Document de Conception : Générateur de Code Python Pédagogique

Julien Mateesco

5 août 2025

Résumé

Ce document décrit la conception et l'architecture du module `code-generator.js`, le cœur du système de génération de code Python pour l'outil d'apprentissage GymInf. L'objectif est de justifier les choix de conception qui permettent de produire un code à la fois aléatoire, syntaxiquement correct, et pédagogiquement pertinent, en réponse aux contraintes d'un environnement entièrement *front-end*.

1 Objectifs et Contexte

L'objectif principal du générateur est de fournir des extraits de code Python variés et adaptés à des fins d'apprentissage, et valides pour pouvoir donner lieu à l'évaluation réelle des variables. Le système doit opérer dans le navigateur de l'utilisateur sans dépendre d'un serveur, ce qui impose des contraintes de performance et d'architecture.

1.1 Principes Directeurs

La conception repose sur trois piliers fondamentaux :

— **Contrôle Pédagogique :** L'utilisateur (enseignant) doit pouvoir sélectionner précisément les constructions syntaxiques (variables, opérations, structures de contrôle, fonctions) à inclure.

— **Variété et Aléatoire :** Chaque génération doit produire un code unique pour garantir une source inépuisable d'exercices.

— **Cohérence et Robustesse :** Le code généré doit toujours être syntaxiquement valide et sémantiquement plausible, évitant les opérations illogiques ou triviales.

2 Architecture Générale

Le générateur est une fonction principale, `generateRandomPythonCode(options)`, qui orchestre un processus de génération en plusieurs phases séquentielles. Cette approche garantit que les prérequis de chaque étape sont satisfaits avant de passer à la suivante, évitant ainsi les erreurs en cascade.

2.1 Flux d'Exécution Séquentiel

Le processus est décomposé en phases logiques, exécutées dans cet ordre :

1. **Phase de Préparation :**
   — `calculateRequiredLines()` : Calcule le nombre minimal de lignes et de variables requis pour satisfaire les options de l'utilisateur (ex : une boucle `for` nécessite au moins 2 lignes et une variable).

— `generateInitialVariables()` : Déclare un ensemble de base de variables en fonction des options de types et de nombre.

— `ensureVariablesForOptions()` : Garantit que les variables nécessaires pour les structures de contrôle (ex : un `bool` pour un `if`) existent, en en créant si besoin.

2. **Phase de Génération des Structures :**
   — `generateControlStructures()` : Orchestre la création des structures de contrôle (`if`, `for`, `while`) et des fonctions (`def`) dans un ordre aléatoire.
   — Chaque fonction de structure (ex : `generateForRangeLoop`) délègue la création de son contenu à `generateStructureBody()`. Cette dernière utilise le contexte pour générer des opérations pertinentes (ex : utiliser la variable d'itération dans le corps d'une boucle).
3. **Phase de Finalisation et d'Enrichissement :**
   — `ensureAllVariablesAreUsed()` : Parcourt toutes les variables déclarées et ajoute une opération les utilisant si elles n'ont pas encore été mobilisées, assurant ainsi qu'aucune variable n'est "morte".
   — `addFiller()` : Si le nombre de lignes cible n'est pas atteint, cette fonction est appelée en boucle pour ajouter des opérations variées jusqu'à atteindre la longueur désirée.
   — `finalVariableCheck()` : Une dernière vérification pour s'assurer que toutes les variables planifiées sont bien déclarées, ajoutant une initialisation par défaut au début du code si nécessaire.

3 Conceptions et Compromis Clés

3.1 Gestion de la Variété : `generateVariedOperation`

Le défi principal est d'éviter la monotonie. La fonction centrale `generateVariedOperation` est conçue pour cela.

— **Conception :** Elle contient un catalogue d'opérations possibles pour chaque type de données, avec des variantes conditionnées par le niveau de difficulté.

— **Compromis :** Plutôt que de générer des opérations purement aléatoires (risquant des nonsens comme `"a" - "b"`), nous utilisons des "templates" d'opérations valides. Cela réduit la liberté créative mais garantit la correction syntaxique et sémantique.

— **Justification :** La robustesse et la pertinence pédagogique priment sur la créativité absolue du code généré.

3.2 Gestion des Dépendances : L'approche "Ensure"

De nombreuses options ont des prérequis (ex : une boucle `for..in list` nécessite une variable de type liste).

— **Conception :** Une série de fonctions "ensure" est appelée durant la phase de préparation pour vérifier l'état actuel et créer les ressources manquantes (ex : `ensureVariableExists` ou `ensureListVariablesCount`).

