

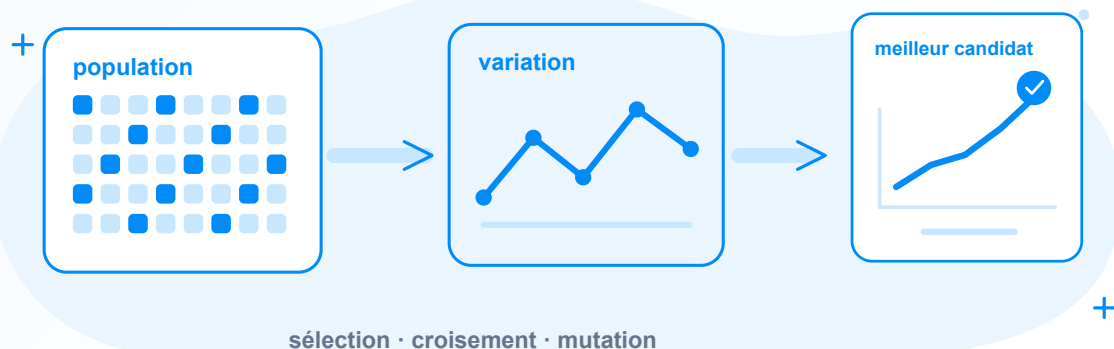
Tutoriel

Algorithmique · optimisation · Python

Implémenter un algorithme génétique à partir de zéro

Construire un algorithme génétique sans bibliothèque d'optimisation pour sélectionner un portefeuille de projets Data/IA sous budget.

Auteur **Youness Boutyour**

**DURÉE**

60 minutes

NIVEAU

Intermédiaire

FORMAT

Tutoriel pratique

Algorithmique · optimisation · Python · publié le 2026-10-16 · local

Objectifs et livrables

Choisir un portefeuille de projets Data/IA revient souvent à arbitrer entre un coût et un impact. Le portefeuille idéal est facile à décrire, mais devient moins évident lorsque le nombre de combinaisons augmente. Nous allons construire un algorithme génétique capable de proposer une sélection de projets sous un budget de 28 kEUR.

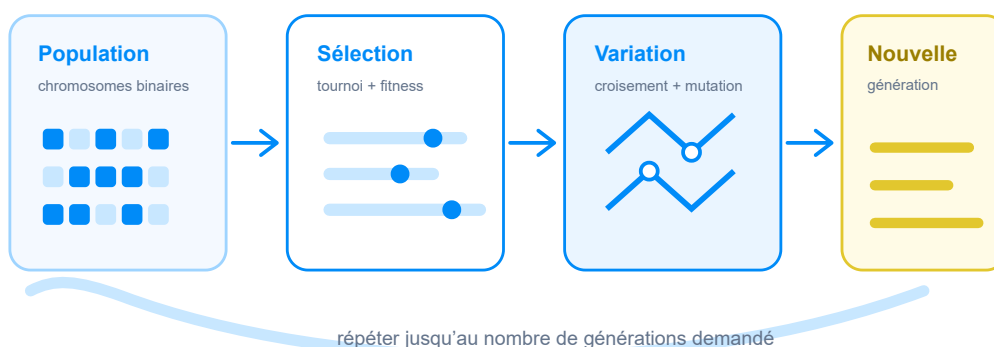
Chaque projet sera représenté par un gène binaire : 1 signifie qu'il est sélectionné, 0 qu'il est écarté. Un chromosome complet représente donc un portefeuille. La fonction de fitness additionnera les points d'impact et appliquera une pénalité si le coût dépasse le budget. Une population de chromosomes sera ensuite améliorée par sélection de tournoi, croisement à un point, mutation et élitisme.

Tout sera écrit en Python avec la bibliothèque standard. La graine aléatoire sera fixée pour obtenir une expérience reproductible. Pour contrôler le résultat sur notre petit catalogue, nous calculerons aussi l'optimum par énumération exhaustive ; cette comparaison sert à tester l'algorithme, pas à remplacer la recherche génétique lorsque le nombre de projets devient grand.

