Observations sur l'initiation au test unitaire dans une UE d'apprentissage de la programmation Python en L1

Mirabelle Nebut[1 [0009−0007−9993−0935]], Yvan Peter[1 [0000−0002−1145−357X]], and Maude Pupin[1 [0000−0003−3197−0715]]

Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
prenom.nom@univ-lille.fr

Abstract. Le but de ce travail est d'observer si, dans le contexte de l'apprentissage de la programmation Python, des étudiant·es peuvent utiliser un outil de test unitaire conçu spécifiquement pour des débutant·es. Pour notre expérimentation, conduite en L1 portail Mathématiques-Informatique, nous avons introduit dans l'UE d'apprentissage de la programmation une sensibilisation au test unitaire avec le support de l'outil `L1Test`, intégré à l'IDE `Thonny`. Les traces d'activité des étudiant·es sur `Thonny` ont été collectées puis nettoyées pour ne garder que les exercices de programmation guidés réalisés dans l'UE. Nous avons observé l'usage de l'outil : dans quelle mesure les étudiant·es ont écrit des tests unitaires pour leurs fonctions et dans quelle mesure ils ou elles ont exécuté ces tests avec `L1Test`. On constate qu'une majorité d'étudiant·es écrit des tests et les exécute, y compris les "vrais débutant·es" n'ayant pas suivi la spécialité NSI.

Keywords: Test unitaire, Apprentissage de la programmation, Python

1 Introduction

Il est maintenant largement reconnu que le test du logiciel doit être abordé dans les enseignements au plus tôt. Plusieurs revues et cartographies systématiques de la littérature traitent la question de l'enseignement du test dans les cours de programmation introductifs [8,5]. Ces travaux mettent en lumière différentes méthodes d'enseignement du test (par exemple le processus de développement encouragé, l'outil utilisé) et les résultats produits par ces méthodes en terme d'apprentissage. Parmi les avantages identifiés figurent une meilleure qualité des programmes et une meilleure compréhension de l'activité de programmation. Parmi les inconvénients identifiés on trouve des tests écrits de mauvaise qualité (notamment parce que les compétences de programmation nécessaires ne sont pas encore acquises) et une absence de motivation de la part des étudiant·es. On note aussi dans [2] une augmentation de la charge cognitive (apprentissage de la syntaxe des tests, prise en main de l'environnement de test). On remarquera que, parmi les 204 articles étudiés dans [2], seuls 78 (environ 38%) ont une démarche expérimentale de validation des contributions.

Notre expérimentation se déroule dans le contexte de la première année de licence (L1) du portail Mathématiques-Informatique (MI) qui propose une UE obligatoire de programmation Python au premier semestre, sans pré-requis. Son contenu couvre les bases de la programmation impérative (pas de programmation objet). Les travaux pratiques (TP) de cette UE sont historiquement basés sur l'utilisation de l'IDE `Thonny`, qui propose des fonctionnalités adaptées aux débutant·es. L'enseignement de la programmation se fait en même temps que l'apprentissage d'un environnement de travail assez riche (système de fichiers, débogueur, etc.). Cette approche contraste avec l'utilisation d'éditeurs de code simplifiés qui s'exécutent dans le navigateur et peuvent donner de bons résultats quant aux apprentissages en programmation [6].

Dans ce contexte, nous nous sommes posé la question d'intégrer des notions de test du logiciel à l'UE en essayant de ne pas augmenter la charge mentale des étudiant·es. On notera que l'activité de test diffère de l'approche par essai-erreur [1] qui consiste à exécuter son code de manière non structurée. Si on reprend les considérations de [8] : nous avons choisi de faire du *test unitaire*, l'unité testée étant une *fonction qui calcule un résultat*. Les tests écrits restent simples : par exemple nous ne faisons pas tester les fonctions basées sur les entrées-sorties (saisie, affichage, etc), voir par exemple [3] pour d'autres choix. Nous avons choisi une démarche de développement itérative de type "écriture des tests - écriture du code - exécution des tests" (*test first*) sans aller jusqu'à une démarche de type *Test Driven Development* (TDD, mentionné par 27% des articles étudiés par [8]). Nous avons enfin fait le choix de faire écrire leurs propres tests aux étudiant·es (et non de les fournir à des fins d'évaluation), en les sensibilisant à la notion de test nominal et test aux bornes.