— **Compromis :** Cette approche est plus verbeuse qu'une génération "au fil de l'eau", mais elle découple la logique de création des prérequis de la logique de génération des structures.

— **Justification :** Cela rend le code plus maintenable. Si la logique de génération d'une boucle change, il n'est pas nécessaire de modifier la manière dont ses dépendances sont créées.

3.3 Gestion de la Complexité : Le paramètre `difficulty`

Le niveau de difficulté influence de multiples aspects de la génération.

- **Conception :** Le paramètre `difficulty` est passé à presque toutes les fonctions de génération. Il est utilisé pour :
  - Déterminer la plage des valeurs numériques (`getValueRange`).
  - Débloquer des opérations plus complexes dans `generateVariedOperation`.
  - Augmenter le nombre d'instructions dans les corps de boucle via la fonction `generateStructureBody`.
- **Justification :** L'utilisation d'un seul paramètre global pour la difficulté simplifie l'interface et le code, tout en offrant un contrôle suffisant sur la complexité du résultat.

4 Critiques Anticipées et Réponses

Critiques Académiques Potentielles

1. **Critique : Le code généré manque de "sens" ou d'intentionnalité. Il ressemble à une suite d'opérations aléatoires plutôt qu'à un algorithme résolvant un problème.**
   - **Réponse Argumentée :** C'est un compromis fondamental et assumé. L'objectif de l'outil n'est pas de générer des solutions à des problèmes complexes, mais de fournir des extraits de code ciblés pour l'apprentissage de la **syntaxe** et de la **sémantique locale** (l'effet d'une seule ligne ou d'un petit bloc). La fonction `chooseAppropriateParameterNames` et les opérations contextuelles dans `generateStructureBody` sont des tentatives pour injecter une plausibilité sémantique, mais l'objectif principal reste la maîtrise des constructions du langage, pas la conception algorithmique.

2. **Critique : La génération basée sur des templates et des listes de noms de variables prédéfinis limite la créativité et ne prépare pas les élèves à la résolution de problèmes authentiques.**
   - **Réponse Argumentée :** L'outil se positionne au début du spectre de l'apprentissage, correspondant aux phases "Predict" et "Run" du modèle PRIMM. À ce stade, la familiarisation avec des formes idiomatiques et des noms de variables conventionnels (`i`, `count`, `items`) est un objectif pédagogique en soi. La créativité et la résolution de problèmes authentiques interviennent dans les phases ultérieures ("Investigate", "Modify", "Make"), que l'outil supporte en permettant à l'élève de modifier librement le code généré. Le générateur fournit donc un "échafaudage" stable sur lequel l'élève peut ensuite construire.

3. **Critique : L'approche par phases séquentielles et la gestion manuelle des dépendances sont complexes et pourraient être remplacées par un système plus déclaratif ou basé sur des contraintes.**
   - **Réponse Argumentée :** Un solveur de contraintes ou un système déclaratif serait en effet une approche plus élégante sur le plan théorique. Cependant, étant donné la contrainte d'un environnement *full front-end* en JavaScript, l'implémentation d'un tel système serait extrêmement complexe et potentiellement lente. L'approche impérative par phases, bien que plus verbeuse, est performante, déterministe dans son comportement, et plus facile à déboguer et à maintenir avec les outils standards du développement web. C'est un choix pragmatique privilégiant la faisabilité et la performance dans le contexte technologique imposé.

5 Comparaison de Différentes Approches de Génération de Code

5.1 L'approche actuelle impérative :

```
function generateRandomPythonCode(options) \{
// Étape 1 : Calculer les besoins
calculateRequiredLines(options);
// Étape 2 : Créer des variables
generateInitialVariables(options);
```

```
// Étape 3 : S'assurer que les prérequis sont là
ensureVariablesForOptions(options);
// Étape 4 : Créer les structures
generateControlStructures(options);
// ... etc.
\}
```

5.2 Une approche déclarative :

```
// On ne fait que décrire le résultat final
const codeDescription = {
    targetLines: 20,
    difficulty: 3,
    mustContain: ['if', 'for_list'],
    mustUseVariables: ['int', 'list'],
    allVariablesMustBeUsed: true,
    noInfiniteLoops: true
};

// Le système se charge de trouver un code qui respecte cette description
const generatedCode = declarativeGenerator.generate(codeDescription);
```

Le système pourrait alors explorer l'espace des programmes possibles et en trouver un qui respecte toutes les contraintes spécifiées, ou à défaut proposer une solution approximative.

5.3 Une approche par solveur de contraintes :

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

Les variables : Chaque ligne de code potentielle, chaque nom de variable, chaque valeur littérale.

Les contraintes :

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

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