Rapport de projet GymInf

?? UniFr , HEP ?

Conception et implémentation d'un exerciseur automatique pour l'enseignement de la programmation

Julien Mateesco

8 septembre 2025

Résumé

Ce rapport présentera un outil-auteur et sa réalisation, faite dans le cadre du programme de formation GymInf, répondant à la question "comment soutenir l'apprentissage par la méthode PRIMM de la programmation Python chez les débutants, avec visualisation logigrammme". L'outil existe, visible sous https://github.com/edu-mateescoj/gyminf et propose principalement quatre fonctionnalités :

- ✓ un générateur automatique (infini) de code Python valide (syntaxiquement correct, qui s'exécute dans l'environnement Pyodide et qui termine) et de niveau pédagogique adapté aux besoins définis par l'utilisateur, exécuté dans le navigateur *full front-end*.
- ✓ un traducteur de code Python en logigramme, exécuté et affiché dans le navigateur *full front-end*, et modifiable par l'utilisateur.
- ✓ le questionnement des élèves sur les valeurs des variables à l'issue du code proposé, avec la rétroaction réussite/échec correspondante, ainsi qu'une simulation de console d'exécution, dans le navigateur *full front-end*.
- ✓ la constitution sur serveur d'une base de données enregistrant les codes Python générés et les interactions élèves correspondantes, laissant la possibilité de suivre individuellement l'évolution des élèves dans leurs interactions avec les codes proposés.

Nous présenterons d'abord les contextes qui ont présidé à la réalisation de cet outil prévu pour être utilisé en classe à très brève échéance, ainsi que son architecture logique. Nous alternerons donc entre les côtés technique et fonctionnel. Le travail de développement informatique prenant une coloration didactique, nous donnerons une description logicielle de l'ensemble, ne fournissant des détails d'implémentation que sur les contributions essentielles que sont :

1. le générateur Javascript de codes Python tournant dans le navigateur, aléatoires mais contrôlables et de niveau pédagogique
2. le traducteur Javascript de code Python en graphique logigramme rendu dans le navigateur par Mermaid
3. le serveur Flask (Python) enregistrant les interactions élèves face aux codes proposés pour construire une base de données didactiques.

Le présent rapport n'a pas vocation à être un mode d'emploi ou une documentation technique exhaustive, mais une explication des fonctionnalités implémentées et des choix qui ont été faits.

Table des matières

Table des figures

Liste des tableaux

1 Introduction

1.1 Préambule philosophique

Notre projet a été construit à une époque marquée simultanément par trois phénomènes apparemment contradictoires :

- la massification de l'apprentissage de la programmation, aujourd'hui proposée à **tous les élèves de filière secondaire académique** (gymnasiale, générale)
- une **disparition annoncée de la pratique de l'écriture humaine de code** informatique, ce qui questionne la possible disparition de l'écriture de code comme compétence socialement valorisée
- l'ubiquité du **numérique dans la vie quotidienne**, en particulier pour les actuels élèves du secondaire, nés après l'avènement du smartphone et du Web 2.0

Il nous semble donc d'autant plus urgent de proposer des *outils conviviaux* susceptibles de redonner aux futurs citoyens une capacité d'agir dans notre monde numérique. Nous empruntons ici l'expression d'Ivan Illich (Illich, 1973) pour qui des "outils conviviaux" sont le contraire de l'outil "dominant" (exemple aujourd'hui : l'algorithme qui décide pour nous, à notre insu). Sont "conviviales" les techniques autolimitées, compréhensibles et modifiables par leurs usagers, que chacun peut utiliser, réparer et adapter sans dépendre d'experts ni d'institutions, afin d'accroître l'autonomie et la coopération plutôt que la dépendance aux systèmes. À l'époque contemporaine, Shoshana Zuboff décrit notre "capitalisme de surveillance" - aux antipodes de la convivialité d'Illich - où l'outil capture les comportements plutôt que d'étendre la capacité d'agir. Les deux autres concepts qui nous semblent essentiels pour essayer de penser la valeur sociale d'un nouveau service numérique sont : le "Panoptique" (Foucault, 1975) de Michel Foucault (métaphore d'un pouvoir disciplinaire fondé sur la visibilité asymétrique) et la "Société de contrôle" (Deleuze, 1986) de Gilles Deleuze (très proche des logiques de plateformes et de scoring algorithmique d'aujourd'hui, des décennies après sa mort). L'outil "convivial" est ainsi un critère normatif contre ces logiques panoptiques et "contrôlantes" : gouvernance par les usagers, transparence et autolimitation vs. extraction des données, opacité et dépendance. Je milite pour une évaluation critique des services numériques que nous faisons utiliser aux élèves, avec comme objectif de favoriser l'appropriation collective (Illich) et d'identifier là où elles se trouvent les logiques de surveillance et de contrôle diffus (Foucault/Deleuze).

1.1.1 Position de principe sur l'IA générative comme outil dans les mains des élèves

Il apparaît naturel pour beaucoup de gens aujourd'hui de déléguer aux LLM des tâches d'enseignants tant les modèles et leurs interfaces sont devenus qualitativement et quantitativement performants. On peut en effet noter à la fois la qualité des réponses proposées, instructives pour nos élèves, et d'un point de vue quantitatif l'infrastructure qui nous sert ces modèles autorise un passage à l'échelle permettant une productivité face aux élèves littéralement inhumaine, leur ubiquité soulignant la rareté et la singularité de l'interaction pédagogique enseignant - apprenant. De façon générale, plusieurs raisons de refuser d'inciter les élèves à utiliser massivement l'IA générative peuvent être mises en avant :

1. l'impact écologique (climat et environnement : énergie, eau et terres rares, etc.),
2. l'inadéquation par rapport à une démarche scientifique "explicative et contrôlée",
3. les dérives sociales des géants du numérique (non respect des normes de traitement des humains engagés dans l'entraînement des modèles),
4. la cannibalisation des communs numériques pour l'entraînement des LLM privés (appropriation de biens publics pour la constitution de profits privés, non respect des bonnes pratiques de crawlers web pour alimenter les modèles),

Maintenant dans une perspective d'*empowerment*, telle que les fournisseurs d'IA nous le promettent (Business Insider, 2024), une précision nous semble importante : déléguer l'interaction maître-élève à une machine et déléguer l'observation et la "compréhension" de l'erreur élève à une machine prive l'enseignant de la possibilité de développer son expertise didactique Plus spécifiquement, pour notre cas d'usage qui est la génération aléatoire d'exercices par les élèves, adapté aux besoins pédagogiques, une solution acceptable par principe aurait été la distillation d'un gros modèle produit dans des conditions *fair-trade*, assez frugale pour tourner sur un serveur interne à l'école, mais c'est un autre sujet qui ne rentre pas dans le cadre de ce travail.
Mettre dans les mains des élèves un outil basé sur l'API d'un modèle commercial, performant mais en voie

de monétisation, reste inconcevable tant cela enfreindrait mes principes que je viens de mentionner ci-dessus, en exposant les élèves au risque d'être instrumentalisés dans le processus de monétisation des IA génératives.

Les ambitions qui ont guidé mon projet se situent :

- **Chez les élèves** : en leur proposant une traduction de la syntaxe textuelle Python en une syntaxe graphique logigramme, en les exerçant à la traduction d'une syntaxe à l'autre comme un entraînement ; en les invitant à se faire une image mentale de la sémantique associée pour tracer la valeur des variables. Ces savoirs syntaxiques et conceptuels sont les premières étapes à mobiliser pour la résolution de nouveaux problèmes. Nous formulons donc l'espoir que notre outil contribuera au développement de leurs compétences stratégiques.
- **Pour la relation enseignants - apprenants**, en mettant à libre disposition un outil-auteur adaptable et capable de proposer des codes prédéfinis par l'enseignant doublé d'un système de journalisation des interactions élèves qui a le potentiel de permettre à l'enseignant un suivi très fin des élèves pour pouvoir apporter une rétroaction différenciée jusqu'au niveau individuel.
- **Chez les enseignants** : en proposant à la communauté enseignante un outil évolutif, nous cherchons à inviter les collègues à la collaboration pour améliorer le projet, et à l'échange sur leurs pratiques. Nous nous situons dans l'esprit de la constitution d'une "communauté de pratique" (Wenger, 1998)
- **Chez les enseignants** : en refusant de déléguer la rétroaction élève à un LLM d'IA générative et en donnant les moyens aux enseignants d'évaluer immédiatement et à long terme l'efficacité de leur action enseignante, notre espoir est de donner les moyens de développer une expertise didactique à des experts informatiques qui s'adressent maintenant à des apprenants culturellement très éloignés.

Maintenant que nous avons défini le cadre social, voyons tout d'abord d'où vient le projet, en alternant les points de vue : professionnel, personnel, pédagogique et pratique.

1.2 Éléments de contexte

Notre démarche tire son essence dans un but : aider les enseignants chargés de faire découvrir la programmation textuelle à des élèves débutants et culturellement aux antipodes. Situons donc ici le public cible et l'environnement concerné afin de justifier les objectifs poursuivis dans la création de notre outil.

1.2.1 Contexte institutionnel

Institutionnel et pédagogique : École secondaire genevoise ; enseignement de l'informatique comme discipline obligatoire pour tous les élèves de 1ère et 2ème année, mais avec très peu d'heures hebdomadaires pour chaque élève (respectivement 2 périodes et 1 seule) et dont la note ne sera pas reprise sur le diplôme de maturité : massification des effectifs d'élèves pour l'enseignant, difficulté voire opposition de certains élèves face à la programmation.

Professionnel et personnel : Ce projet rentre dans le cadre de la validation de ma formation GymInf et doit se terminer durant le semestre académique courant.

Professionnel et institutionnel :

- Constat de l'absence d'une culture de groupe de discipline, besoin de créer une "communauté de pratique" ;
- Très grand nombre d'élèves pour chaque enseignant. Et donc besoin de rationaliser au maximum la pratique en créant des outils automatiques, besoin d'autonomiser les élèves dans leurs apprentissages ;
- Indisponibilité (technique ou budgétaire) ou inadéquation des outils existants (curriculum différent, objectifs "venus d'en haut" et nécessité de se conformer à une épreuve commune)

Le calendrier scolaire est rythmé par le curriculum et l'évaluation certificative qui y est attachée, aussi le développement de l'outil et le présent rapport se sont trouvé désynchronisés, ce qui explique qu'il n'y a pas eu d'analyse de l'efficacité d'un dispositif basé sur l'outil.

1.2.2 Contexte didactique : surtout des hypothèses appuyées par la littérature

1. Hypothèse fondatrice, appuyée par la littérature : la pertinence de la méthodologie PRIMM (Sentance & Waite, 2017) dans le cadre de l'enseignement de la programmation en particulier en école secondaire avec des classes hétérogènes (Sentance et al., 2019), que nous étendons par hypothèse aux

syntaxes graphiques de type logigramme. Les mêmes auteurs relèvent l'attrait de cette approche pour les enseignants, qui a priori justifie qu'on cherche à les aider dans cette approche si elle a du succès chez eux !

2. Hypothèse forte appuyée par la littérature : les intérêts multiples d'un générateur d'exercices aléatoires (Messer et al., 2023), pour autant que l'aléa réponde à des critères pédagogiques fixés et contrôlables par l'enseignant et/ou par l'apprenant. Citons notamment comme intérêt la grande variété des questions, qui évite la mémorisation de la solution sans investissement personnel, évitant ainsi le *plagiat de solutions* ; le contrôle de la difficulté peut permettre une différenciation (par adaptation au niveau réel ou supposé de l'élève).
3. Hypothèse : intérêt de la représentation logigramme (*flowchart*) comme moyen d'apprentissage (Zimmermann et al., 2024), et en particulier de la programmation (Crews & Ziegler, 1998) - en plus d'être un objectif d'apprentissage qui se justifie eu égard au contexte institutionnel.
4. Hypothèse : intérêt de proposer des exercices de lecture de programmes textuels et graphiques avant ceux d'écriture, en référence aux taxonomies d'Anderson & Krathwohl (Anderson & Krathwohl, 2001) et plus spécifiquement (Fuller et al., 2007) pour l'enseignement de la programmation. Ces taxonomies ont en commun de placer "comprendre" à un niveau cognitif plus bas que "appliquer""comprendre" est censé être de niveau cognitif plus bas que "appliquer" ou encore "créer". ou encore "créer".
5. Hypothèse : l'intérêt d'une évaluation formative immédiate pour tester la compréhension et l'intérêt de la possibilité laissée à l'apprenant de modifier les codes proposés - textuel et logigramme - citons notamment (Hattie & Timperley, 2007) et (Shute, 2008).

1.3 Objectifs fonctionnels

Nos objectifs fonctionnels apparaissent comme des conclusions de la section précédente : nous voulons un outil qui aide les enseignants à proposer des exercices de niveau débutant en accord avec les étapes "Predict Run Investigate Modify" de la célèbre approche PRIMM.

1.3.1 Besoin d'un générateur de code et de questions pertinentes

Au niveau débutant, les classes sont hétérogènes. Nous visons donc des exercices de lecture de code fondés sur la taxonomie des tâches au premier échelon, de difficulté ajustable pour différencier au sein du groupe et accompagner sa progression. L'outil doit aussi décharger l'enseignant avec : une génération automatique,un déploiement simple, un contrôle fin des paramètres.
Du côté de l'élève (dimension "Predict"), il faut recevoir programmatiquement un script Python, l'afficher dans une fenêtre avec coloration syntaxique et numérotation des lignes, afin de soutenir la lecture analytique. L'aléa doit être contrôlé pour garantir une plus-value pédagogique : on choisit explicitement les constructions à inclure dans le script, on règle des variables didactiques (nombre de variables, domaine de valeurs, longueur totale), et on fixe les types à utiliser. Nous nous limitons volontairement aux types de base utiles au début de l'apprentissage ; nous renonçons à la simulation de programmes réalistes" au profit de fragments signifiants. Les noms de variables sont cohérents avec leurs types pour donner une apparence plausible et guider l'inférence de l'élève.

Techniquement, l'interface collecte les choix utilisateurs et les transmet au générateur qui produit un script conforme : c'est le point d'entrée de tout le flux.

1.3.2 Besoin d'une sorte d'éditeur Python en ligne

Pour solliciter les dimensions "Run" et "Investigate", nous utilisons un interprète Python centré débutants (sans viser l'exhaustivité syntaxique) et un éditeur qui teste l'élève en cohérence avec les objectifs déclarés par l'enseignant. Les questions se calent automatiquement sur le code proposé et une rétroaction immédiate est fournie. Cela suppose des échanges DOM ↔ exécution : renvoi vers la page des valeurs attendues, lecture des réponses élèves, comparaison, puis affichage d'un feedback adapté.
Pour engager à la fois néophytes et aficionados, l'éditeur offre un rendu familier type appli web ou IDE avec des aides activables (aplatissement de blocs, éventuellement auto-complétion) utiles dans un cadre pédagogique.
Enfin, pour "Investigate" et "Modify", l'élève peut sauvegarder (`Download`) son script et revenir au script initial (`Reload`) afin d'annuler ses modifications depuis la génération.

1.3.3 Un rendu logigramme

Le soutien visuel paraît approprié pour aider des élèves dans leur découverte de leur premier langage de programmation textuelle. Le logigramme est un objectif d'apprentissage curriculaire en soi, au même titre que des pseudo-codes. De façon à proposer un soutien visuel le logigramme parait donc un choix naturel car il matérialise le flux d'exécution logique du programme. Un outil proposant des bouts de code Python et leur représentation logigramme a le défaut de mettre au même niveau deux langages qui n'ont pas le même statut : un seul est un langage de programmation ! mais nous faisons l'hypothèse que c'est le prix à payer pour pouvoir entraîner de façon systématique les élèves à cet exercice particulier qui est de passer d'une syntaxe à l'autre. Pour créer le rendu logigramme pour des codes valides, deux options étaient a priori envisageables :

- **Soit** à la génération du script, un logigramme statique (plus facile car le rendu logigramme est encapsulé avec la génération du code)
- **Soit** dynamique, modifiable en continu avec le code Python modifiable (ce qui alourdit l'UI qui devient moins lisible).

La première option aurait été en contradiction avec l'objectif d'investigation et de modification active ! aussi c'est la seconde option qui a été choisie. Le résultat est un couplage direct entre code affiché et logigramme—le support visuel se met à jour avec le script modifié, ce qui renforce l'activité de lecture, de prédiction et de vérification.

1.4 Applications similaires et sources d'inspiration

Essayons de classifier les solutions proposées parmi les plus connues, en guise d'état de l'art sur les exerciseurs accessibles facilement aux débutants et/ou à leurs enseignants pour organiser le travail des élèves ou animer la classe. On résumera la ou les différences essentielles rencontrées à chaque fois par rapport à notre outil.

Des outils auteurs

Les outils auteurs sont des logiciels permettant à des non-développeurs (souvent des enseignants ou des concepteurs pédagogiques) de créer des contenus d'apprentissage interactifs sans écrire de code. Il existe des fournisseurs privés (exemple : Didask Didask, 2025 et des solutions open-source. Les outils auteurs proposent généralement des modèles d'activités (quiz, glisser-déposer, etc.) à assembler, aujourd'hui les acteurs privés proposent de déléguer la création des contenus par IA générative.

H5P https://h5p.org/ (consulté le 5 août 2025)
H5P intègre des frameworks open-source très populaires qui permettent de créer une grande variété de contenus interactifs directement intégrables dans des plateformes existantes. L'enseignant choisit un type d'activité (ex : "Fill in the Blanks") et remplit les champs de formulaire ad hoc pour créer l'exercice.

Différenciation : Notre application se distingue fondamentalement par sa **spécialisation et sa capacité de génération et de traduction en logigramme**. Alors qu'un outil auteur comme H5P demande à l'enseignant de fournir lui-même le contenu (le code Python, les questions), *notre outil génère ce contenu de manière procédurale. C'est un outil auteur spécifique au domaine* de la syntaxe Python, qui ne fournit pas des modèles d'exercices vides mais produit l'exercice (code + questions) lui-même, en fonction de contraintes pédagogiques.

Des cours et tutoriels en ligne, francophones

Cette catégorie regroupe les innombrables sites web qui proposent des parcours d'apprentissage structurés, souvent sous forme de textes, d'exemples et d'exercices intégrés. Leur but est de guider l'apprenant de façon réfléchie à travers un curriculum défini y compris en utilisant des ressources vidéos.

Des exemples 100% gratuit (consultés le 24 août 2025)

France-ioi https://www.france-ioi.org/algo/chapters.php

code.org https://code.org/fr/teachers

Des exemples avec parcours certifiant (consultés le 24 août 2025)

OpenClassroom https://openclassrooms.com/fr/courses/7168871-apprenez-les-bases-du-langage-python

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

Des IDE pédagogiques

Les Environnements de Développement Intégrés (IDE) pédagogiques sont des versions simplifiées des outils professionnels, conçues pour les débutants. Ils mettent l'accent sur la clarté, la visualisation de l'exécution et des messages d'erreur plus compréhensibles.

Thonny https://thonny.org/ (consulté le 5 août 2025)
Thonny est un IDE Python pour débutants très apprécié. Il inclut un débogueur visuel simple qui permet de suivre l'exécution pas à pas, d'inspecter les variables et de comprendre le fonctionnement de la pile d'appels. Il doit être installé sur l'ordinateur de l'utilisateur.

Spyder https://docs.spyder-ide.org/current/index.html (consulté le 24 août 2024)
Spyder se présente comme un "IDE scientifique pour Python", utilisé à notre connaissance en fin de cursus scolaire durant les cours d'application des maths ou encore pour des TP de physique. Le site officiel en fait la promotion ainsi "Spyder is a powerful scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. It features a unique combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package."

Différenciation : La finalité est différente. Thonny est un **environnement pour écrire et déboguer du code**. Notre application est un **générateur d'exercices pour lire et analyser du code valide**. Bien que notre outil intègre un éditeur, son but n'est pas la création de projets mais l'analyse et la modification de fragments de code (pour les phases "Run", "Investigate" et "Modify" du modèle PRIMM). De plus, notre solution de génération aléatoire & traduction en logigramme est en ligne, elle ne nécessite donc aucune installation et est disponible pour les élèves sur smartphone, contrairement à un IDE de bureau comme Thonny qui est disponible pour Windows, Mac et Linux.

Des défis pédagogiques

Ces sites proposent des collections de problèmes ou "défis" de programmation que l'utilisateur doit résoudre en écrivant du code. L'accent est mis sur la résolution de problèmes et la validation algorithmique. Nous différencions ces sites des autres solutions par leur absence assumée de gestion de classe, d'adéquation à un quelconque curriculum officiel ou affiliation scolaire, et par leur centration sur la résolution de problèmes (présentés sous forme de "challenges" ludiques) par écriture de code.

Codewars https://www.codewars.com/ (consulté le 5 août 2025)
Codewars propose une approche ludique où les développeurs améliorent leurs compétences en résolvant des défis de programmation appelés "kata". L'utilisateur doit écrire une fonction qui passe une série de tests cachés.

Projet Euler https://projecteuler.net/ (consulté le 5 août 2025)
Le Projet Euler est une série de problèmes mathématiques complexes qui nécessitent des solutions de programmation efficaces. L'objectif est de trouver la bonne réponse numérique au problème, la qualité du code n'étant pas évaluée.

Exercism https://exercism.org/ (consulté le 5 août 2025)
Exercism se distingue en proposant des parcours d'apprentissage pour de nombreux langages. Après avoir résolu un problème, l'élève peut soumettre sa solution pour recevoir des commentaires et des conseils de la part de mentors bénévoles, mettant l'accent sur la qualité et l'*idiomaticité* du code.

Différenciation : La démarche pédagogique est inversée. Sur ces plateformes, le **problème est fourni** et l'élève doit **produire le code**. Dans notre application, le **code est fourni** et l'élève doit **analyser son comportement** pour prédire la valeur finale des variables. Notre objectif n'est pas de tester la capacité à concevoir un algorithme, mais de renforcer la compréhension de la sémantique des constructions du langage. De plus, le contenu de notre outil est généré dynamiquement selon les besoins de l'enseignant, alors que les défis de ces plateformes sont des problèmes fixes (éventuellement créés par la communauté).

Des plateformes pédagogiques

Ces plateformes sont des écosystèmes complets qui intègrent un IDE en ligne, la gestion de cours, des devoirs, et souvent une infrastructure serveur pour exécuter le code et procéder à une évaluation automatique. Certaines plateformes intègrent ainsi tout ou partie des autres formes d'outils déjà mentionnées ci-dessus, par exemple avec simulation de console IPython, cours en ligne interactifs avec Jupyter Notebooks, défis progressifs, etc.

Replit https://replit.com/ (consulté le 5 août 2025)
Replit est une plateforme en ligne extrêmement puissante qui fournit un IDE collaboratif dans le cloud pour des dizaines de langages. Elle permet aux enseignants de créer des classes, de distribuer des devoirs avec des tests unitaires pour l'auto-correction, et aux élèves de développer et d'héberger des projets complets.

Codex (La Forge Numérique) https://codex.forge.apps.education.fr/ (consulté le 5 août 2025)
Codex est une plateforme d'exercices et d'évaluation développée pour les écoles du secondaire (lycée français). Elle permet aux enseignants de créer des épreuves de programmation sécurisées, où les élèves soumettent leur code qui est ensuite évalué par des tests automatiques dans un environnement contrôlé.

AlgoPython https://algopython.fr/ (consulté le 5 août 2025)
AlgoPython est un site de référence dans le monde francophone pour l'apprentissage de Python. Il propose un cours très structuré, allant des bases de la syntaxe à des notions plus avancées, avec de nombreux exemples et des exercices à réaliser.

moodle https://moodle.com/ (consulté le 5 août 2025)
moodle va plus loin que tout ce qui a été jusqu'ici, et coche toutes les cases, en effet de nombreux outils ont été développés permettant par exemple de tester du code écrit par les élèves. moodle intégre en particulier la gestion des élèves sur tout le parcours d'apprentissage défini par l'enseignant, c'est donc à la fois un outil-auteur et un "learning management system".

Différenciation : La principale différence réside dans la nature du contenu, la possibilité offerte à l'élève d'expérimenter en dehors de l'exercice strict, et la rétroaction immédiate sans blocage de l'élève dans son avancement. Ces plateformes proposent généralement un **contenu fixe et éditorialisé**, conçu pour être suivi de manière linéaire. Ainsi avec AlgoPython l'élève est stoppé dans sa progression en cas d'échec (sauf intervention de l'enseignant qui peut débloquer le niveau). Notre outil, à l'inverse, est un **générateur de contenu à la demande et non-linéaire**, offrant la possibilité supplémentaire de modifier le contenu et de l'exporter. Notre outil ne propose pas plus de cours que le seul nom des éléments syntaxique, mais est une source inépuisable d'exemples uniques sur des points de syntaxe précis, que l'enseignant peut utiliser pour illustrer une notion spécifique ou créer un exercice ponctuel. Notre application est volontairement **plus légère, plus ciblée sur l'expérimentation autonome par l'élève et axée sur la formation plutôt que l'évaluation**. Contrairement à ces plateformes (AlgoPython, Replit ou Codex) qui sont des solutions lourdes, forcément basées sur une infrastructure client-serveur et requérant des comptes, notre outil de génération d'exercices peut fonctionner comme une application web statique, sans serveur d'exécution grâce à Pyodide, donc anonyme, et conçue pour une tâche unique : générer des exercices ponctuels. Même avec la partie serveur notre outil n'a pas vocation à gérer un cours ou à organiser des examens, mais à fournir une ressource formative, que l'enseignant peut intégrer dans son propre cheminement.

Des sources d'inspiration

A citer également :

- La source d'inspiration pour l'utilisation du *Control Flow Graph* de Python fourni par le module `ast` : https://www.fuzzingbook.org/html/GrammarFuzzer.html
- Un papier universitaire présentant une initiative de génération automatique d'exercices dans le cadre de l'apprentissage de la programmation : https://arxiv.org/abs/2205.11304
- Un générateur de scripts Python, totalement aléatoire et intégrant un très grand nombre d'éléments syntaxiques : https://github.com/radomirbosak/random-ast
- Le célèbre visualiseur d'exécution de bouts de codes : https://pythontutor.com/

- L'initiative emblématique Blackbox de l'enregistrement de (millions de) code snippets d'étudiants utilisant l'IDE BlueJ pour l'apprentissage de Java, pour identifier des patterns d'erreurs (Brown et al., 2018)
- Le jeu Silent Teacher (Toxicode, 2025) qui écarte l'enseignant de l'apprentissage en proposant des mini questions avec rétroaction immédiate, sans aucune référence à la théorie ni contexte, isolant l'apprenant dans une sorte de behaviorisme radical

En conclusion, la multitude des ressources pédagogiques disponibles est une source inépuisable d'inspiration, et de possible évolution, pour une prochaine version de l'outil proposant une génération sémantique (*TBC...*)

1.4.1 Différenciation et apport de notre solution

L'objectif poursuivi a été de proposer aux enseignants d'investir en un seul outil les dimensions Predict, Run, Investigate et Modify de l'approche PRIMM d'introduction à la programmation.

L'application présentée dans ce travail de mémoire diffère des applications existantes par son panachage unique (à notre connaissance) de plusieurs aspects :

- le travail demandé aux élèves, **de lecture et traçage systématique de code, avec traduction de la syntaxe en flux sémantique**, alors que les outils connus actuellement demandent généralement d'écrire du code, à un stade trop précoce pour les élèves qui en sont encore à "comprendre" ce qu'est un bout de code écrit en Python
- le caractère **automatique**, aléatoire et infini, contrôlable mais non éditorialisé de la génération d'exercices, simple à déployer,accessible en classe et à domicile
- la possibilité d'alterner entre les syntaxes **textuelle et graphique : Python et logigramme** d'un même programme, visualisée côte-à-côte dans l'interface, avec juste au-dessus à l'écran un rappel des noms des éléments syntaxiques tels qu'ils apparaissent dans la théorie scolaire ('Vars' pour le rappel du concept de variable avec affectation de valeur et opérations 'Op' selon leur type, 'Ctrl' pour les structures de contrôle conditionnelles, 'Loop' pour les boucles et enfin 'Func' pour les fonctions Python).
- avec le serveur, la possibilité de **tracer le parcours des élèves avec leurs réussites/échecs individuels avec une granularité extrêmement fine** (au niveau de la valeur de chaque variable), et donc la possibilité de construire des indicateurs décrivant leur investissement et leur ténacité (par exemple : la durée passée entre la génération de l'exercice et les propositions de réponses croisée avec le taux de succès, ou encore le taux de succès selon le type de variables et le niveau de difficulté enregistré par l'application, etc.)
- la rétroaction immédiate donnée à l'élève sur la réussite ou l'échec **sans bloquer les élèves face à un échec** tout en laissant avec l'apport du serveur la possibilité pour l'enseignant d'être informé de la nature exacte des échecs.

2 L'outil et ses fonctionnalités

Dans cette partie : l'interface, l'architecture interne et les choix technologiques.

2.1 Fonctionnalités pour les utilisateurs

Pour rappel, deux types d'utilisateurs sont envisagés : enseignants et élèves, qui n'auront évidemment pas le même usage (au moment de la rédaction l'outil n'a pas encore été utilisé en classe) mais enseignants et élèves partagent la même interface et les mêmes fonctionnalités.

