Le code implémente un algorithme de programmation génétique pour résoudre un problème de régression symbolique en cherchant à approximer la fonction cible $f(x) = x^2 + 2x + 1$ à l'aide d'arbres d'expressions. L'algorithme évolue une population d'arbres, sélectionnant, croisant et mutant les meilleurs arbres au fil des générations.

Voici une explication ligne par ligne :

### 1. Définition des Primitives et des Terminaux

```
PRIMITIVES = {
    'add': (operator.add, 2),
    'sub': (operator.sub, 2),
    'mul': (operator.mul, 2),
    'neg': (operator.neg, 1)
}
TERMINALS = ['x'] + [random.randint(-3, 3) for _ in range(5)]
```

Où **Primitives** signifie le dictionnaire qui contient les opérations mathématiques de base utilisées pour construire des arbres d'expressions. Chaque clé est une opération (par exemple, add, sub, mul, neg), et la valeur associée est un tuple contenant la fonction correspondante (operator.add, operator.sub, etc.) et l'arity (le nombre d'arguments requis par la fonction). Par exemple, add prend 2 arguments, tandis que neg prend 1 argument et les **Terminaux** : Ce sont les valeurs possibles pour les feuilles des arbres. Il y a une variable x (le terme indépendant) et cinq valeurs aléatoires générées entre -3 et 3. Ces terminaux sont utilisés pour construire des expressions.

### 2. Génération d'un Arbre Aléatoire :

```
def generate_random_tree(depth):
    if depth == 0 or (depth < 3 and random.random() < 0.3):
        return random.choice(TERMINALS)
    func_name = random.choice(list(PRIMITIVES.keys()))
    _, arity = PRIMITIVES[func_name]
    return [func_name] + [generate_random_tree(depth - 1) for _ in range(arity)]
```

Cette fonction génère un arbre d'expressions aléatoire en fonction de la profondeur spécifiée. L'arbre est construit de manière récursive :

- Si la profondeur atteint zéro, ou si un nombre aléatoire décide qu'une feuille doit être terminée, la fonction retourne un terminal (soit x, soit une constante).
- Sinon, elle choisit une opération aléatoire à partir des primitives disponibles, puis appelle récursivement generate_random_tree pour créer les sous-arbres nécessaires (en fonction de l'arity de l'opération).

### 3. Évaluation d'un Arbre

```python
def evaluate_tree(tree, x):
    if isinstance(tree, (int, float)):
        return tree
    elif tree == 'x':
        return x
    else:
        func_name = tree[0]
        func, arity = PRIMITIVES[func_name]
        args = [evaluate_tree(subtree, x) for subtree in tree[1:]]
        try:
            return func(*args)
        except:
            return 0  # Gérer les erreurs d'exécution
```

Cette fonction évalue un arbre donné pour une valeur spécifique de x.

- Si l'élément de l'arbre est une constante (entier ou flottant), il est retourné directement.
- Si l'élément de l'arbre est la variable x, la fonction retourne la valeur de x.
- Sinon, l'arbre est une opération. La fonction récupère l'opération et ses arguments, évalue récursivement chaque sous-arbre et applique l'opération.
- Si une erreur se produit pendant l'évaluation (par exemple, une division par zéro), la fonction renvoie zéro pour éviter l'échec du programme.
-

### 4. Fonction de Fitness (MSE)

```python
def fitness(tree, X, Y):
    try:
        y_pred = np.array([evaluate_tree(tree, x) for x in X])
        return np.mean((y_pred - Y)**2)
    except:
        return float('inf')
```

La fonction de **fitness** évalue la qualité de l'arbre en fonction de son erreur quadratique moyenne (MSE) par rapport aux valeurs cibles.

- X est un ensemble de valeurs d'entrée pour la variable x.
- Y est le vecteur des valeurs cibles pour chaque x.
- La fonction prédit les valeurs de sortie en évaluant l'arbre pour chaque valeur de x dans X, puis calcule l'erreur quadratique moyenne entre les prédictions et les valeurs réelles Y.

