```
!pip install deap
```

```
Collecting deap
  Downloading deap-1.4.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl.
metadata (13 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from
deap) (2.0.2)
Downloading deap-1.4.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(135 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/135.4 kB ? eta -:--:--
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 135.4/135.4 kB 9.7 MB/s eta 0:00:00
```

```
import random
from deap import base, creator, tools, algorithms

# Define the evaluation function (minimize a simple mathematical function)
def eval_func(individual):
    # Example evaluation function (minimize a quadratic function)
    return sum(x ** 2 for x in individual),

# DEAP setup
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))  # Minimize the fitness
value
creator.create("Individual", list, fitness=creator.FitnessMin)  # Individuals are
lists of floats

toolbox = base.Toolbox()

# Define attributes and individuals
toolbox.register("attr_float", random.uniform, -5.0, 5.0)  # Float values between -5
and 5
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_float, n=3)  # 3-dimensional individual
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Evaluation function and genetic operators
toolbox.register("evaluate", eval_func)
toolbox.register("mate", tools.cxBlend, alpha=0.5)  # Blend crossover
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)  # Gaussian
mutation
toolbox.register("select", tools.selTournament, tournsize=3)  # Tournament selection

# Create population
population = toolbox.population(n=50)

# Genetic Algorithm parameters
generations = 20

# Run the algorithm
for gen in range(generations):
    # Apply genetic operations (crossover and mutation)
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)

    # Evaluate the fitness of the offspring
    fits = list(map(toolbox.evaluate, offspring))  # Evaluate using the map function
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit  # Assign fitness values to individuals
```

```
    # Select the next generation
    population = toolbox.select(offspring, k=len(population))

# Get the best individual after generations
best_ind = tools.selBest(population, k=1)[0]
best_fitness = best_ind.fitness.values[0]

# Print the results
print("Best individual:", best_ind)
print("Best fitness:", best_fitness)
```

```
Best individual: [0.04589889109872622, 0.06070583418507594, 0.043944660740236376]
Best fitness: 0.007723039715773135
```