Apprentissage de la programmation Python - Une première analyse exploratoire de l'usage des tests

Mirabelle Marvie-Nebut, Yvan Peter

▶ To cite this version:

Mirabelle Marvie-Nebut, Yvan Peter. Apprentissage de la programmation Python - Une première analyse exploratoire de l'usage des tests. Atelier " Apprendre la Pensée Informatique de la Maternelle à l'Université ", dans le cadre de la conférence Environnements Informatiques pour l'Apprentissage Humain (EIAH), Jun 2023, Brest, France. pp.1-8. hal-04144206

HAL Id: hal-04144206
https://hal.science/hal-04144206

Submitted on 28 Jun 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.

Actes de l'atelier "Apprendre la Pensée Informatique de la Maternelle à l'Université"
Brest, France, June 13, 2023

Apprentissage de la programmation Python - Une première analyse exploratoire de l'usage des tests

Mirabelle Nebut[1] Yvan Peter[1]
(1) Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
mirabelle.nebut@univ-lille.fr, yvan.peter@univ-lille.fr

RÉSUMÉ

Cet article présente un travail qui démarre sur l'usage des tests pour l'apprentissage de la programmation en Python en première année d'université. L'article présente l'instrumentation qui a été réalisée sur l'environnement de programmation `Thonny` ainsi que les premières visualisations issues du traitement de données exploratoire.

ABSTRACT

Here the title in English.

This article presents a starting research work about the use of tests for the learning of Python programming in the first year at university. The article presents the instrumentation developped in the `Thonny` developpement environnement as well as the first visualizations from the preliminary data exploration.

MOTS-CLÉS : Apprentissage de la programmation, tests unitaires, analyse exploratoire des données.

KEYWORDS: Programming learning, unit tests, exploratory data analysis.

1 Introduction

L'apprentissage de la programmation est notoirement difficile et génère beaucoup d'échecs et d'abandons. Il existe donc une recherche conséquente sur le sujet afin de comprendre les difficultés des apprenant·es et leurs comportements de programmation (Luxton-Reilly et al., 2018). Notre travail cible l'apprentissage des tests dans le cadre de l'apprentissage de la programmation en Python, dans le contexte de la première année de licence. Nous souhaitons, à terme, évaluer si l'usage des tests dès l'initiation à la programmation a un effet sur cet apprentissage (positif ou négatif). Nous souhaitons aussi évaluer quel dose de test introduire, entre sensibilisation et apprentissage.

Dans la mesure où les outils de tests existants peuvent être compliqués à mettre en œuvre par des novices en programmation et génèrent une charge cognitive supplémentaire, nous proposons un outil adapté à une initiation : `L1Test`. Celui-ci a été intégré sous forme de greffon à l'environnement de développement `Thonny` (Annamaa, 2015). Nous avons par ailleurs instrumenté `Thonny` afin de capturer des traces d'usage pour nos analyses.

Dans un premier temps nous présentons un état de l'art lié à l'utilisation des tests durant l'apprentissage de la programmation. Nous présentons dans un deuxième temps l'instrumentation réalisée sur `Thonny`. Enfin nous abordons une première exploration des données sur les traces collectées depuis septembre 2022 avant de définir les suites de ce travail.

APIMU 2023, pages 1–8.
Cette œuvre est mise à disposition sous licence Attribution 4.0 International.

2 État de l’art : test et apprentissage de la programmation

On trouve dans (Garousi et al., 2020) une synthèse des travaux de recherche existants concernant l’enseignement du test du logiciel dans son sens le plus large et dans (Scatalon et al., 2019) une synthèse exclusivement dédiée aux cours introductifs à la programmation. L’idée générale est que les étudiant·es apprennent traditionnellement le test dans des cours intermédiaires ou avancés, alors qu’il est maintenant largement reconnu qu’il devrait être abordé dès les cours introductifs à la programmation.