1. Enseignants **ET** élèves peuvent :
   - utiliser l'outil sur les machines fournies à l'école pour générer des *code snippets*, et regénérer à l'infini
   - sélectionner les éléments du langage à mobiliser pour chaque code qui sera généré
   - afficher le code sous forme Python et/ou *flowchart*, au choix
   - afficher les valeurs des variables en fin de script
   - sauvegarder localement (pour plus tard) codes et flowcharts jugés intéressants
   - modifier le code Python généré automatiquement pour l'afficher en flowchart, pour l'évaluer, pour l'exporter, et revenir au code généré initialement
2. "En tant qu'enseignant, je peux ..." en plus des éléments ci-dessus :
   - utiliser les codes générés pour préparer des questions pour animer un cours dialogué, pour interroger les élèves en classe, ou pour impressions (asynchrone)
   - types d'exercices envisagés : lecture de code Python ; lecture de flowcharts ; traduction d'un langage à l'autre ; code à modifier pour obtenir un certain résultat attendu (prédéfini par l'enseignant)
   - laisser les élèves être autonomes dans leur progression, rendre les élèves conscients que c'est l'ordinateur qui "donne la réponse"
   - rendre les élèves conscients des contenus (les éléments à cocher/décocher) et du caractère *presque scientifique* de la démarche
3. "En tant qu'élève, je peux ..." en plus des éléments ci-dessus :
   - m'exercer à la lecture de code Python et à la lecture de flowchart, par l'évaluation de variables en fin de script, de façon autonome avec une rétroaction (juste ou faux)
   - investiguer des modification du code Python et voir leur effet sur le flowchart et sur les valeurs des variables en sortie du script
   - utiliser la plateforme à l'école, sur smartphone, sur tablette et autres écrans personnels (*fully responsive design*)

2.2 Côté client : l'interface et l'architecture logicielle

L'UI : la page *Front end*

A l'ouverture voici l'interface :

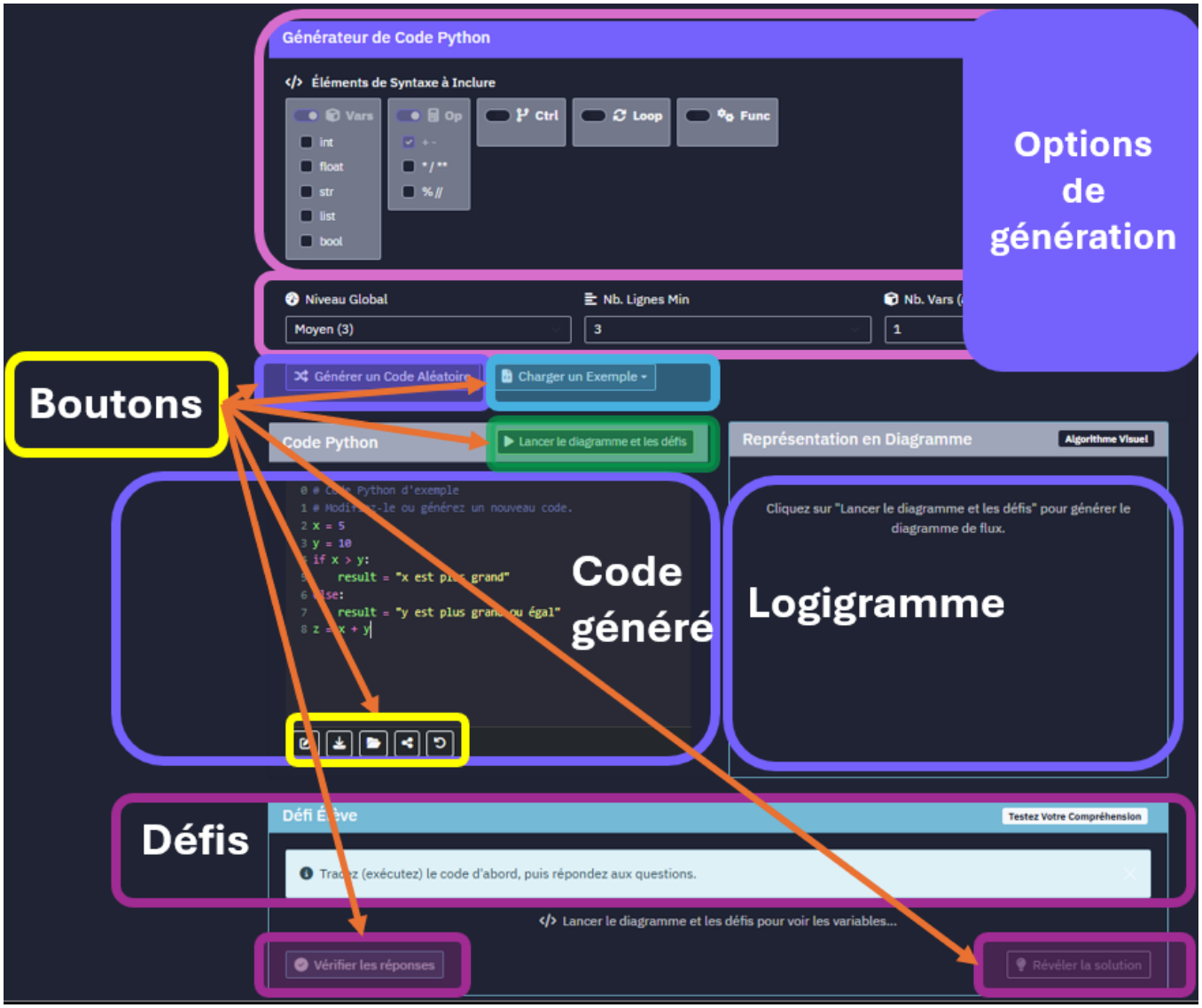

FIGURE 1 : Présentation générale de l'application, vierge

On reconnaît :

- Les options, à choisir dans l'interface
  1. qualitatives (les éléments syntaxiques)
  2. quantitatives (nombre de lignes, de variables et une mesure indicative de difficulté)
- Les affichages côte-à-côte du code et du logigramme
- En bas la section "Défis" correspondant à l'exercice sur le logigramme et au code
- Différents types de boutons

L'interface : les sorties code et diagramme

Prenons ce code ci-dessous, généré aléatoirement avec les options 'def' & 'def f(a)' & 'return'.

```
enabled = True
y = 1
def build(flag):
    output = not flag
```

```
    return output
found = build(enabled)
print("Le résultat de " + "build" + "(" + str(enabled) + ")" + " est " + str(found))
```

Après avoir cliqué sur ”Lancer le diagramme et les défis” l'utilisateur pourra visualiser le logigramme ci-dessous, qui invitera l'élève à détacher conceptuellement le flux d'exécution de la fonction de celui du programme principal, qui ici appelle effectivement la fonction qui y est définie.

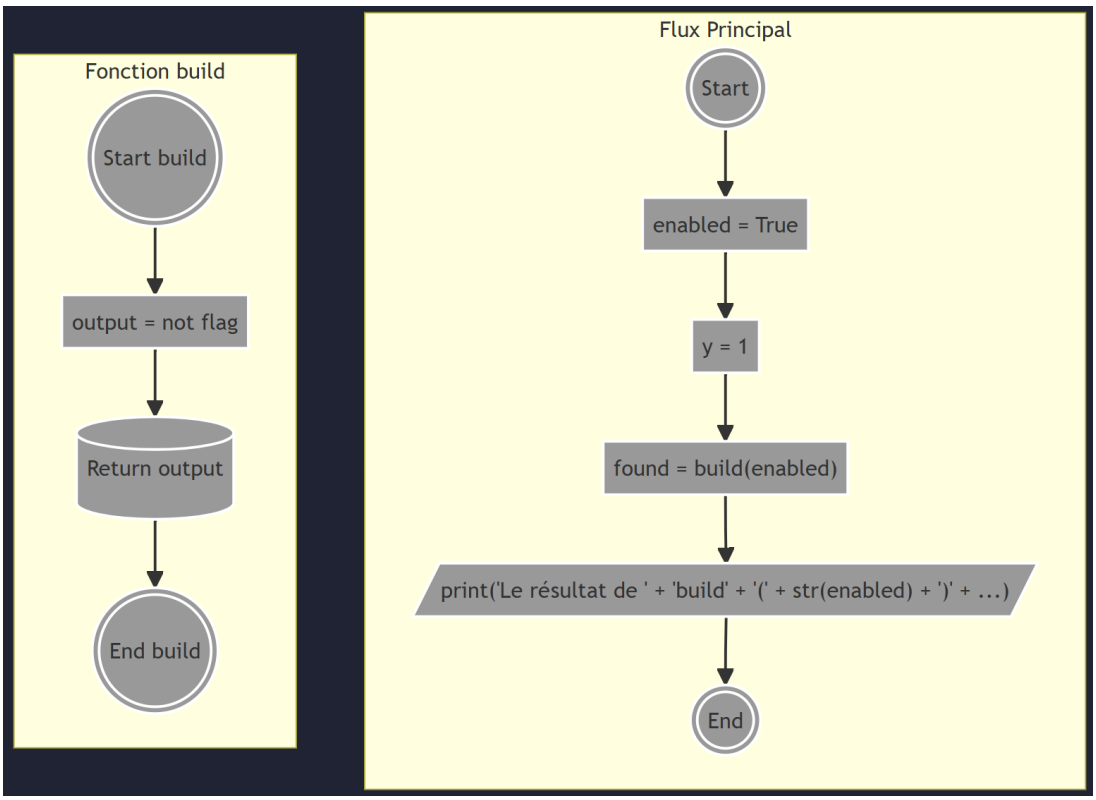

FIGURE 2 : Flowchart d'un code avec appel d'une fonction 'f(a)' & 'return'

On voit que la totalité de la chaîne n'est pas rendue pour des raisons d'affichage (coupée au 55ème caractère et suivie des ”...” dans le parallélogramme dénotant les fonctions d'entrée/sortie).
Ensuite se présente la section ”Défi” composée des variables à tracer.
Enfin l'outil propose une ”Console d'exécution” qui a été ici dépliée mais qui est par défaut masquée, utile pour vérifier les sorties écran du code présenté dans l'éditeur.

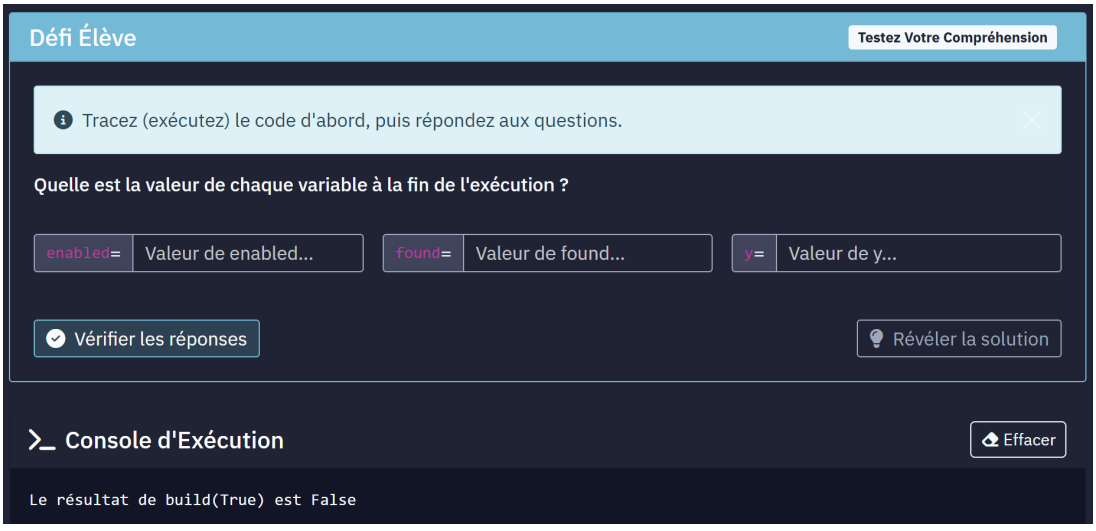

FIGURE 3 : ”Défi Élève” avec appel de fonction 'return'. La variable locale est bien absente.

2.2.1 Les interactions entre objets : entre les *cards*, entre les *divs*...

Voici un *zoom* sur les options de génération de code aléatoire (explications ci-dessous) :

FIGURE 4 : Présentation des choix de génération

- Les *checkboxes* de choix des éléments syntaxiques s'affichent lorsque la *card* est sélectionnée. Exemple : si 'Loop' est sélectionné l'interface déplie la *card* et les différents choix de boucles sont rendus visibles.
- Certains choix de *checkboxes* en impliquent d'autres : si 'Ctrl' est sélectionnée alors il y a forcément un 'if', si 'Func' est sélectionnée alors le code généré montrera au moins une 'def' de fonction, si 'elif:_else' est sélectionné, le générateur de code fonctionnera forcément comme si un 'if:_elif' avait aussi été sélectionné) ⇒ l'interface rend visible cette logique.
- Certains choix de structures rendent plus naturels certains choix de types de variables : l'interface rend l'utilisateur attentif par ajout d'une classe bootstrap sur l'élément concerné (encadré vert dynamique)
- Les encadrés du bas présentent des critères quantitatifs qui obéissent aux choix des éléments de syntaxe (les valeurs sont recalculées dynamiquement en foncion des éléments cliqués ou non)
- Les encadrés du bas sont en fait des valeurs modifiables dans des listes déroulantes : les valeurs peuvent être choisies manuellement pour définir les options de génération.

2.2.2 L'interface : ses éléments de choix pour la création d'un exercice

Deux logiques indépendantes sont implémentées :

1. la génération aléatoire, opérée avec le choix des options et le bouton "Générer un code aléatoire"
2. l'ouverture d'un code Python pré-existant
   - soit en chargeant un des codes pré-définis stockés actuellement dans un fichier JavaScript du projet (la figure ci-dessous montre ce bouton actionné)
   - soit en important un fichier avec le bouton "Ouvrir"

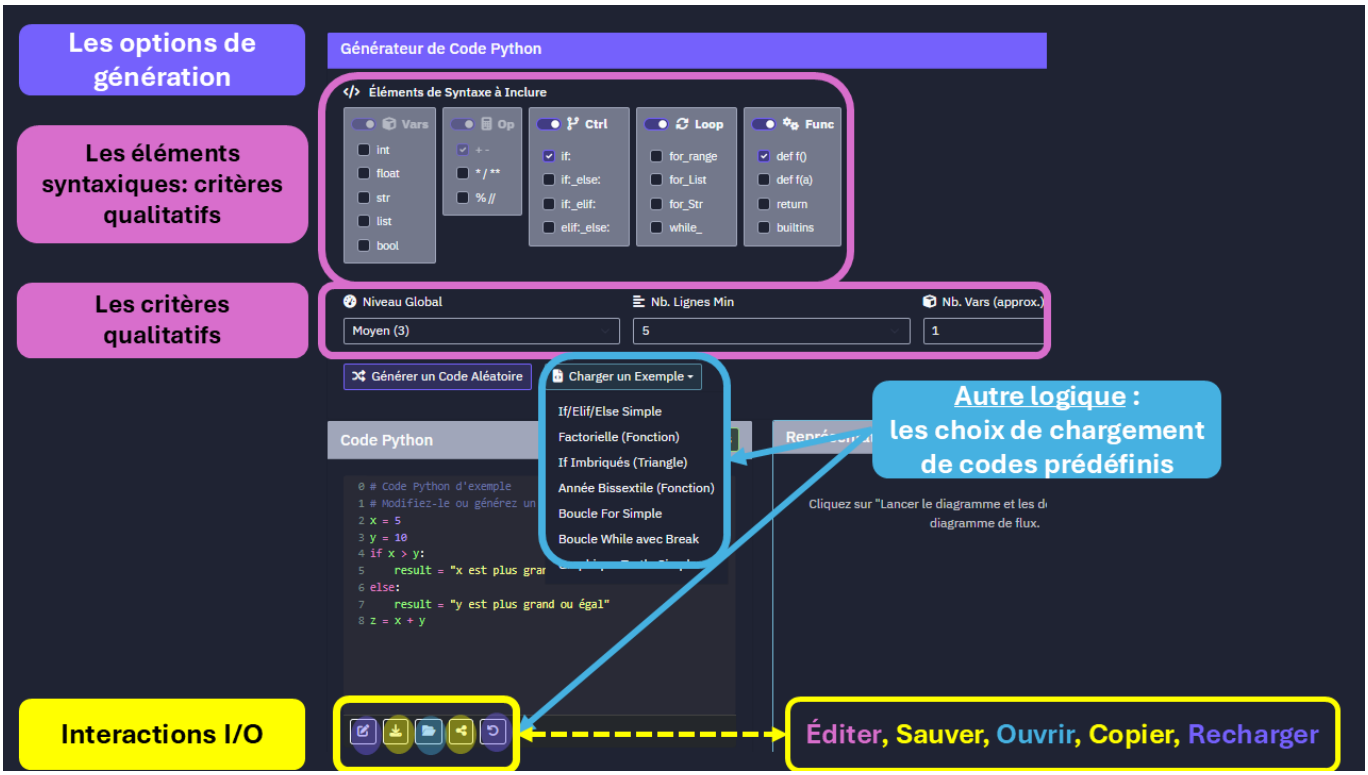

FIGURE 5 : Identification des éléments de choix pour la création d'un exercice

2.2.3 Un outil "tout-en-un" : fonctionnalités additionnelles

La volonté d'aider les enseignants s'exprime dans les possibilités supplémentaires qu'offre l'outil :

1. modifier le code et l'exécuter dans le navigateur : simulation d'un `input()` Python à l'écran (en plus de la console d'exécution déjà évoquée),
2. sauvegarder sur disque dur le code affiché dans l'éditeur,
3. copier-coller dans le presse-papier le code,
4. recharger le code tel qu'il était avant modifications (depus la dernière génération aléatoire ou depuis le dernier chargement de code pré-défini).

2.2.4 Les technologies employées côté client

L'affichage des options (choix qualitatifs et quantitatifs) est géré dynamiquement par l'application de classes Bootstrap sur les éléments HTML, qui bénéficient des éléments visuels Font-Awesome.

L'éditeur de code est une instance CodeMirror 6 (HAVERBEKE, 2021) choisi pour ses multiples paramétrages et facilité d'intégration (par défaut à l'état `readOnly`).

Le rendu graphique dans la *div* Diagramme est fait grâce à Mermaid.js qui interprète la chaîne qui est produite par l'outil à partir du code (cf. la section dévolue à cet aspect).

L'élément clé derrière l'outil est Pyodide (portage de CPython en WebAssembly), qui rend l'exécution de Python dans le naigateur, sans installation ni problèmes logistiques potentiels de connexion à un serveur. Plus lourd que Skulpt (« Skulpt », 2024) et Brython (« Brython », 2024), mais garantit une exécution Python très complète avec la plupart des modules standard, notamment ast sur lequel repose l'exécution du code et sa représentation visuelle en logigramme.

2.3 L'architecture globale (en coulisse)

2.3.1 Résumé en un schéma général

Le processus global peut se schématiser comme ceci :

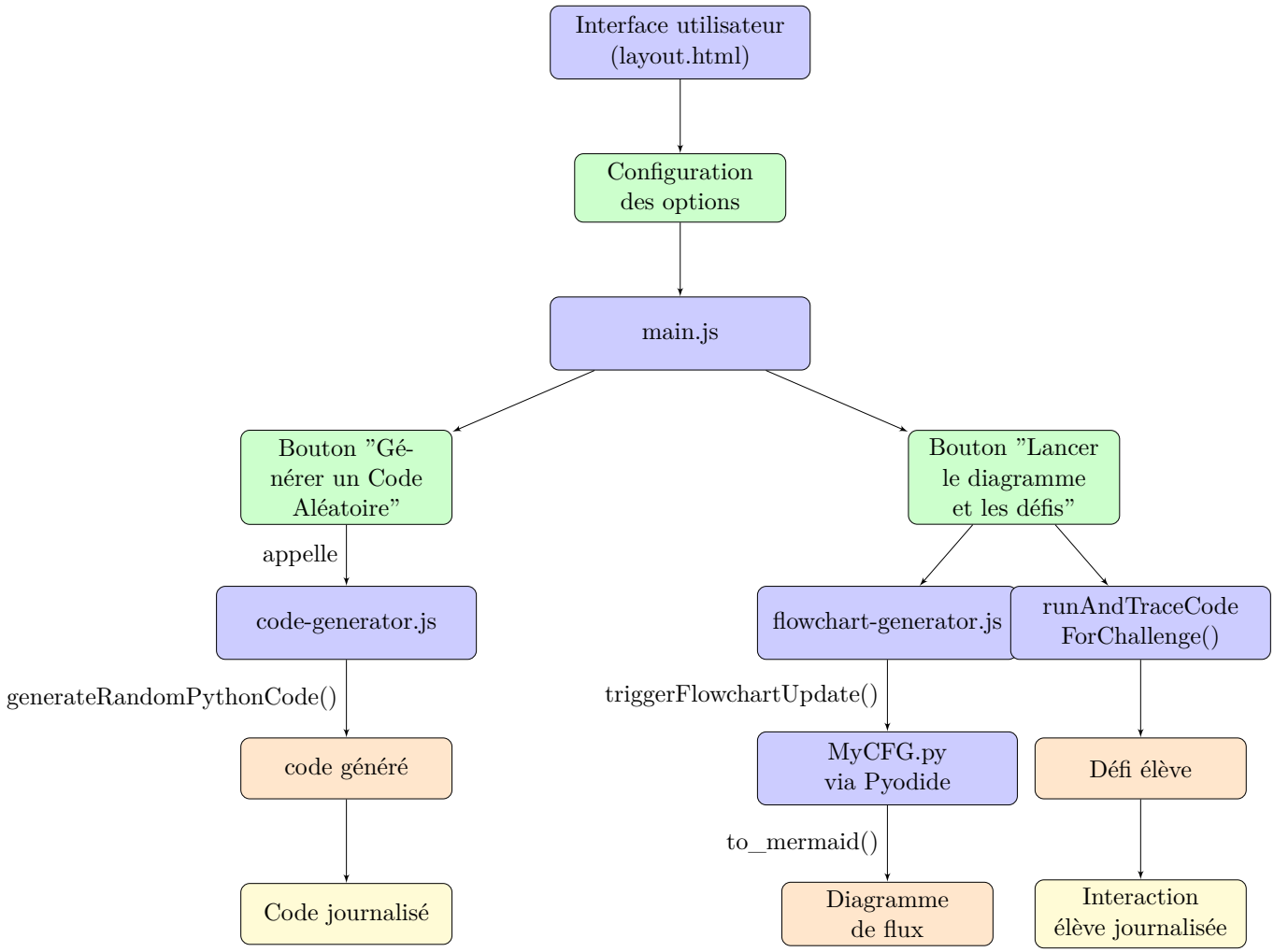

FIGURE 6 : Architecture résumée de l'exerciseur automatique

2.3.2 Résumé en quatre phrases

Le fonctionnement du générateur repose sur une interaction entre l'interface utilisateur (HTML - CSS - JavaScript) permettant de définir les options et le moteur de génération de code (JavaScript) qui écrit sur la page dans l'éditeur de script CodeMirror.

Le fonctionnement du visualiseur logigramme se base lui sur cette chaîne de code générée, pour la traduire en un AST (Python fourni par l'environnement Pyodide) pour en extraire une sémantique traduite en une syntaxe Mermaid passée au navigateur pour être rendue sous forme graphique par Mermaid.js

La partie "Défi" (avec l'interrogation des élèves sur la valeur des variables) est basée sur l'exécution du code (chaîne Python) dans le navigateur par Pyodide pour connaitre les variables à évaluer puis alimenter l'interface avec leur valeur et ainsi pouvoir proposer une rétroaction (HTML - CSS/bootstrap - JavaScript).

Enfin, un *back-end* de l'application a été créé par Flask pour enregistrer (*journaliser*) dans une base de données relationnelle MySQL les exercices et les interactions élèves qui ont découlé de la partie "Défi".

3 Présentation de la génération automatique du code

L'objectif poursuivi est multiple : aider les élèves débutants en proposant des exercices de lecture de code, automatique et à l'infini, de niveau variable et contrôlable ; aider leurs enseignants en les déchargeant de cette charge mentale avec un outil en ligne. La plus-value pédagogique de notre outil dépend de la pertinence des codes proposés, car ceux-ci formeront le support aux exercices proposés aux élèves, c'est donc le cœur de la solution !

3.1 Objectifs et principes directeurs du générateur

Les critères d'une bonne génération d'exercices correspondent à différentes contraintes :

- **Validité et éxécutabilité :** Le code généré doit être syntaxiquement correct et s'exécuter sans erreur (sauf si une erreur logique est un objectif pédagogique explicite), afin d'être analysable par notre module `MyCFG.py` et compréhensible pour l'élève.
- **Complexité Contrôlée :** La difficulté doit être adaptable, non seulement via un niveau global, mais aussi par la sélection fine des constructions syntaxiques à inclure.
- **Pertinence pédagogique :** Le code doit être sémantiquement plausible et illustrer clairement les concepts visés, sans être trivial ni inutilement complexe.
- **Variété et aléa :** Chaque génération doit produire un code unique pour constituer une source inépuisable d'exercices (noms, ordre des structures, opérations, appels de fonctions).

3.2 Vue d'ensemble du processus de génération de code aléatoire et contrôlé

La génération d'un exercice suit un flux contrôlé, orchestré par la fonction `generateRandomPythonCode(options` du fichier *code-generator.js*. Ce processus peut être décomposé en quatre phases principales, comme l'illustre la figure **??**.

Phase 1 : Initialisation. Le contexte de génération est préparé : les options de l'utilisateur sont lues depuis l'interface (types et nombres respectifs d'éléments : c'est l'objet `options`), les constantes (comme les listes de noms de variables, etc.) sont disponibles, et les structures de données pour suivre l'état du code en cours de création (lignes de code, variables déclarées) sont initialisées.

Phase 2 : Planification. Avant d'écrire la moindre ligne de code, le générateur analyse les options pour anticiper les besoins. La fonction `calculateRequiredLines` estime le nombre minimal de lignes et de variables nécessaires pour satisfaire les contraintes de l'utilisateur (par exemple, une boucle `for` nécessite au moins deux lignes et une variable d'itération). La liste des structures est mélangée aléatoirement.

Phase 3 : Génération. C'est le cœur du processus. Le générateur peuple le code en suivant un ordre logique : d'abord les déclarations de variables en choisissant leurs noms au hasard parmi des listes prédéfinies par type, puis les structures de contrôle. Les fonctions comme `ensureVariablesForOptions` et `generateControlStructures` sont appelées pour construire le squelette du programme.

Phase 4 : Finalisation. Une fois les éléments principaux en place, le générateur s'assure que toutes les contraintes sont respectées. La fonction `ensureAllVariablesAreUsed` vérifie qu'aucune variable n'est déclarée inutilement. Ensuite, `addFiller` ajoute des opérations simples pour atteindre le nombre de lignes cible. Enfin, `finalVariableCheck` effectue une dernière passe de validation et ajoute des opérations de remplissage si besoin.

Ce processus structuré garantit que même si de nombreux éléments sont générés de manière aléatoire, le résultat final est toujours cohérent et conforme aux exigences pédagogiques définies par l'utilisateur.

Le code généré est ensuite retourné à l'interface, où l'instance CodeMirror le rend visible. Puis, indépendamment, le processus pourra continuer avec la visualisation du diagramme et la création de la section "défis" - qui demande de déterminer les valeurs contenues dans les variables à la fin de l'exécution du programme - et côté serveur avec la journalisation des interactions élèves.

NB : Le processus de génération de code constitue la branche gauche dans la figure **??** depuis le bouton "Générer un Code Aléatoire" jusqu'à la journalisation.

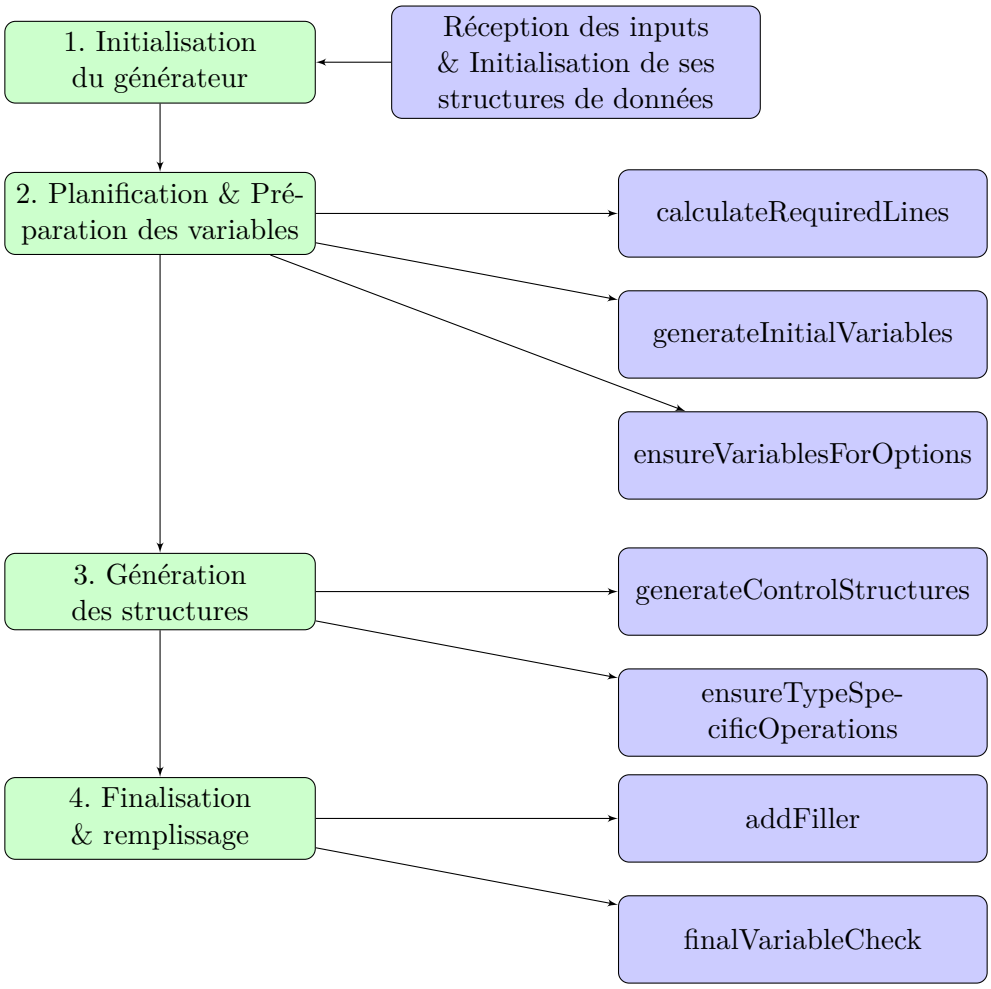

FIGURE 7 : Architecture fonctionnelle du générateur de code. Le processus suit quatre phases séquentielles, chacune invoquant un ensemble de fonctions spécifiques pour planifier, générer et finaliser le code.

3.3 Logique générale de la génération des structures

Une fois passée l'étape nécessaire de création des variables (qui propose aussi des réaffectations et des premières opérations sur ces variables, cf. explications infra) les structures de contrôle sont générées selon un processus en deux phases :

1. **Préparation** :
   - Construction d'une liste des structures à générer (une seule de chaque sorte, par souci pédagogique) selon les options reçues de l'interface
   - Mélange aléatoire de cette liste pour varier l'ordre d'apparition. Les structures peuvent "piocher" éventuellement parmi les variables disponibles (déjà déclarées dans le code généré) mais le mélange aléatoire ne provoque pas d'imbrication non voulue
   - Vérification et éventuellement création des variables nécessaires
2. **Génération** :
   - Parcours de la liste des structures
   - Appel de la fonction appropriée pour chaque structure : les structures se suffisent à elles-mêmes donc le fait d'en avoir plusieurs ne provoque pas d'imbrication qui ne soient pas souhaitées explicitement
   - Gestion de l'indentation et du comptage des lignes

3.4 Implémentations

Examinons les différentes options de générations (i.e. les éléments syntaxiques disponibles) dans l'ordre où elles apparaissent dans l'interface afin d'expliquer comment est construit le code pédagogique. Dans l'interface les structures de contrôle sont regroupées en 3 sortes : les conditions dénotées '*Ctrl*', les boucles '*Loop*' regroupant `for` et `while`, enfin les fonctions '*Func*'.

