sommet initial) de classe liste de taille 3 avec 3 sous objets (sommets voisins) dans l'ordre : deux int de valeur resp. 1 et 2 et une sous liste de taille 2 avec 2 sous objets (voisins) int valant 3 et 4.

Notons que le code suivant donne un objet `r` cyclique, illustré en figure 2.

⁴Par 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.

⁵Avec la convention habituelle Python sur les attributs dits privés

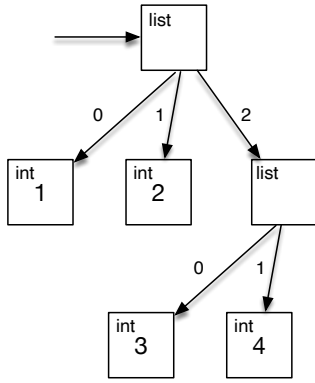


Figure 1: Structure de donnée liste avec sous liste simples

```

r = [1, 2]
r[0] = r[:] # [1, 2] dans r[0]
r[1] = r # référence cyclique dans r[1]
r[0][0] = r # réf cyclique indirecte dans r[0][0]

```

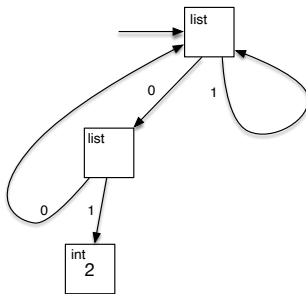


Figure 2: Structure de donnée de listes avec références cycliques

De façon générale, on peut imaginer n'importe quelle structure de graphe en utilisant ce genre de code. Ici, comme on utilise des listes, les voisins sont numérotés et ordonnés.

Algorithme de validité (égalité) des objets r_1 et r_2

Le principe de l'algorithme, est celui qui teste l'isomorphisme entre deux graphes (ici les deux objets r_1 et r_2). Il procède par un parcours en profondeur, effectué en parallèle sur les deux graphes, qui teste si pour chaque sommet du graphe depuis le sommet "initial" correspond bien (dans les deux sens) un sommet dans r_2 . Cela revient à construire une bijection entre les sommets (objets et sous objets) de r_1 et ceux de r_2 en testant que les attributs (classes et attributs de données) correspondent.

L'algorithme gère deux dictionnaires dic_id_1 et dic_id_2 qui vont permettre de vérifier que les structures sont isomorphes en enregistrant les correspondances entre les sommets du graphe r_1 et ceux de r_2 . A chaque dictionnaire dic_id_1 et dic_id_2 , initialement vides, l'algorithme ajoute lors du parcours parallèle des éléments rencontrés, en mettant comme clés des identificateurs d'objets ou sous objets rencontrés lors de l'analyse de r_1 (resp. r_2) et comme valeur, l'identificateur ($id()$) correspondant dans r_2 (resp. r_1). L'algorithme construit ainsi lors du parcours, la bijection entre les noeuds du graphe r_1 et r_2 .

Lors du parcours, l'algorithme, qui utilise la fonction récursive `test_result`, reçoit les deux objets r_1 et r_2 et les deux dictionnaires dic_id_1 et dic_id_2 .

Cas possibles

D'abord si les noms de classe des deux objets sont différents, ou si un des deux objets a déjà été rencontré mais ne correspondant pas à l'autre ($dic_id_1[id(r_1)] \neq id(r_2)$) l'algorithme renvoie `False` (r_1 différents de r_2). Si par contre les objets ont déjà été analysés et correspondent ($dic_id_1[id(r_1)] == id(r_2)$) : couple (r_1 , r_2) déjà analysés ou en cours d'analyse en cas de cycle), il renvoie `True` à l'instance appelante ; Si ce n'est pas le cas, outre la gestion des dictionnaires dic_id_1 et dic_id_2 déjà expliqué, les différents cas suivants peuvent se produire :