Nous nous intéressons pour notre part spécifiquement à l’enseignement du test unitaire intégré à l’enseignement des bases de la programmation avec Python. Dans (Garousi et al., 2020) on trouve une énumération des défis concernant à la fois les enseignant·es et les étudiant·es : les étudiant·es sont souvent peu motivé·es par l’activité de test ; la pratique du test automatisé demande souvent d’utiliser des outils et des méthodes nécessitant de l’expérience en programmation ; l’apprentissage du test augmente la charge cognitive (syntaxe des tests, prise en main de l’environnement de test). Ces deux derniers points sont particulièrement problématiques pour des étudiant·es qui affrontent l’apprentissage des bases de la programmation. Dans (Scatalon et al., 2019) on trouve une liste de bénéfices apportés par l’enseignement précoce du test (ex : le résultat des tests est un retour utile à l’étudiant·e) ainsi qu’une liste d’inconvénients, entre autres la difficulté pour les étudiant·es d’écrire leurs propres tests de manière pertinente et la difficulté à dégager du temps pour l’apprentissage du test (défi aussi présent dans (Garousi et al., 2020)). Les outils et les tâches liées au test doivent rester simples (notamment la syntaxe des tests, l’environnement de test, la méthodologie de développement et les activités liées au test) et s’intégrer à l’apprentissage de la programmation.

Nous nous intéressons ici particulièrement aux outils permettant l’apprentissage du test. Notre cours de première année n’aborde que la programmation impérative, or les bibliothèques de test unitaire utilisent majoritairement l’objet (à l’exception de la bibliothèque `pytest` [1] qui est sans syntaxe objet). Dans des cours plus avancés ou des cours de vérification et validation dédiés au test, l’utilisation d’outils de test réels est pertinent (Garousi et al., 2020). Dans notre contexte un outil dédié à l’apprentissage fait sens. Par ailleurs nous avons choisi d’intégrer notre outil à l’environnement de développement intégré (IDE) `Thonny` utilisé par les étudiants. Nous n’abordons donc pas du tout ici les outils type plate-forme web.

L’apprentissage du test chez des débutant·es en Python se fait maintenant dès le lycée dans la spécialité NSI, avec une problématique d’outillage similaire. Les documents d’accompagnement sur Eduscol pour la classe de première [2] et de terminale [3] mentionnent les outils `pytest` et `doctest` [4] ainsi que l’utilisation du mot-clé Python `assert`. La bibliothèque `pytest` demande d’écrire les tests dans un fichier séparé de celui du programme, à l’intérieur de fonctions dont le nom évoque l’objectif du test. Ce principe peut poser problème aux débutant·es malhabiles dans leur usage d’un gestionnaire de fichier et apprenant tout juste à écrire des fonctions. L’utilisation du mot-clé `assert` à l’intérieur des programmes ne nécessite pas d’apprendre une bibliothèque, mais éloigne beaucoup de l’usage classique des outils de test. En licence nous avons utilisé jusqu’à la rentrée 2022 l’outil `doctest` (inclus dans la distribution Python standard) : cet outil permet d’écrire des tests basiques très simplement dans la documentation des fonctions Python (docstring), en réutilisant la syntaxe

1. https ://pytest.org/
2. https ://eduscol.education.fr/document/30058/download
3. https ://eduscol.education.fr/document/7298/download
4. https ://docs.python.org/3/library/doctest.html

de l'interpréteur Python (voir aussi la section 3.1). Ce principe a été repris pour Java dans l'outil `comTest` (Lappalainen et al., 2010).

3 Instrumentation

Nous utilisons depluis plusieurs années l'IDE `Thonny` (Annamaa, 2015) dédié à l'apprentissage de la programmation avec Python. `Thonny` propose des fonctionnalités d'apprentissage utiles et un débogueur simple d'accès. L'exécution des programmes et les interactions avec l'interpréteur Python se font à l'intérieur de `Thonny`. Nous souhaitions intégrer l'activité de test à celle de programmation et donc l'inclure de même à l'intérieur de `Thonny`.

Les fonctionnalités de `Thonny` peuvent être étendues par un mécanisme de greffons que nous avons utilisé. La figure 1 présente l'interface globale de l'environnement ainsi que les deux greffons ajoutés qui sont décrits dans la suite. `Thonny` se présente sous la forme d'une interface graphique (Tkinter) avec une fenêtre principale contenant une fenêtre d'édition de code, une fenêtre contenant une console Python, ainsi que des boutons permettant notamment d'exécuter le contenu de l'éditeur.

FIGURE 1 – Environnement `Thonny` et ses greffons

3.1 Greffon de test pour `Thonny` L1test

L'outil `L1Test` s'inspire de `doctest` quant au principe d'écrire les tests unitaires directement dans la docstring des fonctions. Ces exemples servent à la fois de documentation pour le code et de tests exécutables. Comme `doctest`, la syntaxe de `L1Test` reprend très simplement la syntaxe de l'interpréteur Python, l'invite >>> ayant été changée en `$$$` pour ne pas mélanger les 2 outils. La figure 2 montre des tests pour des fonctions `somme_n_premiers_entiers` et `affiche_prenom` avec `doctest` et `L1Test`.

