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

# Define the evaluation function
def eval_func(individual):
    target_sum = 15
    return (len(individual) - abs(sum(individual) - target_sum),)  # Note the comma to make it a tuple

# Create the toolbox
def create_toolbox(num_bits):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()
    toolbox.register("attr_bool", random.randint, 0, 1)
    toolbox.register("individual", tools.initRepeat, creator.Individual,
                     toolbox.attr_bool, num_bits)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    # Register the evaluation operator
    toolbox.register("evaluate", eval_func)

    # Register the crossover operator
    toolbox.register("mate", tools.cxTwoPoint)

    # Register the mutation operator
    toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)

    # Register the selection operator
    toolbox.register("select", tools.selTournament, tournsize=3)

    return toolbox

if __name__ == "__main__":
    num_bits = 45
    toolbox = create_toolbox(num_bits)
    random.seed(7)

    # Create an initial population
    population = toolbox.population(n=500)

    # Define probabilities for crossover and mutation
    probab_crossing, probab_mutating = 0.5, 0.2
    num_generations = 10

    print('\nEvolution process starts')

    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, population))
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit

    print('\nEvaluated', len(population), 'individuals')

    # Iterate through generations
    for g in range(num_generations):
        print("\n- Generation", g)

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population))
        offspring = list(map(toolbox.clone, offspring))

        # Apply crossover and mutation
        for child1, child2 in zip(offspring[::2], offspring[1::2]):
            if random.random() < probab_crossing:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values
```

```
        # Apply mutation
        for mutant in offspring:
            if random.random() < probab_mutating:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # Evaluate individuals with invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        print('Evaluated', len(invalid_ind), 'individuals')

        # Replace population with next generation
        population[:] = offspring

        # Print statistics for the current generation
        fits = [ind.fitness.values[0] for ind in population]
        length = len(population)
        mean = sum(fits) / length
        sum2 = sum(x*x for x in fits)
        std = abs(sum2 / length - mean**2)**0.5

        print('Min =', min(fits), ', Max =', max(fits))
        print('Average =', round(mean, 2), ', Standard deviation =', round(std, 2))

    print("\n- Evolution ends")

    # Print the final output
    best_ind = tools.selBest(population, 1)[0]
    print("\nBest individual:\n", best_ind)
    print("\nNumber of ones:", sum(best_ind))
```

```python
# Install DEAP if not already installed
# !pip install deap

import operator
import math
import random
import numpy as np
from deap import base, creator, gp, tools, algorithms

# Define a set of operators and terminals (Primitive Set)
pset = gp.PrimitiveSet("MAIN", 1)  # 1 input variable (X)

# Adding basic operators
pset.addPrimitive(operator.add, 2)    # +
pset.addPrimitive(operator.sub, 2)    # -
pset.addPrimitive(operator.mul, 2)    # *
pset.addPrimitive(operator.neg, 1)    # negation (unary -)

# Add ephemeral constant (random constants)
pset.addEphemeralConstant("rand101", lambda: random.randint(-1, 1))

# Rename the argument for clarity
pset.renameArguments(ARG0='x')

# Define Fitness and Individual
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))  # Minimize error
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

# Toolbox setup
toolbox = base.Toolbox()
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=3)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Define evaluation function
def eval_symb_reg(individual):
    func = toolbox.compile(expr=individual)

    # Training points
    X = np.linspace(-10, 10, 50)
    Y_true = 5*X**3 - 6*X**2 + 8*X - 1  # Slightly rearranged to equal zero

    # Compute predicted Y
    Y_pred = np.array([func(x) for x in X])

    # Compute RMSE
    rmse = np.sqrt(np.mean((Y_true - Y_pred)**2))
    return (rmse,)

toolbox.register("compile", gp.compile, pset=pset)
toolbox.register("evaluate", eval_symb_reg)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)

# Decorate (limit height of trees to avoid bloat)
toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"), max_value=17))
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"), max_value=17))

# Genetic Algorithm parameters
def main():
    random.seed(42)

    pop = toolbox.population(n=300)
    hof = tools.HallOfFame(1)  # Best individual
```

```python
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)

    print("Starting Evolution...")

    pop, log = algorithms.eaSimple(pop, toolbox,
                                   cxpb=0.5, mutpb=0.2,
                                   ngen=40,
                                   stats=stats, halloffame=hof,
                                   verbose=True)

    print("\nBest individual:")
    print(hof[0])
    print("\nFitness (RMSE):", hof[0].fitness.values[0])

if __name__ == "__main__":
    main()
```