

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

## Résumé

Ce rapport présentera un outil-auteur et sa réalisation, faite dans le cadre du programme de formation GymInf. L'outil existe et 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

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Préambule . . . . .	3
1.2	Éléments de contexte . . . . .	4
1.2.1	Contexte institutionnel . . . . .	4
1.2.2	Contexte didactique : surtout des hypothèses appuyées par la littérature . . . . .	4
1.2.3	Position de principe sur l'IA générative comme outil dans les mains des élèves . . . . .	4
1.3	Objectifs fonctionnels . . . . .	5
1.3.1	Un éditeur Python... . . . .	5
1.3.2	... mais un éditeur augmenté + un rendu logigramme . . . . .	5
1.3.3	Un générateur de code et de questions pertinentes : les valeurs attendues . . . . .	6
1.3.4	Des idées abandonnées . . . . .	6
1.4	Applications similaires et sources d'inspiration . . . . .	6
1.4.1	Des outils auteurs . . . . .	7
1.4.2	Des cours et tutoriels en ligne, francophones . . . . .	7
1.4.3	Des IDE pédagogiques . . . . .	7
1.4.4	Des défis pédagogiques . . . . .	8
1.4.5	Des plateformes pédagogiques . . . . .	8
1.4.6	Des sources d'inspiration . . . . .	9
<b>2</b>	<b>L'outil et ses fonctionnalités</b>	<b>10</b>
2.1	L'architecture globale . . . . .	10
2.1.1	Schéma général . . . . .	10
2.2	Côté Client . . . . .	10
2.2.1	CodeMirror ; Bootstrap ; f-a ; Pyodide . . . . .	10
2.3	Côté Serveur . . . . .	10
2.3.1	Flask (Python) + <u>AJAX(JS) ??</u> . . . . .	10
2.4	Côté BDD . . . . .	10
2.4.1	Tables SQL + <u>AJAX(JS) ??</u> . . . . .	10
2.4.2	Les interactions entre objets : entre les <i>cards</i> , entre les <i>divs</i> ... . . . .	12
<b>3</b>	<b>Présentation de la génération automatique du code</b>	<b>15</b>
3.1	Schématisation de la logique et cycle de vie des variables . . . . .	15
3.2	Génération des "structures" . . . . .	16
3.2.1	Logique générale de la génération des structures . . . . .	16
3.2.2	Définitions de fonctions . . . . .	16
3.2.3	Itérateurs et variables locales . . . . .	21
3.3	Schéma des appels de fonctions internes à l'exécution de la génération de code aléatoire . . . . .	23
<b>4</b>	<b>Présentation du traducteur automatique de code en logigramme</b>	<b>25</b>
4.1	L'AST Python : théorie et inspirations . . . . .	25
4.1.1	Le choix de l'AST comme source de vérité . . . . .	25
4.1.2	Le <i>template</i> (patron de conception) "Visiteur" . . . . .	25
4.2	Vocabulaire AST et sémantique des nœuds du CFG . . . . .	26
4.3	Philosophie et architecture de <i>MyCFG.py</i> . . . . .	26
4.4	Fonctionnement de l'application pour la traduction de codes en logigramme . . . . .	28
<b>5</b>	<b>Présentation des interactions élèves et de leur journalisation</b>	<b>30</b>
5.1	Principes directeurs de la collecte de données . . . . .	30
5.1.1	Une base de données à double vocation pédagogique et didactique . . . . .	30
5.1.2	Contraintes techniques et éthiques . . . . .	30
5.1.3	Une preuve de concept extensible . . . . .	30
5.1.4	Présentation visuelle de la collecte de données . . . . .	32
5.2	Architecture et flux de journalisation . . . . .	33
5.2.1	Description générale des étapes d'une journalisation . . . . .	33
5.2.2	Types d'Événements Journalisés . . . . .	33

5.3	Description détaillée des étapes : scénario d'un lancement de logigramme et de défi . . . . .	33
5.4	Visualisation logique et exhaustive de la journalisation . . . . .	37
5.5	Défis techniques et solutions implémentées . . . . .	38
<b>6</b>	<b>Discussions et conclusions</b>	<b>39</b>
6.1	Critiques Anticipées... et Réponses . . . . .	39
6.2	Travaux pour améliorer l'approche actuelle . . . . .	39
6.3	Travaux pour une nouvelle approche de génération de code . . . . .	39
<b>7</b>	<b>Annexes</b>	<b>43</b>
7.1	Check-list de <i>convivialité</i> au sens d'Illich pour un service numérique et invitation à la réflexion	43
7.2	Documentation technique . . . . .	43
7.2.1	Documentation Technique du Générateur de code . . . . .	43
7.2.2	Documentation technique de l'implémentation MyCFG.py . . . . .	47
7.2.3	Comparaison : MyCFG $\neq$ PyCFG . . . . .	47
7.2.4	Documentation technique de l'implémentation MyCFG.py . . . . .	49
7.3	Documentation des modifs UI du 24/06/25 . . . . .	62
7.4	Documentation des modifs <b>code-generator.js</b> du 30/06/25 . . . . .	64
7.5	Objectifs du générateur . . . . .	64
7.6	Flux d'exécution du processus de génération actuel . . . . .	64
7.7	Principales fonctions de génération . . . . .	64
7.8	Stratégie actuelle de gestion des variables . . . . .	65
7.9	Solutions proposées . . . . .	65

## Table des figures

1	Présentation générale de l'application, vierge . . . . .	11
2	Présentation des choix de génération ou de chargement . . . . .	12
3	Identification des éléments de choix . . . . .	13
4	Architecture détaillée de l'exerciseur automatique . . . . .	14
5	Architecture du générateur . . . . .	15
6	Flowchart d'un code avec appel d'une fonction 'f(a)' & 'return' . . . . .	17
7	"Défi Élève" avec appel de fonction 'return'. La variable locale est bien absente. . . . .	17
8	Diagramme de flux de la logique de la fonction <i>generateFunction</i> . . . . .	22
9	idée de flowchart "for each" . . . . .	27
10	Génération du logigramme : Business Process Model des appels et des objets retournés . . . .	29
11	Présentation générale de l'application, vierge . . . . .	32
12	L'UI : événements journalisés . . . . .	32
13	Diagramme BPMN du flux de journalisation des actions utilisateur . . . . .	34
14	Présentation générale de l'application, vierge . . . . .	37
15	L'UI : événements journalisés . . . . .	37

# 1 Introduction

## 1.1 Préambule

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 [Illich1973] 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" [1] de Michel Foucault (métaphore d'un pouvoir disciplinaire fondé sur la visibilité asymétrique) et la "Société de contrôle" [2] 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. En définitive on pourra ainsi tester notre service numérique : favorise-t-il l'appropriation collective (Illich) ou la surveillance et le contrôle diffus (Foucault/Deleuze) ?

En raison de contraintes de temps, le dispositif n'a pas pu être passé en classe, toutefois les ambitions assumées de notre 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" [Wenger1998]
- **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

### 1.2.1 Contexte institutionnel

**Institutionnel et pédagogique :** École secondaire genevoise ; enseignement de l’informatique comme “Discipline Fondamentale” C’est-à-dire un cours obligatoire pour tous les élèves de 1ère et 2ème année avec très peu d’heures hebdomadaires pour chaque élève : 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)

### 1.2.2 Contexte didactique : surtout des hypothèses appuyées par la littérature

1. Pas d’analyse prévue de l’efficacité du dispositif par manque de temps et désynchronisation par rapport à l’année scolaire.
2. Hypothèse forte appuyée par la littérature : les intérêts multiples d’un générateur d’exercices aléatoires [Messer2023], pour autant que l’aléa réponde à des critères pédagogiques fixés et contrôlables par l’enseignant et/ou par l’apprenant → grande variété des questions, évite la mémorisation de la solution sans investissement personnel, évite le *plagiat de solutions* ; 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 de la programmation [Zimmermann2024] - en plus d’être un objectif d’apprentissage.
4. Hypothèses fortes appuyées par la littérature : la pertinence de la méthodologie PRIMM [Sentance2017WiPSCE] dans le cadre de l’enseignement de la programmation, en particulier en école secondaire avec des classes hétérogènes [Sentance2019CSE], que nous étendons par hypothèse aux syntaxes graphiques de type logigramme
5. Hypothèse : intérêt de proposer des exercices de lecture de programmes textuels et graphiques avant ceux d’écriture, en référence aux taxonomies d’Anderson & Krathwohl [AndersonKrathwohl2001] et plus spécifiquement [Fuller2007] 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”.
6. 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 [HattieTimperley2007] et [Shute2008].

### 1.2.3 Position de principe sur l’IA générative comme outil dans les mains des élèves

Il apparaît naturel 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 [5], 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 les principes déjà mentionnés précédemment, tout en exposant les élèves au risque de les instrumentaliser dans le développement commercial des fournisseurs d'IA générative au prétexte de les instrumenter.

### 1.3 Objectifs fonctionnels

Ces objectifs apparaissent comme des conclusions ( $\Rightarrow$ ) de la section précédente.

#### 1.3.1 Un éditeur Python...

1. Une fenêtre pour afficher du code Python  $\Rightarrow$  besoin de recevoir du code Python de façon programmatique (voir plus loin "Générateur de codes")
2. Implémentation de la méthode PRIM(M)  $\Rightarrow$  besoin d'un outil dynamique permettant de donner un feedback instantané
3. Encore un corollaire : besoin d'un outil permettant d'interpréter Python, qui puisse "résister" un minimum à l'apprenant afin que celui-ci s'y confronte, qui permette à l'apprenant de s'investir et d'investiguer en modifiant le code  $\Rightarrow$  autres besoins ci-dessous
4. Encore un corollaire : besoin d'un outil permettant de tester l'élève de façon adaptée aux objectifs contrôlables par l'enseignant  $\Rightarrow$  autres besoins ci-dessous
5. Un affichage "joli" pour engager tous les élèves, et pour ressembler aux *IDE* que rencontreront les élèves les plus motivés, et pour ensuite pouvoir bénéficier de fonctionnalités utiles dans un cadre pédagogique, comme l'aplatissement de blocs voire l'auto-complétion.

#### 1.3.2 ... mais un éditeur augmenté + un rendu logigramme

1. Besoin d'interactions entre le DOM et l'interprétation du code Python :
  - (a) réception des choix utilisateurs par l'UI pour les passer à la génération de scripts ;
  - (b) retour des valeurs attendues et/ou des erreurs attendues vers la page ;
  - (c) lecture des réponses élèves pour comparaison avec les valeurs & erreurs attendues ;
  - (d) affichage du feedback approprié selon étape précédente.
2. Possibilité d'enregistrer le script sur un support (un bouton **Download** de sauvegarde) + Possibilité de revenir au script généré (un bouton **Reload**) annulant les modifications apportées par l'élève depuis la génération, pour investir les dimensions "Investigate" et "Modify" du modèle PRIMM.
3. Créer le rendu logigramme pour les codes valides : A priori, deux options étaient envisageables :
  - **Soit** à la génération du script, 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 (besoin d'ajouter encore un bouton : alourdit l'UI qui devient moins lisible).