Le greffon `L1Test` ajoute à la fenêtre principale de `Thonny` un bouton permettant de lancer les tests (icône à gauche du drapeau ukrainien, figure 1) et à gauche une fenêtre dans laquelle s'affichent les verdicts des tests. Les tests qui passent apparaissent en vert, les tests qui ne passent pas apparaissent

```
>>> liste_somme_premiers_entiers(3)
[1, 3, 6]
>>> affiche_presentation("Paul")
Je suis Paul
```

```
$$$ liste_somme_premiers_entiers(3)
[1, 1 + 2, 1 + 2 + 3]
$$$ affiche_presentation("Paul")
None
```

FIGURE 2 – Exemples de tests avec `doctest` et `L1Test`

en rouge (conventions des bibliothèques de test classiques). Les fonctions sans test apparaissent en orange. Par contraste `doctest` est uniquement textuel dans la console, en noir et blanc.

Le différence avec `doctest` est sémantique. Le principe de `doctest`, initialement non dédié au test unitaire, est compliqué à comprendre à un stade d'initiation. Un test `doctest` passe uniquement si la valeur attendue est, comparée caractère par caractère, celle que fournirait l'interpréteur Python. Dans le test de la figure 2, l'évaluation de `liste_somme_premiers_entiers(3)` produit la liste `[1, 3, 6]`. Le test échouera si la valeur attendue est écrite `[1,3,6]` faute de caractère espace. Il est donc conseillé d'écrire puis exécuter le code de la fonction puis de copier la sortie dans la docstring. Cela incite à écrire les tests après le code, ce qui va à l'encontre des approches préconisant de commencer par les tests. De plus `doctest` permet de "tester" une procédure qui affiche une chaîne de caractère sur la sortie standard (cf `affiche_presentation` en figure 2), ce qui n'aide pas à lever la traditionnelle confusion des débutant·es entre `print` et `return`.

L'outil `L1Test` utilise au contraire l'opérateur de comparaison d'égalité == de Python pour comparer les valeurs attendue et obtenue. Les listes `[1,2]` et `[1, 2]` sont ainsi équivalentes. Il est possible d'utiliser une expression pour la valeur attendue (cf figure 2). Enfin une procédure d'affichage, qui renvoie la valeur `None` comme toute fonction Python sans instruction `return`, ne pourra être testée que via cette valeur et est non testable.

3.2 Greffon de collecte de traces L1log

Les étudiant·es génèrent un gros volume de données pendant les activités de programmation, quand ils ou elles interagissent avec l'environnement de programmation. Un grand nombre d'articles s'intéressent à la collecte et à l'analyse de ces données pour en tirer des conclusions sur le comportement des étudiant·es (Luxton-Reilly et al., 2018). Nous souhaitions récolter des informations sur l'utilisation de `L1Test` (nombre de clics sur le bouton "Run test", nombre de tests présents dans les programmes, etc) mais aussi sur la manière dont le test s'insère dans les pratiques de programmation des étudiant·es. Nous avons collecté des traces d'usage dépassant la seule utilisation de `L1Test`.

`Thonny` génère un événement Tkinter pour chaque interaction de l'utilisateur avec l'interface. Ces événements peuvent être enregistrés sous forme de fichiers JSON en local et un greffon permet de rejouer la session de programmation. Toutefois, ce niveau de trace est trop fin pour nos objectifs (chaque caractère ajouté dans l'éditeur génère un événement) et les traces doivent être exportées manuellement. Pour cette raison nous avons ajouté à `Thonny` un greffon (`L1Log`) chargé de filtrer, d'abstraire des événements plus sémantiques (par ex. exécution du code) et de les envoyer en temps réel dans un dépôt de données (*Learning Record Store* – LRS) au format xAPI [5] (Figure 3).

xAPI est un format de donnée JSON standardisé pour la représentation des activités d'apprentissage. Un enregistrement xAPI est composé a minima d'un *acteur* (l'apprenant), réalisant une action (le

5. `https://xapi.com/`

verbe, par ex. soumettre une réponse) sur un *objet* (par ex. un quiz). Les verbes permettant de décrire l'action ne sont pas définis par xAPI et sont tirés de vocabulaires extérieurs.

ProgSnap2 (pour *Programming Snapshot*) est une spécification pour la représentation et le stockage des données issues d'activités de programmation (Price et al., 2020). Cette spécification offre un vocabulaire spécifique aux activités liées à la programmation (manipulation de fichiers, compilation, exécution d'un programme, etc.).

