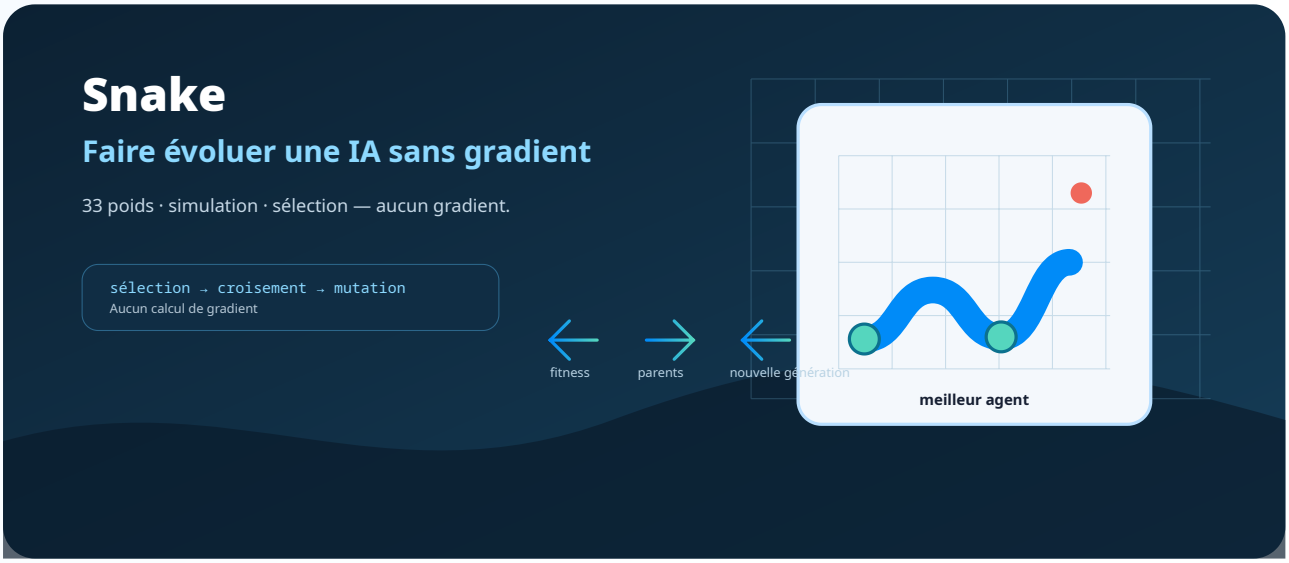
Tutoriel Algorithmique · IA évolutive · Python · Pygame

Snake — Faire évoluer une IA sans apprentissage par gradient

Construire from scratch une politique qui joue à Snake, puis la faire progresser par sélection, croisement et mutation sans calcul de gradient.