- Les objets sont des float : UpyLaB teste l'égalité modulo epsilon (test absolu avec ici $\epsilon = 10^{-5}$, notons qu'un test relatif pourrait être réalisé pour tenir compte de la taille des valeurs) ;
- Les objets sont des itérables (par exemple des listes) sans être des dict, set ou str : l'algorithme teste que les deux objets ont le même nombre de sous objets ($len(r_1) == len(r_2)$) et que les sous objets correspondants sont équivalents (appels récursifs) ;
- si les objets sont des dictionnaires, l'algorithme teste que l'ensemble des clés sont égaux et que chacun des couples de valeurs correspondant sont valides (appels récursifs) ;
- si les objets sont des classes définies, le travail est similaire à celui effectué pour le cas des dictionnaires ; mais avec l'ensemble des attributs publics. On considère ainsi que les attributs privés ne doivent pas intervenir dans le test de validité.
- enfin dans les autres cas, le test d'égalité (`==`) est réalisé (`int`, `str`, `set`, `bool`, ...).

UpyLaB

L'environnement et juge en ligne UpyLaB (UpyLaB (2023)) 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. 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, 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'École 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 (Hoarau et al (2019)) qui comptabilise fin 2022, plus de 130,000 inscrits depuis son début en 2019 et Numerique et sciences informatiques : les fondamentaux.

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 en annexe.

- UpyLaB (2023). Site de l'outil UpyLaB (consulté en juillet 2023), <https://upylab2.ulb.ac.be>

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

Promouvoir les 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 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.

Références

- Glenford J. Myers, Corey Sandler, Tom Badgett (2011), The Art of Software Testing, 3rd Edition, Wiley
- (Hoarau S., Massart T. Poitier I. (2019)), Apprendre à coder avec Python, <https://www.fun-mooc.fr/fr/cours/apprendre-a-coder-avec-python/>
- Hsieh Shu-Ming, Hsu Chiun-Chieh, Hsu Li-Fu (2006), Efficient Method to Perform Isomorphism Testing of Labeled Graphs in: Computational Science and Its Applications - ICCSA 2006, LNCS 3984, pp 422–431
- IEEE-829-2008 (2008), IEEE Standard for Software and System Test Documentation, IEEE Computer Society. <https://standards.ieee.org/ieee/829/3787/>
- Pythontutor (2023), Site de Python Tutor (consulté en juillet 2023) : <https://pythontutor.com/>

Annexe : validité dans UpyLaB

```
from collections.abc import Iterable
EPSILON = 1.0e-5

def test_validity(r1, r2):
    """tests that the objects r1 and r2 have same values and structures"""
    dic_id_1 = {}
    dic_id_2 = {}
    return test_result(r1, r2, dic_id_1, dic_id_2)

def get_public_attributes_keys(obj):
    """get the public attributes of the object"""
    class_name = obj.__class__.__name__
    class_attrs = obj.__dict__.keys()
    public_attributes_keys = [k for k in class_attrs if not (k.startswith("_{}__".format(class_name)))]
    return public_attributes_keys













def test_result(r1, r2, dic_id_1, dic_id_2):
    """
    test that r1 & r2 have the same values and structure
    :param r1, r2: objects to test
    :param dic_id_1, dic_id_2: mapping of the objects/subobjects
    :return:
    """
    if r1.__class__.__name__ != r2.__class__.__name__:
        res = False
    elif isinstance(r1, float) and isinstance(r2, float): # floats
        res = abs(r1 - r2) < EPSILON
    elif isinstance(r1, Iterable) and not isinstance(r1, str) \
        and not isinstance(r1, dict) and not isinstance(r1, set): # iterables not str, dic, set
        res = len(r1) == len(r2)
        deja_vu = False
        if id(r1) in dic_id_1:
            res = res and dic_id_1[id(r1)] == id(r2)
            deja_vu = True
        if id(r2) in dic_id_2:
            res = res and dic_id_2[id(r2)] == id(r1)
            deja_vu = True
        if not deja_vu:
            dic_id_1[id(r1)] = id(r2)
            dic_id_2[id(r2)] = id(r1)
            i = 0
            while res and i < len(r1):
                res = test_result(r1[i], r2[i], dic_id_1, dic_id_2)
                i += 1
    elif isinstance(r1, dict) and set(r1.keys()) == set(r2.keys()): # dict
        res = True
        deja_vu = False
        if id(r1) in dic_id_1:
            deja_vu = True
            res = res and dic_id_1[id(r1)] == id(r2)
        if id(r2) in dic_id_2:
            deja_vu = True
            res = res and dic_id_2[id(r2)] == id(r1)
        if not deja_vu:
            dic_id_1[id(r1)] = id(r2)
            dic_id_2[id(r2)] = id(r1)
            for i in r1.keys():
                res = res and test_result(r1[i], r2[i], dic_id_1, dic_id_2)
    elif '__dict__' in dir(r1) and '__dict__' in dir(r2): # classes
        res = True
        deja_vu = False
        if id(r1) in dic_id_1:
            deja_vu = True
            res = res and dic_id_1[id(r1)] == id(r2)
        if id(r2) in dic_id_2:
            deja_vu = True
            res = res and dic_id_2[id(r2)] == id(r1)
        if not deja_vu:
            keys1 = set(get_public_attributes_keys(r1))
            keys2 = set(get_public_attributes_keys(r2))
            res = keys1 == keys2
            if res:
                dic_id_1[id(r1)] = id(r2)
                dic_id_2[id(r2)] = id(r1)
                for i in keys1:
                    res = res and test_result(r1.__dict__[i], r2.__dict__[i], dic_id_1, dic_id_2)
    elif r1 == r2: # equals
        res = True
    else:
        res = False
    return res
```