Quant au choix de l'outil de test, les documents d'accompagnement sur Eduscol pour la classe de première[1] et de terminale[2] mentionnent les outils `pytest` et `doctest`. L'utilisation de la bibliothèque `pytest`[3] est simple comparée aux outils à la xUnit, mais les tests sont stockés dans un fichier séparé de celui du programme et chaque test est écrit à l'intérieur d'une fonction dont le nom indique l'objectif du test. Ce fonctionnement semble compliqué pour des débutant·es qui apprennent la notion de fonction, et ont du mal à gérer des fichiers. Notre expérience préalable du module Python `doctest`[4], qui propose d'écrire les tests dans la docstring des fonctions, semblait indiquer que sa syntaxe très simple était adaptée à des débutant·es. Ce principe a été repris pour Java dans l'outil `comTest` [4]. Néanmoins l'expérience semblait indiquer que `doctest` présente aussi des inconvénients : sémantique difficile à expliquer car basée sur l'affichage obtenu sur la sortie standard, affichage des verdicts rébarbatif.

Nous avons donc développé `L1Test`[5], outil de test unitaire pensé pour les débutant·es en programmation, fourni sous la forme d'un greffon pour `Thonny`.

[1] `https://eduscol.education.fr/document/30058/download`
[2] `https://eduscol.education.fr/document/7298/download`
[3] `https://pytest.org/`
[4] `https://docs.python.org/3/library/doctest.html`
[5] `https://pypi.org/project/L1test/`

Les tests reprennent la syntaxe de `doctest` mais avec une sémantique qui, selon nos constatations empiriques, est plus adaptée à des débutant·es. L'affichage basé sur le code vert/rouge est classique dans les bibliothèques et les IDE proposant des tests unitaires. Nous avons ajouté à `Thonny` un autre greffon permettant de collecter les traces d'activité de programmation des étudiant·es. Dans cet article nous présentons les premiers résultats d'une analyse des traces collectées lors du premier semestre 2024-2025. Notre objectif est de répondre à la question de recherche suivante : *dans le contexte d'un cours d'apprentissage de la programmation Python pour débutant·es, les étudiant·es ont-ils ou elles été en mesure d'utiliser un outil de test conçu spécifiquement ?*

Nous expliquerons d'abord le cadre expérimental (section 2) avant de présenter l'analyse des résultats obtenus (section 3) puis de conclure par des perspectives.

2 Dispositif expérimental

2.1 Outil de test `L1Test`

Nous présentons ici les caractéristiques principales de `L1Test`.

Syntaxe des tests `L1Test` prend à `doctest` l'écriture des tests dans la docstring des fonctions : les tests sont à proximité immédiate du code. La syntaxe (figure 1) est similaire à celle de `doctest`, nous avons remplacé l'invite `>>>` par l'invite `$$$` pour éviter toute confusion. La syntaxe est donc très proche de celle de l'interpréteur Python. L'expression à droite de l'invite (généralement un appel à la fonction testée, dans notre exemple la fonction `carres`) et l'expression sous l'invite (la valeur attendue pour cet appel) sont des expressions Python. Dans son expression la plus simple la syntaxe des tests est donc entièrement basée sur la syntaxe du langage et de l'interpréteur Python, que les étudiant·es apprennent déjà (pour le langage) et observent déjà (pour l'interpréteur) dans le cours de programmation. Notons que `L1Test` ne permet pas de tester les affichages (instruction Python `print`)[6].

Sémantique des tests La sémantique de `L1Test` est basée sur l'égalité (opérateur Python `==`) des valeurs obtenues en évaluant les expressions à droite de l'invite et sous l'invite. Les verdicts des tests sont classiques : *succès* (le résultat de la comparaison des valeurs obtenue et attendue vaut vrai, par exemple le premier test de la figure 1 est un succès car la valeur de `carres([]) == []` est `True`), *échec* (le résultat de la comparaison vaut faux, par exemple le second test est en échec car la valeur de `carres([2, 4, 0]) == [4, 16, 0]` est `False`) et *erreur* (l'évaluation de la comparaison produit une erreur). Cette sémantique correspond au `assertEqual` des bibliothèques de test standard et est explicable dès que les étudiant·es ont abordé le type booléen de Python.

[6] Cette limitation est un choix délibéré, dans le but de séparer clairement les fonctions (testables) qui renvoient une valeur et les fonctions (non testables) qui réalisent un affichage mais ne renvoient rien, en soutien à la partie du cours qui explique la différence entre "renvoyer une valeur" et "afficher une valeur".