À la fin, le programme produira un portefeuille, un historique de convergence et un résumé JSON. Il retrouvera une valeur de 231 points pour un coût de 28 kEUR avec la configuration fournie. L'algorithme n'utilise ni LLM ni modèle génératif : le choix des projets repose uniquement sur les données, la fonction objectif et la graine définies dans le code.

Une génération devient la suivante

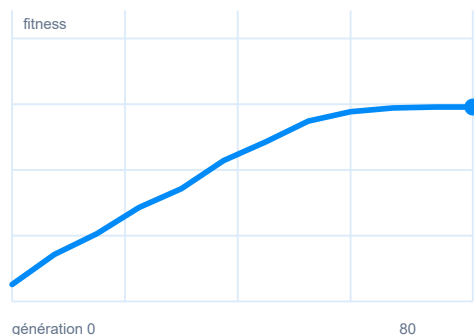
La fitness guide la sélection sans garantir une solution optimale



À chaque génération, les meilleurs candidats sont sélectionnés puis recombinaés et mutés.

Lire la convergence

La fitness progresse puis se stabilise autour d'un portefeuille faisable



Meilleure solution faisable



Budget : 28 k€ · valeur : 246

DQ01 · Audit qualité

CV01 · Contrôle vision

MLO01 · Monitoring modèle

+ autres projets sélectionnés

La courbe suit la meilleure fitness et le tableau résume le portefeuille final.

Faire évoluer une population de solutions

Schéma du cycle génétique qui évalue une population, sélectionne les parents, crée des variations et conserve le meilleur candidat.



Le cycle sépare clairement l'évaluation de la fitness, la création de variantes et l'élitisme.

Lire le cycle avant de coder

Le schéma montre qu'une génération n'est pas une simple liste de scores. La population fournit les candidats, la fitness mesure leur qualité, les opérateurs créent des variantes et l'élitisme protège une bonne solution contre une dégradation aléatoire. Cette séparation rend chaque paramètre observable dans le code.

Pour interpréter la convergence, regardez simultanément la meilleure fitness, le coût du portefeuille et la diversité des chromosomes. Une courbe plate peut signifier que l'optimum est atteint, que la mutation est trop faible ou que la représentation empêche d'explorer une zone utile. La comparaison exhaustive sur le petit catalogue permet de distinguer ces cas pendant la vérification.

Quelques repères sur les algorithmes génétiques

Un algorithme génétique est une métaheuristique : il explore l'espace des solutions avec une population de candidats plutôt que de suivre une seule trajectoire. Il s'inspire de quatre mécanismes opérationnels :

1. **Évaluation** : calculer la fitness de chaque chromosome.
2. **Sélection** : donner plus de chances aux bons candidats de devenir parents.
3. **Variation** : croiser deux parents puis muter certains gènes.
4. **Remplacement** : former une nouvelle population, en conservant éventuellement un élite.

La représentation est déterminante. Ici, un chromosome de longueur 12 correspond aux 12 lignes du catalogue CSV. Le gène placé à l'index 0 concerne DQ01, celui à l'index 1 concerne BI01, et ainsi de suite. Cette correspondance rend le décodage explicite et évite de cacher la logique métier dans une structure opaque.

La fitness n'est pas une vérité universelle : elle traduit le compromis choisi. Nous utilisons :

Sortie / exemple

```
fitness = impact_total - 100 * max(0, coût_total - budget)
```

Une solution faisable est prioritaire dans le classement final. La pénalité permet néanmoins de comparer les individus pendant l'exploration et de faire revenir progressivement la population vers la zone autorisée.