3.4.1 `'Vars'` & `'Op'` : déclarations et opérations élémentaires

L'objectif pédagogique est de découvrir les affectations de valeurs aux variables de types élémentaires, et de faire prendre conscience aux élèves que les types sont des ensembles qui viennent avec leurs opérations. Ce qui provoque l'envie de confronter les élèves à des opérations arithmétiques sur chaînes pour dévoiler la signification de la syntaxe Python : c'est ce que fait le générateur en mobilisant des opérations conformes à chaque type sélectionné.

Préparation des variables À partir des compteurs saisis (`var_int_count`, `var_str_count`, etc.), le générateur :

- détermine quoi créer (`generateInitialVariables`, ligne 366) ;
- garantit les quotas (`ensureVariablesForOptions`, ligne 1452), et crée au besoin via `declareVariable` (#168) des noms sémantiques (`generateUniqueVarName`, ligne 141) cohérents avec le type (ex. `enabled` pour `bool`).

Génération des opérations Deux mécanismes assurent que les variables "vivent" dans le code :

- **Opérations par type minimalement garanties** : `ensureTypeSpecificOperations` (ligne 1806) force au moins une opération pertinente pour chaque type demandé (ex. `str.replace`, concaténations…).
- **Remplissage aléatoire jusqu'à la cible** : `addFiller` (ligne 1500) choisit un type disponible, une variable, puis une opération parmi un éventail (`generateVariedOperation`, ligne 1586), en évitant les répétitions.

Conséquences. Le code contient toujours des affectations/opérations effectives, même si l'utilisateur n'active aucune structure de contrôle : cela garantit un minimum d'"activité" à analyser. Les opérations peuvent aussi apparaître **dans** les corps des structures (cf. `generateStructureBody`, ligne 551), ce qui renforce la variété.

3.4.2 'Ctrl' : les structures conditionnelles `if... elif... else:`

Préparation Si `options.main_conditions && options.cond_if`, la structure `'if'` est ajoutée à la liste `structures` qui est ensuite mélangée (ligne 513) pour assurer la variabilité d'une génération à l'autre pour simuler l'aléa.

Génération Au passage, `generateIfStatement` (ligne 874) :

- génère une condition en privilégiant des variables existantes, en se limitant aux types souhaités dans le plan d'études (`generateCondition`, #409). Ex : pour les booléens ce sont les comparaisons et opérations logiques qui sont mobilisées, pour les listes ce sont les opérateurs `in` ou `not in` et la fofnction `len`, etc.
- construit un corps non vide via `generateAppropriateStatement` (ligne 1245), qui ajoute des opérations adaptées au contexte ;
- garantit qu'un `pass` est inséré si aucun type n'est disponible et que rien n'a été produit (ligne 1252), ce qui a été utile pour le déboguage car ce cas ne devrait pas se produire.

Conséquences (aléa, imbrication, cardinalité) :

- **Aléa** : la présence d'un `if`, son ordre relatif, sa condition et ses opérations internes varient d'une génération à l'autre (liste mélangée + choix stochastiques).
- **Imbrication** : dans la version actuelle, il n'y a pas d'`if` dans un `if`. Les corps contiennent des opérations, pas des structures récursives.
- **Cardinalité** : au plus une instance de chaque sorte de structure est ajoutée par génération.

3.4.3 'Loop' : les boucles `for`/`while`

L'enjeu pédagogique du concept de boucle est majeur et il importe de proposer des codes plausibles en plus d'être valides. L'enjeu technique est la gestion appropriée des variables itérables et des itérateurs : préférence pour l'utilisation de variables déjà déclarées dans le code mais attention à la portée des variables locales, et existence ou non de conditions pré-existantes dans le code.

"Dispatch" des structures : appel de la fonction correspondante.
Lors de la phase **Génération**, on parcourt la liste mélangée `structures` et on appelle la fonction adaptée (ligne 531 dans `generateControlStructures`) :

`'if'` → `generateIfStatement`, `'for_range'` → `generateForRangeLoop`, `'for_list'` → `generate`

Un soin particulier a été apporté aux variables d'itération. Il nous a semblé inutilement complexe que le code généré réutilise des variables existantes pour les réaffecter comme itérateur dans la création des boucles `for ... range`. Toutefois l'utilisation de variables déjà initialisées dans le code comme itérables nous a semblé particulièrement judicieux pour obliger les élèves à tracer chacune de ces variables et pour illustrer la réaffecation des variables et le typage dynamique Python. Ainsi :

- Les noms d'itérateurs sont générés via `generateUniqueIteratorName()`
- Un compteur global `iteratorCounter` garantit des noms distincts
- Les itérateurs sont traités différemment des variables ordinaires

3.4.4 Concrétisation d'une génération de fonction : définition et appel

Le diagramme ci-dessous **??** illustre la séquence de décisions et d'actions prises par la fonction **generateFunction** pour construire une fonction Python complète, de la planification de sa signature à la génération de son appel.

Méthodes et fonctions, avec ou sans argument Les enseignants peuvent avoir la volonté de présenter la définition des fonctions séparément de leur appel, ou la volonté de présenter séparément les fonctions avec ou sans argument, aussi il nous est apparu important de séparer ces différents éléments dans notre interface : le choix 'func' implique que le code définisse une fonction ce qui équivaut au choix 'def f()',

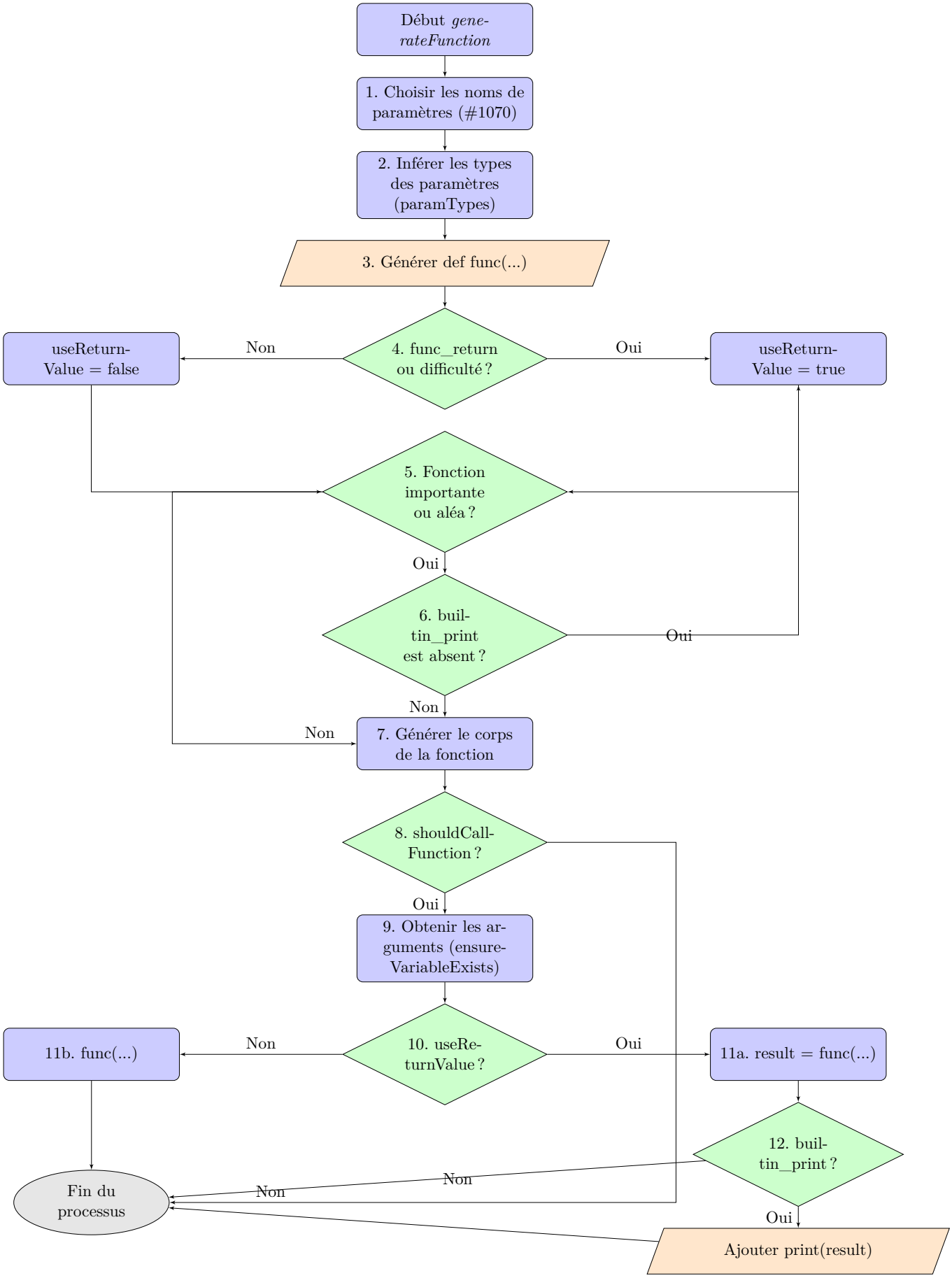

FIGURE 8 : Diagramme de flux de la logique de la fonction *generateFunction*.

mais 'return' et 'def f(a)' sont les options utlisateurs qui permettront de configurer la génération de code impliquant ces éléments.

Assurer que les fonctions définies ne sont pas juste des blocs de code "morts", mais qu'elles sont activement utilisées, est une propriété importante d'un bon programme. Toutefois, nous nous autorisons une certaine probabilité (paramétrable) de produire de tels codes "morts" avec des fonctions définies mais non appelées. Nous nous situons en effet au niveau de l'apprentissage de la programmation, et nous faisons l'hypothèse que susciter l'interrogation chez l'apprenant est quelque chose de productif. Nous pensons qu'il est intéressant que l'élève soit confronté à la conception naïve que le code de la fonction s'exécute dès l'écriture de sa définition, par exemple. C'est pour ça qu'il nous a semblé très important de proposer un éditeur de code dans le navigateur pour que l'élève, éventuellement guidé par l'enseignant, puisse modifier le code généré et interagir avec lui en rajoutant des print() ou des appels de fonction. Notre effort a donc porté sur comment faire pour que les fonctions définies par le générateur ne soient pas inutiles.

Éviter les fonctions inutiles Si la fonction retourne une valeur (`useReturnValue` est true), cocher `builtin_print` permet au générateur d'ajouter une ligne supplémentaire pour afficher ce résultat (le print(resultat)). Si `builtin_print` n'est pas cochée, le générateur sait qu'il ne peut pas rendre l'effet d'une fonction visible via un print. Pour éviter de générer un appel de fonction "invisible" (ce qui pourrait être perçu comme pédagogiquement inutile), il va forcer `useReturnValue` à true. Ainsi, l'appel se fera obligatoirement avec une affectation (resultat = ma_fonction(...)), rendant l'opération traçable pour l'élève.

Exemple Reprenons ce code ci-dessous, généré aléatoirement avec les options 'def' & 'def f(a)' & 'return'.

```
enabled = True
y = 1
def build(flag):
    output = not flag
    return output
found = build(enabled)
print("Le résultat de " + "build" + "(" + str(enabled) + ")" + " est " + str(found))
```

Comment est assurée la correspondance entre les types ? et comment se fait-il que le type de la valeur de retour a un nom cohérent avec le nom de la variable ? Cette cohérence n'est pas le fruit du hasard, mais d'une chaîne logique délibérée en deux étapes qui se déroule entièrement au sein de la fonction *generateFunction*. La correspondance entre le type attendu par la fonction (ici celui de `flag`) et le type de la variable passée en argument (ici celui de `enabled`) est assurée par le fait que le générateur planifie d'abord les types des paramètres lors de la définition, puis utilise ce plan pour générer à la fois la définition, le corps et pour choisir les arguments lors de l'appel.
Nous allons détailler le processus en traçant l'appel à **generateFunction** toujours pour notre exemple de code généré ci-dessus :

Étape 1 : Planification de la fonction et génération de sa définition)

1. **Choix d'un nom de paramètre sémantique** (cf. ligne 1083) : Évidemment la génération de la fonction commence avec un choix de nom, parmi une liste maintenue en constante pour assurer une cohérence sémantique (un verbe d'action, choisi au hasard parmi FUNCTION_NAMES). Le processus commence donc véritablement par l'appel à **chooseAppropriateParameterNames**. Cette fonction ne choisit pas des noms au hasard, mais dans des listes thématiques (ex : dataParams = ['data', 'items', 'elements', ...], mathParams = ['x', 'y', ...]). Pour une fonction nommée `build`, elle pourrait sélectionner un nom dans la liste `utilParams`, comme `flag`. Le tableau `params` contient maintenant `['flag']`.
2. **Inférence du type à partir du nom (#1099)** : Le générateur parcourt le tableau `params`. Il trouve que le nom `flag` est inclus dans la constante `BOOL_VAR_NAMES`. Il en déduit que le type attendu est bool.

```
// #1100
const paramTypes = params.map(param => {
// ...
```

```
    } else if (BOOL_VAR_NAMES.includes(param)) {
        return 'bool'; // <-- Le type 'bool' est inféré pour 'flag'
    } // ...
});
```

À ce stade, le "plan" est établi : la fonction attend un paramètre de type bool. Le tableau `paramTypes` contient `['bool']`.

3. **Génération de la définition (#1116)** : La ligne `def build(flag):` est générée en utilisant les informations du plan.

Étape 2 : Génération de l'appel de la fonction (utilisation du plan)

Après la logique de décision d'appeler ou non la fonction (#1120-1138) et la génération du corps de la fonction (#1139-1152), le générateur va dans notre exemple préparer l'appel.

1. **Sélection d'un argument du bon type (#1168)** : Le code parcourt le tableau `paramTypes` qu'il a créé précédemment.

```
// #1161
const args = params.map((param, index) => {
const paramType = paramTypes[index]; // Récupère le type planifié ('bool')
return ensureVariableExists(paramType); // Demande explicitement une variable de
});
```

L'appel **ensureVariableExists('bool')** (ligne 189) est effectué. Cette fonction va chercher dans `declaredVarsByType.bool` une variable existante. Dans notre cas, elle la trouve et retourne `enabled`. Si elle ne l'avait pas trouvée, elle en aurait crée une sur-le-champ en appelant **declareVariable(type)** qui suivra la même logique de création avec un nom sématiquement en ligne avec le type. Le tableau *args* contient maintenant `['enabled']`.

2. **Création de la variable de résultat (#1171)** : La logique a déterminé que la fonction doit retourner une valeur (`useReturnValue` est `true`). Elle doit donc créer une variable pour stocker ce résultat.
    - **Inférence du type de retour (#1170)** : Le type de retour est inféré à partir du type du premier paramètre. Puisque `paramTypes[0]` est bool, le type de retour est également défini comme bool.
    - **Création d'une variable nommée sémantiquement (#1171)** : Un appel à **generateUniqueVarName('bool')** est fait. Cette fonction choisit un nom sémantique dans `BOOL_VAR_NAMES`, comme `found`.
3. **Génération de l'appel final (#1172)** : Le générateur assemble toutes les pièces : la variable de résultat, le nom de la fonction et l'argument, pour créer la ligne `found = build(enabled)`. Le code prévoit la possibilité de fonctions à plusieurs paramètres, non évoquée ici, pertinente pour une plus grande complexité.

En résumé, la cohérence est le résultat d'un processus structuré : le nom sémantique du paramètre dicte son type, et ce type dicte à la fois le choix de la variable passée en argument et le nom de la variable qui reçoit le résultat (dont le type est fixé par commodité au type du premier paramètre de la fonction). Ainsi l'obtention d'un argument est obligatoirement du bon type, le tableau *paramTypes* agissant donc comme une "mémoire" entre la phase de définition et la phase d'appel, assurant que les deux sont synchronisées. C'est ainsi qu'un nom comme *is_valid* ou *enabled* est choisi pour une fonction qui traitera un booléen (une information de vérité).

3.5 Conclusion sur le processus de génération de code

En définitive, l'architecture actuelle du générateur, qui repose sur une inférence de type à partir de noms de variables sémantiques, peut sembler inutilement complexe. Une approche alternative, plus directe, consisterait à utiliser les options de l'interface pour forcer directement la génération de types et de noms spécifiques. Par exemple : "Si options.func_def_a est cochée et options.var_list_count > 0, alors créer une fonction def process_list(data_list):". Cependant, cette approche directe, bien que plus simple à concevoir

initialement, présente des inconvénients majeurs rencontrés pendant le développement de notre outil, qui ont amené ce choix d'une architecture indirecte avec inférence.

Explosion combinatoire et *edge cases* L'inconvénient principal de l'approche directe, qui a été rencontré très vite, est qu'elle n'a pas pu *passer pas à l'échelle* de l'ajout des options et des mutiples combinaisons envisagées. Au fur et à mesure des ajouts de structures possibles, il devient de plus en plus probable que l'utilisateur choisisse des options contradictoires ou incomplètes. Atteindre la robustesse totale impose d'anticiper chaque cas, ce qui pose des questions parfois insolubles ou nécessite une résolution arbitraire. Concrètement, chaque nouvelle option ajoutée à l'interface multiplie le nombre de cas spécifiques à gérer.

- if (option A) -> faire X
- if (option B) -> faire Y
- if (option A & option B) -> faire Z
- if (option A & option C) -> faire W
- etc...

Avec l'ambition que notre application puisse couvrir un grand nombre de syntaxes Python, le nombre de combinaisons possible a paru difficilement gérable, en tout cas la cascade de conditions imbriquées que le code était devenu paraissait impossible à maintenir et à étendre.

Au contraire avec l'architecture actuelle par inférence le système est décomposé en règles simples et indépendantes qui collaborent.

- Règle 1 : "Choisir un nom de paramètre sémantique." (chooseAppropriateParameterNames)
- Règle 2 : "Déduire le type d'un paramètre à partir de son nom." (paramTypes.map(...))
- Règle 3 : "Pour un appel, trouver une variable existante du type requis." (ensureVariableExists)

Ces règles fonctionnent ensemble de manière *émergente*. On n'a pas besoin de coder explicitement le cas "fonction avec un paramètre de type liste". Le système le découvre de lui-même en quelque sorte : il choisit un nom de paramètre comme *items* (Règle 1), en déduit que c'est une list (Règle 2), puis trouve une variable de type list pour l'appel (Règle 3). L'ajout d'une nouvelle option ne nécessite que d'ajuster une ou deux règles locales, sans casser l'ensemble du système.

Besoin d'un code "organique" et varié mais "débuggable" L'objectif n'est pas seulement de générer du code valide (syntaxiquement correct et qui termine), mais aussi du code qui semble avoir été écrit par un humain, qui puisse simuler une certaine variété "naturelle" mais dont les structures puissent être reconnues par l'auteur (au moins au moment du développement des fonctionnalités). Une approche directe avait été essayée (ça a été la version *brouillon*) mais donnait des résultats insatisfaisants : soit une variété trop faible ou à l'aspect *robotique*, soit au contraire un code généré "indébuggable". En effet, en faisant appel à une génération totalement aléatoire et centralisée il devenait impossible pour l'auteur - à la simple lecture d'un code généré - de savoir d'où venaient les structures, rendant fastidieuse la phase de développement par lecture des nombreux `console.log` pour pouvoir tracer dans le détail l'exécution exacte. Avec l'approche finalement implémentée l'aléa est introduit à plusieurs niveaux, créant une plus grande diversité de résultats cohérents.

- Le nom de la fonction est aléatoire (calculate, process, analyze...).
- Le nom du paramètre est aléatoire, mais sémantiquement lié (items, data, values...).
- La variable utilisée pour l'appel est choisie au hasard parmi celles du bon type.

Le générateur est donc "Automatique" au sens où le pipeline est déterministe une fois les options fixées (vérifications, utilitaires, remplissage, validation).
Le générateur est aussi "Aléatoire" : noms, littéraux, ordre des structures, choix d'opérations et certaines décisions (appeler ou non la fonction) introduisent de la variabilité encadrée.
Cette cascade de choix aléatoires mais contraints produit un code qui est à chaque fois différent, tout en restant logiquement et sémantiquement correct, et qui nous semble donc convenir pour un outil pédagogique. Cette approche a été choisie car elle transforme un problème complexe de combinaisons multiples en un système de règles simples et collaboratives. Elle a été le résultat de la poursuite de nombreux objectifs contradictoires : *scalabilité* et robustesse pour l'ajout de nouvelles fonctionnalités, variété mais qualité pédagogique du code généré.
Au final, la plus grande perte dans ce processus serait peut être - au-delà de la maintenabilité - la possibilité d'une collaboration entre collègues utilisateurs de l'outil-auteur, invités à contribuer sur le processus de développement du générateur actuel.

4 Présentation du traducteur automatique de code en logigramme

Le choix du langage Python comme support d'apprentissage de la programmation tient à sa lisibilité et à sa large adoption en milieu scolaire. Or le langage Python, comme d'autres langages de haut niveau, fournit un module intégré, `ast`, qui permet de décomposer du code source Python en une structure arborescente appelée Arbre de Syntaxe Abstraite (AST). On entend et lit souvent *Arbre Syntaxique Abstrait* mais l'arbre n'a rien d'abstrait aussi nous dirons AST pour Abstract Syntax Tree. Cet arbre est une représentation hiérarchique du code qui ignore les détails syntaxiques superflus pour l'exécution (comme les parenthèses ou les commentaires) pour se concentrer sur la structure logique du programme. Chaque nœud de l'AST correspond à une construction du langage : une assignation, une condition, une boucle, un appel de fonction, etc.

Dans le contexte de l'apprentissage de la programmation, la visualisation des structures de contrôle est un outil pédagogique fondamental. Nous faisons l'hypothèse que le passage d'un langage à un autre est un travail stimulant les compétences qui forment la si élusive *pensée computationnelle*. Le logigramme, ou diagramme de flux, ou *flowchart* offre une représentation graphique - certes plus intuitive - de l'algorithmique sous-jacente à un programme, mais il est constitué d'éléments syntaxiques à apprendre. Chaque élément syntaxique Python n'a pas sa correspondance un-pour-un dans la syntaxe logigramme, et la finalité du logigramme est de rendre compte de l'exécution du flux de contrôle **logique** du programme.
Notre implémentation `MyCFG.py` est conçue pour analyser du code source Python fournie sous forme de chaîne, construire un graphe de flux de contrôle (*CFG, pour Control Flow Graph*, et le traduire en une syntaxe de diagramme Mermaid. L'objectif principal est de fournir un retour visuel immédiat aux apprenants dans un environnement d'exercice interactif, en mettant l'accent sur la clarté pédagogique plutôt que sur une analyse exhaustive de tous les aspects du langage.

4.1 L'AST Python

Pour traduire un programme textuel en un CFG, il est nécessaire d'en extraire la structure logique. Plutôt que de recourir à une analyse fragile basée sur des expressions régulières, nous avons adopté une approche robuste et standard en informatique : l'analyse de l'AST.

4.2 AST, CFG, et logigramme

La génération d'un logigramme à partir d'un code source est un processus de traduction qui transforme une représentation purement syntaxique en une représentation sémantique du flux d'exécution. Notre module `MyCFG.py` opère cette traduction. Détaillons les fondements théoriques de cette transformation et illustrons comment notre implémentation l'a concrétisée.

Prenons un code minimal :

```
x = 1
if x > 0:
    y = 1
else:
    y = -1
print(y)
```

Lorsqu'un interpréteur Python lit un script, sa première étape est de le *parser* (analyser syntaxiquement). Ce processus convertit la chaîne de caractères brute en cette structure de données hiérarchique qu'est l'AST. Le `dump` de l'AST pour ce code minimal est :

```
    Module(body=[Assign(targets=[Name(id='x', ctx=Store())], value=Constant(value=1)
```

que nous préférons immédiatement représenter d'une façon plus humaine, plus lisible car déjà arborescente. On voit bien que le Module de l'AST est la racine. Elle contient une liste d'instructions (body) qui sont soit des affectations (Assign), soit une condition (If) ou un appel de fonction (Expr avec Call).

```
    Module(
  body=[
    Assign(
      targets=[
```

```
        Name(id='x', ctx=Store())],
      value=Constant(value=1)),
    If(
      test=Compare(
        left=Name(id='x', ctx=Load()),
        ops=[
          Gt()],
        comparators=[
          Constant(value=0)]),
      body=[
        Assign(
          targets=[
            Name(id='y', ctx=Store())],
          value=Constant(value=1))],
      orelse=[
        Assign(
          targets=[
            Name(id='y', ctx=Store())],
          value=UnaryOp(
            op=USub(),
            operand=Constant(value=1)))]),
    Expr(
      value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          Name(id='y', ctx=Load())],
        keywords=[]))],
  type_ignores=[])
```

Cette arborescence pourra être résumée sous une forme presque lisible ci-dessous :

```
Module
├─ Assign: Name('x', Store) := Constant(1)
├─ If (test: Compare(Name('x', Load) > Constant(0)))
│  ├─ Assign: Name('y', Store) := Constant(1)
│  ├─ else:
│  └─ Assign: Name('y', Store) := UnaryOp(-, Constant(1))
└─ Expr: Call(Name('print', Load), [Name('y', Load)])
```

Cette dernière écriture de l'AST est lisible car elle fait apparaitre des noeuds qui structurent notre lecture mais chacun de ces noeuds correspondent à une construction syntaxique du langage. Chaque nœud a des attributs qui décrivent ses composants (ex. `targets`, `value` pour une affectation). Cet arbre est une description parfaite de *ce qu'est* le code, mais pas de *comment il s'exécute*. Par exemple, il ne montre pas explicitement que la branche `orelse` est une alternative à la branche `body`, ni que l'exécution continue après la structure `If`.

L'AST est une représentation statique et hiérarchique. Le CFG, quant à lui, est une représentation graphique du flux d'exécution. On a vu que l'AST était un arbre : c'est un graphe non dirigés, acyclique et connexe (les éléments sont tous reliés). Comme l'illustre l'implémentation `PyCFG.py` de *The Fuzzing Book*, un CFG est un graphe dirigé, non connexe et potentiellement cyclique (dès qu'une itération est présente) où :

- Les **nœuds** représentent des blocs d'instructions exécutés séquentiellement sans branchement.
- Les **arêtes** représentent les transferts de contrôle (sauts) entre ces blocs.

Tant que le CFG fourni par le module ast de Python n'a pas été "visité" il est vide :

```
--- CFG : les noeuds ---
[]
```

```
--- CFG : les arêtes ---
set()
```

Une fois notre "visite" effectuée (cf. ci-dessous) on aura construit un CFG :

```
    --- CFG : les noeuds ---
[('node01', 'Start'), ('node02', 'x = 1'), ('node03', 'x > 0'), ('node04', 'y = 1'),
1'), ('node06', '.'), ('node07', 'print(y)'), ('node08', 'End')]
--- CFG : les arêtes ---
{('node02', 'node03', ''), ('node04', 'node06', ''), ('node03', 'node05', 'Non'), ('
```

On dira que le CFG est l'ensemble de ces noeuds et de ces arêtes : il rend explicite la notion de **flux**, il n'est plus hiérarchique mais séquentiel, avec des branches et des fusions, qu'il nous appartient de visualiseur pour l'élève.

4.3 Pourquoi PyCFG (Fuzzing Book) était utile mais insuffisant pour nos objectifs

Apports de PyCFG :

- Démonstration claire de la **construction d'un CFG à partir de l'AST** (*The Fuzzing Book* Zeller et al., 2022).
- Base technique pour le parcours *visitor*, gestion des blocs, arêtes et séquences.

Limites pour l'usage pédagogique visé (et adaptation `MyCFG.py`).

- **Granularité trop bas niveau** : PyCFG reflète étroitement la syntaxe (blocs techniques), alors que nous visons des **concepts algorithmiques** lisibles (`Process` / `Decision` / `Junction` / `IoOperation`).
- **Absence de nœuds de jonction** explicites pour la convergence visuelle des branches ; `MyCFG` insère des `Junction` dédiés, pour réduire la charge cognitive en fournissant à l'élève des repères visuels clairs et pour visuellement éviter les croisements illisibles au moment du rendu par l'algorithme Mermaid.
- **Sortie Graphviz** : adaptée à la recherche mais moins *portable web* ; nous ciblons **Mermaid** pour un rendu immédiat dans le navigateur, sans dépendances serveur pour cette partie du projet.
- **Sémantique pédagogique ajoutée par MyCFG** : typage léger des I/O en conformité avaec ce qui est fait en classe, des libellés plus "human-friendly", une séparation en `subgraph` pour séparer le flux principal des définitions de fonctions, ce que PyCFG ne vise pas.
- **Contrôle de la "verbosité" par MyCFG** : heuristiques pour condenser les expressions (ex. conditions courtes, noms de variables) afin d'essayer de garder le diagramme lisible.

Conclusion : PyCFG fournit l'ossature algorithmique ; `MyCFG.py` ajoute une **couche sémantique et ergonomique** nécessaire à un *retour visuel immédiat et didactique* pour les élèves, directement dans le navigateur via Pyodide + Mermaid, conformément à l'objectif de traduction du code dans le navigateur.

4.3.1 Le choix de l'AST comme source de vérité

Le module `ast` de la bibliothèque standard Python permet déjà de transformer une chaîne de code source en une structure arborescente (l'AST) qui représente sans ambiguïté les relations entre les différentes instructions. Ce choix présente plusieurs avantages déterminants :

- **Robustesse :** L'analyseur de Python fait le travail complexe de validation de la syntaxe. Notre traducteur opère sur une structure déjà validée.
- **Précision sémantique :** L'AST ne représente pas le texte, mais sa signification. Un nœud `ast.For` est distinct d'un nœud `ast.While`, et nous pouvons exploiter cette distinction pour analyser l'arbre, même si leur rendu textuel est arbitraire et peut varier d'une implémentation à l'autre.
- **Extensibilité :** L'approche est modulaire. Pour supporter une nouvelle construction du langage (par exemple, `try...except`), il suffit d'ajouter une méthode pour traiter le nœud AST correspondant (`ast.Try`), sans impacter le reste du code, implémentée dans l'approche "visiteur".