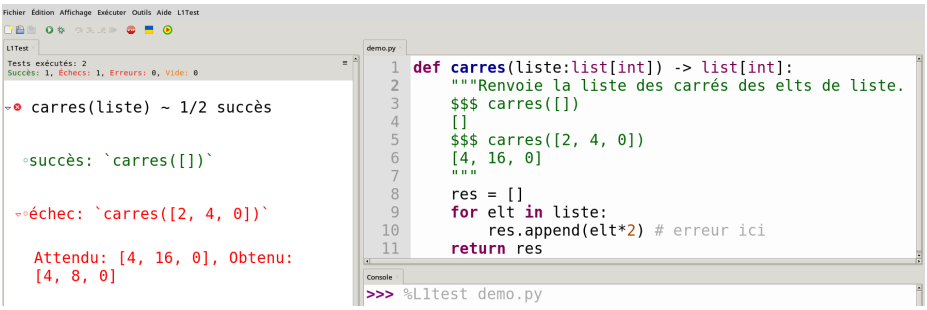


Fig. 1: Syntaxe des tests et exemples d'utilisation avec `L1Test`

Exécution des tests Le greffon `L1Test` ajoute à `Thonny` deux moyens d'exécuter les tests. Le premier est un bouton "Exécuter tous les tests" (appelé `RunTest` par la suite) qui exécute tous les tests de toutes les fonctions présentes dans le fichier contenu dans l'éditeur. C'est le fonctionnement classique des bibliothèques de test unitaire. Son défaut est que l'exécution de tous les tests produit une masse d'information à assimiler, ce qui peut poser problème pour des débutant·es. `L1Test` permet aussi de sélectionner le nom de la fonction dont on souhaite exécuter tous les tests. Par contraste l'IDE `BlueJ`, qui propose des fonctionnalités simplifiées pour l'apprentissage de la programmation objet et du test unitaire avec Java, a un grain d'exécution encore plus fin, au niveau du test [7].

Affichage des verdicts Les verdicts sont affichés dans une fenêtre dite *fenêtre des tests* qui s'ouvre à gauche de la fenêtre principale de `Thonny`, dans l'ordre de définition des fonctions dans le programme et des tests dans les docstrings. On utilise le code couleur vert et rouge traditionnel en test unitaire. Notons que les débutant·es doivent déjà être relativement à l'aise avec les interfaces graphiques pour utiliser cette fenêtre de manière optimale, pour penser à l'agrandir ou à utiliser son ascenseur.

Processus de programmation et de test `L1Test` suggère d'écrire les tests avant le code : une docstring pré-complétée avec une invite de test est automatiquement insérée dans l'éditeur quand l'étudiant·e termine la ligne de code qui définit la signature d'une fonction Python. Une fois les tests et le code de la fonction écrits, l'étudiant·e peut exécuter les tests (tous ou ceux de la fonction qui vient d'être écrite). Le code étant importé automatiquement dans la console, l'étudiant·e peut entrer directement dans un processus de mise au point en faisant des essais dans la console. `L1Test` n'est pas dirigiste sur le processus de développement. Nos observations en TP (non quantifiées) montrent que certain·es étudiant·es préfèrent se passer dans un premier temps de la fenêtre de test : ils utilisent le bouton "Exécuter le script courant" de `Thonny` (appelé par la suite bouton `RunProgram`) puis effectuent un par un les appels correspondant aux tests prévus dans la console de `Thonny`, en terminant éventuellement par une exécution des tests dans `L1Test`.

2.2 Enseignement du test dans la première année de licence

L'UE de programmation dure 12 semaines, avec une notion liée à la programmation introduite par semaine durant les 9 premières semaines. Les 2 séances de travaux pratiques (TP) hebdomadaires sont basées sur des exercices divers (ex : découverte d'un type). L'analyse de la section 3 considère uniquement les exercices qui consistent à programmer des fonctions de manière très guidée : pour ces feuilles d'exercices on fournit un fichier à compléter qui donne le nom et la signature de ces fonctions. Pour certaines fonctions, dites non testables par la suite, nous n'attendons pas de tests (par exemple : fonction dépendant de l'aléatoire, ou d'affichage). Pour les fonctions dites testables : les consignes fournissent certains tests, avec ou sans leur syntaxe, ou donnent des indications sur les tests à réaliser ou simplement incitent à écrire des tests. En fin de semestre, 3 semaines sont dédiées à des TPs non guidés qui visent à réaliser sans aucune indication des jeux de type casse-tête sur grille 2D en mode texte, mêlant fonctions testables et non testables choisies par les étudiant·es.