### 5. Mutation et Croisement

```python
def mutate(tree, depth=3):
    if isinstance(tree, list) and random.random() < 0.2:
        return generate_random_tree(depth)
    elif isinstance(tree, list):
        return [tree[0]] + [mutate(sub, depth-1) for sub in tree[1:]]
    else:
        return tree

def crossover(t1, t2):
    return random.choice([t1, t2])  # Simplifié
```

**Mutation :** La fonction mutate modifie aléatoirement un arbre d'expressions. Parfois, elle remplace l'arbre entier par un nouvel arbre généré aléatoirement. Sinon, elle applique la mutation de manière récursive à chaque sous-arbre. Et le **croisement** combine deux arbres en choisissant l'un des deux arbres au hasard. Cette fonction est simplifiée, et un croisement plus complexe pourrait combiner des sous-arbres de chaque parent.

## 6. Données Cibles

```python
X = np.linspace(-10, 10, 100)
Y = X**2 + 2*X + 1  # Fonction cible
```

Où X est un ensemble de valeurs d'entrée pour la variable indépendante x, allant de -10 à 10 avec 100 points. Et Y est la fonction cible que l'algorithme GP cherche à approximer.

## 7. Boucle de Programmation Génétique

```python
# ------------------------------
POP_SIZE = 50
N_GEN = 30
best_fitness_history = []

population = [generate_random_tree(3) for _ in range(POP_SIZE)]
```

```python
for gen in range(N_GEN):
    scored = [(tree, fitness(tree, X, Y)) for tree in population]
    scored.sort(key=lambda x: x[1])
    best_fitness_history.append(scored[0][1])

    print(f"Génération {gen+1}, Meilleure fitness : {scored[0][1]:.4f}")

    new_population = [scored[0][0]]  # Élitisme
    while len(new_population) < POP_SIZE:
        p1 = random.choice(scored[:10])[0]
        p2 = random.choice(scored[:10])[0]
        child = crossover(p1, p2)
        child = mutate(child)
        new_population.append(child)

    population = new_population
```

```
Génération 1, Meilleure fitness : 851.1684
Génération 2, Meilleure fitness : 851.1684
Génération 3, Meilleure fitness : 43.0067
Génération 4, Meilleure fitness : 35.0067
Génération 5, Meilleure fitness : 35.0067
Génération 6, Meilleure fitness : 35.0067
Génération 7, Meilleure fitness : 35.0067
Génération 8, Meilleure fitness : 16.0000
Génération 9, Meilleure fitness : 16.0000
Génération 10, Meilleure fitness : 16.0000
Génération 11, Meilleure fitness : 1.0000
Génération 12, Meilleure fitness : 1.0000
Génération 13, Meilleure fitness : 1.0000
Génération 14, Meilleure fitness : 1.0000
Génération 15, Meilleure fitness : 1.0000
Génération 16, Meilleure fitness : 1.0000
Génération 17, Meilleure fitness : 1.0000
Génération 18, Meilleure fitness : 1.0000
Génération 19, Meilleure fitness : 1.0000
Génération 20, Meilleure fitness : 1.0000
Génération 21, Meilleure fitness : 1.0000
Génération 22, Meilleure fitness : 1.0000
Génération 23, Meilleure fitness : 1.0000
Génération 24, Meilleure fitness : 1.0000
Génération 25, Meilleure fitness : 1.0000
Génération 26, Meilleure fitness : 1.0000
Génération 27, Meilleure fitness : 1.0000
Génération 28, Meilleure fitness : 1.0000
Génération 29, Meilleure fitness : 1.0000
Génération 30, Meilleure fitness : 1.0000
```