La première option aurait été en contradiction avec le point précédent, qui cherche à inciter l'investigation chez les élèves ! aussi c'est la seconde option qui a été choisie.

### 1.3.3 Un générateur de code et de questions pertinentes : les valeurs attendues

1. Besoin de contrôler l'aléa pour assurer une plus-value pédagogique  
⇒ abandonner l'idée de générer des programmes réalistes
2. Besoin de choisir les constructions utilisées dans la génération du script qui servira de support à l'exercice  
⇒ besoin de fixer les types à utiliser  
⇒ limitation aux types de base, utiles pédagogiquement
3. De préférence : la capacité à s'adapter à l'ajout de fonctionnalités ultérieures
4. Idée d'implémentation : en constante une liste pré-existante de noms de variables avec leurs types parmi lesquelles choisir de façon aléatoire, et pour chaque variable un choix aléatoire de valeurs prises parmi un ensemble pré-défini qui s'ajuste selon niveau de difficulté choisi

### 1.3.4 Des idées abandonnées

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* significantes 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.

## 1.4 Applications similaires et sources d'inspiration

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 panache unique (à ma 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
- le caractère **automatique**, aléatoire et infini, contrôlable mais non éditorialisé
- 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.

#### 1.4.1 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 [14] 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)

**moodle** <https://moodle.com/> (consulté le 5 août 2025)

H5P et moodle intègrent 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. Notons que moodle va plus loin en intégrant en particulier la gestion des élèves, et pour cela est plus souvent mis dans ce qui est une catégorie à part entière : les "learning management systems".

**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.

#### 1.4.2 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-la>

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

#### 1.4.3 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.

#### 1.4.4 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.

**Exemple : 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.

**Exemple : 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.

**Exemple : 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).

#### 1.4.5 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.

**Exemple : 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.

**Exemple : 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.

**Exemple : 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.

**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.

#### 1.4.6 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/>

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...*)

## 2 L'outil et ses fonctionnalités

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

### 2.1 L'architecture globale

#### 2.1.1 Schéma général

### 2.2 Côté Client

#### 2.2.1 CodeMirror ; Bootstrap ; f-a ; Pyodide

### 2.3 Côté Serveur

#### 2.3.1 Flask (Python) + AJAX(JS) ??

### 2.4 Côté BDD

#### 2.4.1 Tables SQL + AJAX(JS) ??

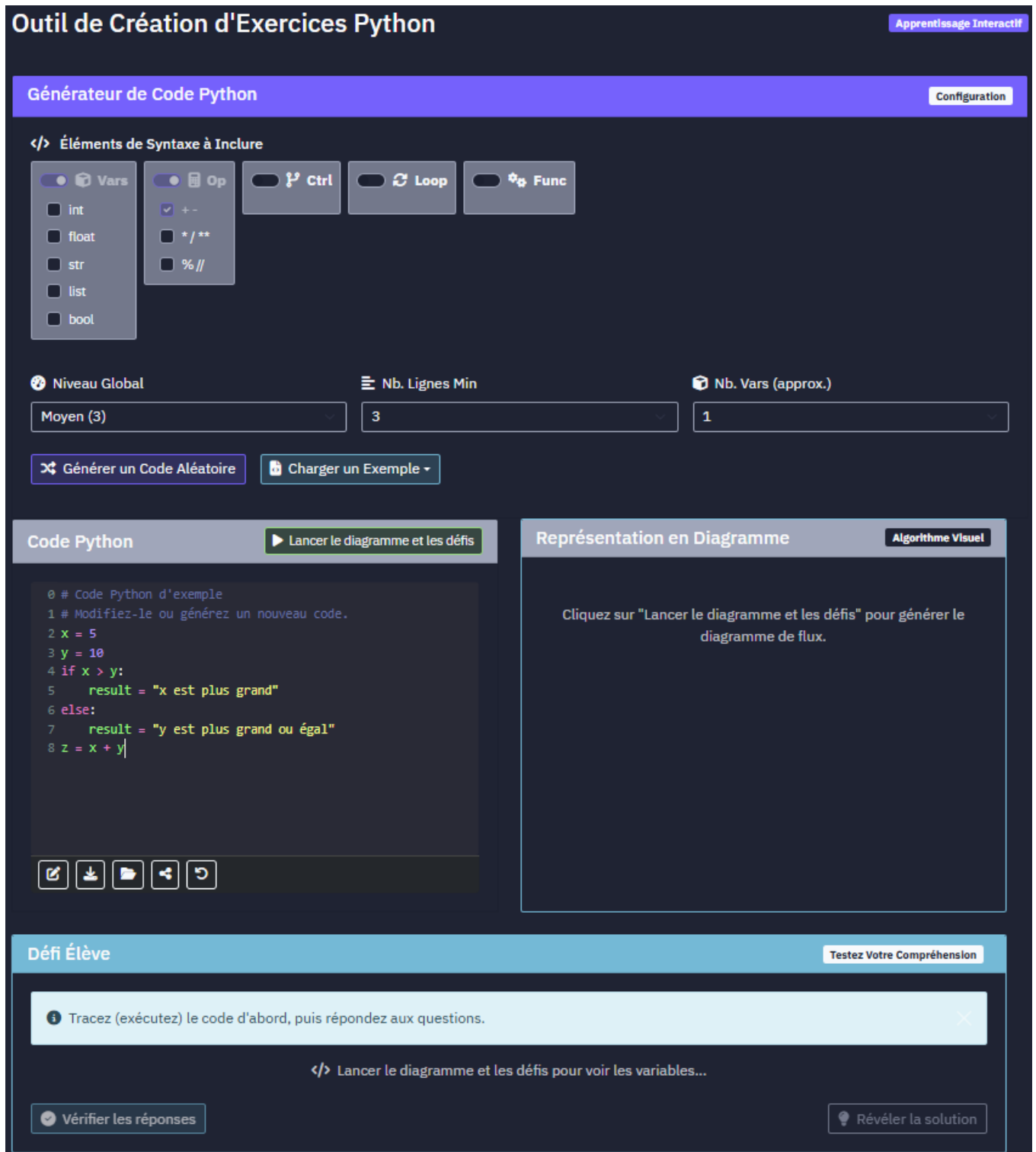


FIGURE 1 : Présentation générale de l'application, vierge

## 2.4.2 Les interactions entre objets : entre les *cards*, entre les *divs*...

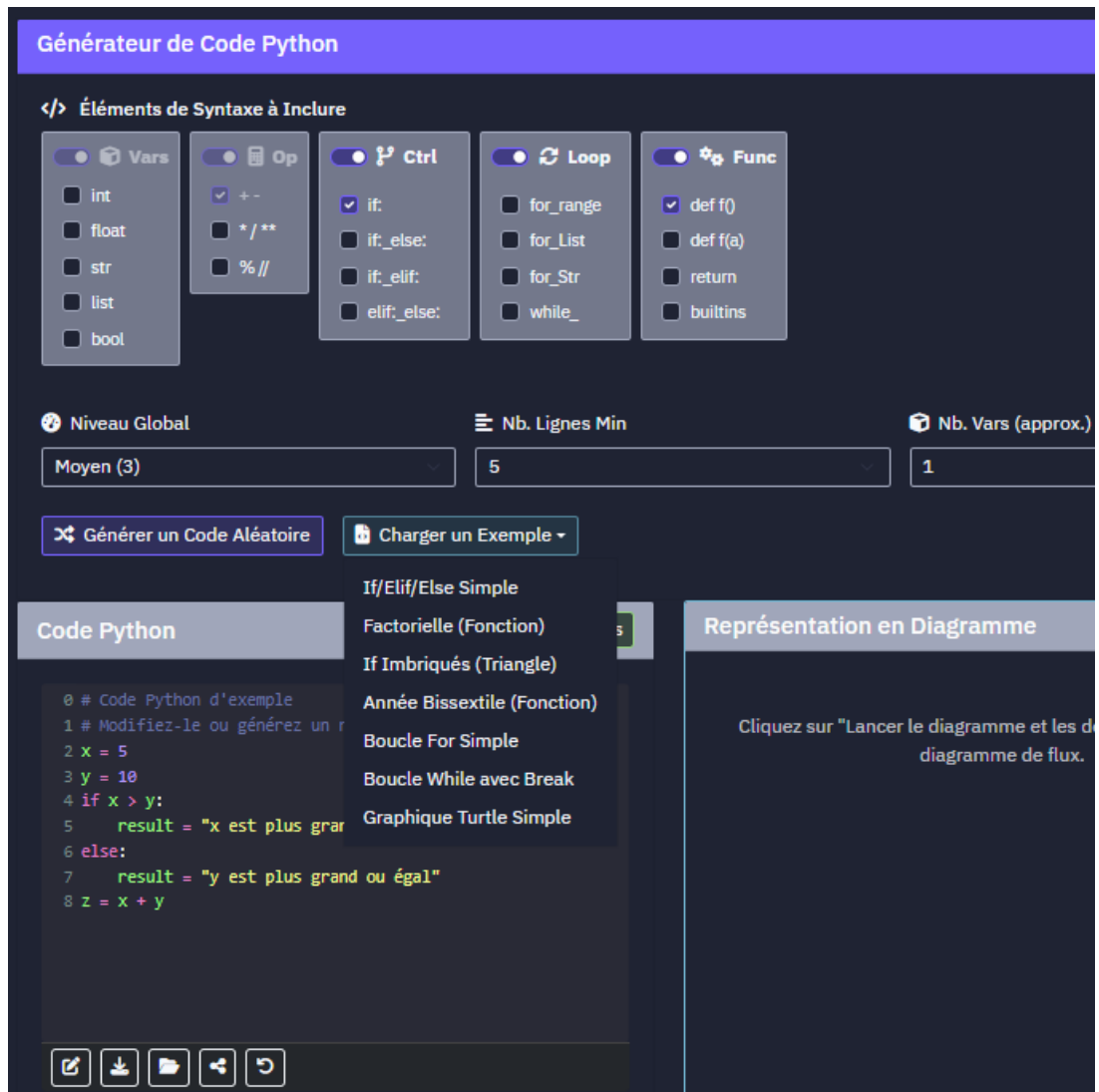


FIGURE 2 : Présentation des choix de génération ou de chargement

- Les *checkboxes* de choix des éléments syntaxiques s'affichent lorsque la *card* est sélectionnée : 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)  $\Rightarrow$  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)