4.3.2 Le *template* (patron de conception) "Visiteur"

Pour parcourir l'AST, nous utilisons le patron de conception *Visiteur*. La classe `ControlFlowGraph` implémente une méthode `visit(node, parent_id)` qui agit comme un répartiteur. Selon le type du nœud AST visité (ex : `ast.If`), elle délègue le traitement à une méthode spécialisée (ex : `visit_If`). Cette approche structure le code de manière claire et alignée sur la grammaire du langage Python.

Notre approche s'inspire de projets académiques et open-source qui exploitent l'AST pour l'analyse de code, notamment les travaux déjà mentionnés, présentés dans *The Fuzzing Book* par Zeller et al. (Zeller et al., 2022), qui démontrent la construction d'un CFG à partir d'un AST. De même, la documentation *Green Tree Snakes* (Kluyver, 2012) illustre la puissance de la manipulation et de la visite de l'AST pour des tâches d'analyse statique. Notre `MyCFG.py` adopte une philosophie similaire en utilisant le patron de conception *Visiteur* pour parcourir l'AST. Une méthode de visite spécifique est implémentée pour chaque type de nœud AST pertinent, permettant de traduire progressivement la structure du code en une structure de graphe composée de nœuds et d'arêtes.

Le processus, orchestré par la méthode principale `visit`, fonctionne comme suit :

1. La visite commence au nœud racine de l'AST (`ast.Module`).
2. Pour chaque nœud AST rencontré, la méthode `visit` appelle le visiteur spécifique (ex : `visit_Assign` pour un nœud `ast.Assign`).
3. Chaque méthode de visite est responsable de :
   - **Créer des nœuds CFG :** Traduire le nœud AST en un ou plusieurs nœuds de logigramme (ex : un `ast.If` devient un nœud "Decision").
   - **Créer des arêtes :** Connecter le(s) nouveau(x) nœud(s) au(x) nœud(s) parent(s) du flux de contrôle.
   - **Visiter les enfants :** Appeler récursivement `visit` sur les nœuds enfants de l'AST (ex : `visit_If` doit visiter les instructions dans ses branches `body` et `orelse`).
   - **Retourner les points de sortie :** Chaque visiteur retourne une liste des "points de sortie" de la structure qu'il vient de traiter. Ce sont les nœuds CFG qui devront être connectés à l'instruction suivante.

4.4 Mon implémentation *MyCFG.py*

Pour rappel l'architecture de `MyCFG.py` est guidée par une philosophie centrée sur l'objectif pédagogique. Plutôt que de viser une représentation exhaustive de tous les mécanismes d'exécution de Python (comme la gestion des exceptions, les définitions en compréhension ou l'asynchronisme), notre implémentation se concentre sur les constructions algorithmiques fondamentales enseignées aux débutants.

Séparation des préoccupations. La classe `ControlFlowGraph` est conçue en trois phases distinctes :

1. **Initialisation et Parsing :** Le constructeur (`__init__`) prend le code source en entrée et utilise `ast.parse()` pour générer l'AST. La gestion des erreurs de syntaxe est effectuée à ce stade précoce, avec un retour immédiat si le code est syntaxiquement invalide. Ne sont pas capturées ici les erreurs sémantiques (exemple : l'utilisation dans une syntaxe Python valide de variables non initialisées)
2. **Construction du Graphe (Visite de l'AST) :** La méthode `visit` et ses sous-méthodes spécialisées (`visit_If`, `visit_For`, etc.) parcourent l'AST. Cette phase ne fait qu'ajouter des nœuds et des arêtes aux structures de données internes (`self.nodes`, `self.edges`). C'est ici que la logique de traduction de la syntaxe en flux de contrôle est implémentée. Une caractéristique clé est l'ajout de nœuds de **jonction** structurels, qui ne correspondent à aucun nœud AST mais sont essentiels pour représenter la convergence des flux de manière claire, notamment après les blocs conditionnels.
3. **Sérialisation en Mermaid :** La méthode `to_mermaid` lit les structures de données internes (nœuds et arêtes) et génère la chaîne de caractères finale au format Mermaid. Cette séparation garantit que la logique de construction du graphe est indépendante du format de sortie final.

Ce qu'il faut bien comprendre est la distinction entre le parcours structurel de l'AST du programme qui construit le CFG et le parcours suivant qui sérialise ce CFG en une chaîne Mermaid. Le premier parcours

est guidé par la syntaxe du langage Python, tandis que le second est guidé par les besoins de visualisation. L'ambivalence vient du fait que j'ai choisi le même vocabulaire pour désigner des choses différentes.

4.5 Vocabulaire AST et sémantique des nœuds du CFG

La traduction d'un AST en un CFG implique une cartographie entre les nœuds syntaxiques de l'AST et les nœuds sémantiques du logigramme. Nous avons défini un ensemble de types de nœuds internes à notre graphe, chacun correspondant à un ou plusieurs types de nœuds AST et représentant un concept algorithmique distinct. Le tableau **??** détaille cette correspondance pour les éléments actuellement implémentés dans `MyCFG.py` pour lever l'ambiguïté introduite par l'emploi des mêmes termes "noeuds" et "visite" pour des choses différentes.

Table 1 : Correspondance des Types de Nœuds Internes, Nœuds AST et Sémantique

Node Type (Interne)	**Nœud(s) AST Correspondant(s)**	**Sémantique (Langage Naturel)**
StartEnd	`ast.Module` (implicite), `ast.FunctionDef` (implicite)	Représente le point d'entrée ('Start') ou de sortie ('End') global du script/module ou d'une fonction spécifique.
Decision	`ast.If`, `ast.While`, `ast.For`	Nœud où le flux de contrôle se divise en fonction d'une condition (If, While) ou de l'état d'une itération (For). Représenté par un losange.
Process	`ast.Assign`, `ast.Expr` (contenant `ast.Call`), `ast.Pass` (via `generic_visit`)	Représente une étape d'exécution séquentielle : une affectation, l'évaluation d'une expression, un appel de fonction, ou une instruction vide. Représenté par un rectangle.
Junction	N/A (Nœud structurel ajouté par notre visiteur pour améliorer le rendu graphique et la lisibilité)	Point de convergence où plusieurs chemins d'exécution se rejoignent (typiquement après une structure `if` / `else`). Assure la clarté du flux en unifiant les branches. Représenté par un petit cercle.
Return	`ast.Return`	Indique la fin de l'exécution d'une fonction et le retour d'une valeur. Termine le chemin d'exécution dans cette fonction.
Jump	`ast.Break`, `ast.Continue`	Représente un saut inconditionnel dans le flux de contrôle vers un autre point défini (sortie de boucle pour 'Break', début de l'itération suivante pour 'Continue'). Termine le chemin séquentiel local.
IoOperation	`ast.Call` (avec `func.id` étant 'print' ou 'input')	Un sous-type de 'Process' pour distinguer sémantiquement les opérations d'entrée/sortie, souvent représentées par un parallélogramme.

Table suite en page suivante

Enfin, clarifions le vocabulaire :

Qu'est-ce qu'un "node" dans l'AST Python ? Un "node" (nœud) est toute instance d'une sous-classe de ast.AST. Chaque structure syntaxique du code Python (module, fonction, boucle, expression, etc.) est représentée par un nœud spécifique : ast.Module, ast.FunctionDef, ast.For, ast.If, ast.Assign, ast.Expr, etc. Exemple de nœuds : Module, For, If, Assign, Expr, Name, Call, Constant, etc.

Quels éléments sont de simples arguments ? Les arguments sont des valeurs ou des attributs des nœuds, mais ne sont pas eux-mêmes des nœuds SEULEMENT s'ils sont de type primitif (int, str, etc.).

On aura donc beaucoup de nœuds qui seront des arguments d'un nœud "parent". Exemple:Dans ast.For, les champs target, iter, et body sont des nœuds ou des listes de nœuds. Dans ast.Call, le champ args est une liste de nœuds représentant les arguments de la fonction appelée. Un champ comme lineno (numéro de ligne) ou col_offset est un simple attribut (pas un nœud).

Dans l'AST Python, aucun élément n'est à la fois une liste et un nœud : Un nœud est une instance d'une sous-classe de ast.AST (ex : ast.Module, ast.For, ast.Assign, etc.). Une liste dans l'AST est toujours un champ (field) d'un nœud, qui contient zéro, un ou plusieurs nœuds (mais la liste elle-même n'est pas un nœud). D'après la doc officielle (ast) Les champs (fields) d'un nœud peuvent être : un nœud unique (ast.AST) une liste de nœuds (List[ast.AST]) une valeur simple (int, str, None, etc.)

Exemples de champs qui sont des listes de nœuds : Module.body : liste d'instructions (nœuds)
FunctionDef.body : liste d'instructions (nœuds)
If.body, If.orelse : listes d'instructions (nœuds)
For.body, For.orelse : listes d'instructions (nœuds)
Call.args : liste d'expressions (nœuds)
Assign.targets : liste de cibles (nœuds)
List.elts, Tuple.elts : liste d'éléments (nœuds)
Compare.ops : liste d'opérateurs (nœuds)
Compare.comparators : liste d'expressions (nœuds)
etc.
Mais la liste elle-même n'est jamais un nœud : C'est toujours le champ d'un nœud, contenant des nœuds.
Dans l'AST Python, les champs comme target, iter, func sont des attributs d'un nœud, mais leur valeur est (presque toujours) un nœud AST lui-même. Exemple dans la doc officielle : "Each node class has a number of attributes, which are usually child nodes or lists of child nodes."

Vocabulaire officiel pour la structure de l'arbre Parent/Child (parent/enfant) :
Un nœud peut avoir des enfants (child nodes), accessibles via ses champs (fields). Le nœud qui contient un autre nœud est son parent. Fields (champs) : Les attributs d'un nœud qui peuvent eux-mêmes être des nœuds ou des listes de nœuds. Voir ast.iter_fields(node) dans la doc.
Descendant (descendant) : Tout nœud accessible en descendant l'arbre à partir d'un nœud donné.
Siblings (frères/soeurs) : Nœuds ayant le même parent.
Tree (arbre), subtree (sous-arbre) : L'ensemble des nœuds à partir d'un nœud racine.
En anglais dans la doc : "node", "child node", "parent node", "field", "attribute", "descendant", "tree", "subtree", "AST node". Vocabulaire pour la structure (français/anglais)
Imbrication / Nesting : profondeur d'un nœud dans l'arbre.
Appartenance / Membership : un nœud fait partie d'un sous-arbre.
Inclusion / Containment : un nœud contient un autre nœud via un champ.
Structure arborescente / Tree structure.

L'attribut `body` joue un rôle central dans cette représentation, car il contient les instructions ou expressions qui composent les différentes structures du code. Dans l'AST Python, même une simple ligne contient plusieurs nœuds imbriqués. Prenons un exemple concret.

4.5.1 Exemple minimal détaillé : la visite d'un code source avec `ast.If`

Le défi principal n'est pas seulement de parcourir l'arbre (la documentation Python et Green Tree Snakes en donne les moyens), mais de "tisser" un graphe séquentiel à partir d'une structure arborescente en rattachant les "têtes" de séquences aux "queues" correspondantes. Pour cela, la définition et l'enregistrement des points d'entrée et des points de sortie à chaque étape de la visite sont cruciales. Pour chaque séquence ces points d'entrée et de sortie peuvent être multiples, il faut donc construire les structures de données correspondantes.
Mes définitions de structures utilisées dans ce cadre sont essentiellement :

- **Point d'entrée :** `parent_id` ou `entry_node_ids`

- Pour une instruction simple, c'est le `parent_id` (une chaîne de caractères) passé à `visit()`. Il représente le nœud CFG précédent auquel le nouveau nœud doit être connecté.
- Pour un bloc d'instructions, c'est la liste `entry_node_ids` passée à

- **Point de sortie :** valeure de retour de `visit()` et
- Chaque appel à `visit()` ou

Reprenons l'exemple minimal montré supra pour tracer le flux d'appels et la transmission des points d'entrée/sortie :

```
x = 1
if x > 0:
    y = 1
else:
    y = -1
print(y)
```

1. **Début du processus : `visit_Module`**
   L'appel initial est `cfg.visit(module_node, None)`. La méthode `visit_Module` crée le premier nœud du CFG, `node01 ("Start")`. Ce `start_id` devient le point d'entrée initial. `visit_Module` appelle alors `visit_body` pour la liste des instructions du module, en lui passant `["node01"]` comme `entry_node_ids`.
2. **Premier niveau de `visit_body` (au niveau du module)**
   - **Instruction `x = 1` :** `visit_body` appelle `visit(assign_node, "node01")`. Le visiteur `visit_Assign` crée le nœud `node02 ("x = 1")`, le connecte à `node01`, et retourne `["node02"]` comme point de sortie. La liste des points actifs de `visit_body` devient `["node02"]`.
   - **Instruction `if x > 0: ...` :** `visit_body` appelle `visit(if_node, "node02")`. La méthode `visit` relaie l'appel à `visit_If`. C'est ici que la **récursivité s'approfondit**.
3. **Visite de la structure `If` : `visit_If`**
   - `visit_If` reçoit `"node02"` comme `parent_id`. Il crée le nœud de décision `node03 ("x > 0")` et le connecte à `node02`. Ce `node03` devient le point d'entrée pour les deux branches.
   - **Visite de la branche `body` (`y = 1`) :** `visit_If` appelle `visit_body` avec `entry_node_ids=["nod`
     Ce `visit_body` de second niveau appelle `visit_Assign`, qui crée `node04 ("y = 1")`, le connecte à `node03`, et retourne `["node04"]`. Le `visit_body` de second niveau retourne donc `["node04"]` à `visit_If`.
   - **Visite de la branche `orelse` (`y = -1`) :** De même, `visit_If` appelle `visit_body` avec `entry_node_ids=["node03"]`. Un troisième niveau de `visit_body` est invoqué, qui crée `node05 ("y = -1")`, le connecte à `node03`, et retourne `["node05"]`.
   - **Fin de `visit_If` :** Ayant traité ses deux branches, `visit_If` a récolté deux points de sortie. Il retourne la liste combinée `["node04", "node05"]`.
4. **Retour au premier niveau de `visit_body`**
   - `visit_body` reçoit `["node04", "node05"]` comme points de sortie de la structure `if`.
   - Il détecte qu'il y a plus d'un point de sortie et que ce n'est pas la dernière instruction. Il crée alors un nœud de jonction `node06 (".")` et connecte `node04` et `node05` à ce nouveau nœud. La liste des points actifs pour l'instruction suivante devient `["node06"]`.
   - **Instruction `print(y)` :** `visit_body` appelle `visit(expr_node, "node06")`. Le visiteur `visit_Call` crée `node07 ("print(y)")`, le connecte à `node06`, et retourne `["node07"]`. La liste des points actifs devient `["node07"]`.

5. **Fin du processus : `visit_Module`**
   `visit_body` a traité toutes les instructions et retourne son dernier point de sortie actif, `["node07"]`, à `visit_Module`. Ce dernier crée le nœud final `node08 ("End")` et le connecte à `node07`. Le processus est terminé.

Synthèse : La **récursivité** se manifeste par les appels imbriqués `visit` → `visit_If` → `visit_body` → `visit` → `visit_Assign`. C'est une descente dans l'arbre syntaxique. La **gestion du flux** est assurée par el chaînage des points d'entrée et de sortie, exemplifié ici par l'appel à `visit_body` dans l'appel à `visit_if` :

- `visit_body` agit comme un chef d'orchestre séquentiel. Il prend les points de sortie de l'instruction N et les utilise comme points d'entrée pour l'instruction N+1.
- Les visiteurs de structures complexes comme `visit_If` sont responsables de la **division** du flux (en créant un point de branchement) et de la **récolte** des multiples points de sortie qui en résultent.

- La méthode `visit_If` dans `MyCFG.py` illustre ce "tissage" :
  1. **Réception :** Le visiteur reçoit le nœud `ast.If` et l'ID du nœud CFG précédent (`parent_id`).
  2. **Création du nœud de Décision :** Il appelle `self.add_node` pour créer un nœud de type "Decision" dont le label est le texte de la condition (`ast.unparse(node.test)`). Une arête est créée depuis `parent_id` vers ce nouveau nœud de décision.
  3. **Visite de la branche "Oui" :** Il appelle `self.visit_body` pour la liste d'instructions de `node.body`. Le point d'entrée pour ce corps est le nœud de décision. `visit_body` retourne les points de sortie de cette branche.
  4. **Visite de la branche "Non" :** Il fait de même pour `node.orelse`.
  5. **Labellisation des arêtes :** Il identifie les premières instructions de chaque branche et s'assure que les arêtes partant du nœud de décision vers ces instructions sont bien labellisées "Oui" et "Non".
  6. **Retour des points de sortie :** Il retourne une liste contenant tous les points de sortie des deux branches. Si une branche est vide, le nœud de décision lui-même est un point de sortie.
- `visit_body` est également responsable de la **fusion** de ces multiples points de sortie en un seul via un nœud de jonction, afin de fournir un point d'entrée unique et propre à l'instruction suivante.

C'est la méthode `visit_body` qui, en recevant potentiellement plusieurs points de sortie de la structure `if`, décidera de créer un nœud "Junction" pour les fusionner avant de passer à l'instruction suivante. C'est ce passage constant de "qui me précède ?" (`parent_id`) et "qui me succède ?" (la valeur de retour `List[str]`) qui permet en définitive construit le flux séquentiel à partir de l'arbre.
La construction du CFG est donc un processus récursif et hiérarchique, où chaque structure de contrôle est traduite en une sous-structure de nœuds et d'arêtes, tout en maintenant la continuité du flux d'exécution.
Dans mon implémentation MyCFG.py, cette récursivité vient naturellement par l'interaction entre la méthode `visit` générique et la méthode

Gestion des portées et des sous-graphes. Une décision architecturale importante a été de séparer visuellement le flux principal du script des définitions de fonctions. `MyCFG.py` utilise une pile de portées (`_function_scope_stack`) pendant la visite de l'AST pour identifier quels nœuds appartiennent à quelle fonction. Lors de la sérialisation en Mermaid, cette information est utilisée pour générer des `subgraph` distincts, améliorant considérablement la lisibilité en évitant le croisement d'arêtes entre des contextes d'exécution logiquement séparés.

4.5.2 Le cas des boucles : une représentation pédagogique

Contrairement à `PyCFG.py` de *Fuzzing Book* qui traduit une boucle `for` en une structure `while` équivalente, notre approche dans `MyCFG.py` vise une **explicitation pédagogique**.

Représentation pédagogique des boucles. Pour les boucles `for`, une attention particulière a été portée à la représentation. Plutôt qu'un simple nœud de décision, nous avons choisi de décomposer la boucle en une structure plus détaillée inspirée des logigrammes classiques (voir figure **??**. La méthode `visit_For` ne crée pas un simple nœud de décision, mais décompose la sémantique de l'itération en plusieurs étapes distinctes, chacune représentée par un nœud :

- Un test initial : "L'itérable est-il vide ?"
- Une initialisation : "Prendre le premier élément"
- Le corps de la boucle
- Un test de continuation : "Y a-t-il un élément suivant ?"
- Une mise à jour : "Prendre l'élément suivant"

Voici concrètement un exemple de rendu graphique d'une boucle "for" parcourant une chaîne, mis en regard d'une structure théorique "for each".

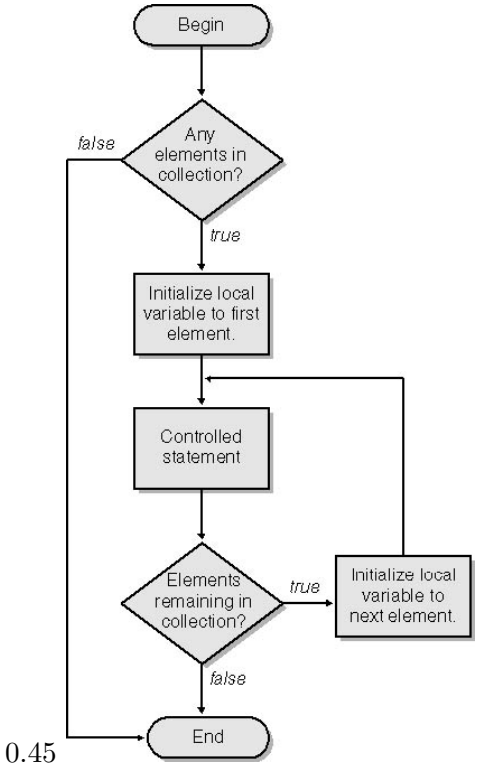

FIGURE 9 : Exemple de flowchart ”for each” (« Flowchart for-each loop (Stack Overflow discussion) », 2013)

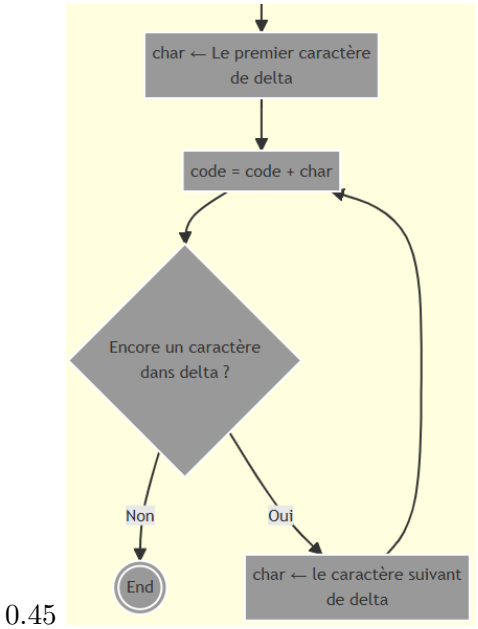

FIGURE 10 : Copie d’écran d’une partie de logigramme visualisant une boucle “for ... str”

FIGURE 11 : Comparaison de notre “for ... in <str>” avec un exemple de “for each”

Cette décomposition, bien que plus verbeuse, vise à rendre le mécanisme d’itération plus explicite pour l’apprenant, faisant abstraction des constructions internes et méthodes

Robustesse et limitations. L’implémentation actuelle gère les structures de contrôle de base (`if` / `else` , `for` , `while`), les assignations, les appels de fonction, et les sauts (`break` , `continue` , `return`). Elle ne gère pas encore les constructions plus avancées comme la gestion des exceptions (`try` / `except` / `finally`), les contextes (`with`), les classes, ou les fonctionnalités asynchrones. Cette limitation est suffisante pour couvrir un sous-ensemble du langage pertinent pour les exercices du niveau des élèves, de débutant à intermédiaire.

4.6 Fonctionnement de l'application pour la traduction de codes en logigramme

Essayons de situer plus globalement l'instance CFG (Python) dans le contexte de la traduction d'un code source issu du navigateur (JS) et rendu graphiquement par Mermaid.js à l'aide du graphique ci-dessous.

Citation de la doc officielle Pyodide (Pyodide Development Team, 2023) :

> When we proxy a JavaScript object into Python, the result is a JsProxy object.
> When we proxy a Python object into JavaScript, the result is a PyProxy object.

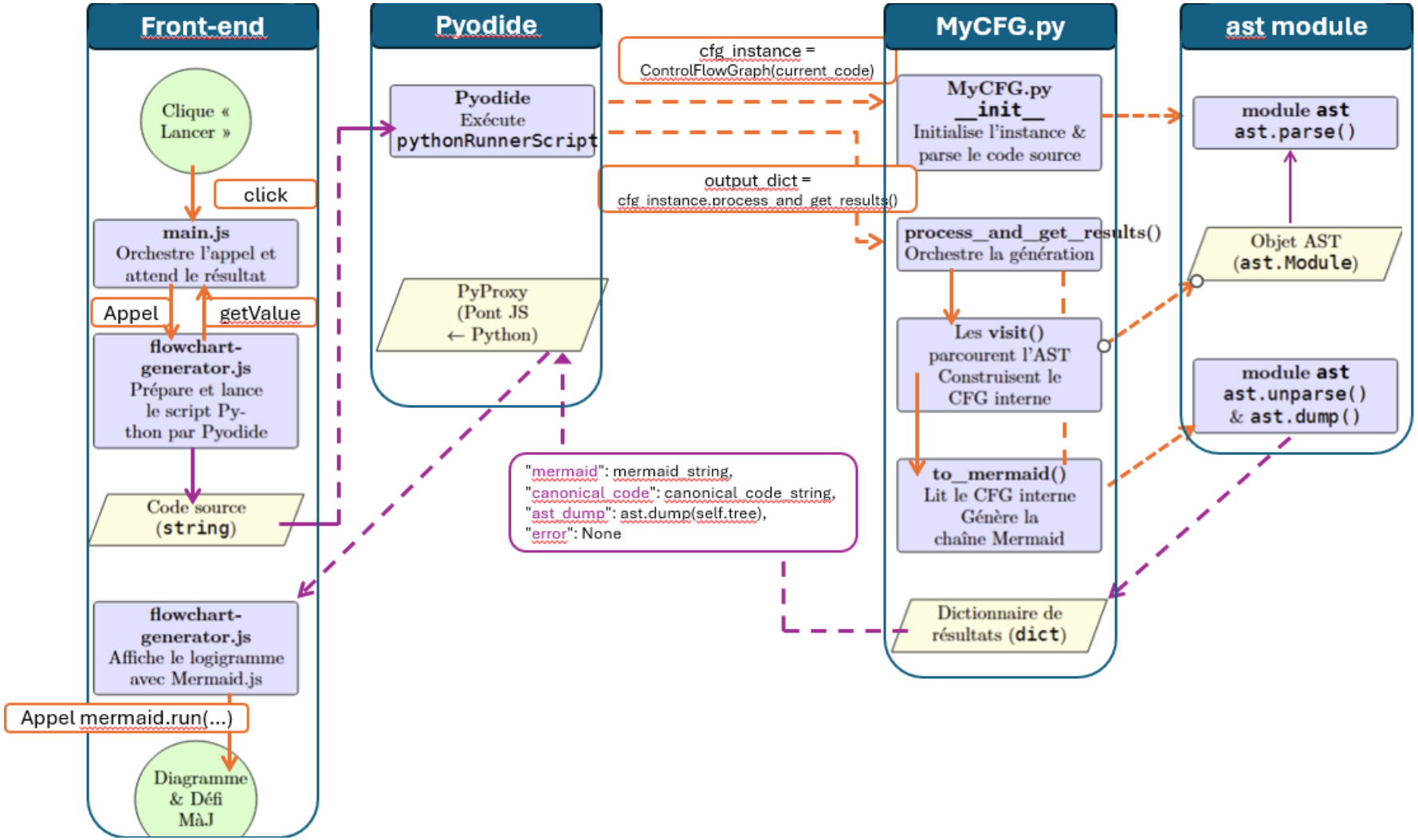

FIGURE 12 : Génération du logigramme : Business Process Model des appels et des objets retournés

5 Présentation des interactions élèves et de leur journalisation

L'enjeu de la didactique est de pouvoir fournir aux apprenants la réponse appropriée pour favoriser leur apprentissage. Notre interface *full front-end*, qui confronte les élèves à des exercices de traçage de code de difficulté variable et mobilisant des éléments de syntaxe Python de façon contrôlée, en leur proposant une évaluation formative par rétroaction immédiate, se positionne comme un outil-auteur capable de délivrer déjà un environnement d'apprentissage assez riche.

La possibilité de plus-value apportée par l'enregistrement des codes générés, des modifications faites dans l'éditeur, des éventuelles réponses proposées par l'élève ou la révélation des solutions et leur *timing* s'inscrit dans l'ambition didactique formulée par Qian et Lehman (2017) (Qian & Lehman, 2017) :

> we recommend that computing education research move beyond documenting misconceptions to address the development of students' (mis)conceptions [...]

Alors que l'enseignement de la programmation est maintenant proposée à des cohortes massives d'élèves (tous les élèves du secondaire de filière académique, à notre connaissance dans tous les pays de l'OCDE) nous rejoignons leur postulat :

> we believe that developing and enhancing instructors' [...] ability to apply effective instructional approaches and tools to address students' difficulties, is vital to the success of teaching introductory programming.

5.1 Principes directeurs de la collecte de données

Il serait évidemment intéressant de pouvoir évaluer l'efficacité de dispositifs didactiques permis par notre outil, en classe et en dehors, et de pouvoir analyser l'évolution des élèves. Cela nécessite un système de journalisation (*logging*) robuste et réfléchi.

5.1.1 Une base de données à double vocation pédagogique et didactique

La collecte de données dans un contexte éducatif répond à une double finalité. D'une part, elle a une **vocation pédagogique** : en assurant une gestion fiable et éthique des utilisateurs, elle permet de tracer le parcours individuel de chaque élève pour lui fournir une rétroaction personnalisée et suivre sa progression. D'autre part, elle a une **vocation didactique** : en agrégeant des données riches et fiables sur les interactions, elle ouvre la voie à des analyses à plus grande échelle sur les stratégies d'apprentissage, les erreurs fréquentes et l'efficacité des exercices proposés.
L'architecture de notre base de données et de notre système de journalisation a été conçue pour servir ces deux objectifs, sous deux contraintes majeures.

5.1.2 Contraintes techniques et éthiques

Des **contraintes d'anonymat** strictes doivent être appliquées relatives à l'absence de traçage : le mot de passe rentré par l'utilisateur n'est jamais stocké en clair mais haché avec Bcrypt, le compte utilisateur n'est pas croisé avec d'autres identifiants et plus généralement les données utilisateurs collectées n'existent que pour le traitement pédagogique des interactions entre l'élève et le défi qui lui est proposé. Il appartiendra à l'enseignant de dé-anonymiser les informations reçues dans le cadre de sa relation avec ses apprenants, dans le respect de son *contrat didactique*. Par ailleurs, pour augmenter les chances de récolter des données riches, en grand nombre, nous voulons proposer **une expérience utilisateur de qualité** à l'image de ce à quoi sont habitués les élèves dans leur vie quotidienne, pour augmenter l'engagement des apprenants. Ceci nous contraint à utiliser les bonnes pratiques de développement web moderne pour la gestion des utilisateurs et pour les interactions de journalisation qui doivent être asynchrones, sans blocage ni rechargement de la page entière lorsque l'utilisateur est dans l'interface.