1. Préparer les données et le projet

Le catalogue contient 12 projets Data/IA avec un coût et un impact estimé. Le format CSV permet de remplacer facilement les valeurs par un jeu réel, mais ces scores restent une hypothèse de démonstration : dans un projet professionnel, ils devraient être documentés, normalisés et validés par les personnes qui prennent la décision.

Créez l'environnement et vérifiez que le projet n'a aucune dépendance externe :

Terminal · Bash

```
mkdir -p algorithme-genetique-python/{data,src,tests,outputs}
cd algorithme-genetique-python
python3.11 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -c "import sys; print(sys.version.split()[0])"
```

Le fichier data/projects.csv est fourni dans l'archive. Ses colonnes sont :

- project_id : identifiant stable du projet ;
- name : nom lisible ;
- cost_k_eur : coût estimé en milliers d'euros ;
- impact_points : valeur utilisée par la fonction objectif.

Extrait de data/projects.csv

Identifiant	Projet	Coût (k€)	Impact (points)
DQ01	Audit qualité des données	4	38
BI01	Tableau de bord des ventes	6	44
ML01	Prévision de la demande	9	62
MLO01	Monitoring du modèle	5	41
LLM01	RAG documentaire interne	8	58
SEC01	Audit de confidentialité	7	50

Cet extrait montre la structure du catalogue utilisé pour calculer le coût, la valeur et la fitness de chaque portefeuille.

Chargez les lignes et validez les contraintes de base avant de créer les chromosomes :

Python

```
import csv
from pathlib import Path

path = Path("data/projects.csv")
with path.open(newline="", encoding="utf-8") as handle:
    rows = list(csv.DictReader(handle))

assert len(rows) == 12
assert len({row["project_id"] for row in rows}) == 12
assert all(int(row["cost_k_eur"]) > 0 for row in rows)
assert all(int(row["impact_points"]) > 0 for row in rows)
print("catalogue:", len(rows), "projets")
```

Sortie obtenue

Sortie / exemple

```
3.14.x  
catalogue: 12 projets
```

La valeur de la version Python varie. Le nombre de projets et l'unicité des identifiants sont les premiers éléments du contrat de données.

Commande de vérification

Terminal · Bash

```
python - <<'PY'  
import csv  
from pathlib import Path  
  
with (Path("data") / "projects.csv").open(newline="", encoding="utf-8") as handle:  
    rows = list(csv.DictReader(handle))  
print("lignes:", len(rows))  
print("colonnes:", ",".join(rows[0]))  
assert len(rows) == 12  
assert set(rows[0]) == {"project_id", "name", "cost_k_eur", "impact_points"}  
PY
```

Sortie / exemple

```
lignes: 12  
colonnes: project_id,name,cost_k_eur,impact_points
```

2. Encoder et évaluer un portefeuille

Nous commençons par les structures qui rendent l'algorithme lisible. Un objet `Project` conserve les données métier ; un `Candidate` contient le chromosome et les valeurs calculées. La fitness pénalise les dépassements, tandis que `feasible` permet de préférer une solution autorisée au moment de présenter le résultat.

Python

```
from dataclasses import dataclass  
  
@dataclass(frozen=True)
```

```

class Project:
    project_id: str
    name: str
    cost: int
    value: int

@dataclass(frozen=True)
class Candidate:
    chromosome: tuple[int, ...]
    cost: int
    value: int
    fitness: float
    feasible: bool

```

Le chromosome est un tuple de zéros et de uns. Pour l'évaluer, nous multiplions le coût et la valeur de chaque projet par le gène correspondant. Le contrôle de longueur est important : un chromosome trop court pourrait ignorer une contrainte sans provoquer d'erreur visible.