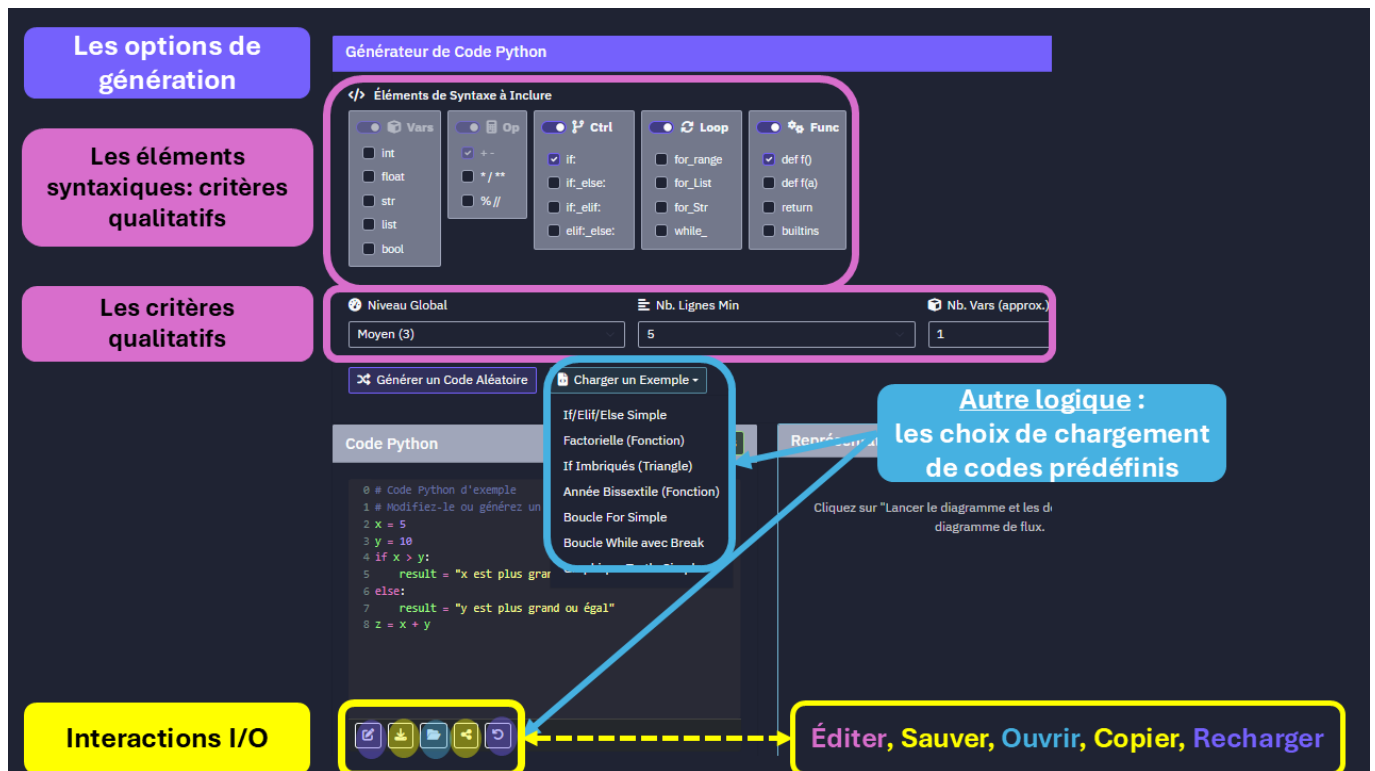


FIGURE 3 : Identification des éléments de choix

## L'architecture globale (en coulisse)

### Résumé en 4 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 connaître les variables à évaluer puis alimenter l'interface avec leur valeur et ainsi pouvoir proposer une rétroaction (HTML - CSS/bootstrap - JavaScript).

Enfin, une partie indépendante de l'application *full front-end* a été pensée pour enregistrer (*journaliser*) dans une base de données relationnelle les codes générés et les interactions élèves qui ont découlé de la partie "Défi".

### Résumé en 1 schéma

Le processus global peut se schématiser comme ceci : (voir page suivante selon place disponible...)

## Les technologies et techniques utilisées

**Le navigateur : CodeMirror, Mermaid-js, Bootstrap et Font-Awesome**

**Pyodide : environnement Python pour le navigateur**

**L'extension *serveur* : requêtes AJAX (JS) traitées par Flask (Python)**

1. Les événements enregistrés en BDD (journalisés)
2. La sécurité : confidentialité et anonymat

Les événements enregistrés en BDD (journalisés) : generation d'un code. Amélioration UX : méthode P-R-G : POST-REDIRECT-GET pratique standard pour éviter le problème de soumission multiple des formulaires

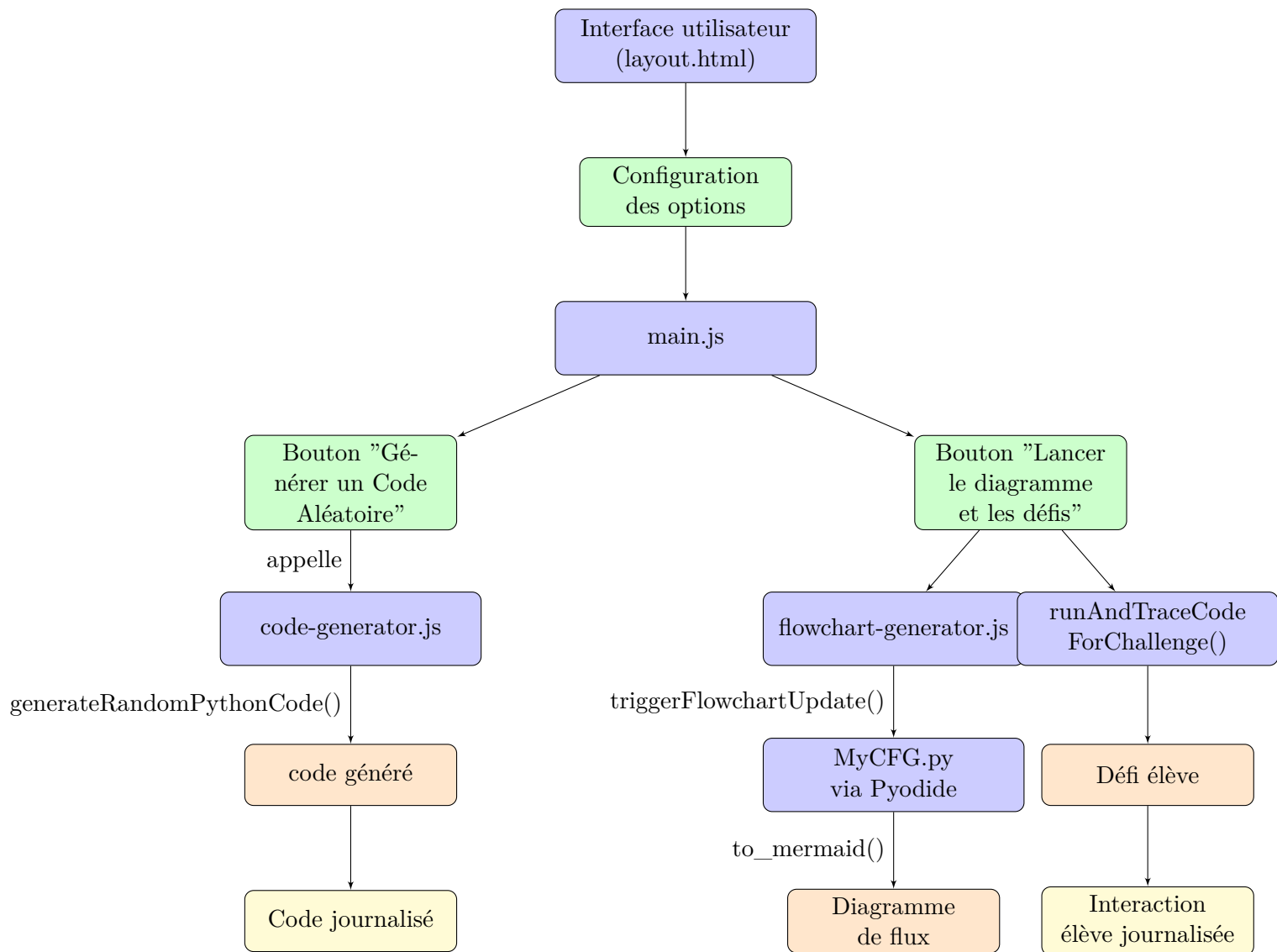


FIGURE 4 : Architecture détaillée de l'exerciseur automatique

et ce qui rend donc l'application plus robuste face aux comportements des utilisateurs comme le rafraîchissement de page ou l'utilisation du bouton "retour" du navigateur.



### 3 Présentation de la génération automatique du code

C'est la branche gauche dans la figure 4.

#### 3.1 Schématisation de la logique et cycle de vie des variables

L'organisation des fonctions peut être représentée ainsi : Voir Figure 5.

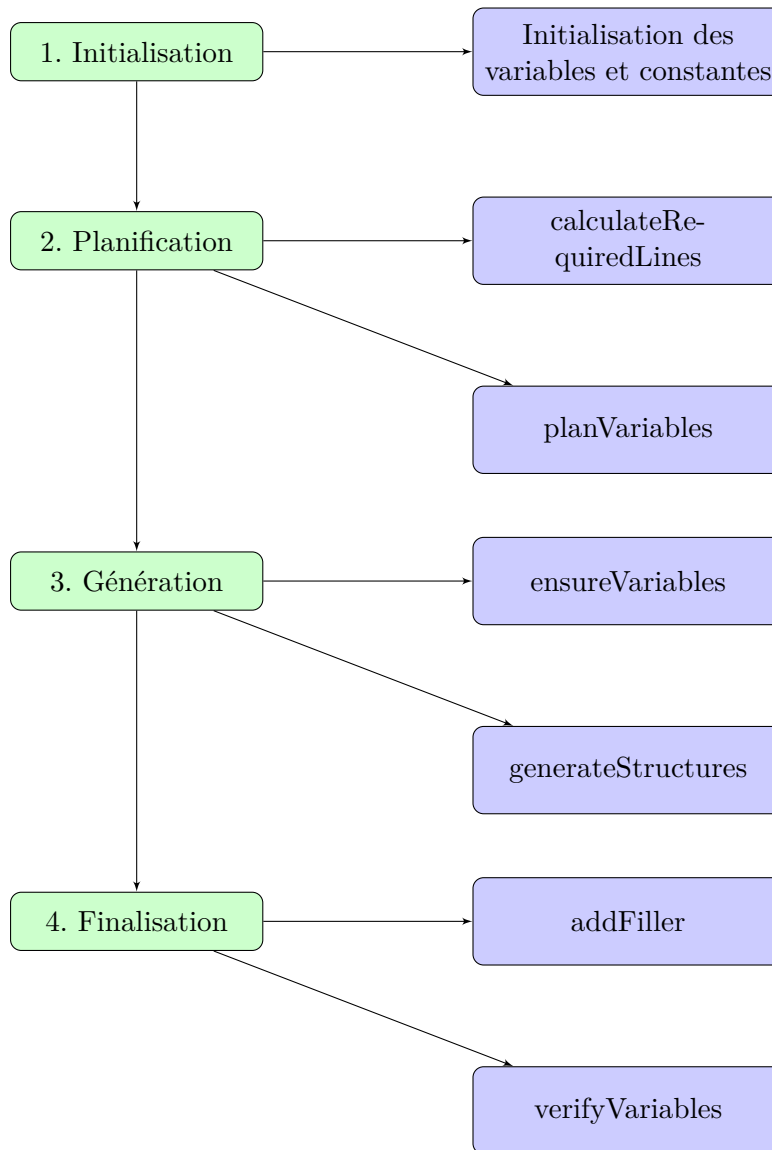


FIGURE 5 : Architecture du générateur

De même le cycle de vie des variables peut être résumé par l'énumération :

1. **Planification** : selon les besoins identifiés par les options utilisateur
2. **Déclaration** : un nom pertinent et une valeur initiale sont écrites dans le code
3. **Utilisation** : les constructeurs de structures de contrôle sont appelés avec les variables prévues
4. **Vérifications** : s'assurer que toutes les variables utilisées ont bien été déclarées, et que celles déclarées ont bien été utilisées (si possible)

L'argument `options` sera la structure regroupant les choix faits par l'utilisateur dans l'interface, sous la forme d'un ensemble de clé : valeur, chacune passée depuis html au code JS. La fonction principale `generateRandomPythonCode(options)` est appelée par le `main.js` avec la constante `generationOptions` passée en paramètre.