Auteur **Youness Boutyour**



DURÉE	NIVEAU	FORMAT
60 minutes	Débutant	Tutoriel pratique

Algorithmique · IA évolutive · Python · Pygame · publié le 2026-10-22 · local

Objectifs et livrables

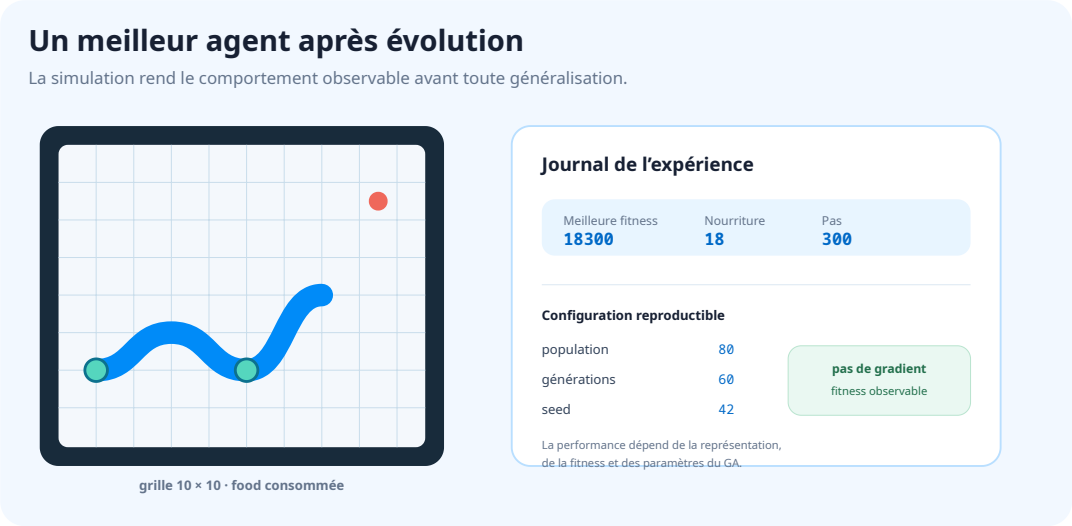
Un Snake est un petit environnement idéal pour comprendre une IA évolutive : les règles sont visibles, les actions sont discrètes et la qualité d'un agent peut être mesurée en quelques secondes. Nous allons partir d'une politique très simple qui ne sait rien du jeu, la faire jouer dans une simulation déterministe, puis conserver les variantes qui mangent le plus de nourriture et survivent le plus longtemps.

Le tutoriel n'utilise ni réseau de neurones, ni rétropropagation, ni bibliothèque de machine learning. Chaque agent possède seulement 33 poids numériques : trois actions possibles — tourner à gauche, continuer tout droit ou tourner à droite — multipliées par onze observations de l'état. Les scores obtenus par ces poids sont comparés pour choisir l'action suivante.

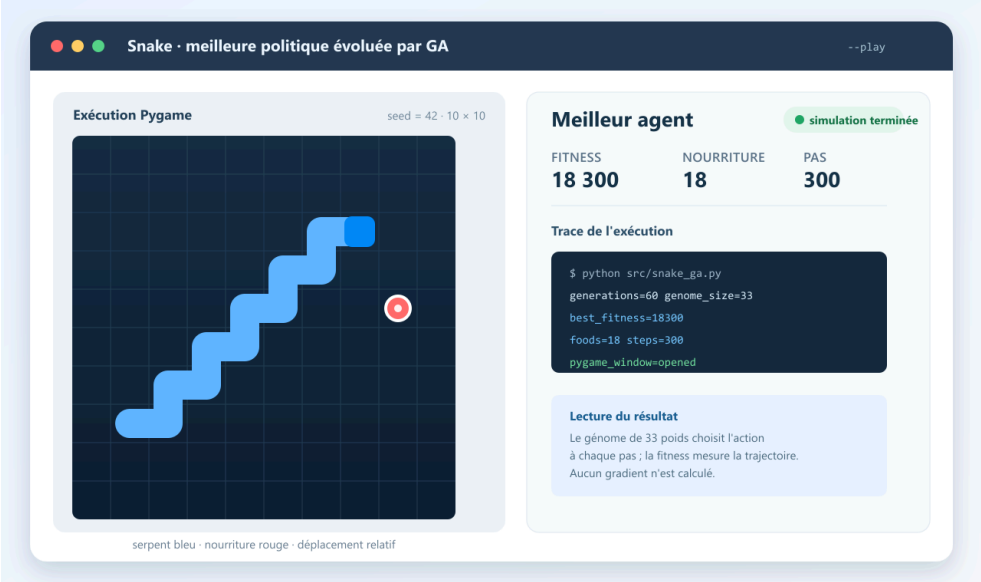
Nous coderons ensuite un algorithme génétique complet avec une population, une fonction de fitness, une sélection par tournoi, un croisement à un point, une mutation gaussienne et un élitisme. Le même génome sera évalué avec la même graine afin que les agents soient comparés dans des conditions identiques. L'option `--headless` produira une trace reproductible ; l'option `--play` ouvrira une partie Pygame avec le meilleur agent.