**Population initiale** : Une population de 50 arbres est générée aléatoirement, chacun ayant une profondeur de 3.

**Évaluation** : À chaque génération, chaque arbre est évalué en fonction de sa fitness (erreur quadratique moyenne).

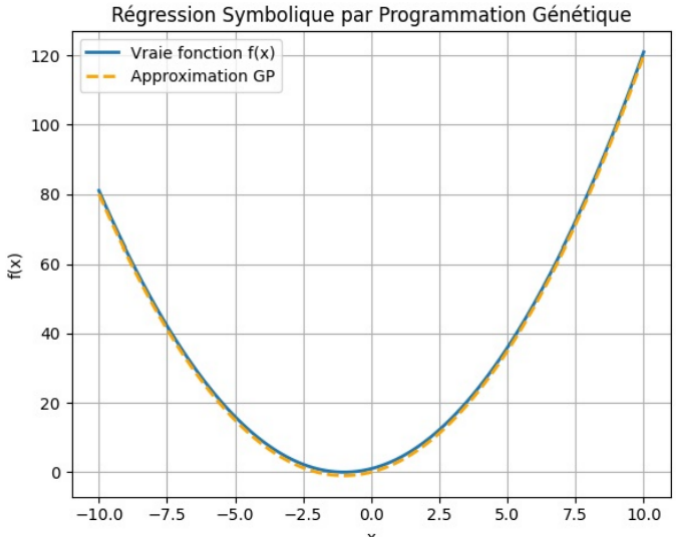
**Sélection et élitisme** : L'arbre avec la meilleure fitness est toujours conservé. Ensuite, un nouvel arbre est créé par croisement et mutation des meilleurs arbres de la population.

**Itérations** : L'algorithme évolue sur 30 générations, en affinant progressivement les arbres pour mieux correspondre à la fonction cible.

```python
best_tree = scored[0][0]
y_pred = np.array([evaluate_tree(best_tree, x) for x in X])

plt.figure(figsize=(12, 5))
```

```python
plt.subplot(1, 2, 1)
plt.plot(X, Y, label="Vraie fonction f(x)", linewidth=2)
plt.plot(X, y_pred, label="Approximation GP", linestyle='--', color='orange', linewidth=2)
plt.title("Régression Symbolique par Programmation Génétique")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
```

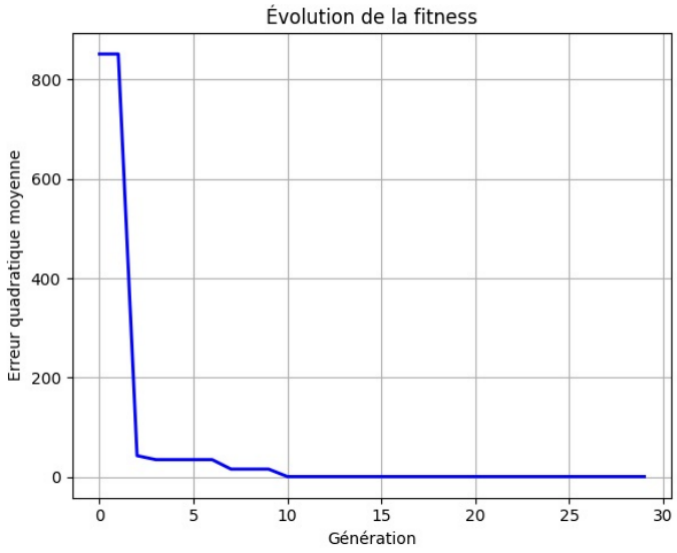



Ce graphique montre : La fonction réelle $f(x) = x^2 + 2x + 1$ comparée avec l'approximation obtenue par l'algorithme de programmation génétique.

```
plt.subplot(1, 2, 2)
plt.plot(best_fitness_history, color='blue', linewidth=2)
plt.title("Évolution de la fitness")
plt.xlabel("Génération")
plt.ylabel("Erreur quadratique moyenne")
plt.grid(True)

plt.tight_layout()
plt.show()

print("\n Arbre final trouvé :", best_tree)
```




Ce graphique illustre L'évolution de la fitness (erreur quadratique moyenne) au fil des générations, montrant comment l'algorithme converge vers une meilleure solution.