Nous avons donc d'une part un format standard liés aux activités d'apprentissage, pour lequel il existe déjà des outils de stockage et de traitements et d'autre part une spécification *ad hoc* qui correspond aux activités qui nous intéressent. Nous avons combiné les deux en intégrant le vocabulaire de ProgSnap2 pour décrire les verbes et objets de nos traces dans le format xAPI. La figure 3 montre un exemple de trace pour un test (verbe `Run.Test` tiré de ProgSnap2). Le tableau 1 recense les propriétés collectées pour chaque événement.


```
{
 "timestamp":{"$date":"2022-10-07T11:15:44.981Z"},
 "actor":
   {
    "openid":"https://www.cristal.univ-lille.fr/users/281b59ea7d35e20",
    "objectType":"Agent"
   },
 "verb":{"id":"https://www.cristal.univ-lille.fr/verbs/Run.test"},
 "object":
 {
  "id":"https://www.cristal.univ-lille.fr/objects/Test",
  "objectType":"Activity",
  "extension":{"https://www.cristal.univ-lille.fr/objects/File/Filename":"file_-8712108631165266730",
  "https://www.cristal.univ-lille.fr/objects/Test/TestedExpression":"maximum(4, 1)",
  "https://www.cristal.univ-lille.fr/objects/Test/TestedLine":27,
  "https://www.cristal.univ-lille.fr/objects/Test/TestedFunction":"maximum"}
 },
 "result":
 {
  "success":true,
  "extension":{"https://www.cristal.univ-lille.fr/objects/Test/expectedResult":"4",
  "https://www.cristal.univ-lille.fr/objects/Test/obtainedResult":"4"}
 }
}
```


FIGURE 3 – Exemple de trace xAPI pour un test.

Propriété	**Obligatoire**	**Signification**
identifiant utilisateur	oui	hash basé sur le login de l'étudiant
identifiant session	oui	session `Thonny`, de son lancement à son arrêt
timestamp	oui	horodotage de l'événement
type d'événement	oui	correspond aux actions effectuées dans l'éditeur
résultat	oui	vrai si l'action s'est déroulée sans erreur, faux sinon
buffer	non	contenu de l'éditeur au moment de l'action
sortie	non	sortie produite lors de l'exécution d'un programme, des tests ou de l'utilisation du shell

TABLE 1 – Propriétés collectées pour les événements

4 Collecte et analyse exploratoire

4.1 Contexte de la collecte de données

L'expérimentation s'est déroulée lors du premier semestre de l'année universitaire 2022-2023 au sein de la première année du portail SESI. Ce portail SESI regroupe des étudiant·es se destinant à diverses licences scientifiques (Informatique, Chimie, etc). L'origine des étudiant·es et leur niveau en programmation est très hétérogène : bacs français avec ou sans spécialité NSI, bacs étrangers, réorientations. La collecte a eu lieu uniquement dans les groupes de TP de 2 amphis, soit environ 300 étudiant·es. Les autres groupes de TP ont utilisé la version de `Thonny` non instrumentée (sans collecte des traces) et `doctest`. Il est donc impossible de comparer l'utilisation des outils `doctest` et `L1Test`. Il est probable que certain·es enseignant·es, absent·es à la réunion de présentation de `L1Test`, ont fait utiliser `doctest`. Par ailleurs la collecte a souffert de divers défauts de conception de `L1Log` et `L1Test`, qui ont tout de même permis le déroulement des TPs.

4.2 Analyse des traces

Nous avons pour le moment travaillé sur le nettoyage des traces suite aux différents défauts de `L1Log` et `L1Test` et autres aléas de collecte (certains étudiant·es ont par exemple utilisé par erreur la version instrumentée de `Thonny` alors que leur groupe utilisait `doctest`). Les premières analyses exploratoires que nous avons menées ont donné des résultats encore très partiels. 162 étudiants ont utilisé `L1Test` au sens où au moins une de leur session contient des actions de lancement des tests avec le bouton dédié (verbe `Run.test`). La figure 4 montre l'usage du lancement des tests au cours des 11 semaines du semestre [6]. On constate une utilisation non négligeable de `L1Test` durant tout le semestre. La décroissance observée en seconde partie de semestre suit la décroissance générale des actions dans `Thonny`, corrélée au décrochage progressif d'une partie de la promotion.