À la fin, vous aurez un projet Python from scratch, trois tests unitaires, un fichier `outputs/evolution.csv`, un résumé JSON et une visualisation. Vous comprendrez aussi ce que cette expérience ne prouve pas : un agent qui réussit une simulation donnée n'a pas appris une règle générale du Snake, et un algorithme génétique ne garantit pas l'optimum global.

Chaque génération évalue les agents puis applique sélection, croisement, mutation et élitisme.



Le meilleur agent est mesuré par la nourriture consommée et le nombre de pas survécus.



Capture d'écran du résultat de l'exécution : la trajectoire Pygame est accompagnée de la fitness et de la trace terminal.

Comprendre l'idée avant de coder

Un algorithme génétique traite les politiques comme une population de candidats. Une politique est représentée par un chromosome ; la simulation lui donne une note ; les meilleurs chromosomes ont davantage de chances de transmettre leurs paramètres. La variation vient du croisement de deux parents et de la mutation aléatoire de quelques poids.

Le mot « IA » désigne ici une règle de décision ajustée par recherche, un modèle de comportement minimal plutôt qu'un réseau entraîné. Le programme ne calcule pas la dérivée de la fitness par rapport aux poids : il compare des résultats complets de parties. C'est ce qui rend l'approche accessible, mais aussi

coûteuse lorsque la simulation devient complexe. La fitness reste le contrat central : si elle récompense mal le comportement, le GA optimisera le mauvais objectif.

La figure du cycle détaille le flux : un génome produit une action à partir de l'état courant, le Snake joue, la partie retourne une fitness, puis la population est renouvelée. Le serpent n'est pas entraîné par un signal local à chaque mouvement ; il est évalué après une trajectoire et les paramètres des meilleurs individus sont réutilisés.

1. Préparer le projet et vérifier le mode sans interface

Le code fourni sépare la simulation de l'affichage. Cette organisation permet de tester l'algorithme sur un serveur ou dans une CI sans ouvrir de fenêtre, puis d'utiliser Pygame uniquement pour observer le résultat. Le fichier `requirements.txt` ne contient que Pygame : le GA lui-même utilise exclusivement la bibliothèque standard.

Décompressez l'archive et créez l'environnement virtuel :

Terminal · Bash

```
mkdir -p snake-ga-sans-gradient
cd snake-ga-sans-gradient
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Sur Windows PowerShell, activez l'environnement avec `.venv\\Scripts\\Activate.ps1`. Si vous ne disposez pas d'un affichage graphique, ne lancez pas Pygame : les commandes `--self-test` et `--headless` sont suffisantes pour suivre le tutoriel.

Code de l'étape

Le point d'entrée expose trois usages : vérifier les invariants, évoluer une population et afficher une partie :

Terminal · Bash

```
python src/snake_ga.py --self-test
python src/snake_ga.py --evolve --headless
python -m unittest discover -s tests -v
```

Sortie obtenue

Sortie / exemple

```
self-test ok
generations=60 genome_size=33
best_fitness=18300 foods=18 steps=300
best_length=21 last_generation=60
outputs=evolution.csv,run-summary.json
```

La valeur de fitness dépend de la graine et des paramètres. Avec la configuration par défaut, l'exécution est reproductible et le meilleur agent mange 18 nourritures avant d'atteindre la limite de 300 pas.

Commande de vérification

Terminal · Bash