## 3.2 Génération des "structures"

### 3.2.1 Logique générale de la génération des structures

Les structures de contrôle sont générées selon un processus en deux phases :

#### 1. Préparation :

- Construction d'une liste des structures à générer selon les options
- Mélange aléatoire de cette liste pour varier l'ordre d'apparition
- Vérification et création des variables nécessaires

#### 2. Génération :

- Parcours de la liste des structures
- Appel de la fonction appropriée pour chaque structure
- Gestion de l'indentation et du comptage des lignes

A l'heure actuelle les structures de contrôle sont regroupées en 3 types : les conditions dénotés *Ctrl* dans l'interface, les boucles (*loop*) regroupant `for` et `while`, ainsi que les fonctions (*Func*).

### 3.2.2 Définitions de fonctions

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

#### <MINTED>

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.

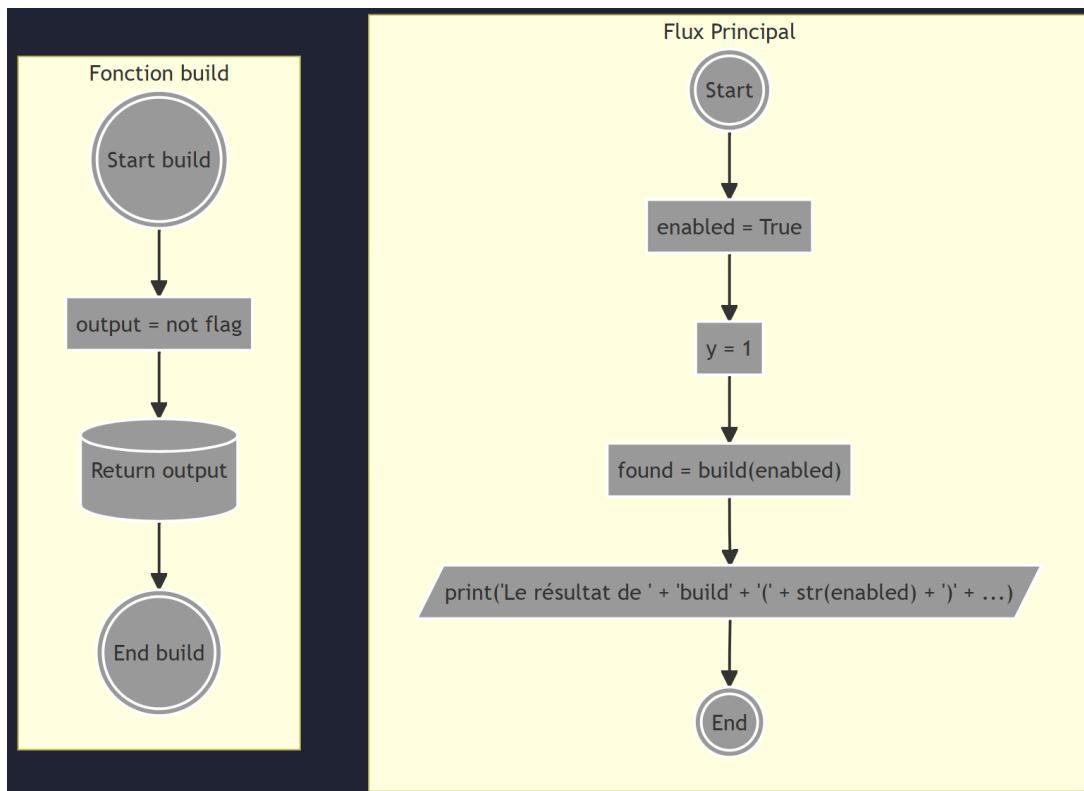


FIGURE 6 : Flowchart d'un code avec appel d'une fonction 'f(a)' & 'return'

Pour des raisons d'affichage la totalité de la chaîne n'est pas rendue (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, et enfin la "Console d'exécution" qui a été ici dépliée mais qui est par défaut masquée :

FIGURE 7 : "Défi Élève" avec appel de fonction 'return'. La variable locale est bien absente.

**Correspondance entre les types** Comment se fait-il que le type du paramètre dans la définition corresponde bien au type de la variable passée en argument dans l'appel ? Comment se fait-il que le type de la valeur de retour a un nom cohérent avec le nom de la variable ? A chaque fois 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 (#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`.  
**<MINTED>**  
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 noter exemple préparer l'appel.

1. **Sélection d'un argument du bon type (#1168)** : Le code parcourt le tableau `paramTypes` qu'il a créé précédemment.  
**<MINTED>**  
L'appel **ensureVariableExists('bool')** (#189) est effectué. Cette fonction va chercher dans `declaredVarsByType` une variable existante. Dans notre cas, elle la trouve et retourne `enabled`. Si elle ne l'avait pas trouvée, elle en aurait créé une sur-le-champ en appelant **declareVariable(type)** qui suivra la même logique de création avec un nom sémantiquement 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).

## Schéma du Processus de Génération de Fonction

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

À première vue, 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 multiples 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 "debuggable"** 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.

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 sur le processus de génération actuel.

### 3.2.3 Itérateurs et variables locales

Un soin particulier a aussi 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 réaffectation des variables et le typage dynamique Python.

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



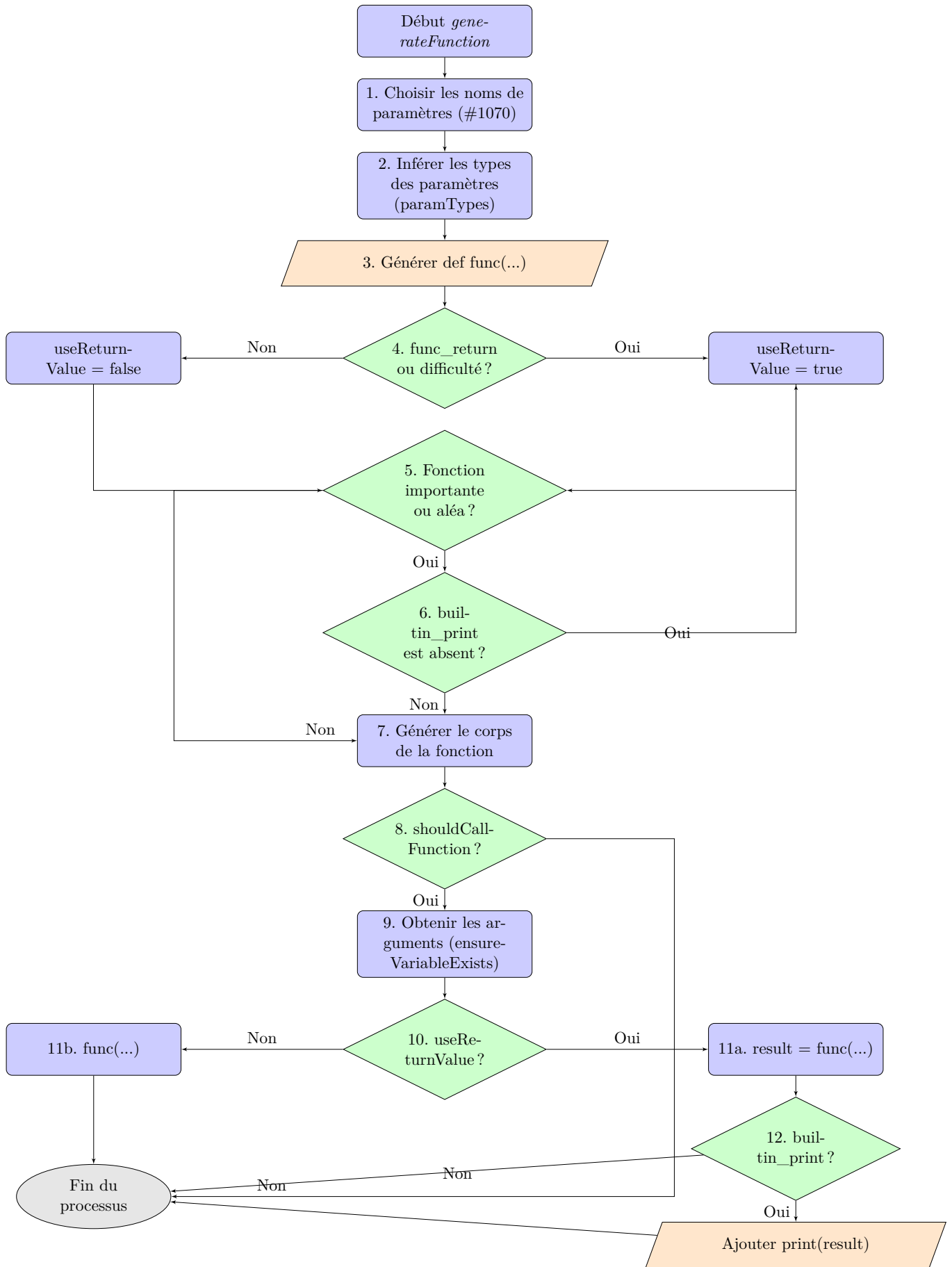
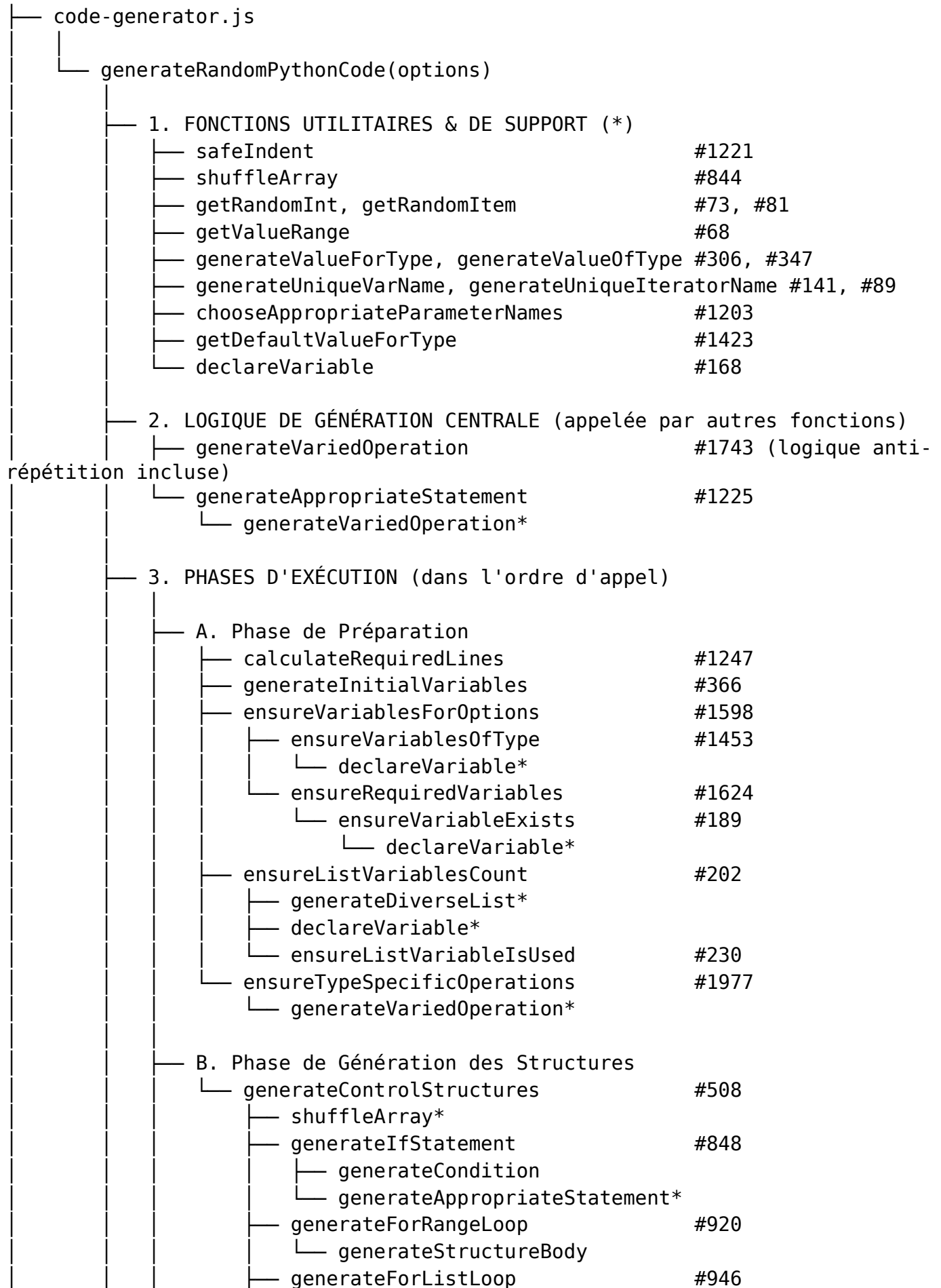