● ● ● Python

```

def evaluate(
    chromosome: tuple[int, ...],
    projects: list[Project],
    budget: int,
) → Candidate:
    if len(chromosome) != len(projects):
        raise ValueError("un gène est attendu par projet")
    cost = sum(project.cost * gene for project, gene in zip(projects, chromosome))
    value = sum(project.value * gene for project, gene in zip(projects, chromosome))
    excess = max(0, cost - budget)
    fitness = value - excess * 100.0
    return Candidate(chromosome, cost, value, fitness, cost ≤ budget)

```

Testez deux cas extrêmes : le chromosome nul ne sélectionne rien, tandis que le chromosome plein sélectionne tout et dépasse le budget. Cette vérification isole la fonction objectif avant d'introduire le hasard.

Sortie obtenue

■ Sortie / exemple

```

chromosome nul : coût=0, valeur=0, faisable=True
chromosome plein : coût=69, valeur=529, faisable=False

```

Le chromosome plein est pénalisé par son dépassement. Les valeurs exactes sont calculées à partir du CSV ; elles changeraient si le catalogue était remplacé.

Commande de vérification

Terminal · Bash

```
python - <<'PY'
from src.genetic_algorithm import evaluate, load_projects
projects = load_projects("data/projects.csv")
empty = evaluate((0,) * len(projects), projects, 28)
full = evaluate((1,) * len(projects), projects, 28)
assert empty.cost == 0 and empty.feasible
assert full.cost == 69 and not full.feasible
assert full.fitness < full.value
print("évaluation: PASS")
PY
```

Sortie / exemple

évaluation: PASS

3. Implémenter sélection, croisement et mutation

La population initiale est composée de chromosomes aléatoires créés avec `random.Random`. Utiliser une instance locale et une graine explicite permet de reproduire le même parcours sans modifier l'état aléatoire global du programme.

La sélection par tournoi tire trois individus et conserve le meilleur selon sa fitness. Elle est plus simple à régler qu'un calcul de roulette et conserve une pression de sélection modérée : un individu faible peut encore être choisi s'il rencontre des concurrents plus faibles dans son tournoi.

Python

```
def tournament_select(population, scored, rng, tournament_size=3):
    indices = rng.sample(
        range(len(population)),
        k=min(tournament_size, len(population)),
    )
    winner = max(indices, key=lambda index: scored[index].fitness)
    return population[winner]
```

Le croisement à un point prend un préfixe du premier parent et un suffixe du second. La mutation inverse chaque gène selon une probabilité indépendante. Le taux doit rester faible : trop élevé, il détruit les bons blocs ; trop faible, il réduit l'exploration de nouvelles combinaisons.

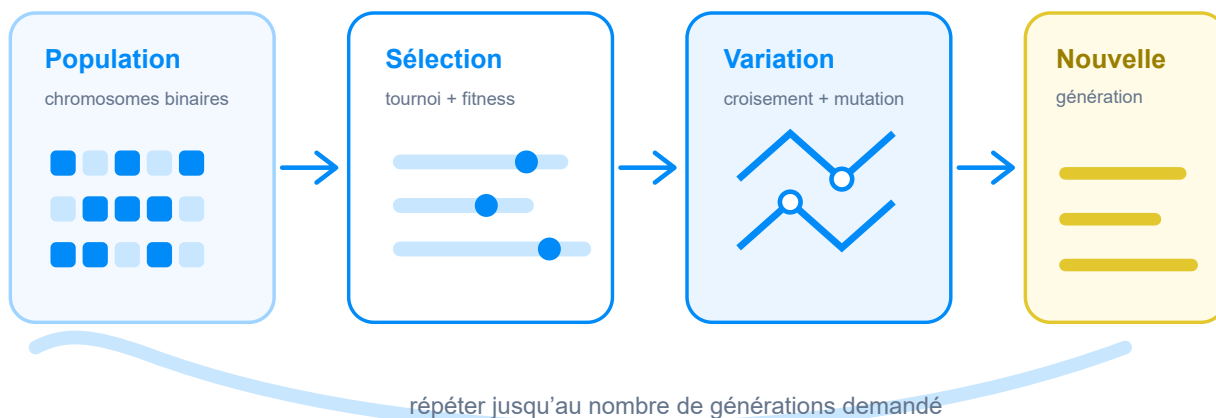
Python

```
def crossover_one_point(first, second, rng):  
    point = rng.randint(1, len(first) - 1)  
    return first[:point] + second[point:]  
  
def mutate(chromosome, rate, rng):  
    return tuple(  
        1 - gene if rng.random() < rate else gene  
        for gene in chromosome  
    )
```