```
python src/snake_ga.py --self-test
test -f outputs/evolution.csv
test -f outputs/run-summary.json
```

Cette vérification confirme que l'algorithme peut être exécuté sans Pygame à l'écran et que ses sorties sont écrites dans le dossier attendu.

2. Modéliser le Snake et ses observations

La simulation représente la tête et le corps par une liste de coordonnées `(x, y)`. La direction est codée par quatre valeurs : haut, droite, bas et gauche. À chaque pas, l'agent choisit une action relative ; le jeu transforme cette action en nouvelle direction, déplace la tête, détecte une collision et retire la queue si aucune nourriture n'a été consommée.

Une politique débutante ne doit pas recevoir toute la grille. Nous lui donnons onze signaux faciles à interpréter : trois indicateurs de danger devant, à gauche et à droite ; quatre indicateurs de position de la nourriture ; quatre indicateurs de direction courante. Les signaux sont binaires, ce qui rend la première expérience lisible et évite de cacher un prétraitement complexe.

Groupe	Nombre	Exemple	Rôle
Danger relatif	3	mur devant	éviter une collision immédiate
Nourriture	4	nourriture à droite	orienter la trajectoire
Direction	4	déplacement vers le haut	interpréter gauche et droite

Groupe	Nombre	Exemple	Rôle
Total	**11**		entrée de la politique

Code de l'étape

La fonction suivante construit les observations à partir de l'état courant :

Python

```python
def state_features(game):
    head_x, head_y = game.snake[0]
    relative_directions = (
        (game.direction - 1) % 4,
        game.direction,
        (game.direction + 1) % 4,
    )
    danger = []
    for direction in relative_directions:
        dx, dy = DIRECTIONS[direction]
        candidate = (head_x + dx, head_y + dy)
        danger.append(float(
            not (0 ⩽ candidate[0] < game.board_size and 0 ⩽ candidate[1] < game.board_size)
            or candidate in game.snake
        ))

    food_x, food_y = game.food or (head_x, head_y)
    food_direction = [
        float(food_x < head_x), float(food_x > head_x),
        float(food_y < head_y), float(food_y > head_y),
    ]
    heading = [float(game.direction == index) for index in range(4)]
    return tuple(danger + food_direction + heading)
```

Dans le code complet, la fonction vérifie aussi qu'elle retourne exactement onze valeurs. Cette assertion est utile : si une observation est ajoutée sans augmenter la taille du chromosome, la politique ne comparera plus les actions sur les mêmes données.

Sortie obtenue

Sortie / exemple

```
features=11 actions=(-1, 0, 1) board=10x10
```

Les actions sont relatives. Le gène `-1` ne signifie pas « aller à l'ouest » : il signifie « tourner à gauche par rapport à la direction actuelle ». Cette convention permet de réutiliser la même politique lorsque le serpent change d'orientation.

Commande de vérification

Terminal · Bash

```
python - <<'PY'
from src.snake_ga import FEATURE_COUNT, SnakeGame, state_features

features = state_features(SnakeGame(seed=7))
assert len(features) == FEATURE_COUNT == 11
assert set(features) <= {0.0, 1.0}
print("observations: PASS", len(features))
PY
```

3. Construire une politique linéaire et une fitness

Le chromosome contient trois blocs de onze poids. Le premier bloc donne un score à l'action gauche, le deuxième à l'action tout droit et le troisième à l'action droite. Pour chaque bloc, nous calculons un produit scalaire avec les observations. L'action dont le score est le plus élevé est choisie ; il n'y a ni couche cachée ni activation entraînée par gradient.

Python

```python
ACTION_VALUES = (-1, 0, 1)
FEATURE_COUNT = 11
GENOME_SIZE = len(ACTION_VALUES) * FEATURE_COUNT

def policy_action(genome, features):
    scores = []
    for action_index in range(3):
        start = action_index * FEATURE_COUNT
        weights = genome[start:start + FEATURE_COUNT]
        scores.append(sum(weight * feature for weight, feature in zip(weights, features)))
    return ACTION_VALUES[max(range(3), key=scores.__getitem__)]
```

La simulation donne ensuite une note à la politique. La formule est volontairement simple : manger une nourriture vaut 1000 points et chaque pas de survie ajoute un point. Le coefficient élevé impose une priorité claire : manger est plus important que tourner en rond longtemps. Dans un projet réel, ce coefficient devrait être analysé et testé avec des scénarios de comportement indésirable.

Python