5.1.3 Une preuve de concept extensible

Le système de journalisation actuel doit être considéré comme une **preuve de concept** fonctionnelle. Il met en place l'infrastructure technique complète pour capturer une variété d'événements (génération de code, exécution, vérification de réponses, etc.) et répond déjà à des besoins identifiés, certes de façon assez intuitive par l'auteur et ses collègues après des échanges informels. Nous insistons ici sur le fait que le travail d'analyse didactique pour définir des **indicateurs de performance pertinents** et des métriques

d'apprentissage complexes reste à l'heure actuelle un champ de recherche ouvert. L'architecture mise en place est intentionnellement modulaire pour permettre l'ajout futur de nouveaux types d'événements et d'analyses plus fines, à mesure que la recherche sur les indicateurs progressera.

5.1.4 Présentation visuelle de la collecte de données et des relations

Revoici l'UI, avec mise en évidence des éléments qui provoquent la collecte des données, déclenchée par chacun des "clics", ici mis en valeur par un petit rectangle plein (les couleurs font référence aux logiques appelées : génération ; chargement ; création du défi ; interaction avec l'exercice).

FIGURE 13 : Présentation générale de l'application, vierge

Revoyons de façon plus schématique cette même architecture, avec mise en valeur des "clicks" par une flèche colorée, évoquant l'appel au serveur Flask et l'insertion en base de données (symbolisée par les disques magnétiques).

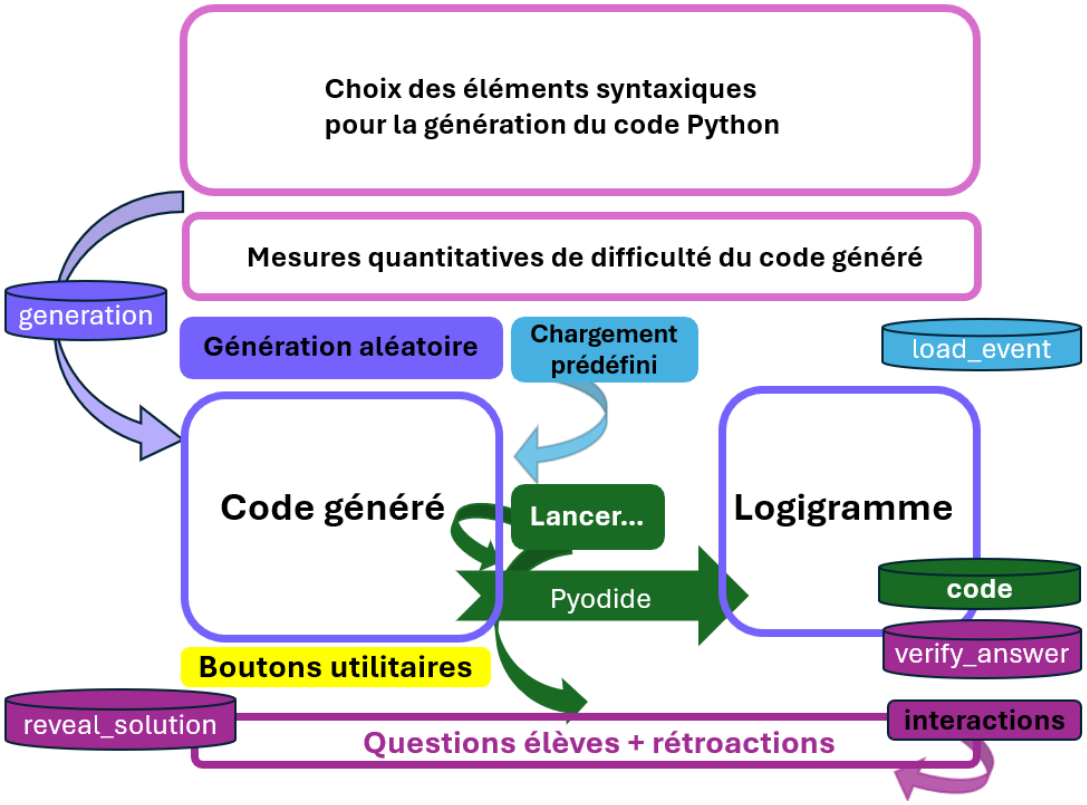

FIGURE 14 : L'UI : événements journalisés

Ce processus schématique se traduit concrètement par la base de données relationnelles `GYMINF_POC` donc le diagramme Entité-Relation est visualisé par la figure **??** ci-dessous.

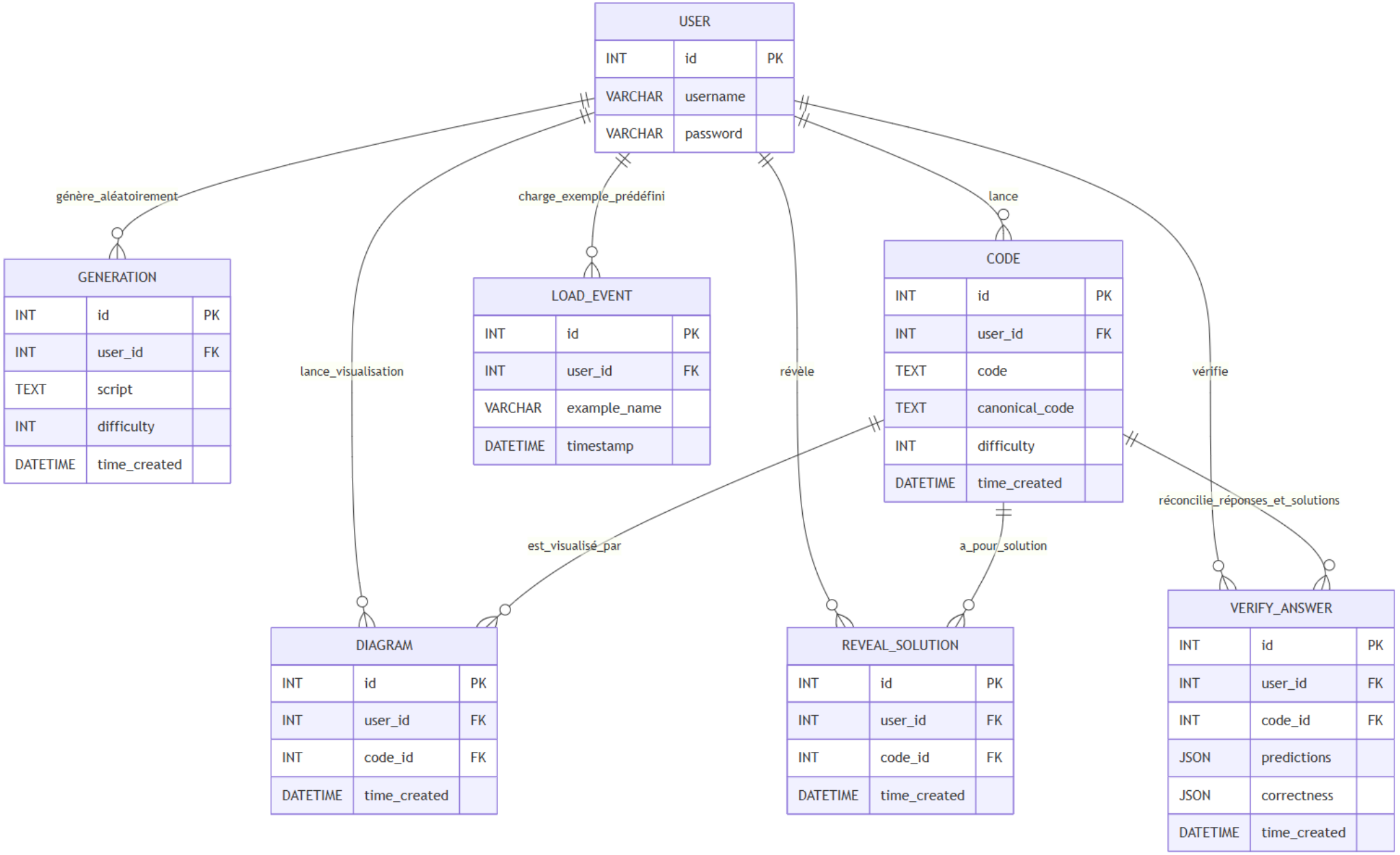

FIGURE 15 : Diagramme Entité Relations complet de GYMINF_POC

5.2 Architecture et flux de journalisation

Quel que soit l'événement, le processus de journalisation suit une architecture client-serveur tri-partite, comme illustré par la figure **??**.

5.2.1 Description générale des étapes d'une journalisation

Le flux débute par une action de l'utilisateur dans le **Navigateur (Front-end JS)**, qui est capturée par le code JavaScript. Ce dernier prépare une requête contenant les données pertinentes et l'envoie de manière asynchrone au **Serveur Web (Flask)**. Le serveur traite la requête, en appliquant d'abord la logique nécessaire (validation, authentification) et interagit avec la **Base de Données (SQL)** pour réaliser effectivement la journalisation par insertion des informations dans la table qui correspond au type d'événement sauvegardé. Enfin, le serveur renvoie une réponse au navigateur pour confirmer le succès de l'opération et, le cas échéant, transmettre des données en retour (comme un identifiant unique pour le défi en cours, qui servira pour le relier aux interactions ultérieures).
Cette architecture garantit une séparation claire des responsabilités et en une interface utilisateur réactive : elle laisse toute la logique gérant l'expérience utilisateur dans le navigateur sans l'altérer par les contraintes liées à la journalisation, et fait intervenir le serveur uniquement comme une extension parallèle chargée de gérer la base de données.

5.2.2 Types d'Événements Journalisés

Le fichier *db_queries.js* définit une énumération `log_enum` qui liste les types d'événements pouvant être journalisés. Chaque type correspond à une route spécifique sur le serveur Flask.

- **`GENERATION` :** Déclenché lors de la création d'un nouveau code Python via le générateur aléatoire.
- **`LOAD_EXAMPLE` :** Déclenché lorsqu'un exemple de code prédéfini est chargé dans l'éditeur.
- **`FLOWCHART_GENERATION` :** Déclenché lorsque l'utilisateur clique sur "Lancer le diagramme et les défis", ce qui implique la génération du diagramme et la préparation du défi. Cet événement enregistre à la fois le code original et sa version canonique (purgée par `ast.unparse(ast.parse())`).
- **`VERIFY_ANSWERS` :** Déclenché lorsque l'élève soumet ses réponses au défi (les réponses peuvent être vides, l'élève a pu voir la solution du défi avant).
- **`REVEAL_SOLUTION` :** Déclenché lorsque l'élève demande à voir la solution du défi (l'élève a pu soumettre ses réponses avant, ou non).

L'événement "Vérifier les réponses" se démarque des autres par l'insertion en base d'une structure de données plus sophistiquée, car il existe plusieurs réponses à donner à chaque exercice. Il s'agit donc de stocker le mapping entre chaque variable et la proposition de réponse correspondante proposée par l'élève, ainsi que le mapping entre le statut vrai/faux et la variable demandée.

5.3 Description détaillée des étapes : scénario d'un lancement de logigramme et de défi

Le processus est similaire pour tous les types d'événements. Pour illustrer concrètement ce processus, considérons le scénario suivant : un élève (identifié par `user_id=1`) décide de "lancer le diagramme et le défi" pour le code Python présent dans son éditeur. Pour rappel, ce bouton lance côté client la création du logigramme et de la section défi (par Pyodide). Cet événement provoquera côté serveur l'action de journalisation (appelée `FLOWCHART_GENERATION` dans l'énumération, cf. `db_queries.js`). Cet événement a été choisi car il a une sophistication supplémentaire qui est de retourner le code_id, qui est une donnée cruciale car c'est la clé étrangère pour les relations "Vérifier les réponses" et "Révéler la solution", tandis que les autres événements se contentent d'une confirmation (statut succès/échec et le message éventuel correspondant). Le fait que le code a peut-être été modifié manuellement par l'élève après sa génération est capturé par une logique en amont côté client (voir explication ci-dessous).

Étape 1 : Le contexte côté client

Au moment du clic, le code visible dans l'éditeur sera appelé **code courant** car il a plusieurs origines possibles (génération aléatoire ou chargement prédéfini) et a peut être subi des modifications par l'utilisateur.

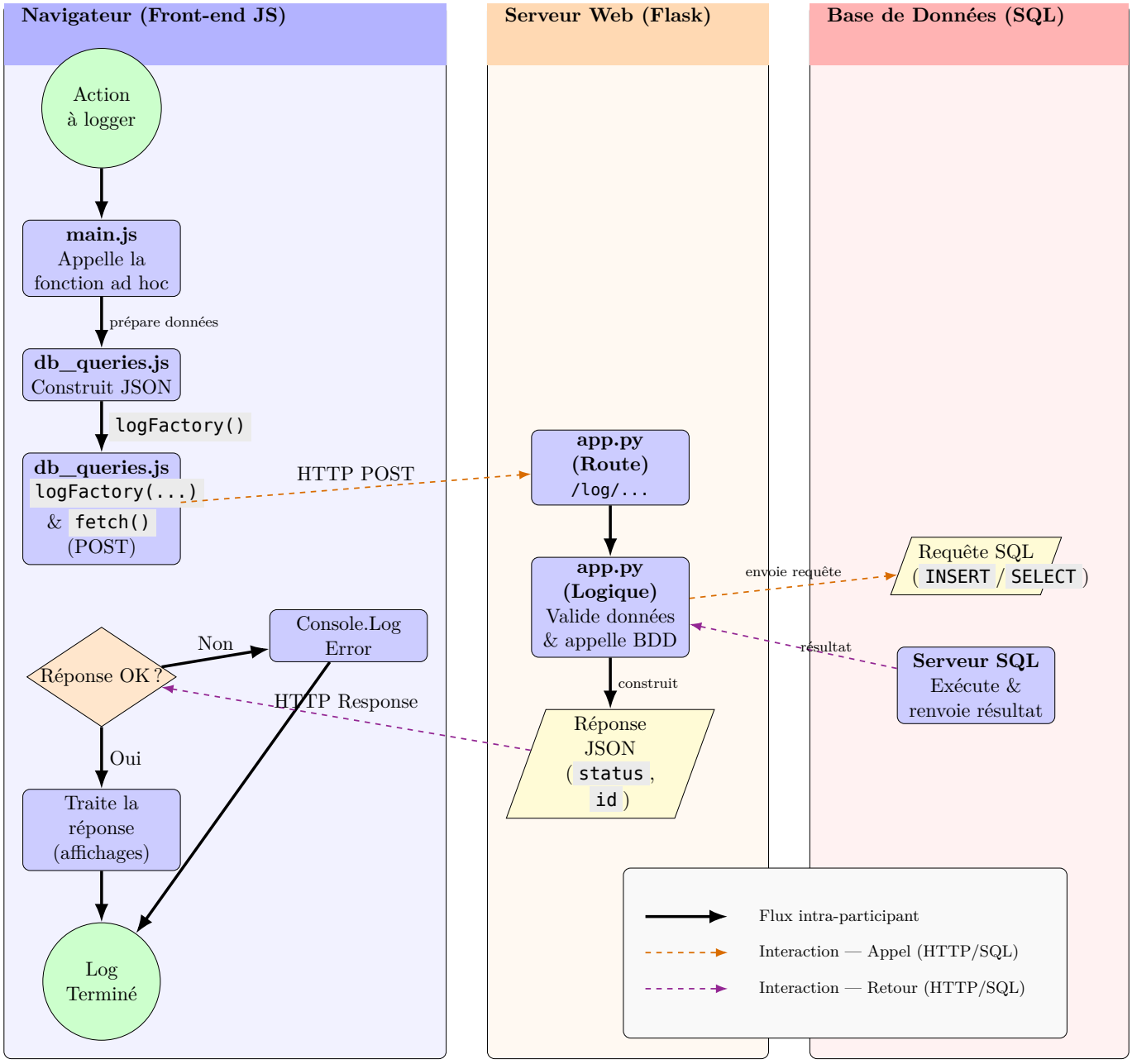

FIGURE 16 : Diagramme BPMN du flux de journalisation des actions utilisateur

Dans notre scénario, reprenons un code déjà évoqué plus haut, résultant d'une génération aléatoire, dont le niveau de difficulté s'établit à 3 (variable `difficulty`). L'élève a marqué des commentaires dans son code, pour s'aider dans l'exercice ou mémoriser la théorie, et il en résulte le code courant suivant :

```
# code avec modif
enabled = True  # affectation, bool
y = 1           # type int
def build(flag):
    output = not flag
    return output
# appel de fonction avec paramètre:
found = build(enabled)
# affichage :
print("Le résultat de " + "build" + "(" + str(enabled) + ")" + " est " + str(found))
```

Au moment où l'utilisateur clique sur le bouton "Lancer le diagramme et les défis" le processus est déclenché (cf. fichier : *main.js*, ligne n°1395.) et la variable `originalCode` reçoit le code dans l'instance CodeMirror.
Le front-end va d'abord envoyer ce code à Pyodide pour analyse. Notre module `MyCFG.py` le normalise en retirant les commentaires et donc en standardisant le formatage. Il en résulte un code qui fait abstraction des marques cosmétiques du point de vue du Control Flow Graph de l'interpréteur et que nous appelons donc le **code canonique** :

```
enabled = True
y = 1

def build(flag):
    output = not flag
    return output
found = build(enabled)
print('Le résultat de ' + 'build' + '(' + str(enabled) + ')' + ' est ' + str(found))
```

A ce stade, après résolution de la promesse Pyodide le front-end dispose du code courant, du code canonique et du niveau de difficulté.

Étape 2 : préparation et envoi de la requête AJAX

Le gestionnaire d'événement dans *main.js* appelle la fonction asynchrone `logExecutedCode` (ligne n°1441) avec ces deux versions du code et la difficulté.
Dans *db_queries.js* (ligne n°79) la fonction `logExecutedCode` prépare le corps (`body`) de la requête au format JSON et appelle ensuite la fonction centrale `logFactory` (ligne n°84) avec le type d'événement (ici `log_enum.FLOWCHART_GENERATION`) et son corps de requête.
Enfin la fonction `logFactory` (ligne n°159) utilise l'API `fetch` pour envoyer la requête `POST` asynchrone à la route Flask qui correspondant au type d'événement journalisé, c'est-à-dire à l'URL . envoie la requête `POST` suivante à l'URL `/log/flowchart_generation`. En suivant notre scénario la requête sera :

```
{
  "code": "# code avec modif\nenabled = True\n...",
  "canonical_code": "enabled = True\ny = 1\n\ndef build(flag):\n...",
  "difficulty": 3
}
```

Étape 3 : Routage puis traitement côté serveur de la requête et interaction avec la base de données

Le serveur Flask reçoit la requête et la dirige vers la fonction Python associée (cf. *app.py*, ligne n°126). Cette fonction de route `flowchart_generation_log_route` extrait les données et l'identifiant de l'utilisateur de la session pour valider les données reçues et appeler la fonction de logique de base de données ici `executed_code_log` (cf. *app.py*, lignes n°127-148) qui obtient une instance de curseur de la connexion mySQL et exécute la requête SQL suivante (*app.py*, lignes n°323-339) :

```
INSERT INTO code (user_id, code, canonical_code, difficulty, time_created)
VALUES (1,
        '# code avec modif...\n...',
        'enabled = True\ny = 1\n...',
        3,
        NOW());
```

La fonction valide ensuite la transaction par la commande `commit()` et retourne le nouvel identifiant du code journalisé (propriété `lastrowid` du curseur), pour finalement refermer explicitement le curseur - bonne pratique d'optimisation, voire de sécurité, pour les serveurs.

Étape 4 : La réponse HTTP et la mise à jour du client

Supposons que la base de données a assigné l'`id=42` à cette nouvelle entrée. Le serveur Flask renvoie une réponse HTTP avec un statut 200 (OK) et le corps JSON suivant (cf. *app.py*, lignes n°138-148) :

```
{
  "status": "success",
  "message": "Code exécuté et diagramme journalisés.",
  "code_id": 42
}
```

La promesse `fetch` dans *db_queries.js* se résout : les blocs `.then()` de la fonction `logFactory` retournent les réponses JSON du serveur (*db_queries.js*, lignes n°155-190). Les valeurs de retour sont remontées jusqu'à l'appelant, c'est-à-dire au code JavaScript (cf. *main.js* lignes n°1438-1444) qui stocke alors la valeur `42` dans la variable globale `currentChallengeCodeId`.
Désormais, toute interaction ultérieure de l'élève avec ce défi (vérification ou révélation de la solution) sera liée à l'entrée `code_id=42` dans la base de données, assurant une traçabilité complète de l'exercice. Si l'élève modifie ensuite le code de façon superficielle (par exemple, en éditant ses commentaires) la référence est conservée. Si la modification est sémantique alors l'interface se grise pour obliger à "(re)lancer le diagramme et les défis".

5.4 Visualisation logique et exhaustive de la journalisation

Revoici la schématisation utilisée précédemment, ajoutant en regard des événements les noms de chaque table et des attributs écrits dedans.

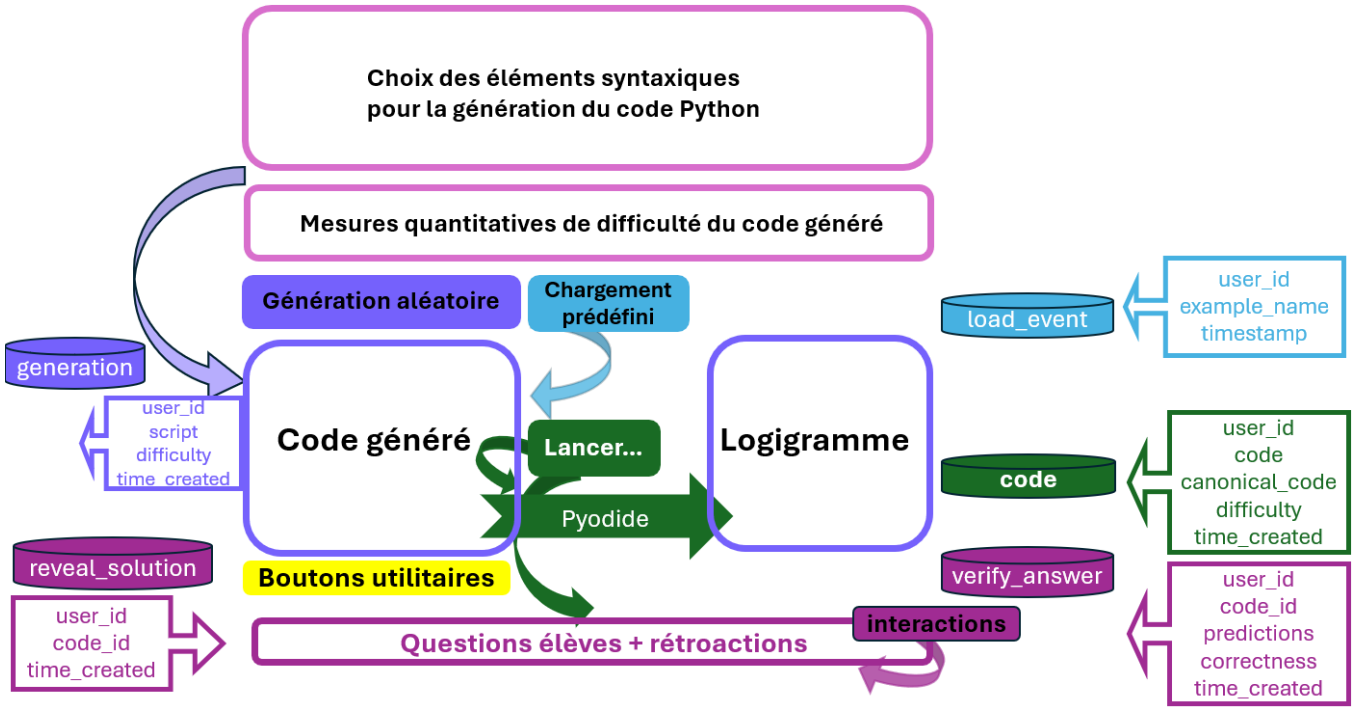

FIGURE 17 : Présentation générale de l'application, vierge

Pour une présentation exhaustive des écritures, toutes déjà implémentées mais non encore utilisées dans un dispositif en classe, voici un tableau situant chaque événement dans son processus "Front-end → (Flask ⇔ SQL) → Front-end" utilisant le même code couleur que ci-dessus.

Événement	LOG (JS) Route (Flask)	Fonction (JS)	Entrées Serveur (JSON depuis JS)	Retours Serveur (JSON vers JS)	Table(s) Cible(s)	Attributs Insérés en Base
Générer un Code Aléatoire	GENERATION /log/generation	logGeneratedCode	{"code", "difficulty"}	{"status", "message"}	generation	user_id, code, difficulty, time_created
Lancer le Diagramme et les défis	FLOWCHART_GENERATION /log/flowchart_generation	logExecutedCode	{"code", "canonical_code", "difficulty"}	{"status", "message", "code_id"}	code,	user_id, code, canonical_code, difficulty, time_created
					diagram	user_id, code_id, time_created
Vérifier les réponses	VERIFY_ANSWERS /log/verify_answers	logVerifyAnswers	{"results", "code_id"}	{"status", "message"}	verify_answer	user_id, code_id, time_created, predictions, correctness
Révéler la solution	REVEAL_SOLUTION /log/reveal_solution	logRevealSolution	{"code_id"}	{"status", "message"}	reveal_solution	user_id, code_id, timestamp
Charger un exemple	LOAD_EXAMPLE /log/load_example	logLoadExample	{"example_name"}	{"status", "message"}	load_event	user_id, example_name,

FIGURE 18 : L'UI : événements journalisés

5.5 Défis techniques et solutions implémentées

La conception d'un environnement d'apprentissage interactif et instrumenté comme le nôtre soulève plusieurs défis techniques. Notre architecture actuelle est le fruit de solutions pragmatiques répondant à des contraintes spécifiques de performance, de logique pédagogique et de robustesse des données.
Cette approche "Problème-Solution" a guidé le développement de notre système de journalisation et d'analyse de code.

TABLE 2 : Tableau de correspondance entre les défis techniques et les solutions implémentées

N°	Défi Technique	Solution Architecturale Implémentée
1	**Suivi Pédagogique Individuel.** Comment lier de manière fiable chaque action de l'élève (génération, vérification, etc.) à son profil et à un exercice précis ?	**Journalisation Authentifiée et Contextualisée.** Chaque requête AJAX est authentifiée via la **session Flask**. Une création d'utilisateur génère un `user_id`, une génération d'exercice génère un `id` unique. Ces ID pourront ensuite être utilisées pour lier toutes les interactions ultérieures (à travers les tables de la base de données), garantissant un suivi cohérent.
2	**Dualité du Code Source.** Comment gérer et journaliser à la fois le code initialement chargé et le code potentiellement modifié par l'élève dans l'éditeur ?	**Double Capture et Restauration.** Un code généré ou chargé peut être modifié dans l'éditeur. La variable JS `lastLoadedCode` sert de point de restauration pour l'élève. Lors de la journalisation, nous enregistrons systématiquement deux versions : le **code original** (brut, de l'éditeur) et sa **version canonique** (via `ast.unparse`). Cela offrira la possibilité d'analyses fines des modifications de l'élève, et ouvrira même la possible constitution de *big data* sur les interactions élèves.
3	**Variations Cosmétiques vs. Changements Structurels.** Comment éviter de journaliser un nouvel exercice si l'élève n'a fait que des changements non-logiques (commentaires, espaces) ?	**Détection de Doublons via le Code Canonique.** La version canonique du code sert d'"empreinte digitale" de sa structure. Une variable JS, `lastLoggedCanonicalCode`, mémorise la dernière empreinte journalisée. Un nouvel enregistrement n'est effectué que si la nouvelle empreinte est différente, évitant ainsi les doublons logiques.
4	**Latence de l'Analyse Côté Client.** Comment minimiser la lenteur perçue par l'utilisateur due aux multiples appels asynchrones (`await`) nécessaires à Pyodide pour analyser le code ?	**Appel Unifié à Pyodide.** Au lieu de multiples appels `await` pour chaque artefact, une méthode centrale en Python, `process_and_get_results()`, est appelée une seule fois. Elle orchestre en une passe le parsing, la construction du graphe, et la génération de tous les artefacts nécessaires (`mermaid`, `canonicalCode`, `ast_dump`), qui sont retournés dans un unique objet.

6 Travaux futurs, discussions et conclusions

Certaines limitations sont connues et nous en ferons une liste ci-dessous. Certaines sont en cours de résolution. Aussi, dans l'attente de voir l'outil utilisé en classe, nous pouvons indiquer dans quelles directions notre outil semble orienter de prochaines recherches.

6.1 Des orientations de recherche qui pourraient être soutenues par notre outil

6.1.1 La question des indicateurs pertinents

La possibilité de confronter des élèves à des difficultés que l'on peut caractériser quantitativement (nombre de structures syntaxiques différentes, longueur du code, nombre de variables) et qualitativement (types de variables, types de structures) tout en mesurant les performances des élèves(réussite/échec sur chaque variable) amène naturellement à se poser la question de la construction d'indicateurs pertinents.

6.1.2 Des questions épistémologiques rendues empiriques

Dans quelle mesure :

- L'apport de la visualisation graphique améliore-t-elle l'apprentissage de la programmation Python chez les débutants ?
- La génération aléatoire de code contrôlée par objectifs soutient-elle l'apprentissage chez les débutants dans le cadre de l'approche PRIMM ?
- La mise à disposition de l'outil, avec sa capacité de génération infinie et sa rétroaction automatique non bloquante, a-t-elle un effet sur la rétention à long terme des élèves ?

6.2 Des idées abandonnées par manque de temps

1. Contrôler l'aléa en utilisant un ensemble de *templates* à remplir pour chaque classe d'exercices à générer (plutôt qu'une approche grammaticale du langage) ⇒ Idée alternative pour faire d'un script un véritable exercice : prédéfinir les questions à poser aux élèves selon le *template* choisi (*aka* la classe d'exercices)

   Exemples : "Quel est le nombre de passages dans la boucle ?" si une boucle est choisie, "Combien de fois le bloc *xyz* a-t-il été exécuté ?" si des boucles imbriquées ont été choisies, etc.
2. Générer les valeurs attendues simultanément aux *erreurs attendues* signifiantes afin de pouvoir fournir un feedback à valeur ajoutée ⇒ Idée plus radicale : se limiter à des affectations de valeurs (les questions aux élèves se limiteraient à "Valeur de x = ...", "valeur de y = ...")
3. Préparer des questions "Comment serait modifiée la valeur si ... *(ici la modification à prévoir)* ?" en plus, pour renforcer/tester la compréhension.
4. Ouvrir la possibilité de générer un script invalide **exprès** comme variable didactique ⇒ besoin de contrôler quelles sont les erreurs permises (Division par zéro ? Index Error ? etc.)
5. La génération d'exercices "Chercher l'erreur", y compris erreurs sémantiques et/ou syntaxiques ? Un intérêt pédagogique avéré (le débogage) mais secondaire par rapport à mes objectifs curriculaires.