L'image ci-dessous résume le cycle exécuté à chaque génération :

Une génération devient la suivante

La fitness guide la sélection sans garantir une solution optimale



Sortie obtenue

■ Sortie / exemple

```
population : 80 chromosomes de longueur 12  
sélection : tournoi de taille 3  
mutation : taux 0.06
```

Ces paramètres ne sont pas des constantes théoriques. Ils forment une configuration de départ qui doit être mesurée sur le problème réel.

Commande de vérification

Terminal · Bash

```
python - <<'PY'
import random
from src.genetic_algorithm import crossover_one_point, mutate

rng = random.Random(42)
first = (1, 0, 1, 0, 1)
second = (0, 1, 0, 1, 0)
child = crossover_one_point(first, second, rng)
mutated = mutate(child, 0.2, rng)
assert len(mutated) == 5
assert set(mutated) <= {0, 1}
print("opérateurs: PASS", mutated)
PY
```

Sortie / exemple

opérateurs: PASS (1, 0, 1, 1, 0)

4. Faire évoluer la population avec élitisme

L'évolution répète le cycle pendant un nombre fixé de générations. À chaque tour, nous évaluons tous les chromosomes, mettons à jour le meilleur candidat, écrivons une ligne d'historique puis construisons la population suivante.

L'élitisme copie le meilleur chromosome sans mutation dans la nouvelle population. Cette précaution empêche une bonne solution de disparaître uniquement parce qu'elle n'a pas été sélectionnée ou qu'un enfant a muté. Le reste de la génération est produit par sélection, croisement puis mutation.

Python

```
for generation in range(generations + 1):
    scored = [evaluate(chromosome, projects, budget)
               for chromosome in population]
    current = max(scored, key=rank)
    if best is None or rank(current) > rank(best):
        best = current
    history.append({
        "generation": generation,
        "best_fitness": round(best.fitness, 2),
        "best_value": best.value,
        "best_cost": best.cost,
    })
```

```
if generation == generations:
    break

next_population = [best.chromosome]
while len(next_population) < population_size:
    first = tournament_select(population, scored, rng)
    second = tournament_select(population, scored, rng)
    child = crossover_one_point(first, second, rng)
    next_population.append(mutate(child, mutation_rate, rng))
population = next_population
```

Le classement final préfère d'abord la faisabilité, puis la valeur, puis la fitness. Dans un problème multiobjectif, ce classement devrait être remplacé par une méthode explicite, par exemple une frontière de Pareto ou une combinaison de scores normalisés.

Sortie obtenue

Sortie / exemple

```
Budget : 28 kEUR
Meilleure fitness faisable : 231 points pour 28 kEUR
Optimum exhaustif : 231 points | écart : 0
```

Le résultat est reproductible avec la graine 42 et la configuration par défaut. Une autre graine peut produire une solution de même valeur avec une composition différente, ou une solution légèrement moins bonne si la recherche est trop courte.

Commande de vérification

Terminal · Bash

```
python -m src.portfolio --budget 28 --seed 42 --generations 80
```

Sortie / exemple

```
- DQ01 | Audit qualité des données | 4 kEUR | 38 points
- BI01 | Tableau de bord des ventes | 6 kEUR | 44 points
- ML001 | Monitoring du modèle | 5 kEUR | 41 points
- ENG01 | Observabilité du pipeline | 3 kEUR | 29 points
- OPS01 | Déploiement d'une API modèle | 6 kEUR | 46 points
- EXP01 | Suivi des expériences | 4 kEUR | 33 points
```

Comparer, mesurer et tester

5.