FIGURE 8 : Diagramme de flux de la logique de la fonction *generateFunction*.

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



	<ul style="list-style-type: none"> <li>└─ generateStructureBody</li> <li>└─ generateForStrLoop #986 <ul style="list-style-type: none"> <li>└─ generateStructureBody</li> </ul> </li> <li>└─ generateWhileLoop #1019 <ul style="list-style-type: none"> <li>└─ generateCondition</li> <li>└─ generateStructureBody</li> </ul> </li> <li>└─ generateFunction #1060 <ul style="list-style-type: none"> <li>└─ chooseAppropriateParameterNames*</li> <li>└─ generateStructureBody</li> <li>└─ ensureVariableExists*</li> </ul> </li> </ul>
	<ul style="list-style-type: none"> <li>└─ C. Phase de Finalisation <ul style="list-style-type: none"> <li>└─ ensureAllVariablesAreUsed #1441 <ul style="list-style-type: none"> <li>└─ generateVariedOperation*</li> </ul> </li> <li>└─ addFiller (boucle de remplissage) #1650 <ul style="list-style-type: none"> <li>└─ generateVariedOperation*</li> </ul> </li> <li>└─ finalVariableCheck #1384 <ul style="list-style-type: none"> <li>└─ declareVariable*</li> <li>└─ getDefaultValueForType*</li> </ul> </li> </ul> </li> </ul>
	<ul style="list-style-type: none"> <li>└─ ***LEGACY*** (finalVariableCheck a un rôle résiduel)</li> </ul>
└─ main.js	<ul style="list-style-type: none"> <li>└─ ... (logique de l'interface utilisateur, gestion des options)</li> </ul>
└─ MyCFG.py	<ul style="list-style-type: none"> <li>└─ ... (logique d'analyse du code Python et de génération du diagramme)</li> </ul>

## 4 Présentation du traducteur automatique de code en logigramme

Le langage Python, comme de nombreux langages de haut niveau, fournit un module intégré, `ast`, qui permet de décomposer le code source en une structure arborescente appelée Arbre Syntaxique Abstrait (Abstract Syntax Tree - AST). Cet arbre est une représentation hiérarchique du code qui ignore les détails syntaxiques superflus (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 évasive *pensée computationnelle*. Le logigramme, ou diagramme de flux, ou `textitflowchart` 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 : théorie et inspirations

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'Arbre de Syntaxe Abstraite (AST). On entend et lit souvent *Arbre Syntaxique Abstrait* mais l'arbre n'a rien d'abstrait.

#### 4.1.1 Le choix de l'AST comme source de vérité

Le module `ast` de la bibliothèque standard Python permet 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".

#### 4.1.2 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 présentés dans *The Fuzzing Book* par Zeller et al. [12], qui démontrent la construction d'un CFG à partir d'un AST. De même, la documentation *Green Tree Snakes* [11] 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.

## 4.2 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 1 détaille cette correspondance pour les éléments actuellement implémentés dans `MyCFG.py`.

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

Node Type (Interne)	Nœud(s) AST Correspondant(s)	Sémantique (Langage Naturel)
StartEnd	<code>ast.Module</code> (implicite), <code>ast.FunctionDef</code> (implicite)	Représente le point d'entrée ('Start') ou de sortie ('End') global du script/module ou d'une fonction spécifique.
Decision	<code>ast.If</code> , <code>ast.While</code> , <code>ast.For</code>	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	<code>ast.Assign</code> , <code>ast.Expr</code> (contenant <code>ast.Call</code> ), <code>ast.Pass</code> (via <code>generic_visit</code> )	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 <code>if / else</code> ). Assure la clarté du flux en unifiant les branches. Représenté par un petit cercle.
Return	<code>ast.Return</code>	Indique la fin de l'exécution d'une fonction et le retour d'une valeur. Termine le chemin d'exécution dans cette fonction.
Jump	<code>ast.Break</code> , <code>ast.Continue</code>	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	<code>ast.Call</code> (avec <code>func.id</code> 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

## 4.3 Philosophie et architecture de *MyCFG.py*

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.

**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.

**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 9 : un test initial de vacuité de l'itérable ("des éléments sont-ils présents dans la structure sur laquelle itérer" ?), l'initialisation de la variable d'itération (l'itérateur), l'exécution du corps, un test de continuation, et la mise à jour de la variable.

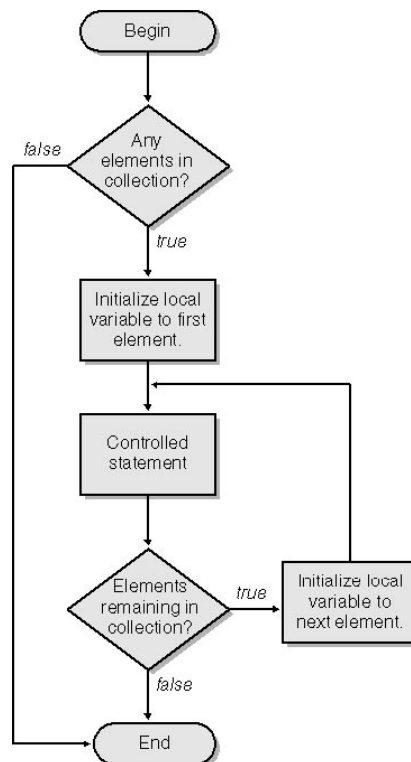


FIGURE 9 : idée de flowchart "for each"

Cette décomposition, bien que plus verbeuse, vise à rendre le mécanisme d'itération plus explicite pour l'apprenant. Une heuristique est également utilisée pour décrire la nature de l'itérable (chaîne, liste, variable, etc.) et de ses éléments (caractère, nombre, etc.) en langage naturel. C'est peut être encore un peu trop verbeux malgré l'attention qui a été portée pour réduire la longueur des chaînes (exemples : utiliser les ' plutôt que les ", couper les longues chaînes et les faire suivre de "...")

**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.

#### 4.4 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 [\[6\]](#) :

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.



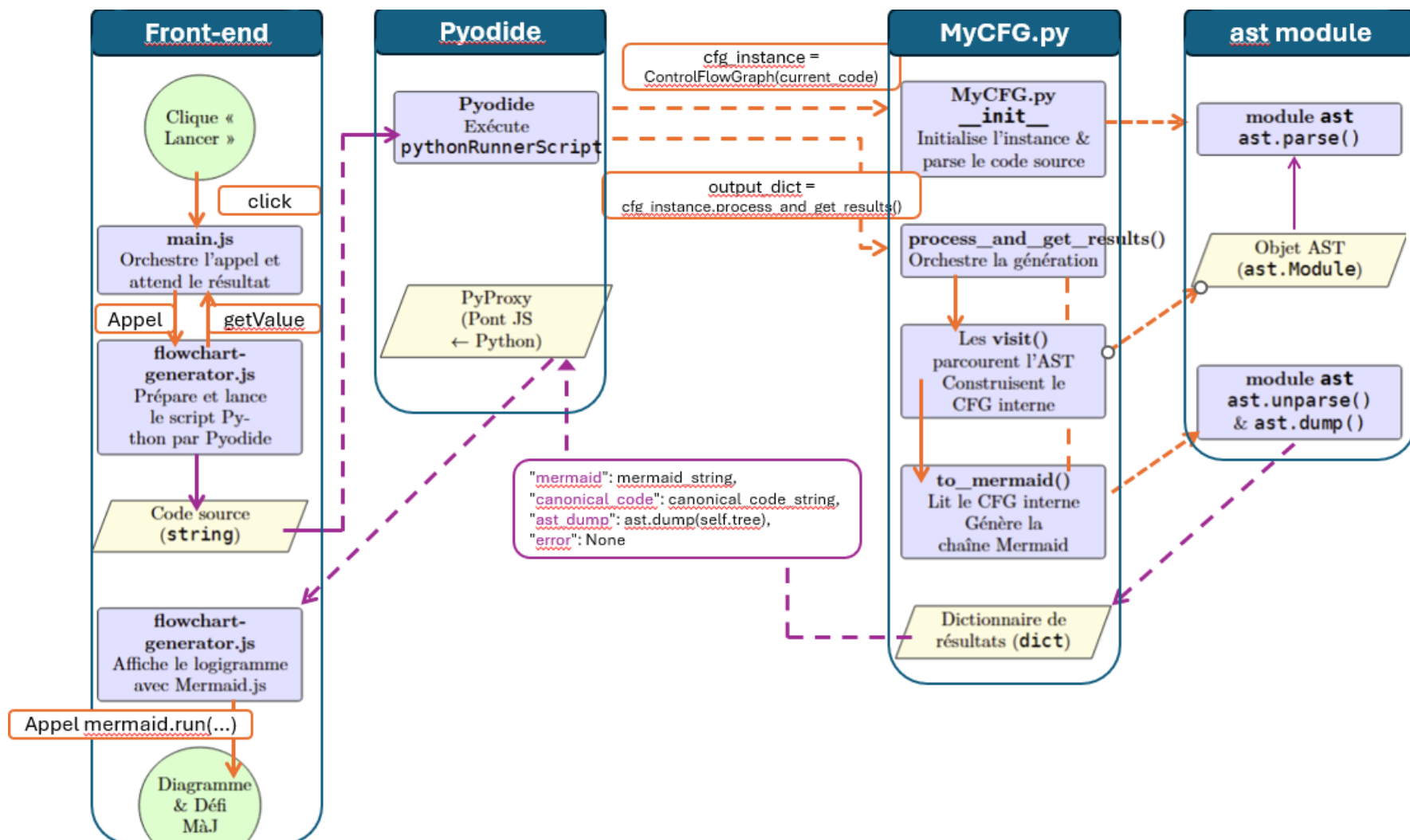


FIGURE 10 : Génération du logigramme : Business Process Model des appels et des objets retournés

## 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) [QianLehman2017] :

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 avancés) 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 éthiques** strictes doivent être appliquées relatives à l'anonymat et 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** quasi professionnelle 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

Revoici l'UI, avec une présentation schématique de la logique de collecte des données déclenchée par chacun des “clicks”, 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).