Annexe: Interfaces d'UpyLaB



Interface Professeur


Accueil

Mes cours

Mnémonique	Titre	Dernière modification	Statut	Actions
CAI-0008	Programmation	21 août 2021 11:22	Ouvert	 
CAI-0009	Bac à sable	21 août 2021 12:56	Fermé	 
CAI-0010	programmation python de base	23 août 2021 10:46	Fermé	 
CAI-0016	cours de test Python	15 février 2022 17:35	Fermé	 
CAI-0086	Numérique et Sciences Informatiques : fondamentaux	22 novembre 2021 16:21	Fermé	 
CAI-0148	INFO-F-101-2022	29 septembre 2022 15:58	Fermé	 

Répertoires dont je suis éditeur

Titre	Dernière modification	Actions
Répertoire de upylab upylab	2 mai 2022 10:35	
Répertoire de F-prof1 L-prof1	15 février 2022 17:18	



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

Figure 3: Accueil

Gestionnaire d'exercices

Général

1. Bases

CAIx-0136 - La petit prince
Auteur: Thierry Massart

CAIx-0132 - affectations de variables
Auteur: Thierry Massart

CAIx-0004 - La règle de trois
Auteur: upylab upylab

Nouveau chapitre

Ajouter

+ Importer un exercice



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

Figure 4: Gestion des exercices d'un cours

← Marketplace d'exercices

Veuillez sélectionner les exercices à importer dans le cours **CAI-0008 - Programmation**.

Rechercher un exercice

Titre de l'exercice, auteur, mots clés ...

Mes exercices

CAIx-0135 - Comment aller sur la lune

Auteur: [Thierry Massart](#)

Le petit Prince voudrait aller sur la lune à l'aide de sa grande feuille de papier. Il suffit de plier celle-ci un certain nombre de fois; de monter sur la pile formée par la feuille pliée quand la lune est juste au dessus de lui, et de sauter sur la lune qui sera juste en dessous. La feuille fait 0.1mm d'épaisseur La distance Terre - Lune est de 384.000 km (en moyenne - on suppose que c'est bien le cas ici).Ecrivez un programme qui reçoit la distance Terre - Astre (en km), et qui affiche le...

☐

CAIx-0136 - La petit prince

Auteur: [Thierry Massart](#)

while

Le petit Prince voudrait aller sur la lune à l'aide de sa grande feuille de papier, il suffit de plier celle-ci un certain nombre de fois; de monter sur la pile formée par la feuille pliée quand la lune est juste au dessus de lui,

Figure 5: Emprunt d'exercices

8

Interface Elève

La règle de trois

Auteur: upylab upylab

- Nombre d'essais: 15
- Date du dernier essai: 7 mars 2023 15:12
- Résolu: **Oui**

[affichage \(print \)](#) [arithmétique](#) [entrée \(input \)](#)

Énoncé:

Source MOOC « Apprendre à coder avec Python » - Session 4

Exercice UpyLab 2.3 - Parcours : **vert bleu rouge**

Auteurs : Sébastien Hoarau - Thierry Massart - Isabelle Poirier

Le but de cet exercice est de vous familiariser avec la lecture (`input()`) de données et l'impression (`print()`) de résultats.