6.3 Travaux pour améliorer l'approche actuelle

Les améliorations envisagées sont d'ordre quantitatif (ajouts d'éléments) et qualitatif.

- Enrichir les types d'exercices et la section "Défis" :
  1. Proposer une option "Debug" qui génère des codes invalides, avec des boucles infinies ou syntaxiquement incorrects, appelant les élèves à identifier les erreurs selon leur type.
  2. Des questions qualitatives qui peuvent être ouvertes ou fermées par menu déroulant : Quels sont les types des variables ? (les types étant à récupérer depuis le générateur de code).
  3. Des questions quantitatives *in situ* : Combien de passages dans la boucle ? Combien de passages à tel point du code ?
  4. Des questions mobilisant des concepts théoriques : Telle boucle du code est-elle bornée ?

- Proposer des QCM sur le code généré, avec erreurs attendues mélangées avec la bonne réponse et des erreurs totalement *random*.
- Améliorer les rétroactions pour le bouton "Vérifier" :
  1. Utiliser la réponse élève pour l'analyser au regard des erreurs attendues pour donner des indices (cf. litérature didactique)
  2. Ajouter des *tooltips* (visibles en survolant l'espace de réponse avec le pointeur souris) pour donner des indications de réponses ou des bonnes questions à se poser, pour baliser le travail des élèves
- Améliorer l'interface pour renforcer l'engagement :
  1. Rendre le *flowchart* cliquable pour un meilleur rendu smartphone
  2. Améliorer la *responsiveness*, notamment les espaces horisontaux pris par les cartes en haut de page
  3. Utiliser des "combines addictives" pour faire revenir les utilisateurs
- Améliorer le rendu *flowchart* :
  - Éléments syntaxiques Python à implémenter :
    1. le *break* à connecter au nœud terminal de sa boucle
    2. les `try: except: else: finally:`, les `Raise`, les `Assert`, ...
    3. Un traitement des annotations de type ?
    4. un traitement de la récursion ?
  - numéroter les nœuds graphique selon `lineno` pour visualiser correspondance syntaxe ⇔ sémantique
  - colorer les flèches "True / False" resp. Bleu/Rouge
  - clarifier (coloriser ?) les rendus graphiques des différents types d'appels : internal_call, appels de type I/O, ...
  - animer les éléments graphique pour réduire la charge cognitive de l'élève ?

6.4 Des réponses aux critiques les plus probables

Les objections qui pourraient être faites : expliciter des choix assumés, notes des limitations actuelles dont les solutions ont déjà été pensées et non encore implémentées, des approches alternatives (en cours de recherche ou à l'état complètement hypothétique)

Critique sur la finalité des codes générés proposés aux élèves

Deux reproches pourraient nous être faits à ce titre :

- Le code généré manque de "sens" ou d'intentionnalité. Il ressemble à une suite d'opérations aléatoires plutôt qu'à un algorithme résolvant un problème.
- La génération basée sur des motifs et listes de noms de variables prédéfinis limite la créativité et ne prépare pas les élèves à la résolution de problèmes authentiques.

Ces critiques sont valides et sont le résultat d'un compromis assumé. L'outil se positionne au début du spectre de l'apprentissage, correspondant aux phases "Predict" et "Run" du modèle PRIMM. À ce stade, la familiarisation avec des motifs standards et des noms de variables conventionnels (`i`, `count`, `items`) est un objectif pédagogique en soi. La créativité et la résolution de problèmes authentiques interviennent dans les phases ultérieures. Nous pensons que notre outil est de nature à supporter les stades "Investigate" et "Modify" en permettant à l'élève de modifier librement le code généré. Le générateur fournit donc un "échafaudage" sur lequel l'élève peut ensuite construire, et un outil qui décharge du temps pour l'enseignant. L'objectif de l'outil n'est pas de générer des solutions à des problèmes complexes, mais de fournir des extraits de code ciblés pour l'apprentissage de la syntaxe. Les fonctions chooseAppropriateParameterNames et generateStructureBody sont des tentatives pour injecter une plausibilité sémantique dans notre approche aléatoire et impérative, mais l'objectif principal reste la maîtrise des constructions du langage, pas la conception algorithmique.

Critique sur l'approche impérative et aléatoire de génération de code

A la suite de la critique précédente, il vient l'observation qu'une autre approche aurait pu être plus élégante sur le plan théorique et pour l'implémentation, comme une approche déclarative, par solveur de contraintes par exemple, ou par modèles de langage.
Nous avons déjà répondu en préambule à la question de la délégation de la solution à une API du commerce ou un modèle propriétaire. Nous répondons ici que le déploiement d'une solution JavaScript s'exécutant dans le navigateur pare aux problèmes logistiques de déploiement d'une solution plus lourde. Ajoutons enfin que que notre approche impérative actuelle avec ses phases (préparation, écriture et vérification) est très verbeuse mais relativement performante et déterministe dans son comportement, qui peut être déboguée avec nos outils standards. C'est un choix pragmatique privilégiant la faisabilité et la performance dans le temps imparti et le contexte technologique imposé.

6.5 Travaux pour une nouvelle approche de génération de code

Approche par "template sémantique"

Un pré-requis pour cette approche est de disposer d'une base de codes "organiques" ayant un sens et résolvant un problème, de différents niveaux, toujours compatibles avec nos objectifs poursuivis : l'enseignement auprès des débutants d'école secondaire. Il serait alors possible par analyse systématique des AST de caractériser chaque code finement (selon les éléments syntaxiques utilisés) et il sera ensuite possible d'*aléatoiriser* chaque code, en substituant aux variables didactiques (noms et valeurs affectées aux variables) d'autres, en accord avec les objectifs pédagogiques sélectionnés par l'utilisateur.

A quoi ressemblerait une approche déclarative

On ne ferait que décrire le résultat final. Le système se chargerait de trouver un code qui respecte cette description :

```
const codeDescription = {
    targetLines: 20,
    difficulty: 3,
    mustContain: ['if', 'for_list'],
    mustUseVariables: ['int', 'list'],
    allVariablesMustBeUsed: true,
    noInfiniteLoops: true
};
const generatedCode = declarativeGenerator.generate(codeDescription);
```

A quoi ressemblerait une approche par solveur de contraintes

La génération de code pourrait être modélisée comme un problème de contraintes :
Les variables :

- chaque ligne de code potentielle,
- chaque nom de variable,
- chaque valeur littérale.

Les contraintes :

- contrainte_longueur : Le nombre total de lignes doit être entre 15 et 20.
- contrainte_syntaxe : Si l'option if est cochée, le code doit contenir au moins un bloc if.
- contrainte_dépendance : Si un bloc for..in..list existe, alors une variable de type list doit exister et être déclarée avant la boucle.
- contrainte_unicité : Deux variables ne peuvent pas avoir le même nom.
- contrainte_utilisation : Chaque variable déclarée doit apparaître au moins une fois dans une expression après sa déclaration.
- contrainte_terminaison : Le programme doit se terminer par une instruction de retour ou une sortie.

- contrainte_sémantique : Les opérations doivent être logiques (ex : ne pas soustraire deux chaînes de caractères).

Le solveur de contraintes recevrait toutes ces règles et tenterait de “construire” un programme valide qui les respecte toutes.

7 Conclusions

Remarques personnelles

Eu égard à son aspect certificatif ce travail mobilise une diversité de compétences, mettant en oeuvre des apprentissage variés (développement web, POO Python, base de données relationnnelles, modèle client - serveur), ce qui apporte une première justification au parcours de formation GymInf qu'il vient clôturer.
Je crois que mon projet a du potentiel dans une démarche de pédagogie active, notamment dans le cadre de l'approche PRIMM, mais il reste à le soumettre aux pairs pour améliorations et à le tester dans de multiples classes à grande échelle pour en valider l'efficacité pédagogique et la pertinence des données collectées. alors seulement je pourrai dire que j'ai apporté ma pierre à l'édifice dans la constitution d'une communauté *scientifique* de praticiens de la pédagogie de l'informatique.
Aussi je crois que l'intérêt principal de mon outil n'est pas de permettre aux élèves de se réapproprier leur apprentissage en leur offrant des outils adaptés à leurs besoins et à leur niveau (ce qu'ils pourraient faire par ailleurs), mais surtout de permettre aux enseignants d'analyser les erreurs et de construire des indicateurs pertinents pour aller au-delà de l'observation des *students misconceptions*.
Je pense que l'enjeu principal dans l'enseignement de la programmation est l'*empowerment* des débutants qui se tiennent à l'écart *socio-culturellement* alors que les IA génératives donnent maintenant cette faculté de décupler un minimum de savoirs pour une puissance de productivité inouïe. Le code n'est pas seulement une technique : c'est une super-puissance que chacun peut apprendre pour imaginer, créer et agir dans le monde numérique. »

Á la suite de Littman (Littman, 2023) je crois que découvrir un peu de programmation, c'est ouvrir une porte vers l'autonomie, l'inventivité et la joie de façonner le futur. L'apprentissage de la programmation n'est pas une fin en soi, mais une ouverture : celle d'une capacité accrue à participer, à innover et à s'approprier un langage pour penser, créer et agir dans une société traversée par le numérique.

Remerciements

Merci à Patrick Wang pour son temps, ses idées et ses conseils tout au long de ce projet, pour son incitation à faire plus et à faire mieux.
Merci à tous les acteurs du programme GymInf : Andreas Humm pour la coordination et tous les enseignants et les participants pour leur engagement.

8 Annexes

8.1 Check-list de *convivialité* au sens d'Illich pour un service numérique et invitation à la réflexion

En français, d'après le *Larousse* le mot "convive" a une signification ancienne :

Personne qui prend ou doit prendre part à un repas

qui a donné son sens au mot "convivialité" qui a - toujours d'après le *Larousse* - deux significations aujourd'hui :

1. Capacité d'une société à favoriser la tolérance et les échanges réciproques des personnes et des groupes qui la composent.
2. Facilité d'emploi d'un système informatique.

Ivan Illich a donné un sens particulier à ce mot dans les années 1970. Pour inciter les enseignants et leurs élèves à se questionner sur le sujet, nous vous proposons une *check-list* pratique et facile à mettre en place, qui ne nécessite pas de faire appel aux auteurs originaux de philosophie politique et sociologie.

L'idée est d'attribuer une note entre 0 et 6 à chaque service numérique, par l'application de 6 critères, chacun évalué entre 0 et 100% selon que la réponse à la question est totalement négative (0%) ou parfaitement juste (100%).

Appropriation & autonomie L'usager peut-il apprendre, utiliser, réparer/modifier l'outil sans dépendre d'experts ou d'un prestataire unique ?

Auto-limitation & échelle humaine Le service fixe-t-il des limites à sa taille/complexité/effet réseau pour éviter la capture (l'addiction, la croissance illimitée, les dépendances opaques) ?

Gouvernance par les usagers La communauté participe-t-elle réellement aux règles (statuts, votes, roadmap publique) ?

Transparence & auditabilité Code/algorithmes/logs sont-ils documentés, vérifiables (ou, a minima, explicables et audités) ?

Portabilité & réversibilité Export complet, formats ouverts, interopérabilité, pas de lock-in, migration/fork sont-ils tous et toujours permis ?

Sobriété & lien social L'outil favorise-t-il la coopération plutôt que la captation/compétition ? ou au contraire contraint-il l'usage des ressources (énergie, attention, données)

Interprétation : 5-6 = outil convivial ; ... <2 = logique de contrôle.

8.2 Documentation technique

8.2.1 Documentation Technique du Générateur de code

Cette section détaille le fonctionnement interne des fonctions principales du script `code-generator.js`. L'architecture repose sur un ensemble de fonctions orchestrées pour garantir la génération d'un code Python à la fois aléatoire, cohérent et pédagogiquement pertinent, en respectant scrupuleusement les options sélectionnées par l'utilisateur.

Schéma des appels de fonctions internes à l'exécution de la génération de code aléatoire

```
├── code-generator.js
│   │
│   └── generateRandomPythonCode(options)
│       │
│       ├── 1. FONCTIONS UTILITAIRES & DE SUPPORT (*)
│       │   ├── safeIndent                                  #1221
│       │   ├── shuffleArray                                #844
│       │   ├── getRandomInt, getRandomItem                 #73, #81
│       │   ├── getValueRange                               #68
│       │   ├── generateValueForType, generateValueOfType #306, #347
│       │   ├── generateUniqueVarName, generateUniqueIteratorName #141, #89
```

```
│       │   ├── chooseAppropriateParameterNames          #1203
│       │   ├── getDefaultValueForType                   #1423
│       │   └── declareVariable                          #168
│       │
│       ├── 2. LOGIQUE DE GÉNÉRATION CENTRALE (appelée par autres fonctions)
│       │   ├── generateVariedOperation                  #1743 (logique anti-
répétition incluse)
│       │   └── generateAppropriateStatement             #1225
│       │       └── generateVariedOperation*
│       │
│       ├── 3. PHASES D'EXÉCUTION (dans l'ordre d'appel)
│       │   │
│       │   ├── A. Phase de Préparation
│       │   │   ├── calculateRequiredLines               #1247
│       │   │   ├── generateInitialVariables             #366
│       │   │   ├── ensureVariablesForOptions            #1598
│       │   │   │   ├── ensureVariablesOfType            #1453
│       │   │   │   │   └── declareVariable*
│       │   │   │   └── ensureRequiredVariables          #1624
│       │   │   │       └── ensureVariableExists         #189
│       │   │   │           └── declareVariable*
│       │   │   ├── ensureListVariablesCount             #202
│       │   │   │   ├── generateDiverseList*
│       │   │   │   ├── declareVariable*
│       │   │   │   └── ensureListVariableIsUsed         #230
│       │   │   └── ensureTypeSpecificOperations         #1977
│       │   │       └── generateVariedOperation*
│       │   │
│       │   ├── B. Phase de Génération des Structures
│       │   │   └── generateControlStructures            #508
│       │   │       ├── shuffleArray*
│       │   │       ├── generateIfStatement              #848
│       │   │       │   ├── generateCondition
│       │   │       │   └── generateAppropriateStatement*
│       │   │       ├── generateForRangeLoop             #920
│       │   │       │   └── generateStructureBody
│       │   │       ├── generateForListLoop              #946
│       │   │       │   └── generateStructureBody
│       │   │       ├── generateForStrLoop               #986
│       │   │       │   └── generateStructureBody
│       │   │       ├── generateWhileLoop                #1019
│       │   │       │   ├── generateCondition
│       │   │       │   └── generateStructureBody
│       │   │       └── generateFunction                 #1060
│       │   │           ├── chooseAppropriateParameterNames*
│       │   │           ├── generateStructureBody
│       │   │           └── ensureVariableExists*
│       │   │
│       │   ├── C. Phase de Finalisation
│       │   │   ├── ensureAllVariablesAreUsed            #1441
│       │   │   │   └── generateVariedOperation*
```

```
│       │   │   ├── addFiller (boucle de remplissage)     #1650
│       │   │   │   └── generateVariedOperation*
│       │   │   └── finalVariableCheck                    #1384
│       │   │       ├── declareVariable*
│       │   │       └── getDefaultValueForType*
│       │   │
│       └── ***LEGACY*** (finalVariableCheck a un rôle résiduel)
├── main.js
│   └── ... (logique de l'interface utilisateur, gestion des options)
│
└── MyCFG.py
    └── ... (logique d'analyse du code Python et de génération du diagramme)
```

Portée : Ces variables sont déclarées au niveau supérieur de la fonction **generateRandomPythonCode**. Grâce au mécanisme de clôture (closure) de JavaScript, elles sont accessibles et partagées par toutes les fonctions auxiliaires imbriquées, agissant comme un état centralisé et encapsulé pour une unique session de génération de code.

Variables Clés :

- *options* : object. Objet contenant toutes les options de configuration booléennes et numériques récupérées depuis l'interface HTML. C'est la source de vérité pour toutes les décisions de génération !
- *codeLines* : Array<string>. Tableau servant d'accumulateur, chaque élément étant une ligne du code Python final. Le rôle des fonctions de génération est d'ajouter des lignes à ce tableau.
- *declaredVarsByType* : object. Structure de données centrale qui catalogue les noms de toutes les variables déclarées, classées par type. Exemple : `int: ['count', 'x'], str: ['name']` .
- *allDeclaredVarNames* : Set<string>. Ensemble stockant tous les noms de variables déjà utilisés. Sa nature de *Set* garantit l'unicité.
- *linesGenerated* : number. Compteur simple qui suit le nombre de lignes de code générées pour respecter la limite fixée par l'option `numLinesGlobal` qui définit l'objectif à atteindre *targetLines* : ni trop, ni trop peu.

Fonctions de Génération

1. **generateInitialVariables** #366

   Signature : `() -> void`

   Description : Initialise le script en créant les variables explicitement demandées par l'utilisateur via les options `var_..._count` . Ajoute à la liste *typesToGenerate* les types et la quantité correspondante `"type : count"` reçus de HTML ; Parcourt *typesToGenerate* jusqu'à atteindre la taille demandée (le nombre de variables "count") ; Génère un nom et une valeur selon le type, par appels à `generateUniqueVarName` et `generateValueForType` ; Ajoute l'initialisation aux lignes de code Python générées ; Met à jour les catalogues centralisés des variables déclarées *allDeclaredVarNames* et *declaredVarsByType*, et le compteur linesGenerated. Une boucle de sécurité répète tant que les nombres de variables et de lignes de code n'ont pas tous les deux atteints les tailles demandées. La condition de sécurité est exactement : `while (allDeclaredVarNames.size < Math.min(MAX_`

2. **generateUniqueVarName(type)** #141

   - Renvoie un nom de variable qui n'a pas encore été utilisé, parmi l'ensemble prédéfini, pour le type passé en argument
   - Construit un tableau des noms disponibles pour ce type, puis filtre ce tableau de noms en ne gardant que les noms qui n'ont pas encore été déclarés (c'est-à-dire qui ne sont pas présents dans le Set allDeclaredVarNames), et renvoie un nom choisi au hasard parmi ceux filtrés
   - Si tous les noms sont pris pour ce type, en crée un nouveau par ajout de suffixe numérique

3. **generateValueForType** #306

   Signature : `( type : string) -> string | number | boolean`

Description : Agit comme un simple dispatcher pour obtenir une valeur littérale. Il utilise le dictionnaire `LITERALS_BY_TYPE` qui associe chaque type à une *fonction anonyme* génératrice de valeur. *Note d'implémentation :* Les *fonctions* de ce dictionnaire capturent la variable *difficulty* depuis la portée parente, centralisant ainsi la gestion du niveau de complexité des valeurs générées.

4. `ensureVariablesForOptions` #1598
   - Pour chaque type parmi int, float, str, list, bool : si le compteur est strictement positif, alors appeler `ensureVariablesOfType` avec en paramètres le type et le compteur
   - Appelle `ensureRequiredVariables`
5. `ensureVariablesOfType` #1453
   - Idem `generateInitialVariables`, en version générique pour tout choix des arguments type et count
6. `ensureRequiredVariables` #1624
   - Fonction de sécurité pour gérer le cas particulier des loop_for_list et loop_for_str et cond_if sans que les types correspondants n'aient été choisis dans l'interface
   - Dans ces cas : appelle `ensureVariableExists` avec le type en paramètre
7. `ensureVariableExists(type)` #189
   - Pour le type passé en paramètre, renvoie soit un nom de variable existante (choisi au hasard parmi celles existantes de ce type dans `declaredVarsByType`) soit appelle `declareVariable` pour le type concerné
8. `declareVariable(type, value = null) -> name` #168
   - Pour le type passé en paramètre, renvoie un nom de variable généré par appel à `generateUniqueVarName` avec le type concerné en argument
   - Si une valeur lui est passée en argument elle l'utilise, sinon elle déclare un objet littéral grâce au **dictionnaire de fonctions** `LITERALS_BY_TYPE[type](difficulty, 'int')` DONC UNIQUEMENT mais bien dépendante de la difficulté
   - Pour `LITERALS_BY_TYPE` voir #46
9. `ensureListVariablesCount` #202
   - Boucle jusqu'à atteindre le compte attendu de listes : initialise une liste itemTypes avec ['int'] puis, si les options de variables ont été choisies par l'utilisateur, en ajoute une seule pour chacun des types selon 'str', 'bool' et/ou 'float'
   - Initialise une variable listValue par appel à `generateDiverseList(itemTypes,difficulty)` pour la passer en argument à `declareVariable` pour initialiser une variable listVar
10. `generateDiverseList(allowedTypes, difficulty) -> string` #310
    - Retourne des items séparés par des ', ' selon les allowedTypes et la difficulty passés en paramètres
    - Selon la difficulté choisie, calule une taille et définit aléatoirement si la liste sera homogène (pour difficulté 1 à 3 ou 70% de proba) ou non
    - Dans le cas hétérogène, choisit le type courant à utiliser de façon non aléatoire par `typesToUse[i % typesToUse.length]` pour générer la valeur de l'item courant par appel à `generateValueOfType`
11. `generateValueOfType` #347
    - Cas 'int' ou 'float' : retourne une valeur dans l'intervalle défini par la difficulté choisie et la fonction utilitaire `getValueRange`
    - Cas 'str' : retourne un mot aléatoire parmi ["alpha", "beta", "gamma", "delta", "epsilon", "kappa", "thet
    - Cas 'bool' : retourne au hasard 'True' ou 'False'
12. **ensureListVariableIsUsed** #230

    Signature : `( listVarName : string) -> void`

Description : S'assure qu'une liste est active dans le code. Elle analyse les lignes existantes pour détecter une "utilisation significative" (une boucle sur cette liste, un accès à une de ses valeurs par indexation [i], ou un appel à une méthode `append` ou `extend`). Si aucune n'est trouvée, elle ajoute une opération simple comme un `.append()` pour illustrer concrètement l'usage d'une liste. *Note d'implémentation :* Utilise un tableau de *fonctions anonymes* pour pouvoir générer des blocs de code potentiellement multi-lignes. Ci-dessous un exemple réel d'utilisation, suivi d'explications sur ce tableau :

```
const operations = [];
operations.push(() => {
    // Opération basique toujours disponible
    codeLines.push(`${listVarName}.append(${getRandomInt(1, 10)})`);
    linesGenerated++;
});
(...)
    // Exécuter une opération aléatoire
    getRandomItem(operations)();
```

Le type de la variable *operations* est appelé tableau de fonctions anonymes en vocabulaire JS. Ensuite l'appel `getRandomItem(operations)()` sélectionne et exécute une de ces "mini-recettes" de code en choisissant aléatoirement un élément parmi le tableau *operations* : `getRandomItem(operations` sélectionne une fonction aléatoire parmi ces fonctions et les () après getRandomItem(operations) exécutent cette fonction sélectionnée. La fonction exécutée contient elle-même des appels à codeLines.push() qui ajoutent du code au tableau codeLines. Cette technique permet à chaque fonction anonyme de générer un/des morceaux de code, avec des indentations et des structures conditionnelles complètes, sans se limiter à une seule ligne.

NB : Si la condition `isNotDeclaration && line.includes(listVarName) && isSignificantUse` est vraie pour au moins une ligne, alors `codeLines.some()` retourne true, donc *isUsed* devient true, et donc le bloc conditionnel ne s'exécute pas !

13. `generateControlStructures()` #508
    - Fonction d'orchestration qui va appeler les fonctions correspondantes aux structures demandées par l'utilisateur
    - Crée un tableau structures et y ajoute dans cet ordre les mots clés correspondants aux options choisies dans l'interface : 'if', 'for_range', 'for_list', 'for_str', 'while' et 'function', puis appelle shuffleArray dessus, et parcourt le tableau mélangé pour appeler dans chacun des cas la fonction "generatestructure" appropriée pour chacun des mots-clés
14. **ensureAllVariablesAreUsed** #1441

    Signature : `() -> void`

    Description : Assure une bonne pratique de codage en évitant les variables inutilisées. Si une variable est jugée "inerte" (déclarée mais jamais lue), la fonction lui donne alors un rôle en ajoutant une opération simple si le nombre de lignes n'a pas atteint le nombre attendu, en appelant **generateVariedOperation** pour le niveau de difficulté reçu du scope global, et ajoute l'opération au script.
15. **generateVariedOperation** #1743

    Signature : `( type : string, varName : string, difficulty : number) -> string`

    Description : Moteur de génération d'opérations atomiques (une/des lignes de code). Il utilise une table de dispatch pour sélectionner une opération possible en fonction du `type` et de la `difficulty`. Il intègre une logique anti-répétition et des patchs de sécurité (ex : détection d'opérations invalides sur les chaînes) pour garantir la qualité et la variété du code.

    Étape 1 : Crée d'abord une table de dispatch (un objet littéral contenant des fonctions fléchées) où chaque type passé en paramètre pointe vers un tableau de fonctions qui génèrent des opérations adaptées à ce type, ce qui permet de construire `availableOps = operations[type]` et de choisir au hasard parmi ce tableau un opération disponbile

Étape 2 : Essaie d'ajouter une opération différente des/de la précédente plusieurs fois, si toujours sans succès : ajoute un commentaire très improbable, pris aléatoirement parmi les caractères alphanumériques (ligne #1870)

Étape 3 : Ajoute un patch regex pour détecter les opérations arithmétiques invalides sur 'str' et les remplacer à la volée par une opération valide, sélectionnée aléatoirement parmi un tableau (ligne #1940)

16. `generateIfStatement` #848
17. `generateForRangeLoop` #920
18. `generateForListLoop` #946
19. `generateForStrLoop` #986
20. `generateWhileLoop` #1019
21. `generateFunction` #1060
22. `generateCondition(varTypes = ['int', 'bool', 'str', 'list'], preferExisting = true)` #409
    - Ajoute au tableau possibleConditions des conditions définies manuellement selon les types rencontrés dans varTypes passé en paramètre, puis en choisis une au hasard parmi possibleConditions.
    - Plan B : si aucune variable existante ne peut être utilisée (exemple : le générateur de boucle while demande une condition while_safe qui ne peut être créée avec les variables existantes), le bloc de 'fallback' va ajouter au code généré une déclaration/initialisation d'une nouvelle variable et une condition simple l'utilisant ('bool' = True|False ou 'int' > 0). Sans oublier de mettre à jour les structures declaredVarsByType, allDeclaredVarNames et le compteur linesGenerated après avoir incrémenté le compteur pour l'ajout de la variable.
23. `generateStructureBody(indentLevel, contextType, contextOptions = {})` #551
    - Fonction appelée par les générateurs de structures pour créer un corps fait d'un nombre d'instructions variable selon la difficulté, situé par indentLevel, pour une structure identifiée par contextType (qui peut être for_range, for_list, for_str ou function). Différentes logiques sont implémentées pour essayer de créer un équilibre entre des opérations de base garanties, l'introduction progressive de complexité, et une variété d'opérations qui restent asez simples pour remplir le corps de la boucle.
    - Explications sur le mapping contextOptions passé en paramètre : c'est le moyen de passer un ensemble de paramètres spécifiques au contexte (exemples : le nom de la variable de boucle loopVar, ou si un return est attendu) à la fonction generateStructureBody, sans avoir à lister une dizaine de paramètres dans sa signature !
    - Le cas function est le context le plus complexe évidemment : la fonction se base sur le type du premier paramètre pour ajouter des opérations adaptées, écrites manuellement chacune dans l'objectif de proposer aux élèves des syntaxes variées et valides pour des opérations sur 'str', 'list', 'bool' et 'int'|'float'.
    - Les cas des boucles 'for' utilisent une première instruction basique typique (addition d'un entier pour une variable dans un range, ajout d'un caractère pour une variable parcourant une chaine) suivie d'instructions choisies aléatoirement. Pour l'instruction basique de début de corps de boucle 'for_range' nous avons au contraire été amenés à pallier la forte probabilité de répétition d'instructions identiques en ajoutant la génération d'un commentaire improbable, pour assurer l'unicité de chacune des lignes :

      ```
      // Vérifier si cette opération est déjà présente dans le corps
      if (addedOperations.has(operation)) {
          // Ajouter un commentaire unique pour la rendre différente
          const uniqueId = Math.random().toString(36).substring(2, 5);
          operation = operation.replace(/\s*#.*$/, '') + `  # var_${uniqueId}`;
                      }
      ```

    - Dans la suite du corps (de boucle ou de fonction) les instructions sont générées par appel à generateVariedOperation. Cette dernière fonction intègre déjà une logique aléatoire avec un vrai évitement des répétitions en choisissant une structure différente, plus robuste que le "patch" ci-dessus qui rend unique une instruction par simple ajout d'un commentaire.

24. **finalVariableCheck** #1384

Signature : `() -> void`

Description : *Rôle legacy.* Cette fonction était un garde-fou dans les anciennes versions pour s'assurer que les variables "planifiées" étaient bien déclarées. Avec la nouvelle architecture où les variables sont déclarées immédiatement, son rôle est devenu mineur et elle est conservée comme une sécurité résiduelle.

8.2.2 Documentation technique de l'implémentation MyCFG.py

Comparaison avec l'implémentation PyCFG

8.2.3 Comparaison : MyCFG ≠ PyCFG

Comme mentionné plus haut, plusieurs sources disponibles fournissent des exemples intéressants d'utilisation du module `ast`. Ce module est en effet fourni avec les implémentations standards de Python et avec Pyodide, ce qui nous permis d'en tirer profit dans notre environnement initialement uniquement *front-end*. D'autres auteurs - comme Rahul Gopinath qui a contribué au projet FuzzingBook - ont fourni la preuve de la pertinence de l'outil `ast`, avec différents projets sur Github (Gopinath, 2024). Andreas Zeller est notamment l'auteur des classes CFGNode et PyCFG dont nous nous sommes directement inspiré. Qu'il en soit remercié ! Je cite le projet FuzzingBook ici :

> CFGNode representing each node in the control flow graph (...) Next, the PyCFG class which is responsible for parsing, and holding the graph.