```python
fitness = foods_eaten * 1000.0 + steps_survived
```

Sortie obtenue

Sortie / exemple

```
genome_size=33
fitness=1000 * foods + steps
```

Un agent qui ne mange rien mais survit 300 pas obtient 300. Un agent qui mange deux nourritures et meurt après 20 pas obtient 2020. Cette différence rend l'objectif visible dans les sorties et facilite la comparaison de deux individus.

Commande de vérification

Terminal · Bash

```
python - <<'PY'
from src.snake_ga import GENOME_SIZE, simulate

genome = tuple(0.1 for _ in range(GENOME_SIZE))
result = simulate(genome, seed=42, max_steps=50)
assert len(genome) == 33
assert result.fitness == result.foods * 1000 + result.steps
print("simulation: PASS", result)
PY
```

La graine est fixée dans la simulation pour que deux génomes soient comparés sur la même séquence de nourriture. Sans cette précaution, un agent pourrait sembler meilleur simplement parce qu'il a reçu un tirage plus favorable.

4. Coder sélection, croisement et mutation

La population initiale contient des poids tirés uniformément entre `-1` et `1`. La sélection par tournoi tire trois candidats et conserve le meilleur. Elle n'impose pas de trier toute la population à chaque choix et conserve une pression de sélection assez modérée pour un premier exemple.

Python

```
def tournament_select(scored, rng, size=3):
    contestants = rng.sample(scored, k=min(size, len(scored)))
    return max(contestants, key=lambda candidate: candidate.result.fitness).genome
```

Le croisement à un point prend une partie du premier parent et le reste du second. La mutation ajoute un bruit gaussien à un poids avec une probabilité de `0.08`. Les poids sont bornés entre `-3` et `3` pour éviter qu'une mutation isolée ne domine tous les scores.

Python

```python
def crossover_one_point(first, second, rng):
    point = rng.randint(1, len(first) - 1)
    return first[:point] + second[point:]

def mutate(genome, rate, rng):
    return tuple(
        max(-3.0, min(3.0, gene + rng.gauss(0.0, 0.35)))
        if rng.random() < rate else gene
        for gene in genome
    )
```

L'élitisme copie les deux meilleurs chromosomes directement dans la génération suivante. Cela garantit qu'une bonne solution ne disparaît pas à cause d'un croisement ou d'une mutation. Le reste de la population est rempli avec des enfants, jusqu'à retrouver la taille initiale.

Sortie obtenue

Sortie / exemple

```
population=80
selection=tournament(size=3)
crossover=one-point
mutation=gaussian(rate=0.08)
elitism=2
```

Ces valeurs sont un point de départ, pas une configuration universelle. Une population plus grande explore davantage mais demande plus de simulations ; une mutation plus forte explore plus largement mais peut détruire des comportements utiles.

Commande de vérification

Terminal · Bash

```bash
python - <<'PY'
import random
from src.snake_ga import GENOME_SIZE, crossover_one_point, mutate

rng = random.Random(42)
first = tuple(0.0 for _ in range(GENOME_SIZE))
second = tuple(1.0 for _ in range(GENOME_SIZE))
child = crossover_one_point(first, second, rng)
```

```
mutated = mutate(child, 0.2, rng)
assert len(child) == len(mutated) == GENOME_SIZE
assert all(-3 <= gene <= 3 for gene in mutated)
print("opérateurs: PASS")
PY
```

5. Faire évoluer et mesurer la population

La boucle d'évolution évalue toute la population avec la même graine, conserve le meilleur global et crée ensuite une nouvelle population. Le fichier `evolution.csv` garde une ligne par génération pour séparer la mesure de la visualisation. Le résumé JSON contient le résultat final sans enregistrer tout le chromosome dans le terminal.

Lancer l'expérience complète prend quelques secondes sur une machine standard : 80 agents, 60 générations et au plus 300 pas par partie. Pour un premier essai plus rapide, utilisez `--population 24 --generations 10`. Le résultat sera moins stable, mais le mécanisme restera identique.

Code de l'étape

Le cœur du remplacement est court :

Python