Une méthode pour trouver le quatrième terme parmi quatre termes ayant un même rapport de proportion $\frac{a}{b} = \frac{c}{d}$ lorsque trois de ces termes sont connus repose sur l'égalité des **produits en croix**.

Elle utilise le fait que le produit des premier et quatrième termes est égal au produit du second et du troisième : $a.d = b.c$ et donc $d = \frac{b.c}{a}$

Exemple : si chacun mange autant de chocolat et que pour 4 personnes il en faut 100 grammes, pour 7 personnes il en faudra donc d tel que $\frac{1}{100} = \frac{7}{d}$

D'où $d = \frac{7.100}{4}$ grammes = 175 grammes.

Écrire un programme qui lit des valeurs de type float pour a , b et c et qui affiche la valeur de d vérifiant l'égalité $\frac{a}{b} = \frac{c}{d}$.

Exemple 1

Avec les données lues suivantes :

4.0

100.0

7.0

Sortie du code

✓ L'appel à votre programme sur l'input
"4.0+12.0+22.0+?" a renvoyé:

66.0

✓ L'appel à votre programme sur l'input
"4.0+4.0+12.0+?" a renvoyé:

12.0

✓ L'appel à votre programme sur l'input
"4.0+12.0+4.0+?" a renvoyé:

12.0

✓ L'appel à votre programme sur l'input
"4.0+12.0+22.0+?" a renvoyé:

66.0

✓ L'appel à votre programme sur l'input
"4.0+22.0+22.0+?" a renvoyé:

121.0

✓ L'appel à votre programme sur l'input
"4.0+4.0+4.0+?" a renvoyé:

4.0

Votre solution :

```
1 a = float(input())
2 b = float(input())
3 c = float(input())
4 print(b*c/a)
```

Valider

Figure 6: Exercice

← Cours : Exercices du MOOC : Apprendre à coder avec Python

DESCRIPTION DU COURS

1. Introduction à UpyLaB
2. Arithmétique et manipulation des textes
3. Instructions
4. Fonctions
5. Structures de données
6. Ensembles et dictionnaires
7. Exercices de synthèse

EXERCICES

1. Introduction à UpyLaB
 - ✓ Imprime "Bonjour UpyLaB !"
2. Arithmétique et manipulation des textes
 - ✓ Assignations simples
 - ✗ Moyenne arithmétique
 - ✓ La règle de trois
 - ✓ Sommets d'un hexagone
 - ✗ Impression d'expressions diverses
 - Impressions de textes
 - Volume d'une sphère

Se désinscrire du cours

Vos statistiques

Progression

4%

À propos du cours

Cours ouvert

87 exercices

Dernière mise-à-jour le 10 septembre 2021 10:50

Figure 7: Liste des exercices d'un cours

ULB UNIVERSITÉ DE LA BIENNE Moodle-Central CTSP01 STSET Transversal Autres Aide

Sebastien Hoarau

Accueil / Cours / STSET / UFR ST / LICENCE INFORMATIQUE / LICENCE 1 INFORMATIQUE / S1IN120 / Contrôles continus en CM / CC2 -- Exercice 1 -- 1 pt

CC2 -- Exercice 1 -- 1 pt

Exercice

ULB UNIVERSITÉ DE BRUXELLES

Exercice

Sortie du code

Indice du minimum
Auteur: Sébastien Hoarau

- Nombre d'essais: 1
- Date du dernier essai: 28 octobre 2022 13:41
- Résultat: **Non**

Énoncé:

Écrire une fonction `indice_min` qui prend une liste non vide d'entiers en paramètre et renvoie l'indice du plus petit élément. Si le plus petit élément apparaît plusieurs fois, la fonction doit renvoyer le premier indice (le plus petit).

Exemples :

```
>>> indice_min([10, 5, 20])
1
>>> indice_min([21, 22, 23, 21])
0
>>> indice_min([10])
0
```

Figure 8: 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

<https://www.ulb.ac.be>
<https://www.ulb.ac.be/di>