En hommage et en clin d'œil aux auteurs de cette approche nous avons nommé notre fichier MyCFG.py, contenant une seule classe (hypertrophiée) appelée ControlFlowGraph. Par commodité présentons sous forme de tableau notre approche MyCFG à l'approche originale PyCFG. **Ci-dessous : le tableau comparatif**

Table 3 : Comparaison des logiques de construction de graphe de flot de contrôle

Aspect	MyCFG	PyCFG
Philosophie générale	Analyse directe de l'AST Python, au plus proche de la syntaxe du code source, approche qui se veut pédagogique car tolérante (accepte des chaînes non valides avec `return` hors d'une définition)	Analyse plus sémantique, mais `lineno` visualisé à chaque nœud ; reconstruit le flot d'exécution réel, "désucre" les structures (ex : for → while + next), pédagogique mais plus exigente au sens où elle révèle des "détails", plus bas niveau !
Entrée acceptée	Une chaîne de caractères représentant un code Python **syntaxiquement valide** (y compris invalide, tant que l'AST peut être construit)	Uniquement du code Python valide (car lorsqu'il rencontre un return, il cherche à rattacher ce nœud à la fonction englobante en remontant la pile (liste Python) des parents, donc si le return n'est pas dans une fonction (cf. mes `exemples.py` comme dans NestedIf), la pile des parents finit par être vide, ce qui provoque un `IndexError` et pas un `SyntaxError: 'return' outside function`
Construction du graphe	Plusieurs visites de l'AST, chaque structure (If, For, While, etc.) a sa méthode dédiée (`visit_If`, `visit_For`, ...)	Similaire (cf. classe `PyCFG`) avec méthodes nommées `on_NodeType` (ex : `on_if`, `on_for`, ...) pour chaque type de nœud AST

Table suite en page suivante

Suite du tableau **??**

Aspect	MyCFG	PyCFG
Gestion des branches et jonctions	Ajoute explicitement des nœuds de jonction pour les structures de contrôle (type `Junction` pour chaque If, For et While), gère ce qui est sensé représenter une sortie terminale (`Return`) ; A MODIFIER : applique la même gestion à `Break` et `continue`	Suit le flot réel, relie les nœuds selon l'exécution, gère les retours et les sorties via la structure du graphe et la pile de parents. Pas de nœud terminal `End` pour le flowchart, autre que les `return` de la fonction définie.
Gestion des boucles	Représente les boucles telles qu'elles apparaissent dans le code source (ex : `For i in ...`)	Transforme les boucles `for` en séquence équivalente (syntaxe un peu cryptique : `iter`, `next`, `while`), chaque étape devient un nœud
Gestion des appels de fonction (hors définition de fonction)	Si l'appel est interprété comme tel par l'AST alors affiché de façon spécifique (parallélogramme), sinon affectation simple (rectangle), NB : rendu visuel différent selon que l'appel est avant ou après la déf° (cf. exemples `defif` et `defif2`). POUR LE MOMENT : pas différencié appels internes/externes, mais parait faisable	Appels de fonctions natives sont comme affectations classiques (rectangles simples) ; appels des fonctions internes : suivis par le flux de la définitino de fonction. NB : `return` est un ovale "exit", même si suivi par du code.
Gestion des définitions de fonctions	Problématique quand le code est uniquement une définition de fonction ! Les nœuds `Start` et `End` sont alors artificiels... logique à redéfinir ??	Impeccable. Caveat idem ci-dessus !
Tolérance aux erreurs	Tolère des return en dehors des fonctions	Requiert du code Python syntaxiquement valide pour fonctionner (ex : expressions avec variables non intitialisées ne lèvent psa d'exception)
Sortie / Visualisation	Génère une chaine en syntaxe Mermaid (un DOT language similaire à Graphviz) mieux adaptée pour un rendu Web il me semble : plus facile à maintenir	Génère du DOT-Graphviz : infrastrucutre plus lourde ! à décortiquer pour rendre un Mermaid.js mais visualisation avancée, couleurs et styles selon le flot
Gros Pour	plus souple = plus proche de ce qui est fait en classe ; infrastructure légère	visualisation plus propre (eg : flux principal vs `def`)
Gros Contre	encore bugs et incertitudes...	c'est Digraph qui fait le graph !!

8.2.4 Documentation technique de l'implémentation MyCFG.py

La classe Python `ControlFlowGraph` est le composant central pour la transformation du code Python généré (une chaine) en un *flowchart* (une chaine Mermaid rendue graphiquement par l'interface).

`__init__`

Signature : `(__init__(self, code: str)) -> None`

Description : Constructeur de la classe. Initialise l'analyseur AST avec le code Python fourni et prépare les structures de données internes pour stocker les informations du graphe de flux de contrôle (ci-après *flowchart*). Cette méthode est appelée une seule fois lors de la création d'une instance de `ControlFlowGraph`.

Variables d'instance initialisées :

- *tree* : ast.Module. L'arbre syntaxique abstrait (AST) généré à partir du `code` source. C'est la structure de base que les méthodes de visite parcourront.
- *nodes* : List[Tuple[str, str]]. Liste de tuples, où chaque tuple représente un nœud du CFG sous la forme `(node_id, label)`. `node_id` est un identifiant unique et `label` est le texte affiché

dans la forme (rectangulaire ou autres). Seuls les nœuds de jonctino, techniques, n'ont pas ce texte.

- *edges* : Set[Tuple[str, str, str]]. Ensemble de tuples représentant les arêtes du CFG. Chaque tuple est de la forme `(from_node_id, to_node_id, label)`, où `label` est le texte affiché sur l'arête (peut être vide). Utiliser un ensemble évite les arêtes dupliquées.
- *node_counter* : int. Compteur entier utilisé par `get_node_id` pour générer des identifiants de nœuds uniques (par exemple, "node01", "node02"). Initialisé à 0.
- *loop_stack* : List[Tuple[str, str, str]]. Pile utilisée pour gérer le contexte des boucles imbriquées. Chaque élément est un tuple `(continue_target_id, break_target_id, retest_target_id)` stockant les ID des nœuds cibles pour les instructions `continue`, `break`, et pour le retour au test de la condition de boucle. NB : les `break` ne sont pas encore bien implémentés même si l'arête est générée !
- *node_labels* : Dict[str, str]. Dictionnaire associant un `node_id` à son `label` textuel. Permet un accès rapide au label d'un nœud.
- *terminal_nodes* : Set[str]. Ensemble des `node_id` qui représentent la fin d'un chemin d'exécution normal (par exemple, un nœud créé pour une instruction `return`, `break`, ou `continue`).
- *node_types* : Dict[str, str]. Dictionnaire associant un `node_id` à son type sémantique (par exemple, "Process", "Decision", "StartEnd", "Junction"). Utilisé pour le style dans le rendu Mermaid, en évitant d'utiliser des mots-clés déjà pris.
- *_function_scope_stack* : List[Set[str]]. Pile interne pour gérer les portées des fonctions (imbriquées ou non). La **"portée d'une fonction" fait ici référence à l'ensemble des nœuds du graphe de flux de contrôle (CFG) qui sont générés** lors de l'analyse du corps de cette fonction spécifique. La principale utilité est de permettre à la méthode `connect_finals_to_end` de fonctionner correctement. Quand on est à la fin de la visite d'une fonction (dans `visit_FunctionDef`), on veut s'assurer que tous les chemins d'exécution à l'intérieur de cette fonction qui n'ont pas explicitement de successeur (fins de chemin implicites ou nœuds Return) soient connectés au nœud End NomFonction de cette fonction spécifique, et non au End du module. Cela permet d'identifier quels nœuds appartiennent à quelle fonction pour les regrouper visuellement dans des `subgraph` distincts. Chaque élément de la liste est un ensemble d'ID de nœuds appartenant à une portée de fonction spécifique. Lorsqu'on commence à visiter une fonction (visit_FunctionDef), un nouvel ensemble vide (set()) est poussé sur cette pile. **Explications** : Chaque fois qu'un nouveau nœud CFG est créé (get_node_id) pendant que cette fonction est en cours de visite (c'est-à-dire, pendant que son ensemble est au sommet de la pile), l'ID de ce nouveau nœud est ajouté à cet ensemble au sommet de la pile. Cela gère correctement les fonctions imbriquées : si func_A contient func_B, lors de la visite de func_A, son scope est sur la pile. Quand visit_FunctionDef est appelée pour func_B, un nouveau scope pour func_B est poussé par-dessus celui de func_A. Les nœuds de func_B vont dans le scope de func_B. Quand la visite d'une fonction est terminée, son ensemble d'ID de nœuds est retiré (poppé) de la pile et stocké dans `self.all_function_scopes` (qui conserve tous les scopes de fonction finalisés). La dernière portée de la liste est la plus interne.
- *function_subgraph_nodes* : Dict[str, Set[str]]. Dictionnaire stockant, pour chaque nom de fonction, l'ensemble des ID de nœuds qui lui appartiennent. Utilisé par `to_mermaid` pour créer les sous-graphes.
- *main_flow_nodes* : Set[str]. Ensemble des ID de nœuds appartenant au flux principal du module (hors définitions de fonctions). Utilisé par `to_mermaid` pour le sous-graphe principal.

get_node_id

Signature : `(get_node_id(self) -> str)`

Description : Génère un identifiant de nœud unique basé sur *node_counter*. Si l'appel se produit dans le contexte d'une fonction (c'est-à-dire si *_function_scope_stack* n'est pas vide), l'ID du nouveau nœud est ajouté à l'ensemble des nœuds de la portée de fonction la plus interne. Sinon, il est ajouté à *main_flow_nodes*. Cette méthode est appelée par `add_node` chaque fois qu'un nouveau nœud est créé.

Variables locales/modifiées :
- *new_id* : str. L'identifiant de nœud nouvellement généré (par exemple, "node01").
- Modifie *self.node_counter* (incrémenté).
- Modifie potentiellement *self._function_scope_stack[-1]* ou *self.main_flow_nodes* en y ajoutant *new_id*.

add_node

Signature : `(add_node(self, label: str, node_type: str = "Process") -> str)`

Description : Crée un nouveau nœud avec le `label` et le `node_type` fournis, l'ajoute aux structures de données internes du graphe (*self.nodes*, *self.node_labels*, *self.node_types*), et retourne l'ID unique du nœud créé. Appelée par les différentes méthodes de visite lorsqu'une instruction ou une structure de contrôle doit être représentée par un nœud.

Paramètres :
- `label` : str. Le texte à afficher sur le nœud.
- `node_type` : str. Le type sémantique du nœud (défaut : "Process").

Variables locales/modifiées :
- *node_id* : str. L'ID du nœud créé, obtenu via `get_node_id`.
- Modifie *self.nodes*, *self.node_labels*, *self.node_types*.

add_edge

Signature : `(add_edge(self, from_node: str, to_node: str, label: str = "") -> None)`

Description : Ajoute une arête dirigée entre deux nœuds identifiés par `from_node` et `to_node`, avec un `label` optionnel. Effectue des vérifications pour éviter les arêtes invalides (nœuds non existants, auto-boucles non labellisées, arêtes sortant de nœuds terminaux non autorisés). Appelée par les méthodes de visite pour connecter les nœuds représentant le flux de contrôle.

Paramètres :
- `from_node` : str. L'ID du nœud de départ de l'arête.
- `to_node` : str. L'ID du nœud d'arrivée de l'arête.
- `label` : str. Le texte à afficher sur l'arête (défaut : chaîne vide).

Variables locales/modifiées :
- Modifie *self.edges* en y ajoutant le tuple de l'arête.

_is_terminal_ast_node

Signature : `(_is_terminal_ast_node(self, node: ast.AST) -> bool)`

Description : Petite méthode utilitaire vérifiant si un `node` AST donné est un type d'instruction qui termine le flux d'exécution normal (i.e. présentement : `ast.Return`, `ast.Break`, `ast.Continue`). Appelée par la méthode `visit` pour déterminer si les nœuds CFG créés doivent être marqués comme terminaux.

Paramètres :
- `node` : ast.AST. Le nœud AST à inspecter.

visit_body

Signature : `(visit_body(self, body: List[ast.AST], entry_node_ids: List[str]) -> List[str])`

Description : Traite une séquence d'instructions (un "corps" de code, comme le corps d'une fonction, d'une boucle ou d'une branche conditionnelle). Elle itère sur chaque instruction du `body`, la visite en utilisant les `entry_node_ids` comme points d'entrée, et gère la création de nœuds de jonction si une instruction produit plusieurs sorties non terminales et n'est pas la dernière du corps. Retourne une liste des ID de nœuds qui constituent les points de sortie finaux de ce corps. Appelée par `visit_Module`, `visit_FunctionDef`, `visit_If`, `visit_For`, `visit_While`.

Paramètres :
- `body` : List[ast.AST]. La liste des nœuds AST représentant les instructions du corps.
- `entry_node_ids` : List[str]. Liste des ID des nœuds CFG qui servent de points d'entrée pour la première instruction de ce corps.

Variables locales/modifiées :
- *active_ids_for_current_statement* : List[str]. Liste des ID de nœuds actifs qui servent d'entrée à l'instruction en cours de traitement ou de sortie de l'instruction précédente. Initialisée avec `entry_node_ids`, puis mise à jour après chaque instruction.
- *i* : int. Index de l'instruction en cours dans la boucle sur `body`.
- *stmt* : ast.AST. L'instruction AST en cours de traitement.
- *current_stmt_entry_points* : List[str]. Sous-ensemble de *active_ids_for_current_statement* ne contenant que les nœuds non terminaux, servant d'entrée à l'appel de `visit(stmt, ...)`.
- *exits_from_current_stmt_all_paths* : List[str]. Liste collectant tous les ID de nœuds de sortie retournés par `visit(stmt, ...)` pour tous ses points d'entrée.
- *parent_id* : str. Un ID de nœud de *current_stmt_entry_points*, passé comme parent à `visit(stmt, pare`
- *exit_nodes_from_stmt_path* : List[str]. Les ID de nœuds de sortie retournés par un appel spécifique à `visit(stmt, parent_id)`.
- *is_last_statement* : bool. Indicateur si *stmt* est la dernière instruction du `body`.
- *non_terminal_active_ids* : List[str]. Sous-ensemble de *active_ids_for_current_statement* (après traitement de *stmt*) ne contenant que les nœuds non terminaux. Utilisé pour décider de créer une jonction.
- *junction_id* : str. ID du nœud de jonction créé si nécessaire. Initialisé uniquement si plus d'un *non_terminal_active_ids* existe et que ce n'est pas la dernière instruction.
- *pid* : str. Un ID de nœud de *non_terminal_active_ids*, utilisé pour connecter ces nœuds à *junction_id*.
- *terminal_active_ids* : List[str]. Sous-ensemble de *active_ids_for_current_statement* contenant les nœuds terminaux. Conservés lors de la création d'une jonction.

visit

Signature : `(visit(self, node: ast.AST, parent_id: Optional[str]) -> List[str])`

Description : Méthode de répartition principale (dispatcher) qui dirige la visite d'un `node` AST vers la méthode `visit_<NodeType>` appropriée (par exemple, `visit_If` pour un `ast.If`). Si aucune méthode spécifique n'existe, elle utilise `generic_visit`. Après l'appel au visiteur spécifique, si le `node` AST est de type terminal (Return, Break, Continue), elle marque les nœuds CFG créés comme terminaux et retourne une liste vide (indiquant aucune continuation de flux normale). Sinon, elle retourne la liste des ID de nœuds de sortie produits par le visiteur. Appelée récursivement pour parcourir l'AST.

Paramètres :
- `node` : ast.AST. Le nœud AST à visiter.
- `parent_id` : Optional[str]. L'ID du nœud CFG parent auquel le(s) nouveau(x) nœud(s) créé(s) pour `node` doi(ven)t être connecté(s). Peut être `None` pour le nœud Module ou les FunctionDef de haut niveau.

Variables locales/modifiées :
- *method_name* : str. Le nom de la méthode de visite spécifique (par exemple, "visit_If").
- *visitor* : Callable. La référence à la méthode de visite à appeler.
- *exit_nodes* : List[str]. Liste des ID de nœuds de sortie retournés par la méthode *visitor*.
- *node_id* (dans la section terminal) : str. Un ID de nœud de *exit_nodes*, utilisé pour l'ajouter à *self.terminal_nodes*.
- Modifie potentiellement *self.terminal_nodes*.

connect_finals_to_end

Signature : `(connect_finals_to_end(self, target_end_id: str, scope_node_ids: Optional[Set[str`

Description : Parcourt les nœuds spécifiés par `scope_node_ids` (ou tous les nœuds si `None`). Si un nœud dans cette portée n'a pas d'arête sortante ou s'il s'agit d'un nœud "Return", une arête est ajoutée de ce nœud vers `target_end_id`. Cela garantit que tous les chemins d'exécution (y compris

les retours implicites et explicites) se terminent correctement au nœud "End" approprié (de module ou de fonction). Appelée par `visit_Module` et `visit_FunctionDef`.

Paramètres :
- `target_end_id` : str. L'ID du nœud "End" auquel connecter les nœuds finaux.
- `scope_node_ids` : Optional[Set[str]]. Ensemble optionnel d'ID de nœuds à considérer. Si `None`, tous les nœuds du graphe sont vérifiés.

Variables locales/modifiées :
- *source_nodes_with_outgoing_edges* : Set[str]. Ensemble des ID de nœuds qui ont au moins une arête sortante.
- *nodes_to_check* : Set[str]. L'ensemble effectif des ID de nœuds à inspecter.
- *node_id* : str. L'ID du nœud en cours d'inspection dans la boucle.
- *is_return_node* : bool. Vrai si le *node_id* actuel est de type "Return".
- Modifie *self.edges* en ajoutant des arêtes vers `target_end_id`.

visit_Module

Signature : `(visit_Module(self, node: ast.Module, parent_id: Optional[str] = None) -> List[st`

Description : Gère la visite du nœud racine du module (`ast.Module`). Crée les nœuds "Start" et "End" globaux. Sépare les définitions de fonctions des autres instructions. Visite d'abord les définitions de fonctions (qui génèrent leurs propres sous-graphes isolés), puis visite les autres instructions pour construire le flux principal du module. Enfin, connecte les points de sortie du flux principal et les fins implicites au nœud "End" global. Appelée une seule fois, au début du processus de visite, avec `parent_id` à `None`.

Paramètres :
- `node` : ast.Module. Le nœud AST du module.
- `parent_id` : Optional[str]. Toujours `None` pour cette méthode.

Variables locales/modifiées :
- *start_id* : str. ID du nœud "Start" du module.
- *function_defs* : List[ast.FunctionDef]. Liste des nœuds AST de définition de fonction dans le module.
- *other_statements* : List[ast.AST]. Liste des autres nœuds AST (non-FunctionDef) dans le module.
- *func_def_node* (dans la boucle) : ast.FunctionDef. Une définition de fonction en cours de visite.
- *module_flow_exits* : List[str]. Liste des ID de nœuds de sortie du flux principal du module, retournée par `visit_body(other_statements, ...)`.
- *module_end_id* : str. ID du nœud "End" du module.
- *node_id* (dans la boucle de connexion) : str. Un ID de nœud de *module_flow_exits*.

visit_FunctionDef

Signature : `(visit_FunctionDef(self, node: ast.FunctionDef, parent_id: Optional[str]) -> None`

Description : Gère la visite d'un nœud de définition de fonction (`ast.FunctionDef`). Crée une nouvelle portée de fonction, puis les nœuds "Start NomFonction" et "End NomFonction" pour le corps de cette fonction. Visite le corps de la fonction. Collecte tous les ID de nœuds créés dans cette portée, les stocke dans *self.function_subgraph_nodes* pour le rendu en sous-graphe, puis dépile la portée. Connecte les points de sortie normaux du corps et les nœuds "Return" au nœud "End NomFonction" de cette fonction. Ne s'insère pas dans le flux du parent et retourne une liste vide. Appelée par `visit_Module` ou par un autre `visit_FunctionDef` (pour les fonctions imbriquées).

Paramètres :
- `node` : ast.FunctionDef. Le nœud AST de la définition de fonction.
- `parent_id` : Optional[str]. L'ID du nœud parent. `None` si c'est une fonction de haut niveau, ou l'ID du nœud de la fonction englobante si elle est imbriquée (Mise à jour : la version actuelle ne crée plus de nœud de déclaration dans le flux parent... seul subsiste l'appel de fonction dans le flux parent, la déclaration est représentée par un sous-graphe pour la définition de fonction).

Variables locales/modifiées :
- *func_body_start_id* : str. ID du nœud "Start NomFonction".
- *func_body_end_id* : str. ID du nœud "End NomFonction".

- *body_normal_exit_nodes* : List[str]. Liste des ID de nœuds de sortie normaux du corps de la fonction.
- *node_id* (dans la boucle de connexion) : str. Un ID de nœud de *body_normal_exit_nodes*.
- *current_function_node_ids* : Set[str]. Ensemble des ID de nœuds appartenant à la fonction en cours, récupéré de *self._function_scope_stack* avant le `pop()`.
- Modifie *self._function_scope_stack* (push et pop).
- Modifie le Dict *self.function_subgraph_nodes* en y ajoutant une entrée pour la clé `node.name` ayant pour valeur *current_function_node_ids*.

visit_If

Signature : `(visit_If(self, node: ast.If, parent_id: str) -> List[str]: final_exit_nodes_afte`

Description : Gère la visite d'une instruction conditionnelle `if` (`ast.If`). Crée un nœud de décision pour la condition. Visite ensuite la branche `body` (si elle existe) et la branche `orelse` (si elle existe, qui peut être un `else` ou un `elif`). Labellise les arêtes sortant du nœud de décision avec "True" et "False" (ou les labels appropriés pour `elif`). Retourne une liste des ID de nœuds de sortie de la structure `if` globale, qui seront potentiellement fusionnés par `visit_body`. Appelée par `visit_body` lorsqu'une instruction `if` est rencontrée.

Paramètres :
- `node` : ast.If. Le nœud AST de l'instruction `if`.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *condition_text* (renommée de *condition*) : str. Le texte de la condition du `if`.
- *if_decision_id* (renommée de *if_id*) : str. ID du nœud de décision pour la condition.
- *final_exit_nodes_after_if* : List[str]. Liste collectant les ID de nœuds de sortie de la structure `if`.
- *true_branch_first_node_id* (renommée de *true_branch_start_node*) : Optional[str]. ID du premier nœud créé dans la branche `body` (True).
- *false_branch_first_node_id* (renommée de *false_branch_start_node*) : Optional[str]. ID du premier nœud créé dans la branche `orelse` (False).
- *nodes_before_true_branch* : Set[str]. Ensemble des ID de nœuds existants avant de visiter la branche `body`.
- *true_branch_exits* : List[str]. Liste des ID de nœuds de sortie de la branche `body`.
- *nodes_after_true_branch* : Set[str]. Ensemble des ID de nœuds existants après avoir visité la branche `body`.
- *new_nodes_in_true_branch* : List[str]. Liste triée des ID de nœuds créés spécifiquement dans la branche `body`.
- *nodes_before_false_branch* : Set[str]. Ensemble des ID de nœuds existants avant de visiter la branche `orelse`.
- *false_branch_exits* : List[str]. Liste des ID de nœuds de sortie de la branche `orelse`.
- *nodes_after_false_branch* : Set[str]. Ensemble des ID de nœuds existants après avoir visité la branche `orelse`.
- *new_nodes_in_false_branch* : List[str]. Liste triée des ID de nœuds créés spécifiquement dans la branche `orelse`.
- Modifie *self.edges* pour ajouter/supprimer des arêtes lors de la labellisation. Plus précisément, une arête *(if_decision_id, true_branch_first_node_id, "")* a été créée par le premier appel à `visit()` dans `visit_body`. Nous la supprimons ici pour la recréer avec le label "True", respectivement pour le label "False".

visit_For

Signature : `(visit_For(self, node: ast.For, parent_id: str) -> List[str])`

Description : Gère la visite d'une boucle `for` (`ast.For`). Pour satisfaire la volonté d'une représentation plus sémantique (initialisation, condition, incrémentation) on distingue les cas `for ... in range(...)` du cas d'un itérable générique (traité par `_visit_for_generic_iterable`). Gère la visite du corps de la boucle et de la clause `orelse` (si présente). Met à jour *self.loop_stack*. Retourne les points de sortie de la boucle. Appelée par `visit_body` .

Paramètres :
- `node` : ast.For. Le nœud AST de la boucle `for` .
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées (pour le cas `range`) :
- *iterator_variable_str* (renommée de *iterator_str*) : str. Nom de la variable d'itération.
- *iterable_node* : ast.AST. Le nœud AST représentant l'itérable (par exemple, l'appel à `range()`).
- *loop_overall_exit_points* : List[str]. Points de sortie de la boucle.
- *range_args* : List[ast.AST]. Arguments de la fonction `range()` .
- *start_val_str*, *stop_val_str*, *step_val_str* : str. Représentations textuelles des arguments de `range()` .
- *init_label* : str. Label du nœud d'initialisation de la variable de boucle.
- *init_node_id* : str. ID du nœud d'initialisation.
- *condition_op* : str. Opérateur de comparaison pour la condition de boucle (< ou >).
- *temp_step_for_eval* : str. Valeur du pas, purgée des '#quot' pour évaluation.
- *evaluated_step* : Union[int, float]. Valeur numérique du pas, si évaluable.
- *loop_condition_label* (renommée de *loop_cond_label*) : str. Label du nœud de condition.
- *loop_condition_id* (renommée de *loop_cond_id*) : str. ID du nœud de condition.
- *increment_label* : str. Label du nœud d'incrémentation.
- *increment_node_id* : str. ID du nœud d'incrémentation.
- *true_branch_first_node_id* (renommée de *true_branch_start_node*) : Optional[str]. ID du premier nœud du corps de la boucle.
- *nodes_before_body* : Set[str]. Nœuds avant la visite du corps.
- *body_exit_nodes* : List[str]. Sorties du corps de la boucle.
- *nodes_after_body* : Set[str]. Nœuds après la visite du corps.
- *new_nodes_in_body* : List[str]. Nouveaux nœuds dans le corps.
- *exit_node* (dans la boucle) : str. Un nœud de sortie du corps.
- *false_branch_first_node_id* (renommée de *false_branch_start_node*) : Optional[str]. ID du premier nœud de la clause `orelse` .
- *nodes_before_orelse* : Set[str]. Nœuds avant la visite de `orelse` .
- *orelse_exit_nodes* : List[str]. Sorties de la clause `orelse` .
- *nodes_after_orelse* : Set[str]. Nœuds après la visite de `orelse` .
- *new_nodes_in_orelse* : List[str]. Nouveaux nœuds dans `orelse` .
- Modifie *self.loop_stack* (push et pop).
- Modifie *self.edges*.

__visit__for__generic__iterable

Signature : `(_visit_for_generic_iterable(self, node: ast.For, parent_id: str, iterator_variab`

Description : Méthode pour gérer les boucles `for` avec des itérables autres que `range()` . Crée un unique nœud de décision pour la boucle. Gère le corps et la clause `orelse` . Met à jour *self.loop_stack*. Appelée par `visit_For` .

Paramètres :
- `node` : ast.For. Le nœud AST de la boucle `for` .
- `parent_id` : str. L'ID du nœud CFG parent.
- `iterator_variable_str` : str. Nom de la variable d'itération.

Variables locales/modifiées :
- *iterable_text* (renommée de *iterable_label*) : str. Texte de l'itérable.

- *loop_decision_label* : str. Label du nœud de décision.
- *loop_decision_id* : str. ID du nœud de décision.
- *loop_overall_exit_points* : List[str]. Points de sortie de la boucle.
- *iteration_branch_first_node_id* (renommée de *iteration_branch_start_node*) : Optional[str]. ID du premier nœud du corps.
- *nodes_before_body* : Set[str].
- *body_exit_nodes* : List[str].
- *nodes_after_body* : Set[str].
- *new_nodes_in_body* : List[str].
- *exit_node* (dans la boucle) : str.
- *terminated_branch_first_node_id* (renommée de *terminated_branch_start_node*) : Optional[str]. ID du premier nœud de `orelse`.
- *nodes_before_orelse* : Set[str].
- *orelse_exit_nodes* : List[str].
- *nodes_after_orelse* : Set[str].
- *new_nodes_in_orelse* : List[str].
- Modifie *self.loop_stack* (push et pop).
- Modifie *self.edges*.

Explication des modifications du 28/05/25 :

1. Ordre de Création des Nœuds :
   - entry_decision_id ("A des éléments à traiter ?")
   - init_var_id ("iter = premier élément")
   - retest_decision_id ("Encore un élément à traiter ?")
   - next_var_id ("iter = élément suivant")
2. Connexion entry_decision_id :
   - entry_decision_id –True–> init_var_id
   - La branche False de entry_decision_id devient une des sorties de la boucle (loop_overall_exit_points).
3. Visite du Corps (node.body) :
   - nodes_before_body = nid for nid, _ in self.nodes : On capture l'état des nœuds avant de visiter le corps.
   - body_exit_nodes = self.visit_body(node.body, [init_var_id]) : Le corps est visité en partant du nœud d'initialisation de la variable.
   - nodes_after_body = ..., new_nodes_in_body = ... : On identifie les nœuds qui ont été créés pendant la visite du corps.
   - first_node_of_body = new_nodes_in_body[0] : Le premier de ces nouveaux nœuds est considéré comme le début effectif du corps.
   - L'arête (init_var_id, first_node_of_body, "") (qui aurait été créée par le premier self.visit dans visit_body) est implicitement gérée car visit_body connecte init_var_id au premier statement. On s'assure qu'elle n'a pas de label "True" superflu.
4. Connexion des Sorties du Corps :
   - for exit_node in body_exit_nodes : self.add_edge(exit_node, retest_decision_id) : Les sorties normales du corps mènent au nœud de re-test.
5. Corps Vide :
   - Si node.body est vide, init_var_id est directement connecté à retest_decision_id.
6. Connexion retest_decision_id :
   - retest_decision_id –True–> next_var_id
   - La branche False de retest_decision_id devient une des sorties de la boucle (loop_overall_exit_points).
7. Retour au Corps depuis next_var_id :

- if first_node_of_body : self.add_edge(next_var_id, first_node_of_body) : Si on a pu identifier le premier nœud du corps, on connecte la mise à jour de la variable (next_var_id) à ce nœud.
- Sinon (corps vide ou problème d'identification), on met un fallback (moins précis).

8. Gestion de orelse :
   - Si node.orelse existe, il est visité en partant de retest_decision_id (conceptuellement, après que la condition "Encore un élément ?" soit fausse).
   - L'arête retest_decision_id –False–> premier_noeud_de_orelse est créée.
   - Les sorties de orelse deviennent des sorties de la boucle.

visit_While

Signature : `(visit_While(self, node: ast.While, parent_id: str) -> List[str])`

Description : Gère la visite d'une boucle `while` (`ast.While`). Crée un nœud de décision pour la condition. Visite le corps et la clause `orelse`. Met à jour *self.loop_stack*. Retourne les points de sortie de la boucle. Appelée par `visit_body`.