Le tableau 1 montre, selon la semaine, la notion abordée, l'utilisation ou non de `L1Test`, le nombre de fonctions pour lesquelles nous attendons des tests, ainsi que les semaines que nous prendrons en compte dans l'analyse de la section 3. Les fonctions non testables d'affichage et de saisie de la semaine 5 ainsi que les jeux dans lesquels nous ne pouvons pas repérer les fonctions testables ne sont pas pris en compte.

semaine	notion travaillée	utilisation de `L1Test`	nb de fonctions testables à coder	analyse des tests effectuée
1	types et variables		-	
2	fonctions et tests		8	
3	type booléen		14	
4	conditionnelles		16	
5	entrées-sorties		-	
6	boucles, listes		8	
7	boucles (suite)		12	
8	boucles (suite)		10	
9	boucles (suite)		24	
10-12	jeux non guidés		-	

Table 1: Détails sur l'organisation de l'UE et de l'analyse par semaine

2.3 Données collectées

Les traces d'activité des étudiant·es sont collectées uniquement sur les ordinateurs des salles TP du département informatique, pendant les TPs. En début d'année certain·es étudiant·es doivent travailler en binômes sur le même

ordinateur, les effectifs étant trop importants pour le nombre de machines. L'identifiant de l'étudiant·e qui n'est pas connecté·e est demandé à l'ouverture de `Thonny`. Chaque trace collectée compte alors pour les 2 étudiant·es, comme si la trace était dupliquée, ce qui constitue un biais pour notre analyse.

Les informations collectées sont : l'identifiant universitaire de l'étudiant·e connecté·e (et de l'éventuel·le binôme), l'horodatage de l'action, les actions de lancement et fermeture de `Thonny`, d'ouverture et de sauvegarde de fichier, d'exécution du fichier ouvert dans l'éditeur (par le bouton qui déclenche le débogueur, le bouton `RunProgram` et le bouton ou le menu `RunTest`), le contenu de l'éditeur pour chaque action et les commandes exécutées dans la console.

Une importante phase de nettoyage des traces a été réalisée avant l'analyse de manière à identifier les traces qui correspondent aux TPs décrits dans la table 1, et parmi elles, celles qui concernent les exercices guidés de programmation de fonctions[7]. Quand la trace concerne une commande exécutée dans la console de `Thonny`, il est difficile de rattacher cette commande à l'activité en cours. Quand il n'est pas possible d'identifier ce sur quoi travaillait l'étudiant·e au moment où sa trace a été collectée, nous ignorons la trace. Les traces ignorées constituent un biais de l'analyse, car on ignore toute l'activité qu'elles contiennent.

Nous n'avons pas considéré les traces des étudiant·es qui n'ont pas consenti à ce que leur activité d'utilisation de `Thonny` soit utilisée à fin de recherche. En 2024-2025 nous avons collecté avant nettoyage les 302763 traces de 302 étudiant·es. Après application de la RGPD nous avons obtenu les 234969 traces de 257 étudiant·es (étudiant·es connecté·es et binômes). Quand on se restreint aux feuilles d'exercices de programmation guidées des semaines 2 à 4 puis 6 à 9 on obtient les 111943 traces de 226 étudiant·es.

3 Analyse des traces collectées

Pour répondre à notre question de recherche (*dans le contexte d'un cours d'apprentissage de la programmation Python pour débutant·es, les étudiant·es ont-ils ou elles été en mesure d'utiliser un outil de test conçu spécifiquement ?*) nous avons analysé les traces des TPs guidés des semaines 2 à 4 puis 6 à 9, comme expliqué en section 2.2. La pratique du test implique principalement 3 activités : 1- écrire des tests, 2- exécuter ces tests une fois le code à tester écrit et 3- interpréter le verdict des tests et agir en conséquence. Dans ce travail nous n'analysons pas l'action qui suit l'exécution des tests et traitons uniquement les points 1 et 2. Nous examinons tout d'abord dans quelle mesure les étudiant·es ont écrit des tests durant les TPs guidés (section 3.1), puis dans quelle mesure ils ont exécuté ces tests en utilisant les fonctionnalités de `L1Test` (section 3.2). Au préalable nous décrivons la composition de la promotion selon le cursus antérieur de programmation.

[7] Dans la suite de cet article l'acronyme "TP2" est utilisé pour désigner la feuille d'exercices de programmation associée à la notion travaillée en semaine 2 (en l'occurrence la feuille sur les fonctions).

La formation n'effectue pas de sélection à l'entrée sur les compétences en programmation ni sur la maîtrise des outils numériques. Elle accueille donc d'ex-lycéen·nes ayant suivi la spécialité NSI et connaissant à l'avance tout le contenu de l'UE, comme des novices n'ayant que très peu touché un ordinateur auparavant. Pour isoler les "vrais débutant·es" nous avons enrichi les traces avec le cursus antérieur de l'étudiant·e pour séparer d'une part les *non débutant·es* : étudiant·es ayant fait au moins un an de spécialité NSI (première ou première et terminale), doublant·es et d'autre part les *débutant·es* : le reste de la promotion.