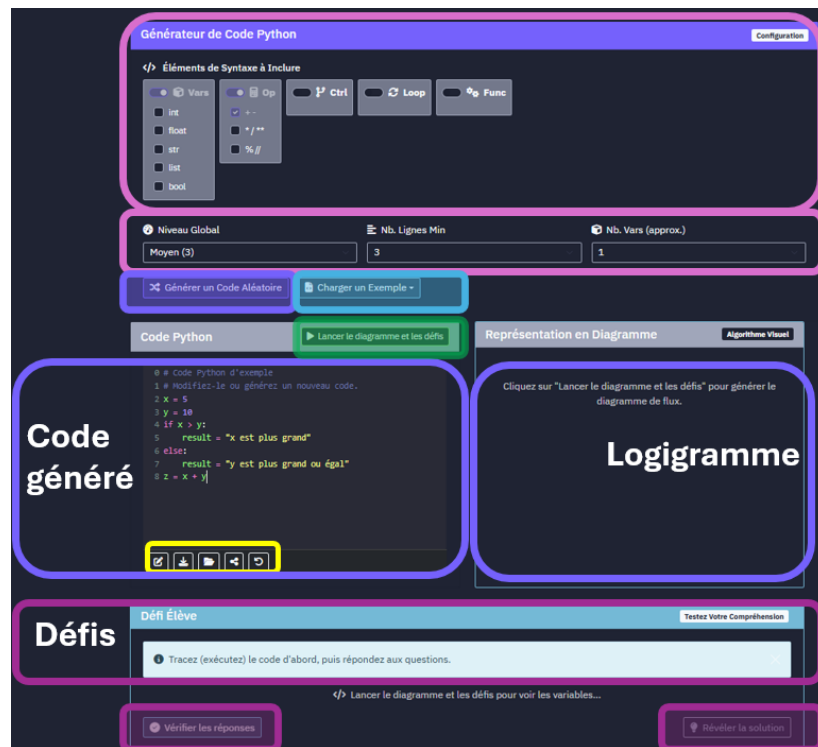


FIGURE 11 : Présentation générale de l'application, vierge

De façon plus schématique, 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).

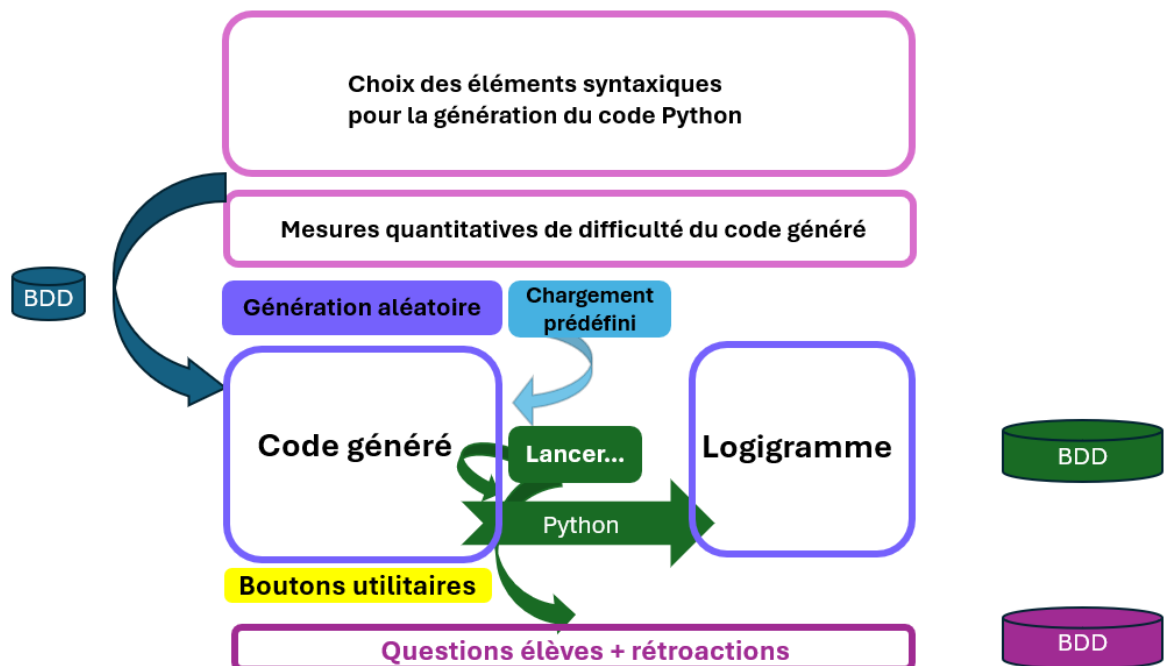


FIGURE 12 : L'UI : événements journalisés

## 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 13.

### 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. Il s'agit en effet 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 une donnée cruciale (le `code_id`) réutilisée pour pouvoir établir un lien avec les événements "Vérifier les réponses" et "Révéler la solution", tandis que les autres se contentent d'une confirmation.

### 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. 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 :

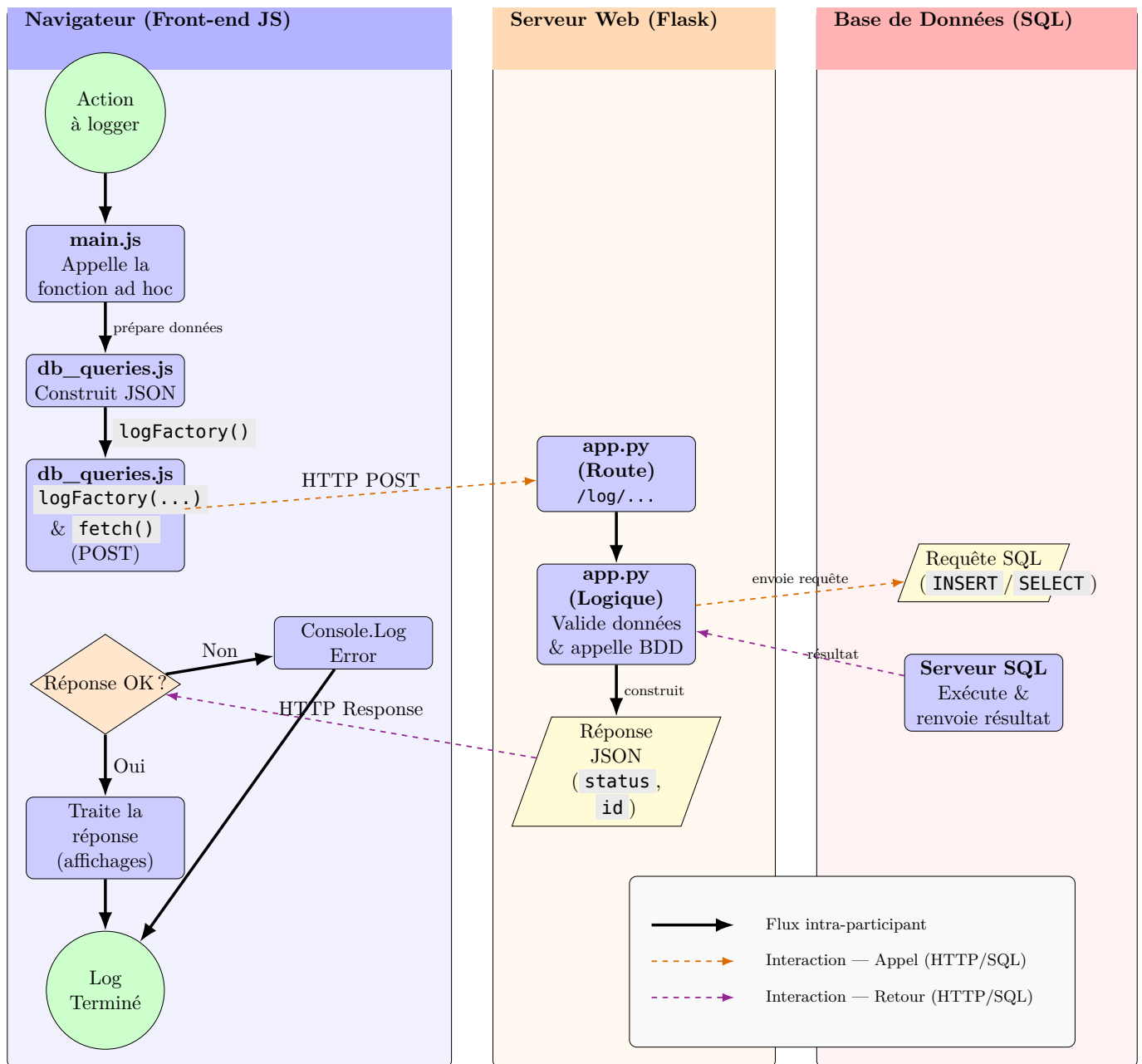


FIGURE 13 : Diagramme BPMN du flux de journalisation des actions utilisateur

### <MINTED>

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** :

### <MINTED>

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 `/log/flowchart_generation`. En suivant notre scénario la requête sera :

### <MINTED>

## 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) :

### <MINTED>

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) :

### <MINTED>

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 action 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'interaction.



## 5.4 Visualisation logique et exhaustive de la journalisation

Revoici la schématisation utilisée précédemment, ajoutant en regard des événements les noms des attributs écrits en base.