Paramètres :
- `node` : ast.While. Le nœud AST de la boucle `while`.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *condition_text* (renommée de *condition*) : str. Texte de la condition.
- *while_decision_id* (renommée de *while_id*) : str. ID du nœud de décision.
- *loop_overall_exit_points* : List[str].
- *true_branch_first_node_id* (renommée de *true_branch_start_node*) : Optional[str].
- *nodes_before_body* : Set[str].
- *body_exit_nodes* : List[str].
- *nodes_after_body* : Set[str].
- *new_nodes_in_body* : List[str].
- *exit_node* (dans la boucle) : str.
- *false_branch_first_node_id* (renommée de *false_branch_start_node*) : Optional[str].
- *nodes_before_orelse* : Set[str].
- *orelse_exit_nodes* : List[str].
- *nodes_after_orelse* : Set[str].
- *new_nodes_in_orelse* : List[str].
- Modifie *self.loop_stack* (push et pop).
- Modifie *self.edges*.

visit_Return

Signature : `(visit_Return(self, node: ast.Return, parent_id: str) -> List[str])`

Description : Gère la visite d'une instruction `return` (`ast.Return`). Crée un nœud "Return" avec la valeur retournée (si présente). Retourne l'ID de ce nœud. La méthode `visit` marquera ensuite ce nœud comme terminal. Appelée par `visit_body`.

Paramètres :
- `node` : ast.Return. Le nœud AST du `return`.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *value_text* (renommée de *value*) : str. Texte représentant la valeur retournée.
- *return_node_id* (renommée de *return_id*) : str. ID du nœud "Return".

visit_Break : en chantier

Signature : `(visit_Break(self, node: ast.Break, parent_id: str) -> List[str])`

Description : Gère la visite d'une instruction `break` (`ast.Break`). Crée un nœud "Break". Ne connecte pas explicitement ce nœud à la sortie de la boucle ici ; cela est géré par le fait que le nœud est terminal et que la boucle a des points de sortie définis. Retourne l'ID de ce nœud. La méthode `visit` le marquera comme terminal. Appelée par `visit_body`.

Paramètres :
- `node` : ast.Break. Le nœud AST du `break`.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *break_node_id* (renommée de *break_id*) : str. ID du nœud "Break".
- *loop_exit_target* : str. Serait l'ID du nœud cible du `break`, récupéré de *self.loop_stack*. Actuellement non utilisé pour créer une arête explicite (commenté dans le code actuel).

visit_Continue

Signature : `(visit_Continue(self, node: ast.Continue, parent_id: str) -> List[str])`

Description : Gère la visite d'une instruction `continue` (`ast.Continue`). Crée un nœud "Continue". Si dans une boucle, connecte ce nœud à la cible de "continue" de la boucle (obtenue de *self.loop_stack*). Retourne l'ID de ce nœud. La méthode `visit` le marquera comme terminal (i.e. qu'il n'y aura pas de continuation du flux normal). Appelée par `visit_body`.

Paramètres :
- `node` : ast.Continue. Le nœud AST du `continue`.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *continue_node_id* (renommée de *continue_id*) : str. ID du nœud "Continue".
- *loop_continue_target* : str. ID du nœud cible du `continue`, récupéré de *self.loop_stack*.
- Modifie *self.edges* si une arête de saut est ajoutée.

generic_visit

Signature : `(generic_visit(self, node: ast.AST, parent_id: str) -> List[str])`

Description : Visiteur par défaut pour les types de nœuds AST qui n'ont pas de méthode `visit_<NodeType>` spécifique. Tente de créer un nœud "Process" en utilisant la représentation textuelle du nœud AST (via `ast.unparse`) comme label. Si `ast.unparse` échoue, utilise le nom du type de nœud comme label. Connecte le nouveau nœud au `parent_id`. Appelée par `visit` si aucun visiteur spécifique n'est trouvé.

Paramètres :
- `node` : ast.AST. Le nœud AST à visiter.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *label_text* (renommée de *label*) : str. Label du nœud.
- *node_type* : str. Type du nœud (par défaut "Process").
- *max_label_length* (renommée de *max_len*) : int. Longueur maximale du label (actuellement 60 caractères).
- *new_node_id* (renommée de *node_id*) : str. ID du nœud créé.

visit_Assign

Signature : `(visit_Assign(self, node: ast.Assign, parent_id: str) -> List[str])`

Description : Gère la visite d'une instruction d'assignation (`ast.Assign`). Crée un nœud "Process" dont le label est la représentation textuelle de l'assignation (par exemple, "x = y + 1"). Gère le raccourcissement des labels trop longs. Appelée par `visit_body`.

Paramètres :
- `node` : ast.Assign. Le nœud AST de l'assignation.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *targets_str* : str. Représentation textuelle des cibles de l'assignation.
- *value_str* : str. Représentation textuelle de la valeur assignée.
- *label_text* (renommée de *label*) : str. Label complet du nœud.
- *node_type* : str. Type du nœud ("Process").
- *max_label_length* (renommée de *max_len*) : int.
- *available_len_for_value* : int. Espace disponible pour la partie valeur du label si raccourcissement.
- *short_value* : str. Version raccourcie de *value_str*.
- *assign_node_id* (renommée de *assign_id*) : str. ID du nœud créé.

visit_Expr

Signature : `(visit_Expr(self, node: ast.Expr, parent_id: str) -> List[str])`

Description : Gère la visite d'une instruction d'expression (`ast.Expr`). Une telle instruction évalue une expression (souvent un appel de fonction comme `print()` ou un appel à une fonction définie par l'utilisateur qui ne retourne rien d'utilisé). Cette méthode délègue simplement la visite à la valeur de l'expression (`node.value`). Appelée par `visit_body` .

Paramètres :
- `node` : ast.Expr. Le nœud AST de l'expression.
- `parent_id` : str. L'ID du nœud CFG parent.

visit_Call

Signature : `(visit_Call(self, node: ast.Call, parent_id: str) -> List[str])`

Description : Gère la visite d'un appel de fonction (`ast.Call`). Crée un nœud (généralement "Process" ou "IoOperation" pour `print/input`) dont le label représente l'appel de fonction avec ses arguments. Gère le raccourcissement de la liste d'arguments si elle est trop longue. Appelée par `visit_Expr` (si l'appel est une instruction autonome) ou potentiellement par d'autres visiteurs si l'appel fait partie d'une expression plus large (par exemple, dans un `Assign`).

Paramètres :
- `node` : ast.Call. Le nœud AST de l'appel de fonction.
- `parent_id` : str. L'ID du nœud CFG parent.

Variables locales/modifiées :
- *func_name_str* : str. Nom de la fonction appelée.
- *args_list_str* (renommée de *args_list*) : List[str]. Liste des arguments positionnels sous forme de chaînes.
- *double_quote_char* : str. Contient le caractère `"` . Utilisé pour éviter les problèmes de backslash dans les f-strings.
- *kwargs_list_str* (renommée de *kwargs_list*) : List[str]. Liste des arguments nommés sous forme de chaînes "nom=valeur".
- *all_args_concatenated_str* (renommée de *all_args_str*) : str. Concaténation de tous les arguments pour l'affichage.
- *max_args_display_length* (renommée de *max_arg_len*) : int. Longueur maximale pour afficher la chaîne des arguments.
- *node_type* : str. Type du nœud ("Process" ou "IoOperation").
- *label_text* (renommée de *label*) : str. Label final du nœud.
- *call_node_id* (renommée de *call_id*) : str. ID du nœud créé.

Note : La variable *label_prefix* a été intégrée dans la logique de construction de *label_text*.

_simplify_junctions

Signature : `(_simplify_junctions(self) -> Tuple[List[Tuple[str, str]], Set[Tuple[str, str, st`

Description : Méthode privée (actuellement non utilisée activement car `visit_body` ne crée pas de jonctions 1-entrée/1-sortie) qui tenterait de simplifier les nœuds de jonction "triviaux" (ceux avec une seule arête entrante et une seule arête sortante, et qui ne sont pas des auto-boucles). Si de telles jonctions existaient, cette méthode les supprimerait et redirigerait les arêtes pour connecter directement le prédécesseur au successeur de la jonction. Retourne une version simplifiée des listes de nœuds et d'arêtes.

Variables locales/modifiées :

- *simplified_nodes_tuples* (renommée de *simplified_nodes*) : List[Tuple[str, str]]. Liste des nœuds après simplification.
- *simplified_edges* : Set[Tuple[str, str, str]]. Ensemble des arêtes après simplification.
- *junction_to_successor_map* (renommée de *junction_map*) : Dict[str, str]. Dictionnaire mappant l'ID d'une jonction simplifiée à l'ID de son successeur.
- *nodes_to_keep_ids* (renommée de *nodes_to_keep*) : Set[str]. Ensemble des ID de nœuds à conserver après simplification.
- *continue_targets* : Set[str]. (Commenté dans le code) Aurait pu être utilisé pour éviter de simplifier les jonctions qui sont des cibles de `continue`.
- *junction_candidate_id* (renommée de *j_id*) : str. ID d'un nœud en cours d'évaluation pour simplification.
- *incoming_edges* : List[Tuple[str, str, str]]. Arêtes entrant dans *junction_candidate_id*.
- *outgoing_edges* : List[Tuple[str, str, str]]. Arêtes sortant de *junction_candidate_id*.
- *predecessor_node* (renommée de *pred_node*) : str. Prédécesseur de la jonction.
- *successor_node* (renommée de *succ_node*) : str. Successeur de la jonction.
- *node_id* (dans la boucle de reconstruction) : str.
- *label* (dans la boucle de reconstruction) : str.
- *from_node* (renommée de *from_n*) : str. Nœud source d'une arête originale.
- *to_node* (renommée de *to_n*) : str. Nœud destination d'une arête originale.
- *edge_label_text* (renommée de *edge_label*) : str. Label d'une arête originale.
- *current_from_node* (renommée de *current_from*) : str. Nœud source après redirection potentielle.
- *current_to_node* (renommée de *current_to*) : str. Nœud destination après redirection potentielle.

to_mermaid

Signature : `(to_mermaid(self) -> str)`

Description : Génère la représentation textuelle complète du flowchart en syntaxe Mermaid. Structure la sortie avec un sous-graphe pour le "Flux Principal" et des sous-graphes séparés pour chaque fonction définie (par exemple, "Fonction nom_fonction"). Définit les styles CSS pour les différents types de nœuds et formate les nœuds et les arêtes. Appelée à la fin du processus pour obtenir la chaîne à rendre par MermaidJS.

Variables locales/modifiées :

- *display_nodes_tuples* (renommée de *display_nodes*) : List[Tuple[str, str]]. Liste des nœuds à afficher (actuellement tous les nœuds, pas de simplification active).
- *display_edges* : Set[Tuple[str, str, str]]. Ensemble des arêtes à afficher.
- *mermaid_lines* (renommée de *mermaid*) : List[str]. Liste de chaînes, chaque chaîne étant une ligne de la sortie Mermaid.
- *node_id* (dans les boucles de sous-graphe) : str.
- *label_text* (renommée de *label*, dans les boucles de sous-graphe) : str.
- *safe_label* : str. Label du nœud, sécurisé pour Mermaid (guillemets remplacés, sauts de ligne convertis en `<br/>`).
- *node_type* : str. Type du nœud en cours de formatage.
- *shape_open*, *shape_close* : str. Délimiteurs de forme Mermaid (par exemple, "[", "]" ou "", "").
- *func_name* (dans la boucle des fonctions) : str. Nom de la fonction pour le titre du sous-graphe.
- *node_ids_in_func* (dans la boucle des fonctions) : Set[str]. Ensemble des ID de nœuds pour la fonction *func_name*.
- *node_style_lines* : List[str]. Lignes pour les définitions de style `class node_id type;`.

- *edge_definitions* : List[str]. Lignes pour les définitions d'arêtes.
- *from_node*, *to_node* : str. Nœuds source et destination d'une arête.
- *edge_label_text* (dans la boucle des arêtes, renommée de *label*) : str.
- *safe_edge_label* : str. Label de l'arête, sécurisé pour Mermaid.

Note : Les variables *node_definitions*, *node_styles*, *processed_node_ids* ont été intégrées dans la nouvelle structure de boucles pour les sous-graphes.

_get_mermaid_node_shape

Signature : `(_get_mermaid_node_shape(self, node_type: str, label: str) -> Tuple[str, str])`

Description : Méthode utilitaire privée appelée par `to_mermaid`. Retourne un tuple contenant les chaînes de caractères pour les délimiteurs d'ouverture et de fermeture de la forme d'un nœud Mermaid, en fonction de son `node_type` et de son `label` (utilisé pour rendre les jonctions sans label plus petites).

Paramètres :
- `node_type` : str. Le type sémantique du nœud.
- `label` : str. Le label (sécurisé) du nœud.

Variables locales/modifiées :
- *shape_open* : str. Délimiteur d'ouverture (par exemple, "[", "((").
- *shape_close* : str. Délimiteur de fermeture (par exemple, "]", "))").

Vocabulaire AST d'éléments de syntaxe non implémentés encore

TABLE 4 : Correspondance des Types de Nœuds Internes, Nœuds AST et Sémantique

Node Type (Interne)	**Nœud(s) AST Correspondant(s)**	**Sémantique (Langage Naturel)**
ExceptionHandler	`ast.ExceptHandler` (dans `ast.Try`)	Bloc de code exécuté lorsqu'une exception spécifique est attrapée dans un bloc 'Try'.
TryBlock	`ast.Try` (partie `body`)	Le bloc de code principal surveillé pour les exceptions.
FinallyBlock	`ast.Try` (partie `finalbody`)	Bloc de code qui est *toujours* exécuté après un bloc 'Try', qu'une exception ait eu lieu ou non. Modifie le flux de sortie.
ElseBlock (Try/Loop)	`ast.Try` (partie `orelse`), `ast.For`/`ast.While` (partie `orelse`)	Bloc exécuté si *aucune* exception n'est levée dans le 'Try' correspondant, ou si une boucle se termine *normalement* (sans 'Break').
Raise	`ast.Raise`	Provoque un saut inconditionnel hors du flux normal vers un gestionnaire d'exception ou termine le programme si non attrapé. Termine le chemin séquentiel local.
ContextManager (With)	`ast.With`, `ast.AsyncWith`	Gère l'entrée et la sortie d'un contexte (ex : ouverture/fermeture de fichier). Représente un bloc avec potentiellement du code setup/teardown implicite.
Assertion	`ast.Assert`	Vérifie une condition et lève une `AssertionError` si elle est fausse. Peut être vu comme une Décision menant potentiellement à un 'Raise'.
ClassDefinition	`ast.ClassDef`	Similaire à 'Subroutine', représente la définition d'une classe.

Yield	`ast.Yield`, `ast.YieldFrom`	Spécifique aux générateurs, met en pause l'exécution et retourne une valeur, permettant la reprise ultérieure. Modifie profondément le flux.
AsyncAwait	`ast.AsyncFunctionDef`, `ast.Await`, `ast.AsyncFor`, `ast.AsyncWith`	Constructs pour la programmation asynchrone, impliquant des points de suspension et une boucle d'événements. (Probablement hors de portée initiale).

Table suite en page suivante

8.3 Documentation des modifs UI du 24/06/25

Défis rencontrés :

- Garantir que l'état visuel de l'interface reflète toujours de manière fiable la synchronisation entre le code dans l'éditeur et le diagramme affiché, à la condition que le flowchart soit effectivement modifié (pas pour les changement cosmétiques, non sémantiques) ;
- L'état "périmé" (outdated) n'a de sens que s'il y a un diagramme de référence à comparer. Si nous rechargeons le code, nous invalidons de fait le diagramme précédent.
- Arriver à un code plus cohérent qui élimine la "condition de concurrence" (race condition) qui se produit lorsque des estionnaires d'événements accèdent à - et manipulent - l'état visuel des bordures, et que le résultat final dépend de l'ordre, imprévisible, dans lequel ils s'exécutent.

Petit scénario explicatif de la concurrence par la métaphore de la "course" (race condition) : 1. Le problème avec le bouton 'reload' : Le "coureur" n°1 démarre : Le gestionnaire d'événement du bouton reload-code-btn est appelé. Le "coureur" n°1 agit : Il exécute la ligne codeEditorInstance.setValue(lastLoadedCode). Le "coureur" n°2 est déclenché : L'action setValue déclenche immédiatement l'événement change de l'éditeur de code. Le gestionnaire codeEditorInstance.on('change', ...) se met en route. Le "coureur" n°2 agit : Il récupère le nouveau code. Il calcule son AST. Il le compare à lastDiagramAstDump, qui contient toujours l'AST de l'ancien code (celui d'avant le rechargement). La comparaison échoue ! Les AST sont différents. Le "coureur" n°2 conclut donc que le diagramme est périmé et appelle setDiagramAndChallengeCardState("outdated"). Les bordures deviennent rouges. Le "coureur" n°1 termine sa course : Après que le "coureur" n°2 a fini, le gestionnaire du bouton "Reload" reprend et exécute sa dernière ligne : setDiagramAndChallengeCardState("default"). Le Résultat ? Le plus souvent, le "coureur" n°2 (le change handler) était plus rapide ou son effet visuel s'appliquait en dernier, laissant les bordures rouges. Le résultat était imprévisible et dépendait du timing du navigateur. C'était une course, et le mauvais coureur gagnait. 2. LA solution : pas faire courir les processus plus vite, mais changer les règles de la course pour qu'elle devienne une coopération orchestrée. 1) Invalider l'État de Référence en Premier Dans le nouveau code du reload-code-btn, la toute première chose que nous faisons est : lastDiagramAstDump = ""; C'est l'équivalent de dire : "Attention, le diagramme qui était affiché n'a plus aucune valeur. Il n'y a plus de référence valide." 2) Rendre le `change` Handler plus "Intelligent" Le gestionnaire codeEditorInstance.on('change', ...) a maintenant une nouvelle instruction au tout début de son code :

```
if (!lastDiagramAstDump) {
    setDiagramAndChallengeCardState("default");
    return; // Arrête-toi ici !
}
```

i.e. "Avant de faire quoi que ce soit, vérifie s'il existe un diagramme de référence valide. Si lastDiagramAstDump est vide, le diagramme est invalidé. Ton seul travail est de t'assurer que les bordures sont bleues (default) et de t'arrêter immédiatement. Ne continue pas la comparaison des AST."

Continuons la métaphore avec des "coopérateurs" et non des coureurs concurrents : Le "coopérateur" n°1 démarre : Le gestionnaire du reload-code-btn est appelé. Le "coopérateur" n°1 prépare le terrain : Il exécute immédiatement lastDiagramAstDump = "". L'état de référence est maintenant invalidé. C'est le passage de témoin. Le "coopérateur" n°1 continue : Il appelle codeEditorInstance.setValue(lastLoadedCode). Le "coopérateur" n°2 est déclenché : L'événement change est émis. Le "coopérateur" n°2 lit les instructions : Il entre dans sa logique et sa première question est : if (!lastDiagramAstDump). La réponse est VRAI ! Le "coopérateur" n°1 a bien invalidé la variable. Il exécute donc setDiagramAndChallengeCardState("default") (les bordures deviennent bleues) et s'arrête net grâce au return. Fin de l'opération : Le gestionnaire change a terminé son travail correctement. Le gestionnaire reload termine aussi. Le résultat final est stable, prévisible et correct : les bordures sont bleues.

Bref, la liste des modis effectuées :

1. setDiagramAndChallengeCardState(state) :
    - Légèrement modifiée pour être plus robuste : elle retire d'abord toutes les classes de bordure potentielles (border-danger, border-info, etc.) ainsi que la classe border de base, avant de réappliquer les bonnes. Cela évite d'avoir plusieurs classes de bordure en même temps.
    - Utilise `?.closest('.card')` pour éviter une erreur si l'élément n'est pas trouvé dans le DOM.

2. lastDiagramAstDump (Variable Globale) :
    - Cette variable est maintenant la seule source de vérité pour l'état du diagramme affiché.
    - Elle est initialisée à "" (chaîne vide).
    - Elle n'est mise à jour que lors d'un clic réussi sur le bouton "Lancer ...".
    - Elle est invalidée (remise à "") chaque fois qu'un nouveau code est chargé, généré ou rechargé, car le diagramme ne correspond plus.
3. Listener du bouton "Lancer..." (run-code-btn) :
    - C'est ici que lastDiagramAstDump est mis à jour avec le ast.dump du code qui vient d'être exécuté pour générer le diagramme.
    - Après cette mise à jour, il appelle `setDiagramAndChallengeCardState("default")` pour mettre les bordures en bleu, car le code et le diagramme sont maintenant synchronisés.
4. Listener du bouton "Reload" (reload-code-btn) :
    - C'est le cœur de la correction. La logique est maintenant :
        (a) Invalider l'état du diagramme en premier : lastDiagramAstDump est remis à "" et le contenu du div $\#flowchart$ est effacé.
        (b) Ensuite, mettre à jour le code dans l'éditeur : codeEditorInstance.setValue(lastLoadedCode).
    - **Pourquoi cet ordre est-il crucial ?** Lorsque `setValue` est appelé, l'événement `change` de l'éditeur se déclenche immédiatement. Ce listener (décrit ci-dessous) verra que `lastDiagramAstDump` est vide et mettra l'état visuel à "default", ce qui est exactement le comportement souhaité. La course est ainsi gagnée en préparant l'état avant de déclencher l'événement !
5. Listener `change` de `codeEditorInstance` : Sa logique a été affinée :
    - a. Il vérifie d'abord si lastDiagramAstDump est vide/invalide. Si c'est le cas, cela signifie qu'il n'y a pas de diagramme de référence. L'état est donc forcément "default" (on ne peut pas être "périmé" par rapport à rien). Il met les bordures en bleu et s'arrête.
    - b. Ce n'est que si lastDiagramAstDump a une valeur qu'il procède à la comparaison des AST et met l'état à "outdated" si nécessaire.

8.4 Documentation des modifs `code-generator.js` du 30/06/25

8.5 Principales fonctions de génération

Implémentées actuellement :

Fonction	Description
`generateRandomPythonCode`	Fonction principale de génération qui orchestre tout le processus
`calculateRequiredLines`	Calcule le nombre minimum de lignes nécessaires
`ensureVariablesForOptions`	Crée les variables demandées par l'utilisateur
`ensureRequiredVariables`	Garantit que les variables nécessaires aux structures sont présentes
`generateControlStructures`	Génère les structures de contrôle (if, boucles, fonctions)
`generateIfStatement`	Génère une structure conditionnelle if/elif/else
`generateForRangeLoop`	Génère une boucle for avec range()
`generateForListLoop`	Génère une boucle for parcourant une liste
`generateForStrLoop`	Génère une boucle for parcourant une chaîne
`generateWhileLoop`	Génère une boucle while
`generateFunction`	Génère une définition de fonction
`addFiller`	Ajoute des opérations simples pour atteindre le nombre de lignes cible

Actuellement abandonnées :

Fonction	Description
`generateSimpleOperation`	Fonction qui écrit un minimum de code
`planVariable`	Fonction de gestion de la présence des variables nécessaires
`ensureVariableForStructure`	Fonction qui crée les variables demandées par les structures sélectionnées
`finalVariableCheck`	...
`availableForVariables`	variable de stockage

8.6 Stratégie actuelle de gestion des variables

La gestion des variables est encore problématique, c'est l'un des aspects les plus complexes du générateur. Elle repose sur plusieurs mécanismes :

- Les variables sont stockées dans `declaredVarsByType`, un objet qui les classe par type
- Un ensemble `allDeclaredVarNames` permet de vérifier l'unicité des noms
- Les variables planifiées mais non encore déclarées sont stockées dans `plannedVarsByType`
- Des fonctions utilitaires génèrent des noms uniques et des valeurs appropriées

Résultat : plusieurs problèmes !

1. **Problème d'indentation déjà mentionné** : La fonction `ensureVariableExists()` ne respecte pas l'indentation courante lors de la création de variables dans des structures de contrôle.
2. **Fonctions redondantes** : Plusieurs fonctions coexistent avec des rôles similaires :
   - `finalVariableCheck()` n'est finalement jamais appelée dans le flux d'exécution actuel
   - `ensureVariableForStructure()` redondante avec `ensureVariableExists()`
   - `generateSimpleOperation()` similaire à `generateAppropriateStatement()`
   - `generateVariables()` et `ensureVariablesForOptions()` ont des rôles qui se chevauchent
3. **Variables inutilisées** : `availableForVariables` est calculée mais jamais utilisée
4. **Distinction confuse** : La distinction entre variables "déclarées" et "planifiées" qui était prévue n'est pas toujours clairement respectée dans le code

8.7 Solutions proposées

Pour résoudre ces problèmes, les modifications suivantes sont recommandées :

1. **Correction de `ensureVariableExists()`** : Modifier cette fonction pour qu'elle tienne compte de l'indentation courante et ajoute les variables avant les structures de contrôle :

```
function ensureVariableExists(type, preferPlanned = false) {
    // Si on est dans une structure (indentLevel > 0)
    if (indentLevel > 0) {
        // Trouver d'abord une variable existante
        if (declaredVarsByType[type].length > 0) {
            return getRandomItem(declaredVarsByType[type]);
        }

        // Sinon, créer une variable AVANT la structure
        const name = generateUniqueVarName(type);
        const currentPosition = codeLines.length;

        // Trouver l'endroit où insérer la déclaration
        let insertPosition = currentPosition - 1;
        while (insertPosition >= 0 &&
               codeLines[insertPosition].startsWith("    ")) {
            insertPosition--;
        }

        // Insérer la déclaration à cet endroit
        codeLines.splice(insertPosition + 1, 0,
            `${name} = ${LITERALS_BY_TYPE[type](difficulty)}`);
        declaredVarsByType[type].push(name);
        allDeclaredVarNames.add(name);
        linesGenerated++;

        return name;
    }

    // Comportement normal hors structures
    // ...
}
```

2. **Simplification des fonctions de gestion des variables** :
   - Supprimer `finalVariableCheck()` et intégrer sa logique dans `ensureRequiredVariables()`
   - Remplacer `ensureVariableForStructure()` par des appels à `ensureVariableExists()`
   - Unifier `generateSimpleOperation()` et `generateAppropriateStatement()`
   - Clarifier les rôles de `generateVariables()` et `ensureVariablesForOptions()`
3. **Nettoyage des variables inutilisées** : Supprimer `availableForVariables` ou l'utiliser effectivement dans le code
4. **Clarification de la distinction entre variables déclarées et planifiées** :
   - Documenter clairement le cycle de vie des variables
   - Utiliser systématiquement `declareVariable()` pour passer une variable de "planifiée" à "déclarée"
   - Vérifier que toutes les variables planifiées sont bien déclarées avant la fin de la génération

Références

Illich, I. (1973). Tools for Conviviality. Harper & Row.

Foucault, M. (1975). Surveiller et punir : Naissance de la prison. Gallimard.

Deleuze, G. (1986). Foucault. Les Éditions de Minuit.

Business Insider. (2024). Emails between Sam Altman and Elon Musk that kicked off OpenAI [Consulté en août 2025]. Consulté le 24 août 2025 à https://www.businessinsider.com/emails-between-sam-altman-elon-musk-kicked-off-openai-2024-11

Wenger, E. (1998). Communities of Practice : Learning, Meaning, and Identity. Cambridge University Press. https://doi.org/10.1017/CBO9780511803932

Sentance, S., & Waite, J. (2017). PRIMM : Exploring pedagogical approaches for teaching text-based programming in school. Proceedings of the 12th Workshop on Primary and Secondary Computing Education (W 113-114. https://doi.org/10.1145/3137065.3137084

Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM : A sociocultural perspective. Computer Science Education, 29(2-3), 136-176. https://doi.org/10.1080/08993408.2019.1608781

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2023). Automated Grading and Feedback Tools for Programming Education : A Systematic Review. arXiv preprint. https://arxiv.org/abs/2306.11722

Zimmermann, A. E., King, E. E., & Bose, D. D. (2024). Effectiveness and Utility of Flowcharts on Learning in a Classroom Setting : A Mixed-Methods Study. American Journal of Pharmaceutical Education, 88(1), 100591. https://doi.org/10.5688/ajpe100591

Crews, T., & Ziegler, U. (1998). The flowchart interpreter for introductory programming courses. Frontiers in Educa 307-312. https://doi.org/10.1109/FIE.1998.736854

Anderson, L. W., & Krathwohl, D. R. (Éd.). (2001). A Taxonomy for Learning, Teaching, and Assessing : A Revi Longman.

Fuller, U., et al. (2007). Developing a computer science-specific learning taxonomy. ACM SIGCSE Bulletin, 39(4), 152-170. https://doi.org/10.1145/1345375.1345438

Hattie, J., & Timperley, H. (2007). The power of feedback. Review of Educational Research, 77(1), 81-112. https://doi.org/10.3102/003465430298487

Shute, V. J. (2008). Focus on formative feedback. Review of Educational Research, 78(1), 153-189. https://doi.org/10.3102/0034654307313795

Didask. (2025). Qu'est-ce qu'un outil auteur ? – Le guide complet. Consulté le 5 août 2025 à https://www.didask.com

Brown, N., Altadmri, A., Sentance, S., & Kölling, M. (2018). Blackbox, Five Years On : An Evaluation of a Large-scale Programming Data Collection Project, 196-204. https://doi.org/10.1145/3230977.3230991

Toxicode. (2025). Silent Teacher. Consulté le 24 août 2025 à https://silentteacher.toxicode.fr/

Haverbeke, M. (2021). CodeMirror 6 Documentation. Consulté le 24 août 2025 à https://codemirror.net/6/

Skulpt. (2024). Consulté le 24 août 2025 à https://skulpt.org/

Brython. (2024). Consulté le 24 août 2025 à https://brython.info/

Zeller, A., et al. (2022). The Fuzzing Book. Consulté le 24 août 2025 à https://www.fuzzingbook.org

Kluyver, T. (2012). Green Tree Snakes : Getting to and from ASTs. Consulté le 24 août 2025 à https://greentreesnakes.readthedocs.io

Flowchart for-each loop (Stack Overflow discussion). (2013). Consulté le 24 août 2025 à https://stackoverflow.com/questions/20580028/flowchart-for-each-loop-loop-without-variable-increment

Pyodide Development Team. (2023). Pyodide Documentation. Consulté le 24 août 2025 à https://pyodide.org

Qian, Y., & Lehman, J. D. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming : A Literature Review. ACM Transactions on Computing Education, 18(1), 1-24. https://doi.org/10.1145/3077618

Littman, M. L. (2023, octobre). Code to Joy : Why Everyone Should Learn a Little Programming. The MIT Press.

Gopinath, R. (2024). Personal GitHub page. Consulté le 24 août 2025 à https://github.com/rahulgopinath/rahulgopinath.github.io