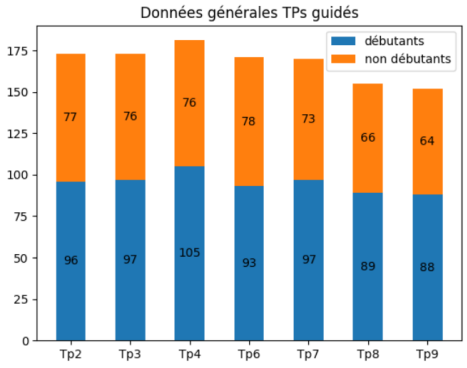


Fig. 2: Composition de la promotion au fil des TP guidés.

La composition des étudiant·es au fil des TP guidés est montrée figure 2. Le nombre d'étudiant·es ayant réalisé en présentiel les TPs guidés oscille entre 152 (semaine 9) et 181 (semaine 4) selon la semaine, avec une moyenne de 168 étudiant·es par semaine. Les chiffres reflètent l'évolution classique d'une promotion de L1 au fil du semestre, avec une décroissance des effectifs à partir du milieu du semestre due aux abandons. Parmi ces étudiant·es, le pourcentage d'étudiant·es débutant·es ayant réalisé en présentiel les TPs guidés oscille entre 54% et 58% selon la semaine, avec une moyenne de 56,4% d'étudiant·es débutant·es (écart-type : 1,5).

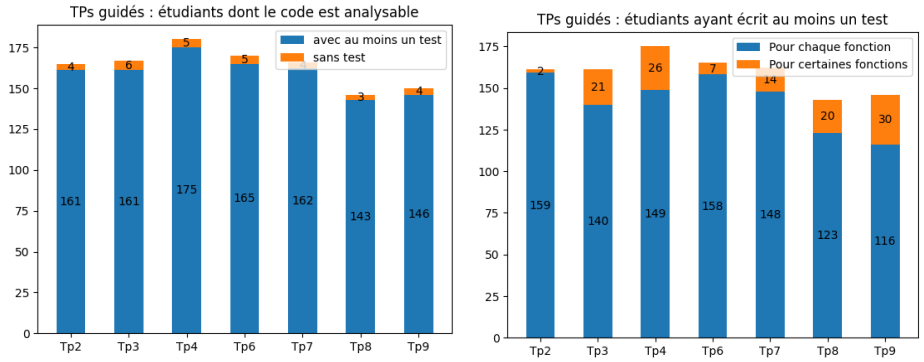
3.1 Écriture des tests

Recherche dans les traces des tests écrits Nous cherchons les tests dans les traces qui comprennent le code contenu dans l'éditeur au moment de l'action, en priorisant les traces les plus récentes (version du code de l'étudiant·e estimée la plus aboutie en fin de TP)[8]. Notre extraction commence par construire l'arbre syntaxique (*Abstract Syntax Tree*) du code Python puis repère les tests qui

[8] Cette heuristique est un biais de notre analyse, surtout dans le cas où 2 étudiant·es ont travaillé seul·es puis en binôme.

respectent la syntaxe de `L1Test`[9]. Nous excluons donc de notre analyse le travail des étudiant·es qui ne maîtrisent pas les aspects syntaxiques de Python. Le nombre de travaux exclus reste toutefois faible en regard du nombre total. Sur les TPs 2 à 9 nous avons exclu en moyenne 2,67% des travaux, qui sont en moyenne à 90% des travaux de débutant·es (écart-type : 17,85). Cette sur-représentation des débutant·es est cohérente avec la non maîtrise syntaxique du langage.

Proportion d'étudiant·es n'ayant écrit aucun test La figure 3a indique que les travaux syntaxiquement analysables ne contenant aucun test syntaxiquement correct sont en très petit nombre. En moyenne sur l'ensemble des TPs 2,7% des étudiant·es dont le travail est analysable n'ont écrit aucun test, avec un écart-type de 0,49.



(a) Présence de tests dans les codes syntaxiquement analysables.

(b) Présence de tests dans les fonctions écrites testables.

Fig. 3: Présence de tests écrits dans les TPs guidés.

