```
import random
import copy
```

En este bloque, se importan los módulos random y copy. random se usa para generar números aleatorios, mientras que copy se utiliza para crear copias profundas de listas y objetos.

```
# Definir el conjunto de terminales y funciones
terminales = ['N', 'S', 'E', 'W']
funciones = ['R']  # Giro a la derecha (puede expandirse con más funciones)
```

Se definen dos listas, terminales y funciones. terminales contiene direcciones cardinales ('N', 'S', 'E', 'W'), y funciones contiene una función 'R' que representa el giro a la derecha.

```
# Funciones auxiliares para el movimiento y giro del robot
def move_forward(current_position, direction):
    # Moverse una unidad en la dirección especificada
    x, y = current_position
    if direction == 'N':
        return (x, y + 1)
    elif direction == 'S':
        return (x, y - 1)
    elif direction == 'E':
        return (x + 1, y)
    elif direction == 'W':
        return (x - 1, y)

def turn_right(current_direction):
    # Girar a la derecha (90 grados)
    if current_direction == 'N':
        return 'E'
    elif current_direction == 'S':
        return 'W'
    elif current_direction == 'E':
        return 'S'
    elif current_direction == 'W':
        return 'N'
```

La función move_forward recibe la posición actual del robot y una dirección ('N', 'S', 'E', o 'W') y devuelve la nueva posición después de moverse una unidad en esa dirección. La función turn_right recibe la dirección actual del robot ('N', 'S', 'E' o 'W') y devuelve la dirección después de girar 90 grados a la derecha.

```
# Función para crear programas genéticos aleatorios
def create_program():
    # Generar un programa genético aleatorio de longitud aleatoria
    program_length = random.randint(10, 50)  # Longitud del programa (ajustar segú
    return [random.choice(terminales + funciones) for _ in range(program_length)]
```

La función create_program genera un programa genético aleatorio de longitud aleatoria, que consiste en una secuencia de elementos tomados de las listas terminales y funciones.

```
# Definir la función de aptitud
def fitness(program):
    max_steps = 100  # Límite máximo de pasos para evitar bucles infinitos
    robot_position = (0, 0)  # Iniciar en la posición (0, 0) de la sala
    robot_direction = 'N'  # Iniciar mirando al norte
    engineer_positions = [(2, 2), (-1, 1), (3, -1), (-2, -2)]  # Posiciones de los
    engineer_scores = [0] * len(engineer_positions)  # Inicializar los puntajes de

    if not program:  # Si el programa está vacío, asignar puntuación mínima
        return 1

    for step in range(max_steps):
        if robot_position in engineer_positions:
            # El robot entrega una galleta al ingeniero en su posición
            engineer_scores[engineer_positions.index(robot_position)] += 1

        if not program:  # Si el programa está vacío, terminamos
            break

        # Obtener la próxima instrucción del programa
        next_instruction = program.pop(0)

        if next_instruction in terminales:
            # Si la instrucción es una terminal (dirección), movemos el robot en es
            robot_position = move_forward(robot_position, next_instruction)
        elif next_instruction == 'R':
            # Si la instrucción es 'R', giramos a la derecha
            robot_direction = turn_right(robot_direction)

    total_score = sum(engineer_scores)  # Puntaje total obtenido por entregar galle
    return max(total_score, 1)  # Devolver al menos 1 punto
```

La función fitness evalúa la aptitud de un programa genético. Simula el movimiento de un robot en un entorno representado por una cuadrícula y evalúa su desempeño al entregar "galletas" a los ingenieros que se encuentran en ciertas posiciones. La función devuelve un puntaje basado en el número de galletas entregadas.

```
# Crear la población inicial
population_size = 100
population = [create_program() for _ in range(population_size)]

# Evolución de la población
generations = 50
mutation_rate = 0.1

for generation in range(generations):
    # Evaluar la aptitud de cada programa en la población
    fitness_scores = [fitness(program) for program in population]

    # Seleccionar a los programas más aptos para reproducirse
    selected_indices = random.choices(range(population_size), k=population_size //
    selected_population = [population[i] for i in selected_indices]

    # Crear la nueva generación mediante cruce y mutación
    new_population = []
    for _ in range(population_size // 2):
        parent1, parent2 = random.choices(selected_population, k=2)
        min_length = min(len(parent1), len(parent2))

        if min_length >= 2:  # Asegurarse de que haya al menos 2 elementos para el
            crossover_point = random.randint(1, min_length - 1)
            child1 = parent1[:crossover_point] + parent2[crossover_point:]
            child2 = parent2[:crossover_point] + parent1[crossover_point:]
        else:
            # Si alguno de los padres tiene una longitud menor a 2, no se realiza e
            child1 = parent1
            child2 = parent2

        # Aplicar mutación a child1
        mutated_child1 = copy.deepcopy(child1)
        for i in range(len(mutated_child1)):
            if random.random() < mutation_rate:
                mutated_child1[i] = random.choice(terminales + funciones)

        # Aplicar mutación a child2
        mutated_child2 = copy.deepcopy(child2)
        for i in range(len(mutated_child2)):
            if random.random() < mutation_rate:
                mutated_child2[i] = random.choice(terminales + funciones)

        new_population.extend([mutated_child1, mutated_child2])

    population = new_population
```

Se crea una población inicial de programas genéticos aleatorios. El tamaño de la población se define como 100 y se utiliza la función create_program para generar los programas. En este bloque, se realiza la evolución de la población a lo largo de 50 generaciones. Se evalúa la aptitud de cada programa en la población y se seleccionan los programas más aptos para la reproducción. Luego se crea una nueva generación de programas mediante cruces y mutaciones.

```
# Encontrar el programa óptimo
best_program = max(population, key=fitness)
best_fitness = fitness(best_program)

print("Mejor programa genético:", best_program)
print("Puntaje del mejor programa:", best_fitness)
```

Finalmente, se encuentra el programa genético óptimo dentro de la población y se imprime en la pantalla junto con su puntaje de aptitud.

En resumen, este código simula la evolución de programas genéticos que controlan un robot en un entorno para maximizar la entrega de galletas a ingenieros en posiciones específicas. Los programas evolucionan a lo largo de generaciones mediante cruces y mutaciones, y se busca encontrar el programa con la mejor aptitud.