La figure 5 présente une visualisation qui permet d'étudier le comportement des étudiant·e·s durant une session. Chaque trait représente une action de l'étudiant·e : celles dont le résultat a provoqué une erreur sont au dessous de l'axe (incluant les tests en échec) et les autres sont au dessus. Sur l'exemple présenté, l'étudiant·e lance beaucoup les tests durant la session avec de nombreux tests en échec en début de session.

5 Perspectives

Cette première phase d'expérimentation nous a permis de tester les greffons développés sur un grand volume d'étudiant·es et de corriger des erreurs sur `L1Test` et `L1Log`. Elle nous a également permis d'avoir une première étude des données produites ainsi que des visualisations. L'outil `L1test` a comme ambition de permettre l'écriture de tests unitaires basiques de manière aussi simple qu'avec `doctest`, avec une sémantique et une interface graphique plus intuitives. Nous souhaitons évaluer les effets de la sensibilisation au test avec `L1Test` sur l'apprentissage de la programmation (notamment la correction du code) et définir des indicateurs en ce sens, dans le cadre de notre expérimentation.

6. `L1Test` n'a pas été utilisé pendant la semaine 37, première semaine des enseignements. Les semaines 44 et 45 correspondent à une vacance des enseignements.

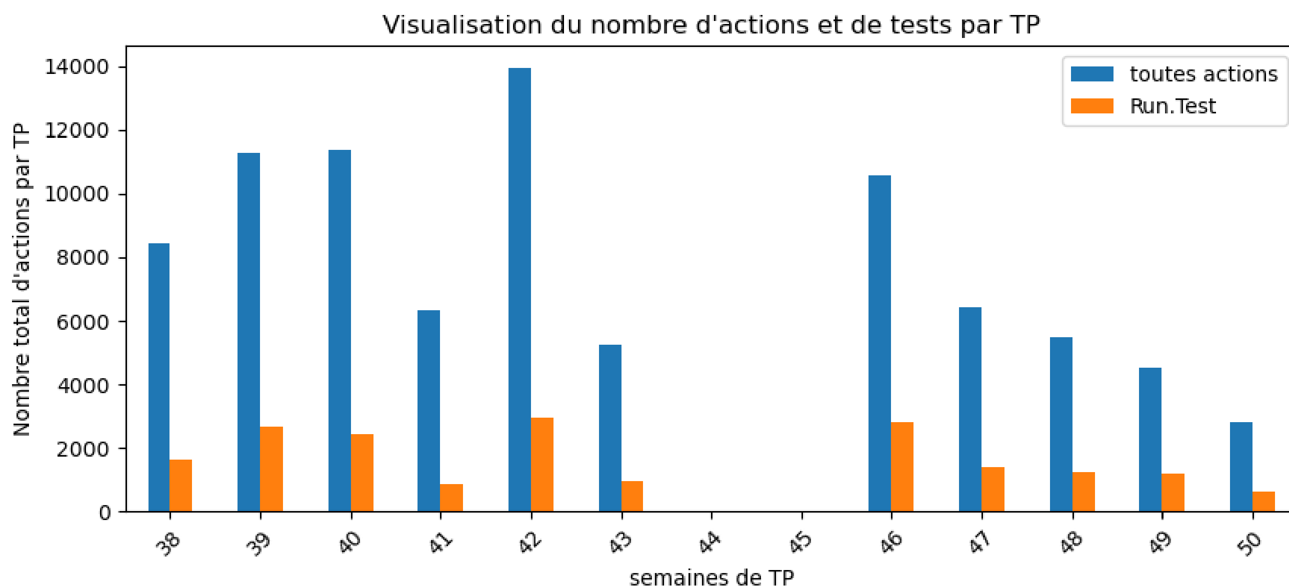

FIGURE 4 – visualisation globale de l'usage des tests.