Proportion d'étudiant·es ayant écrit au moins un test pour chaque fonction codée testable La brique de base pour le test unitaire étant la fonction, l'écriture d'au moins un test par fonction testable peut représenter un comportement nominal pour des débutant·es (le comportement nominal réel consistant à écrire un jeu de tests pertinents). Nous avons regardé (figure 3b) dans quelle proportion les fonctions codées pour lesquelles des tests étaient attendus contiennent effectivement des tests. Les résultats montrent que la plupart des étudiant·es écrivent des tests pour toutes les fonctions testables réalisées. La moyenne sur les TPs indique que 86,7% des étudiant·es dont le code est analysable écrivent des tests pour toutes les fonctions testables, avec un écart-type à 6,5, un maximum

[9] Les expressions Python apparaissant dans les tests ne sont pas nécessairement syntaxiquement correctes à ce stade.

à 96,3% pour le TP2, et un tassement pour le TP9 (pour ce TP environ 77% des étudiant·es ayant écrit des tests l'ont fait pour toutes les fonctions testables).

Nombre de fonctions avec tests Le nombre moyen de fonctions avec au moins un test varie selon les TPs et est systématiquement légèrement inférieur pour les débutant·es, sans doute parce qu'ils ou elles avancent moins vite. En moyenne sur l'ensemble des TPs les étudiant·es ont écrit 8,5 fonctions avec tests (écart-type : 4,7), contre 7,9 fonctions pour les débutant·es (écart-type : 4,8). Ce nombre est le plus bas pour le TP8 (moyenne de 6,5 fonctions écrites pour la promotion et 5,6 pour les débutant·es). On observe un pic pour le TP9 (feuille mal calibrée avec un grand nombre de fonctions demandées), avec une moyenne de 12,75 fonctions testées pour la promotion et 12,11 pour les débutant·es). Ces données (à mettre en regard avec celles de la table 1) montrent qu'en moyenne les étudiant·es programment et testent une partie seulement des fonctions demandées, mais que cette partie est conséquente.

Nombre moyen de tests écrits par fonction Le nombre moyen de tests par fonction varie selon les TPs, mais reste homogène quant au cursus antérieur des étudiant·es (peu de différence constatée entre les débutant·es et la promotion). Le nombre moyen de tests par fonction est de 2,36 (écart-type : 0,7) sur l'ensemble des TPs (les tests vides dans les docstrings générées automatiquement ne sont pas comptabilisés). Le plus petit nombre moyen de tests écrits par fonction est constaté pour les TP2, TP6 et TP7 (autour de 1,7 tests par fonction) et le plus grand est constaté pour le TP9 (3,4 tests par fonction). Ces variations s'expliquent par les différences entre les feuilles d'exercices proposées et la précision des indications données concernant les tests à écrire. Par exemple pour la fonction `carres` de la semaine 6, un test nominal était donné, et une mise en garde globale à la feuille d'exercices rappelait que tous les tests n'étaient pas fournis. Les étudiant·es étaient censés penser à tester `carres` sur la liste vide. Le nombre moyen de tests pour cette fonction est 1,88, ce qui indique qu'un certain nombre d'étudiant·es ne s'est pas contenté du test fourni. Pour la fonction `est_palindrome` du TP9, la consigne était : *Prévoir des tests pour un palindrome de longueur paire, impaire, nulle, égale à 1, pour une chaîne non palindrome*. Le nombre moyen de tests pour cette fonction est autour de 4,5. Pour la fonction `compare` du TP4 (retour de 1, -1 ou 0 selon la valeur des 2 paramètres, 3 tests pertinents attendus) aucune consigne de test n'était donnée, le nombre moyen de tests est de 2,8. Les étudiant·es écrivent donc des tests même quand les consignes sont imprécises ou absentes.

3.2 Exécution des tests

Méthode utilisée pour vérifier que les tests écrits ont été exécutés Écrire des tests sans les exécuter montre une incompréhension du processus de test. Nous cherchons donc à vérifier que chaque étudiant·e a exécuté au moins une fois les tests de chaque fonction qu'il ou elle a écrit. Chaque exécution de test produit une trace de type `RunTest` qui contient les fonctions testées ainsi que les verdicts.