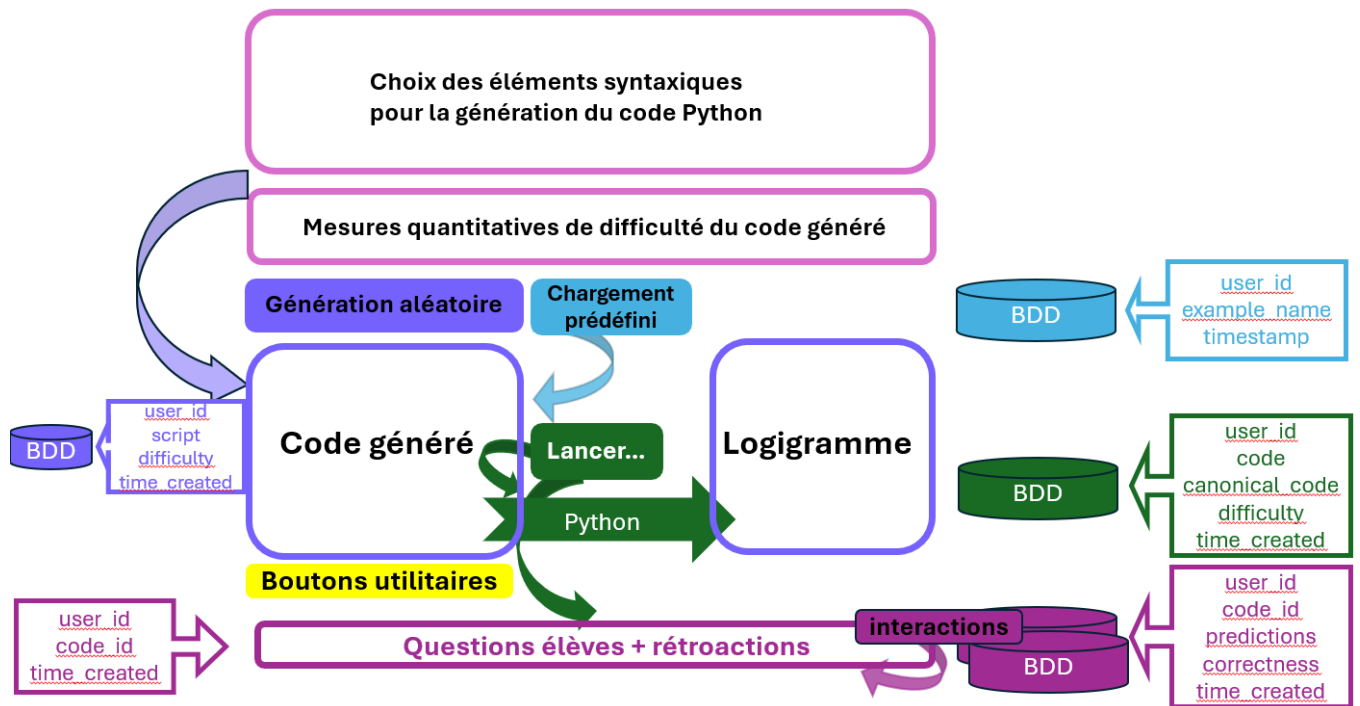


FIGURE 14 : Présentation générale de l'application, vierge

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

## 6 Discussions et conclusions

### 6.1 Critiques Anticipées... et Réponses

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)

### 6.2 Travaux pour améliorer l'approche actuelle

- 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. littérature 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 horizontaux pris par les cartes en haut de page
  3. Utiliser des "combines addictives" pour faire revenir les *users*
- Préparer la base de données pour le suivi de l'évolution des élèves **et pour** le suivi des codes générés **et pour** le suivi des interactions "Défis"
- 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  $\leftrightarrow$  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 ?

### 6.3 Travaux pour une nouvelle approche de génération de code

# Stack technologique : choix & hypothèses

## UI

1. Chargement des dépendances (parmi tous les possibles pour exécuter du Python dans le navigateur, préférence = *Pyodide*, et en plus *Skulpt* si besoin) + la jungle des frameworks pour affichage "joli" : gros travail de debugging à prévoir même avec *LABjs*
2. Des **checkbox** pour choisir les *templates*, avec une logique d'exclusion de choix mutuellement exclusifs à définir et à implémenter (si et seulement si la boucle *for* est choisie il existe la possibilité d'imbriquer des boucles, autre exemple selon le niveau de difficulté choisie on peut griser ou rendre visible certains choix, etc.)
3. Les deux vues **script** dans un éditeur de code, et **flowchart** si implémentées : onglets ou *toggle* entre les `<div>`.
4. Pour la présentation du feedback élève à renvoyer après enregistrement des réponses élèves : coloration des cellules + modales pour retours ad hoc, notamment selon comparaison avec erreurs attendues
5. Une piste : les **tooltips**, pour ajouter des **mouseover** sur les tokens du code, si possibilité d'y avoir accès... Autre piste : à la création de l'arbre syntaxique, possibilité de récupérer les annotations des noeuds par les parseurs Python

## Pistes et Hypothèses spécifiques flowchart... desquelles dépendra la logique de génération du code !

Des solutions très difficiles à créer from scratch, ou des solutions difficiles qui alourdissent la stack.

1. Manipuler directement le DOM de l'UI via Pyodide : créer les éléments HTML appropriés et concevoir leur manipulation (= réinventer la roue, puisque des bibliothèques spécialisées ont été créées pour ça)...  $\Rightarrow$  idées suivantes :
2. Bibliothèques JS existantes, exemple *Mermaid.js* : il faut que Python communique le flowchart à Mermaid  $\rightarrow$  d'où l'idée du parsing AST
3. *Graphviz* et son *DOTlanguage* : à ma connaissance en WebAssembly (accessoirement alourdirait le chargement de la page) donc à ma connaissance impossibilité de modifier le DOM directement rendant compliquée (impossible ?) l'option du rendu dynamique du flowchart... MAJ 26/03/25 des modules Python existeraient pour faire le travail demandé : cf. <https://github.com/pydot/pydot>

## Génération du code - des idées de pistes avec une supposition à questionner sur le besoin de travailler l'AST correspondant

Différentes approches à envisager, pas forcément mutuellement exclusives.

**Approche texte + ast** : Le générateur de code créerait le script à afficher (texte) par application de *templates* prédéfinis en les remplissant au fur et à mesure avec les variables et structures sélectionnées aléatoirement (exemple : "**if** `<condition>` : `<indent>` `<bloc>`"). Approche la plus accessible et la plus facile à contrôler pédagogiquement, et la plus intuitive pour la production du script car on travaille du texte (ce que l'on veut en sortie) mais il manque le lien logique pour nous rapprocher du flowchart : besoin d'analyser la syntaxe du code généré pour en extraire la structure du code ! C'est ce que promettent des fonctions comme `ast.parse()` pour retourner la racine de l'arbre syntaxique, et la classe `ast.NodeVisitor` pour parcourir les noeuds intéressants à traduire en noeuds flowchart (pour en garder que les éléments ciblés par nos objectifs pédagogiques). Pyodide est sensé nous donner la possibilité de créer et manipuler les éléments HTML correspondants, à styliser avec une classe CSS correspondante (à définir... plus facile à dire qu'à faire : il faudra supporter l'ancienneté des navigateurs installés dans les écoles et les terminaux des élèves iOS v18 et plus...).

**Approche naïve** : S'inspirer de [12] qui semble choisir des productions grammaticales correctes, mais à la signification aléatoire.

**Approche AST radicale** : Générer un AST pour le traduire en Python et le traduire en flowchart = utiliser l'AST comme une sorte de DSL en faisant une application intensive du module `ast`. Approche inspirée de ce que j'ai compris de <https://github.com/radomirbosak/random-ast>.

Utiliser l'arbre assurerait une cohérence syntaxique par rapport au langage (et permettrait des erreurs *IndexError* ou *Divisionbyzero* et d'autres plus difficiles à contrôler). Le module **astor** semble faire le job de traduire l'arbre en code (cf. <https://pypi.org/project/astor/>). Le problème principal est la difficulté de créer le générateur d'arbre AST ! Sans parler de la difficulté de le maintenir pérenne selon les mises à jour Python !

**Approche via DSL radicale :** D'après [https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language) il s'agit d'une solution standard dans l'industrie logicielle de concevoir un DSL adapté à un problème spécifique afin de faciliter la génération de solutions... mais qui semble totalement hors de portée, dépassant le cadre de ce projet, à moins d'en trouver un clé en main !

## Fonctionnalités TODO

- Y a-t-il différents types d'utilisateurs? → oui et non :
  1. oui : enseignants et élèves n'auront pas le même usage
  2. non : enseignants et élèves partagent la même interface et les mêmes fonctionnalités
- Formuler les fonctionnalités sous la forme :
  1. Enseignants **ET** élèves peuvent :
    - utiliser l'outil sur les machines fournies à l'école pour générer des *code snippets*, et régénérer 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
    - bénéficier d'une rétroaction plus riche en cas d'erreur (validation des types ? erreurs attendues comme l'indexation pré-évaluées ?)
    - utiliser la plateforme à l'école, sur smartphone, sur tablette et autres écrans personnels (*fully responsive design*)

## 7 Annexes

### 7.1 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.

### 7.2 Documentation technique

#### 7.2.1 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.

**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, calcule 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

**<MINTED>**

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", "theta"]`
- 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 :

**<MINTED>**

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 patches 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 une opération disponible

**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 choisit 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 assez 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 chaîne) 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 :

**<MINTED>**

- 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.

### 7.2.2 Documentation technique de l'implémentation MyCFG.py

#### Comparaison avec l'implémentation PyCFG

#### 7.2.3 Comparaison : MyCFG $\neq$ 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 [13]. 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
<b>Philosophie générale</b>	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 <code>return</code> hors d'une définition)	Analyse plus sémantique, mais <b>lineno</b> visualisé à chaque nœud ; reconstruit le flot d'exécution réel, "désucre" les structures (ex : <code>for</code> $\rightarrow$ <code>while</code> + <code>next</code> ), pédagogique mais plus exigeante au sens où elle révèle des "détails", plus bas niveau !