```python
for generation in range(generations + 1):
    scored = [evaluate(genome) for genome in population]
    best = max(scored, key=lambda candidate: candidate.result.fitness)
    history.append({
        "generation": generation,
        "fitness": best.result.fitness,
        "foods": best.result.foods,
        "steps": best.result.steps,
    })
    if generation == generations:
        break

    elites = sorted(scored, key=rank, reverse=True)[:2]
    next_population = [elite.genome for elite in elites]
    while len(next_population) < population_size:
        first = tournament_select(scored, rng)
        second = tournament_select(scored, rng)
        child = mutate(crossover_one_point(first, second, rng), mutation_rate, rng)
        next_population.append(child)
    population = next_population
```

Sortie obtenue

Sortie / exemple

```
generations=60 genome_size=33
best_fitness=18300 foods=18 steps=300
best_length=21 last_generation=60
outputs=evolution.csv,run-summary.json
```

L'évolution ne signifie pas que le Snake comprend le jeu comme un humain. Elle signifie que, pour cette représentation, cette fitness et cette graine, certains chromosomes produisent des trajectoires plus longues et plus nourries que d'autres. L'image « meilleur agent » reprend cette lecture expérimentale.

Commande de vérification

Terminal · Bash

```
python src/snake_ga.py --evolve --headless
python - <<'PY'
import csv
import json
from pathlib import Path

summary = json.loads(Path("outputs/run-summary.json").read_text())
with Path("outputs/evolution.csv").open(newline="") as handle:
    rows = list(csv.DictReader(handle))
assert summary["foods"] ⩾ 0
assert len(rows) == 61
assert rows[-1]["generation"] == "60"
print("mesures: PASS", len(rows), "générations")
PY
```

Extrait de `outputs/evolution.csv`

Génération	Fitness globale	Nourriture	Pas
0	1300	1	300
2	2020	2	20
10	3027	3	27
60	18300	18	300

La meilleure fitness globale ne redescend pas, car `best` est conservé d'une génération à l'autre. Le comportement de la population courante peut, lui, varier ; ne confondez pas la meilleure valeur historique avec la moyenne de tous les individus.

6. Observer le meilleur agent dans Pygame

Une visualisation aide à relier les nombres à un comportement. Le mode graphique réutilise exactement `SnakeGame`, `state_features` et `policy_action` utilisés pendant l'évaluation. Il ne réentraîne pas l'agent : il rejoue le meilleur génome pour rendre la trajectoire visible.

Lancez l'entraînement puis la partie :

Terminal · Bash

```
python src/snake_ga.py --evolve --play
```

Une fenêtre de grille s'ouvre. Le carré bleu représente le corps, le cercle rouge la nourriture et la partie se termine à la collision ou après la limite de pas. Fermez la fenêtre pour terminer le processus. Sur une machine sans affichage, gardez `--headless` et utilisez la figure de résultat ainsi que les fichiers CSV/JSON.

Sortie obtenue

Sortie / exemple

```
generations=60 genome_size=33
best_fitness=18300 foods=18 steps=300
outputs=evolution.csv,run-summary.json
pygame_window=opened
```

La fenêtre permet surtout de détecter les défauts de la fitness : un agent peut accumuler des pas en tournant dans une zone sûre, ou prendre des risques qui rapportent de la nourriture mais provoquent rapidement une collision. Le comportement observé doit toujours être mis en relation avec les mesures.

La capture ci-dessous montre le résultat obtenu avec la configuration par défaut. Le plateau rend la trajectoire lisible, tandis que le panneau de droite reprend exactement les valeurs imprimées par le programme : la figure ne remplace donc pas la mesure, elle la met en contexte visuel.

Commande de vérification

Terminal · Bash

