TP 1 - Algorithme génétique

Tricks utiles pour ce TP

Mapper une opération à chaque élément d'une liste

```
inputs = [ 1, 2, 3, 4 ]
output = [ x**2 for x in inputs ]
# Output = [ 1, 4, 9, 16 ]
```

S'assurer qu'une condition est vraie

```
assert condition == false # Erreur
assert condition == true # OK
```

Trier une liste avec une comparaison custom

```
class Toto:
  def __init__(self, score):
    self.score = score ** 2;

input = [Toto(4), Toto(3), Toto(9)]

output = sorted(input, key = lambda inp: inp.score)
# Output contient les classes Toto, ordonnées du plus petit score, au plus grand

output = sorted(input, key = lambda inp: inp.score, reverse=True)
# Maintenant ordonnées du plus grand au plus petit :-)
```

Générer des nombres aléatoires

```
import random
n = random.random() # Nombre entre 0.0 et 1.0
n = random.randrange(0, 15) # Nombre entre 0 et 14
```

Afficher un message avec un format particulier

```
name = "Anna"
howami = "depressed"
print(f"Hello {name} how are you today ? Me I'm {howami}")
```

1 - Fonction à optimiser

Choisir entre l'une des fonctions du site https://benchmarkfcns.info/fcns

Doit être:
- n-dimensional
- continuous

Noter la plage d'application que vous voulez définir (`min` et `max` des inputs acceptées)

Noter la valeur à trouver après optimisation (le **Global minima**), et ses coordonnées

2 - Implémenter la fonction

Créer une fonction Python qui implémente cette fonction mathématique

Vous pouvez utiliser le module `math` pour avoir accès à des fonctions de base

La fonction doit accepter une liste d'input à *longueur variable*

Python trick utile ce tp:

Example: `assert ma_fonction([23.3, 0.0]) == 12.5`

3 - Créer un traducteur d'input

On veut pouvoir passer un nombre compris *entre* `0` *et* `1`, et que cela soit *traduit* en un nombre compris entre le `MIN` et `MAX` de la fonction à optimiser.

Pour cela, créer une fonction (notée ici `fct`) devant faire en sorte que:
- `fct(0.0) = min`, min étant l'*input minimum* acceptée par la fonction mathématique
- `fct(1.0) = max`, max étant l'*input maximum* acceptée par la fonction mathématique
- `fct(0.5) = best`, best étant l'*input optimale* produisant la sortie la plus petite

4 - Créer une classe Cell

Cette classe contiendra toutes les attributs et méthodes pour réaliser un "essai":

- un attribut `genome` qui est une liste de nombres entre `0` et `1`, à appliquer à la fonction
- une méthode `apply` qui appelle la fonction à optimiser, avec le génome en input
- une méthode `child` qui créé un *clone* de cette `Cell`, mais en ayant *un gène muté*
- un attribut `output` qui *stocke le résultat* obtenu par la méthode `apply` précédemment
- une méthode `reset` qui réinitialise la classe

Mutation de gène:

- Tirer au hasard un chiffre `n` compris entre `0` et `len(genome) - 1` (compris)
- Remplacer `genome[n]` par un nombre aléatoire

5 - Créer l'algorithme d'optimisation

Le processus d'optimisation va appeler la méthode `apply` de toutes les `Cell`, et ne gardera que les `X%` meilleures (les scores les plus **BAS**)

Il ne restera plus, à chaque génération, de remplir la liste avec de nouvelles `Cell` produites à partir des meilleures, mais contenant une *mutation*.

Préparation:

- Initialiser `N` cellules, ayant tous des génomes aléatoires

À chaque génération:

- Pour chaque cellule, appeler la méthode `apply`
- Trier la liste des cellules de leur plus petit `output` jusqu'au plus grand
- Afficher un message avec le score minime obtenu, le meilleur génome, la génération

Avant de recommencer une génération:

- Effacer de la liste les `X%` de cellules ayant les scores les plus grands
- Compléter la liste, jusqu'à ce qu'on atteigne le nombre `N` de cellules:
  - Traverser la liste de cellules restantes (du plus petit au plus grand score)
  - Appeler la méthode `child`, ajouter la nouvelle cellule à la liste