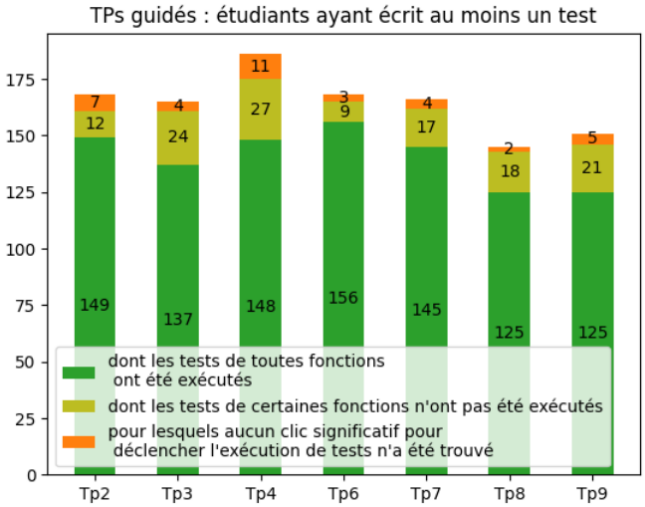


Fig. 4: Exécution des tests écrits durant les TPs guidés

Nous analysons les noms de fonctions présents dans ces traces pour voir si les fonctions dont les tests ont été exécutés par l'étudiant·e couvrent l'ensemble des fonctions présentant des tests qu'il ou elle a écrites, en reprenant les fonctions et les tests trouvés dans les contenus de l'éditeur sélectionnés dans la section 3.1.

Comportements relatifs à l'exécution des tests La figure 4 montre les résultats de cette analyse. On trouve en ordonnée non pas les étudiant·es de toute la promotion, mais celles et ceux ayant écrit au moins un test lors des TPs guidés (sans quoi chercher si les tests ont été exécutés n'a pas de sens). Nous nous concentrons sur deux comportements extrêmes : les étudiant·es qui ont exécuté les tests de toutes les fonctions avec tests qu'ils ont écrites pour le TP et au contraire les étudiant·es qui n'ont jamais exécuté de tests pour le TP.

Les étudiant·es qui ont exécuté via une interaction avec `L1Test` les tests de toutes les fonctions (présentant des tests) qu'ils ont écrites représentent en moyenne 88,5% des étudiant·es ayant écrit des tests (écart-type : 3,88), avec un minimum de 84,6% pour le TP4 et un maximum de 94,5% pour le TP6. Ce chiffre montre que la grande majorité des étudiant·es a compris *a priori* qu'un test s'exécute et a l'habitude d'exécuter ses tests. Notons que l'analyse ne nous informe pas complètement sur l'activité de test : il est possible et observé en TP (observations non quantifiées) que l'exécution des tests dans `L1Test` soit précédée d'une phase d'essai-erreur dans la console.

La figure 4 montre aussi les étudiant·es pour lesquel·les nous n'avons trouvé dans les traces aucune exécution significative[10] de tests durant un TP donné.

[10] "Significatif" signifie ici que nous avons exclu les traces qui résultent d'une interaction dans `L1Test` visant à exécuter des tests, mais pour lesquelles le code dans l'éditeur ne contenait aucun test.

Ces étudiant·es pour qui nous avons trouvé la présence d'au moins un test, mais sans aucune exécution via `L1Test` représentent en moyenne 3,18% des étudiant·es ayant écrit des tests (écart-type : 1,69), sur l'ensemble des TP, avec un maximum de 6,3% pour le TP4 et un minimum de 1,39% pour le TP8. Sur l'ensemble de la promotion, seul·es 7 étudiant·es présentent du code avec au moins un test écrit mais n'ont jamais exécuté un test de tout le semestre.

Le reste des étudiant·es n'a exécuté que partiellement les tests écrits. Ces étudiant·es représentent en moyenne 11,5% des étudiant·es ayant écrit des tests (écart-type : 3,9). En moyenne sur l'ensemble des TPs, les étudiant·es pour lesquel·les seule une fonction a des tests non exécutés représentent 56,9% des étudiant·es avec exécution partielle (écart-type : 15,2). Ces résultats sont difficiles à interpréter et nécessitent une analyse plus fine (fin de TP interrompant la mise au point d'une fonction, heuristique de recherche des tests écrits...).

Nous n'avons pas observé de différence entre les étudiant·es débutant·es et la promotion quant à l'exécution des tests. On trouve 87,3% de débutant·es ayant exécuté (via une interaction avec `L1Test`) les tests de toutes les fonctions (présentant des tests) qu'ils ont écrites. On trouve 3,8% de débutant·es parmi lesquel·les nous n'avons trouvé dans les traces aucune exécution significative de tests durant un TP donné.

4 Conclusion

La question de recherche étudiée dans cet article porte sur la capacité des étudiant·es novices en programmation à utiliser, en parallèle de l'apprentissage de la programmation Python, l'outil de test unitaire `L1Test` conçu pour des débutant·es et intégré à l'IDE `Thonny`.