```
python src/snake_ga.py --evolve --headless --generations 10 --population 24
python -m unittest discover -s tests -v
```

La première commande permet de valider le parcours sans interface graphique ; la seconde vérifie la taille du génome, la reproductibilité de la simulation et la conservation de la longueur après les opérateurs génétiques.

Exemples de sorties

Le résumé produit par la configuration par défaut ressemble à ceci :

JSON

```
{
  "genome_size": 33,
  "fitness": 18300.0,
  "foods": 18,
  "steps": 300,
  "length": 21
}
```

La valeur exacte dépend de la version du code et de la graine, mais une exécution identique doit produire le même résultat. Pour comparer honnêtement deux paramètres, ne changez qu'un paramètre à la fois et conservez la graine ; pour comparer la robustesse, utilisez ensuite plusieurs graines et rapportez la moyenne ainsi que la dispersion.

Dépannage et diagnostics

Le programme se termine immédiatement — vérifiez que vous avez lancé la commande depuis la racine de l'archive et que `src/snake_ga.py` est présent. Le mode `--self-test` permet de distinguer une erreur d'installation d'une erreur d'évolution.

Pygame ne s'installe pas ou aucune fenêtre ne s'ouvre — continuez en mode `--headless`. Le GA n'a pas besoin de Pygame pour être évalué ; sur un serveur distant, un affichage graphique n'est généralement pas disponible.

Le meilleur agent ne mange aucune nourriture — inspectez la fitness et la graine, puis augmentez temporairement la population ou le nombre de générations. Vérifiez aussi que les signaux de position de la nourriture sont bien présents dans les onze features.

La fitness augmente mais le comportement semble mauvais — la fitness mesure uniquement nourriture et survie. Un agent peut exploiter une faiblesse de la grille ou tourner en rond. Ajoutez une pénalité de répétition ou une récompense de distance seulement après avoir observé le problème, afin de garder le lien entre le code et l'objectif.

Les résultats changent à chaque lancement — passez la même `--seed` et utilisez la même version du code. Si vous changez `--population`, `--generations` ou `--mutation-rate`, le résultat attendu change légitimement.

Le Snake se cogne souvent — ce n'est pas forcément un bug : les premières générations contiennent des politiques aléatoires. Vérifiez d'abord `test_simulation_is_reproducible`, puis augmentez le nombre de générations avant de modifier les règles de collision.

Vérification finale

Depuis la racine du projet, exécutez :

Terminal · Bash

```
python src/snake_ga.py --self-test
python -m unittest discover -s tests -v
python src/snake_ga.py --evolve --headless
python - <<'PY'
import json
from pathlib import Path

summary = json.loads(Path("outputs/run-summary.json").read_text(encoding="utf-8"))
assert summary["genome_size"] == 33
assert summary["foods"] >= 0
assert Path("outputs/evolution.csv").exists()
print("contrat final: PASS")
PY
```

Le tutoriel est réussi lorsque les tests passent, que les deux fichiers de sortie existent et que le meilleur agent peut être rejoué avec `--play` sur une machine équipée d'un affichage.

Limites et prolongements

Cette expérience est une introduction à la recherche évolutive, pas une preuve qu'un GA est toujours préférable à l'apprentissage par renforcement ou à une stratégie codée à la main. La politique linéaire est expressive mais limitée ; elle ne mémorise pas la trajectoire et ne voit pas toute la grille. La fitness est également sensible à la règle de fin et au coefficient donné à la nourriture.

Pour aller plus loin, vous pouvez comparer plusieurs graines, ajouter une pénalité de boucle, fournir une vision plus riche de la grille ou remplacer la politique linéaire par un petit réseau dont les poids sont toujours optimisés par GA. Dans tous les cas, gardez une simulation headless, une fitness testable et une baseline simple afin de savoir si l'évolution apporte réellement un gain.

Références

- Python — module random

- Pygame — documentation officielle
- John H. Holland — Adaptation in Natural and Artificial Systems