Table suite en page suivante

Aspect	MyCFG	PyCFG
<b>Entrée acceptée</b>	Une chaîne de caractères représentant un code Python <b>syntactiquement valide</b> (y compris invalide, tant que l'AST peut être construit)	Uniquement du code Python valide (car lorsqu'il rencontre un <code>return</code> , il cherche à rattacher ce nœud à la fonction englobante en remontant la pile (liste Python) des parents, donc si le <code>return</code> n'est pas dans une fonction (cf. mes <code>exemples.py</code> comme dans <code>NestedIf</code> ), la pile des parents finit par être vide, ce qui provoque un <code>IndexError</code> et pas un <code>SyntaxError: 'return' outside function</code> )
<b>Construction du graphe</b>	Plusieurs visites de l'AST, chaque structure ( <code>If</code> , <code>For</code> , <code>While</code> , etc.) a sa méthode dédiée ( <code>visit_If</code> , <code>visit_For</code> , ...)	Similaire (cf. classe <code>PyCFG</code> ) avec méthodes nommées <code>on_NodeType</code> (ex : <code>on_if</code> , <code>on_for</code> , ...) pour chaque type de nœud AST
<b>Gestion des branches et jonctions</b>	Ajoute explicitement des nœuds de jonction pour les structures de contrôle (type <b>Junction</b> pour chaque <code>If</code> , <code>For</code> et <code>While</code> ), gère ce qui est sensé représenter une sortie terminale ( <b>Return</b> ) ; A MODIFIER : applique la même gestion à <b>Break</b> et <b>continue</b>	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 <b>End</b> pour le flowchart, autre que les <b>return</b> de la fonction définie.
<b>Gestion des boucles</b>	Représente les boucles telles qu'elles apparaissent dans le code source (ex : <b>For i in ...</b> )	Transforme les boucles <b>for</b> en séquence équivalente (syntaxe un peu cryptique : <b>iter</b> , <b>next</b> , <b>while</b> ), chaque étape devient un nœud
<b>Gestion des appels de fonction (hors définition de fonction)</b>	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 <b>defif</b> et <b>defif2</b> ). POUR LE MOMENT : pas différencié appels internes/externes, mais paraît faisable	Appels de fonctions natives sont comme affectations classiques (rectangles simples) ; appels des fonctions internes : suivis par le flux de la définition de fonction. NB : <b>return</b> est un ovale "exit", même si suivi par du code.
<b>Gestion des définitions de fonctions</b>	Problématique quand le code est uniquement une définition de fonction ! Les nœuds <b>Start</b> et <b>End</b> sont alors artificiels... logique à redéfinir ??	Impeccable. Caveat idem ci-dessus !
<b>Tolérance aux erreurs</b>	Tolère des <code>return</code> en dehors des fonctions	Requiert du code Python syntaxiquement valide pour fonctionner (ex : expressions avec variables non initialisées ne lèvent pas d'exception)
<b>Sortie / Visualisation</b>	Génère une chaîne 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 : infrastructure plus lourde ! à décortiquer pour rendre un Mermaid_JS mais visualisation avancée, couleurs et styles selon le flot
<b>Gros Pour</b>	plus souple = plus proche de ce qui est fait en classe ; infrastructure légère	visualisation plus propre (eg : flux principal vs <b>def</b> )
<b>Gros Contre</b>	encore bugs et incertitudes...	c'est Digraph qui fait le graph !!

#### 7.2.4 Documentation technique de l'implémentation MyCFG.py

La classe Python `ControlFlowGraph` est le composant central pour la transformation du code Python généré (une chaîne) en un *flowchart* (une chaîne 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 jonction, 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, parent_id)`.
- `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]] = None) -> None)`

**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[str])`

**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 dépèle 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_after_if)`

**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'incrément.
- `increment_node_id` : `str`. ID du nœud d'incrément.
- `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_variable: str)`

**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], str]`)

**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	<code>ast.ExceptionHandler</code> (dans <code>ast.Try</code> )	Bloc de code exécuté lorsqu'une exception spécifique est attrapée dans un bloc 'Try'.
TryBlock	<code>ast.Try</code> (partie <code>body</code> )	Le bloc de code principal surveillé pour les exceptions.
FinallyBlock	<code>ast.Try</code> (partie <code>finalbody</code> )	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)	<code>ast.Try</code> (partie <code>orelse</code> ), <code>ast.For/ast.While</code> (partie <code>orelse</code> )	Bloc exécuté si *aucune* exception n'est levée dans le 'Try' correspondant, ou si une boucle se termine *normalement* (sans 'Break').
Raise	<code>ast.Raise</code>	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)	<code>ast.With</code> , <code>ast.AsyncWith</code>	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	<code>ast.Assert</code>	Vérifie une condition et lève une <b>AssertionError</b> si elle est fausse. Peut être vu comme une Décision menant potentiellement à un 'Raise'.
ClassDefinition	<code>ast.ClassDef</code>	Similaire à 'Subroutine', représente la définition d'une classe.
Yield	<code>ast.Yield</code> , <code>ast.YieldFrom</code>	Spécifique aux générateurs, met en pause l'exécution et retourne une valeur, permettant la reprise ultérieure. Modifie profondément le flux.
AsyncAwait	<code>ast.AsyncFunctionDef</code> , <code>ast.Await</code> , <code>ast.AsyncFor</code> , <code>ast.AsyncWith</code>	Construits pour la programmation asynchrone, impliquant des points de suspension et une boucle d'événements. (Probablement hors de portée initiale).

*Table suite en page suivante*

## 7.3 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 changements 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 :

#### <MINTED>

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 modifs 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/invalidé. 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.

## 7.4 Documentation des modifs **code-generator.js** du 30/06/25

### 7.5 Objectifs du générateur

- Générer du code Python syntaxiquement correct et exécutable
- Adapter la complexité selon le niveau souhaité
- Permettre la sélection précise d'éléments syntaxiques
- Produire du code pédagogiquement pertinent
- Offrir une variété suffisante pour des exercices diversifiés

### 7.6 Flux d'exécution du processus de génération actuel

TODO : Debug indentation dans ForRange

Le processus de génération suit ces étapes principales :

1. L'utilisateur configure les options dans l'interface (types de variables, structures de contrôle, etc.)
2. Le bouton "Générer un Code Aléatoire" déclenche la collecte des options
3. La fonction **generateRandomPythonCode(options)** est appelée par **main.js**
4. Cette fonction est définie dans **code-generator.js**
5. Le générateur crée le code en plusieurs phases :
  - (a) Initialisation des constantes et variables
  - (b) Calcul des lignes requises pour les structures demandées
  - (c) Préparation des variables nécessaires
  - (d) Génération des structures de contrôle
  - (e) Complétion pour atteindre le nombre de lignes cible
6. Le code généré est retourné à l'interface
7. L'utilisateur peut alors simuler l'exécution du code, visualiser le diagramme et répondre aux défis.

### 7.7 Principales fonctions de génération

Implémentées actuellement :

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

Actuellement abandonnées :

Fonction	Description
<b>generateSimpleOperation</b>	Fonction qui écrit un minimum de code

<code>planVariable</code>	Fonction de gestion de la présence des variables nécessaires
<code>ensureVariableForStructure</code>	Fonction qui crée les variables demandées par les structures sélectionnées
<code>finalVariableCheck</code>	...
<code>availableForVariables</code>	variable de stockage

---

## 7.8 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

## 7.9 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 :  
**<MINTED>**
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

- [Illich1973] Illich I. (1973). *Tools for conviviality*. (dans le domaine public, disponible sur Internet)
- [1] Foucault M. (1975) *Surveiller et punir : Naissance de la prison*. Gallimard
- [2] Deleuze G. (1986). *Foucault*. Les Éditions de Minuit.
- [QianLehman2017] Qian Y., Lehman J. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming : A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (October 2017) <https://dl.acm.org/doi/10.1145/3077618>
- [Wenger1998] Wenger, E. (1998). *Communities of practice : Learning, meaning, and identity*. Cambridge University Press. <https://doi.org/10.1017/CB09780511803932>
- [3] Papert, S. (1980). *Mindstorms : Children, Computers, and Powerful Ideas*. Basic Books.
- [Messer2023] Messer, M., Brown, N.C.C., Kölling, M., & Shi, M. (2023). Automated Grading and Feedback Tools for Programming Education : A Systematic Review. *arXiv preprint* arXiv:2306.11722.
- [Zimmermann2024] 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.
- [Senance2017WiPSCE] Senance, S., & Waite, J. (2017). PRIMM : Exploring pedagogical approaches for teaching text-based programming in school. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE '17)* (pp. 113–114). New York, NY : ACM. <https://doi.org/10.1145/3137065.3137084>
- [Senance2019CSE] Senance, 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>
- [AndersonKratwohl2001] Anderson, L. W., & Kratwohl, D. R. (Eds.). (2001). *A taxonomy for learning, teaching, and assessing : A revision of Bloom's taxonomy of educational objectives*. New York, NY : Longman.
- [Kratwohl2002] Kratwohl, D. R. (2002). A revision of Bloom's taxonomy : An overview. *Theory Into Practice*, 41(4), 212–218. [https://doi.org/10.1207/s15430421tip4104\\_2](https://doi.org/10.1207/s15430421tip4104_2)
- [Fuller2007] Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernan-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., McGee Thompson, D., Riedesel, C., & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, 39(4), 152–170. <https://doi.org/10.1145/1345375.1345438>
- [HattieTimperley2007] Hattie, J., & Timperley, H. (2007). The power of feedback. *Review of Educational Research*, 77(1), 81–112. <https://doi.org/10.3102/003465430298487>
- [Shute2008] Shute, V. J. (2008). Focus on formative feedback. *Review of Educational Research*, 78(1), 153–189. <https://doi.org/10.3102/0034654307313795>
- [BlackWiliam1998] Black, P., & Wiliam, D. (1998). Inside the black box : Raising standards through classroom assessment. *Phi Delta Kappan*, 80(2), 139–148.
- [4] Sovietov, P. (2022). Automatic Generation of Programming Exercises. *Institute of Information Technologies MIREA – Russian technological university Moscow, Russia* - [sovetov@mirea.ru](mailto:sovetov@mirea.ru) 10.48550/arXiv.2205.11304.
- [Keuning2018] Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19(1), 1–43. <https://doi.org/10.1145/3231711>
- [5] Révélations mises au jour lors du procès opposant Elon Musk à OpenAi et son PDG (Sam Altman) :  
**On Jun 24, 2015, at 10:24 AM, Sam Altman wrote :**  
1) The mission would be to create the first general AI and use it for individual empowerment—ie, the distributed version of the future that seems the safest. More generally, safety should be a first-class requirement.  
Cité dans la presse en novembre 2024. <https://www.businessinsider.com/emails-between-sam-altman-elon-musk-kicked-off-openai-2024-11> (consulté en août 2025).

- [6] Pyodide Development Team. (2023). *Pyodide Documentation*. Récupéré de <https://pyodide.org> (consulté en 2024).
- [7] Haverbeke, M. (2021). *CodeMirror 6 Documentation*. Récupéré de <https://codemirror.net/6/> (consulté en 2024).
- [8] Ace Editor. (n.d.). *Ace - The High Performance Code Editor for the Web*. Récupéré de <https://ace.c9.io> (consulté en 2024).
- [9] Python Software Foundation. (2023). *Python Documentation : ast — Abstract Syntax Trees*. Extrait de <https://docs.python.org/3/library/ast.html> (consulté en 2024).
- [10] Python Software Foundation. (2023). *Python Documentation : dis — Disassembler for Python bytecode*. Extrait de <https://docs.python.org/3/library/dis.html> (consulté en 2024).
- [11] Kluyver, T. (2012). *Green Tree Snakes : Getting to and from ASTs*. Documentation en ligne, <https://greentreesnakes.readthedocs.io> (consulté en 2024).
- [12] Zeller, A. et al. (2022). *The Fuzzing Book* – Chapitre "Fuzzing with Grammars". En ligne, <https://www.fuzzingbook.org> (consulté en 2024).
- [13] Gopinath, R. *Page personnelle*. En ligne, <https://github.com/rahulgopinath/rahulgopinath.github.io> (consulté en 2024).
- [14] Didask. (n.d.). *Qu'est-ce qu'un outil auteur ? - Le guide complet*. Recopié de [https://www.didask.com\(consultéen2025\)](https://www.didask.com(consultéen2025)).