Sur 12 projets, il est encore possible d'énumérer les 2^{12} chromosomes. La fonction brute force du projet calcule cet optimum exact uniquement pour vérifier l'algorithme génétique. À partir de dizaines ou centaines de variables, cette stratégie devient rapidement trop coûteuse ; c'est précisément l'intérêt d'une métaheuristique.

Le programme écrit `convergence.csv` avec la meilleure fitness, la valeur et le coût à chaque génération. Le fichier `run-summary.json` conserve la graine, les identifiants sélectionnés et l'écart à l'optimum exact. Ces sorties sont plus utiles qu'un simple message terminal lorsqu'il faut comparer plusieurs taux de mutation.

Extrait de `outputs/convergence.csv`

Génération	Meilleure fitness	Valeur	Coût (k€)
0	229,0	229	28
1	229,0	229	28
2	229,0	229	28
3	229,0	229	28
4	229,0	229	28

L'extrait décrit la progression de la meilleure solution ; il s'agit d'une trace de sortie et non d'un second jeu de données d'entrée.

```
Terminal · Bash

python -m src.portfolio \
  --budget 28 \
  --population-size 80 \
  --generations 80 \
  --mutation-rate 0.06 \
  --seed 42
```

Sortie obtenue

■ Sortie / exemple

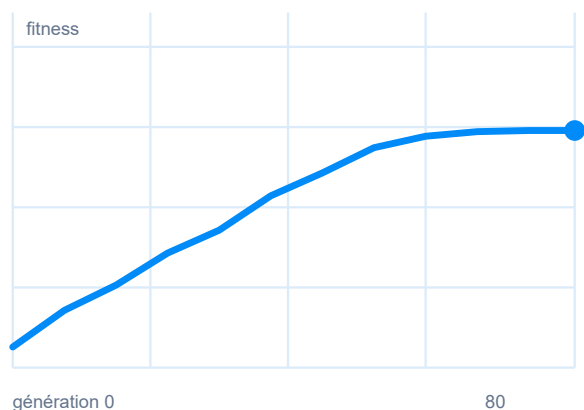
Budget : 28 kEUR
Meilleure fitness faisable : 231 points pour 28 kEUR
Projets sélectionnés :

```
- DQ01 | Audit qualité des données | 4 kEUR | 38 points
- BI01 | Tableau de bord des ventes | 6 kEUR | 44 points
- MLO01 | Monitoring du modèle | 5 kEUR | 41 points
- ENG01 | Observabilité du pipeline | 3 kEUR | 29 points
- OPS01 | Déploiement d'une API modèle | 6 kEUR | 46 points
- EXP01 | Suivi des expériences | 4 kEUR | 33 points
Optimum exhaustif : 231 points | écart : 0
```

La valeur de 231 est atteinte avec un coût exactement égal au budget. La convergence peut être visualisée avec le fichier CSV, sans ajouter de bibliothèque graphique au tutoriel.

Lire la convergence

La fitness progresse puis se stabilise autour d'un portefeuille faisable



Meilleure solution faisable



Budget : 28 k€ · valeur : 246

DQ01 · Audit qualité

CV01 · Contrôle vision

MLO01 · Monitoring modèle

+ autres projets sélectionnés

Commande de vérification

Terminal · Bash

```
python -m unittest discover -s tests -p 'test_*.py' -v
```

Sortie / exemple

```
test_evaluation_penalizes_over_budget_chromosome ... ok
test_operators_keep_binary_chromosome_shape ... ok
test_seeded_genetic_run_reaches_exact_optimum ... ok
```

Ran 3 tests in 0.1xxs

OK

Les tests vérifient la pénalité de contrainte, la forme binaire des opérateurs et la reproductibilité du résultat avec la graine définie.