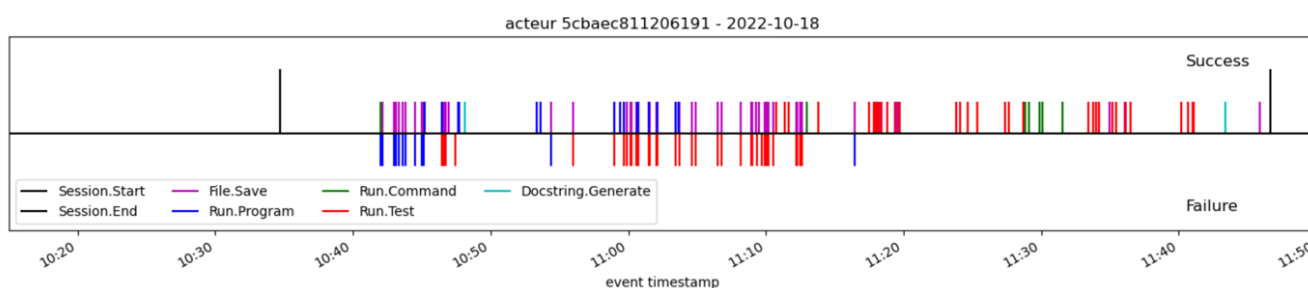

FIGURE 5 – Exemple de frises temporelles.

Nous aimerions comprendre si et à quel moment l'introduction du test répond à un besoin des étudiant·es ou leur apporte une plus-value importante à leur yeux. Parmi les critères à doser, on pourra citer la méthodologie (nous avons choisi de faire écrire les tests avant le code) et le fait de demander aux étudiant·es d'écrire leur propres tests (vs les fournir tout ou partie).

Écrire les tests avant le code permet de porter son attention sur l'utilisation qui sera faite de la fonction et de se représenter son comportement. En particulier le fonctionnement de `L1Test` peut permettre d'aborder par les tests la traditionnelle confusion entre retour et effet de bord. De plus une grosse fonction est difficilement testable : l'utilisation des tests peut permettre d'aborder des questions de conception liées au découpage en fonctionnalités d'un programme. La non-testabilité des procédures d'affichage devrait aussi inciter les étudiant·es à séparer celles-ci des fonctionnalités de calcul.

Exécuter les tests régulièrement fournit un retour formalisé et automatisé sur la présence d'erreurs dans le code. Ce retour semble ressenti comme motivant et autonomisant par les étudiant·es, surtout si la suite de tests est fournie et constitue un indicateur raisonnable de correction du code. Les tests permettent d'ailleurs de rendre plus concrète cette notion de correction, charge aux enseignante·s d'indiquer clairement que les tests prouvent la présence d'erreurs dans le code mais jamais leur

absence, ni le respect des conventions de codage.

Par ailleurs, le Test Driven Learning (Janzen and Saiedian, 2006) est une approche pédagogique entièrement basée sur les tests, beaucoup plus poussée que la nôtre, qui encourage les étudiant·es à écrire leurs propres tests notamment pour passer d'une technique d'essai-erreurs à une réflexion sur le comportement du code (Edwards, 2004). Nous aimerions évaluer si l'écriture des tests engage effectivement les étudiant·es dans une telle démarche de réflexion, et si fournir une suite de tests incite au contraire à faire juste "passer les tests" de manière erratique. Enfin, nous aimerions évaluer via un suivi de cohorte si la sensibilisation au test aura un impact sur l'apprentissage du test unitaire objet classique avec JUnit en 2ème année.

Sur le plan de l'analyse des traces, nous aimerions maintenant commencer à mettre en place un tableau de bord rassemblant différents indicateurs et visualisations afin de faciliter l'analyse du comportement des étudiant·es aussi bien sur le long terme que dans le cadre des TP, dans l'optique de fournir aux étudiants des rétroactions adaptées.

Références

Annamaa, A. (2015). Thonny, : A python ide for learning programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, page 343. Association for Computing Machinery.

Edwards, S. H. (2004). 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*, pages 26–30.

Garousi, V., Rainer, A., Lauvås Jr, P., and Arcuri, A. (2020). Software-testing education : A systematic literature mapping. *Journal of Systems and Software*, 165 :110570.

Janzen, D. S. and Saiedian, H. (2006). Test-driven learning : intrinsic integration of testing into the cs/se curriculum. *Acm Sigcse Bulletin*, 38(1) :254–258.

Lappalainen, V., Itkonen, J., Isomöttönen, V., and Kollanus, S. (2010). 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*, pages 63–67.

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J., and Szabo, C. (2018). Introductory programming : A systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion, page 55–106. Association for Computing Machinery.

Price, T. W., Hovemeyer, D., Rivers, K., Gao, G., Bart, A. C., Kazerouni, A. M., Becker, B. A., Petersen, A., Gusukuma, L., Edwards, S. H., and Babcock, D. (2020). Progsnap2 : A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 356-–362. ACM.

Scatalon, L. P., Carver, J. C., Garcia, R. E., and Barbosa, E. F. (2019). Software testing in introductory programming courses : A systematic mapping study. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 421–427.