Nous avons analysé dans quelle mesure les étudiant·es écrivent et exécutent des tests pour leurs fonctions. Les résultats tendent à montrer que, lorsque la tâche consiste à écrire et tester une fonction dont la signature est donnée, les étudiant·es n'ont pas de problème avec la syntaxe des tests. En moyenne 86,7% de la promotion écrit des tests pour chaque fonction écrite. Les étudiant·es écrivent en moyenne plus de 2 tests par fonction. Le nombre de fonctions avec tests écrites par TP est significatif. De plus on constate que les étudiant·es utilisent les moyens d'exécution des tests fournis par `L1Test`. La proportion d'étudiant·es qui exécutent les tests de toutes les fonctions qu'ils ou elles ont écrite est en moyenne de 88,5%, ce qui semble indiquer un usage des tests dans le processus de développement. Nous n'avons pas constaté que le cursus antérieur de l'étudiant·e ("vrai·es débutant·es" découvrant la programmation ou étudiant·es ayant déjà programmé auparavant) a une influence, même si le nombre de fonctions avec tests écrites par les débutant·es est légèrement inférieur à celui de la promotion.

Ces premiers résultats sont encourageants et doivent être complétés par une analyse plus fine du comportement des étudiant·es. Cette analyse nous permettra de vérifier la bonne compréhension des résultats des tests en observant la prise en compte des tests échoués dans le processus de résolution de problème. Par ailleurs cela nous permettra d'observer les stratégies d'écriture de tests entre

une vraie approche de programmation dirigée par les tests (écriture des tests en premier) et écriture à la fin du TP au pire des cas. Cela nous renseignera sur la compréhension de l'intérêt des tests. Nous souhaitons en particulier comparer le processus de programmation et de test des étudiant·es venant de la spécialité NSI par rapport à celui des débutant·es. En effet, les premiers peuvent avoir déjà pris des habitudes de programmation qui freinent l'adoption du test unitaire. Enfin nous avons essentiellement évalué l'utilisabilité [9] de `L1Test`. Nous souhaitons maintenant étendre notre dispositif pour évaluer son acceptabilité et son utilité, notamment évaluer si l'initiation au test apporte un bénéfice pédagogique concernant l'apprentissage de la programmation.

Acknowledgments. Ce travail a été partiellement financé par le projet InterReg FWV 0100029 Open Badges For IT (OB4IT). Nous remercions les étudiant·es du département Informatique qui ont contribué à ce travail, notamment Reda Id Taleb (`L1Test`), Amadou Barro, Thomas Briche, Corentin Duvivier et Sana Mirahsani.

References

1. Edwards, S.H.: Using software testing to move students from trial-and-error to reflection-in-action. In: Proceedings of the 35th SIGCSE technical symposium on Computer science education. pp. 26–30 (2004)
2. Garousi, V., Rainer, A., Lauvås, P., Arcuri, A.: Software-testing education: A systematic literature mapping. Journal of Systems and Software **165**, 110570 (Jul 2020). https://doi.org/10.1016/j.jss.2020.110570
3. Janzen, D.S., Saiedian, H.: Test-driven learning: intrinsic integration of testing into the cs/se curriculum. Acm Sigcse Bulletin **38**(1), 254–258 (2006)
4. Lappalainen, V., Itkonen, J., Isomöttönen, V., Kollanus, S.: Comtest: a tool to impart tdd and unit testing to introductory level programming. In: Proceedings of the fifteenth annual conference on Innovation and technology in computer science education. pp. 63–67 (2010)
5. Meira, A.C.B.T., Guerrero, D.D.S., Andrade, W.L.: A Systematic Literature Review on the Use of Software Testing and Programming with Novice Students. In: 2024 IEEE Frontiers in Education Conference (FIE). pp. 1–7. IEEE, Washington, DC, USA (Oct 2024). https://doi.org/10.1109/FIE61694.2024.10893450
6. Nguyen, H.A., Bogart, C., Šavelka, J., Zhang, A., Sakr, M.: Examining the trade-offs between simplified and realistic coding environments in an introductory python programming class. In: European Conference on Technology Enhanced Learning. pp. 315–329. Springer (2024)
7. Patterson, A., Kölling, M., Rosenberg, J.: Introducing unit testing with bluej. ACM SIGCSE Bulletin **35**(3), 11–15 (2003)
8. Scatalon, L.P., Carver, J.C., Garcia, R.E., Barbosa, E.F.: Software Testing in Introductory Programming Courses: A Systematic Mapping Study. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education. pp. 421–427. ACM, Minneapolis MN USA (Feb 2019). https://doi.org/10.1145/3287324.3287384
9. Tricot, A., Plégat-Soutjis, F., Camps, J.F., Amiel, A., Lutz, G., Morcillo, A.: Utilité, utilisabilité, acceptabilité: interpréter les relations entre trois dimensions de l'évaluation des eiah. In: Environnements Informatiques pour l'Apprentissage Humain 2003. pp. 391–402. ATIEF; INRP (2003)