Exemples de sorties

Le fichier run-summary.json produit par l'exécution par défaut contient :

```
JSON
{
  "budget_k_eur": 28,
  "seed": 42,
  "generations": 80,
  "selected_project_ids": [
    "DQ01",
    "BI01",
    "ML001",
    "ENG01",
    "OPS01",
    "EXP01"
  ],
  "genetic_value": 231,
  "genetic_cost": 28,
  "exact_value": 231,
  "optimality_gap": 0
}
```

L'historique commence par une génération aléatoire puis conserve la meilleure fitness connue. La courbe ne doit jamais redescendre dans cette implémentation grâce à l'élitisme ; si elle baisse, le code ne suit probablement pas le meilleur global ou remplace l'élite par un enfant muté.

Dépannage et diagnostics

La solution dépasse le budget — vérifiez que le classement final préfère `candidate.feasible` et que la pénalité est calculée avec $\max(0, \text{cost} - \text{budget})$. La population peut contenir des individus irréalisables pendant l'exploration, mais le résultat présenté doit être faisable.

Le résultat change entre deux exécutions — passez explicitement `--seed 42` et utilisez le même catalogue. Une graine ne rend pas déterministes des données ou des fonctions qui utilisent le générateur aléatoire global ailleurs dans le processus.

L'optimum exhaustif est trop lent — il est volontairement réservé au petit jeu de validation. Désactivez cette comparaison pour un grand catalogue et comparez plutôt plusieurs exécutions avec des graines différentes.

La convergence stagne — augmentez progressivement la population ou le nombre de générations, puis testez un taux de mutation légèrement différent. Une mutation trop forte transforme la recherche en tirages presque aléatoires.

Une jointure ou une contrainte supplémentaire est nécessaire — encodez la contrainte dans `evaluate`, ou ajoutez une réparation de chromosome après mutation. Testez séparément cette règle avant de modifier les opérateurs.

Vérification finale

Depuis la racine de l'archive, exécutez les commandes suivantes :

Terminal · Bash

```
python -m unittest discover -s tests -p 'test_*.py' -v
python -m src.portfolio --budget 28 --seed 42
python - <<'PY'
import json
from pathlib import Path

summary = json.loads(Path("outputs/run-summary.json").read_text())
assert summary["genetic_cost"] ≤ summary["budget_k_eur"]
assert summary["optimality_gap"] == 0
assert Path("outputs/convergence.csv").exists()
print("contrat final: PASS")
PY
```

Sortie / exemple

```
OK
Meilleure fitness faisable : 231 points pour 28 kEUR
contrat final: PASS
```

Le tutoriel est terminé lorsque le portefeuille est faisable, que les trois tests passent et que l'historique de convergence est présent. Le résultat peut alors être comparé à une autre configuration sans modifier la fonction objectif.

Limites et prolongements

Un algorithme génétique ne garantit pas l'optimum global. La réussite dépend de la représentation, de la fonction de fitness, de la pression de sélection, du taux de mutation, de la taille de population et du nombre de générations. Une seule exécution ne suffit donc pas pour conclure qu'un paramétrage est meilleur.

Le catalogue d'exemple utilise une somme d'impacts indépendante. Avec des dépendances entre projets, des incompatibilités ou plusieurs objectifs, il faut enrichir `evaluate` et décider comment réparer ou pénaliser les chromosomes invalides. Pour des problèmes continus, remplacez le chromosome binaire par des réels ou des permutations, puis adaptez le croisement et la mutation.

Pour une étude sérieuse, exécutez plusieurs graines, mesurez la moyenne et la dispersion de la meilleure fitness, comparez une baseline gloutonne et conservez les paramètres dans un fichier de configuration. Si un agent ou un LLM pilote l'algorithme, il doit seulement choisir les paramètres autorisés et lire les résultats validés ; il ne doit pas réécrire la fonction objectif sans contrôle.

Références

- [Python — module random](#)
- [Python — unittest](#)
- [Introduction aux algorithmes génétiques — MIT OCW](#)