Server.py

```
from xmlrpc.server import SimpleXMLRPCServer
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

server = SimpleXMLRPCServer(('localhost', 8000))
server.register_function(factorial, 'calculate_factorial')
print("Server is ready to accept RPC calls...")
server.serve_forever()
```

Output

```
PS C:\Users\shrey\OneDrive\Desktop\Code\Ass1> & C:/Users/shrey/anaconda3/python.exe c:/Users/shrey/OneDrive/Desktop/Code/Ass1/Server.py
Server is ready to accept RPC calls...
127.0.0.1 - - [01/Apr/2024 18:32:30] "POST /RPC2 HTTP/1.1" 200 -
```

Client.py

```
import xmlrpc.client
server = xmlrpc.client.ServerProxy('http://localhost:8000')
n = int(input("Enter the number to calculate factorial: "))
result = server.calculate_factorial(n)
print(f"The factorial of {n} is: {result}")
```

Output

```
PS C:\Users\shrey\OneDrive\Desktop\Code\Ass1> & C:/Users/shrey/anaconda3/python.exe c:/Users/shrey/OneDrive/Desktop/Code/Ass1/Client.py
Enter the number to calculate factorial: 5
The factorial of 5 is: 120
PS C:\Users\shrey\OneDrive\Desktop\Code\Ass1>
```

Server.py

```
import Pyro4
@Pyro4.expose
class StringConcatenator:
def concatenate(self, str1, str2):
        return str1 + str2
daemon = Pyro4.Daemon()
uri = daemon.register(StringConcatenator)
print("Server URI:", uri)
daemon.requestLoop()
```

OUTPUT

```
Server URI: PYRO:obj_3b026bba00294d4ebc03dee3d74a29b9@localhost:50011
```

Client.py

```
import Pyro4
uri = input("Enter the URI of the server: ")
concatenator = Pyro4.Proxy(uri)
str1 = input("Enter the first string: ")
str2 = input("Enter the second string: ")
result = concatenator.concatenate(str1, str2)
print("Concatenated string:", result)
```

OUTPUT

```
Enter the URI of the server: PYRO:obj_3b026bba00294d4ebc03dee3d74a29b9@localhost:50011
Enter the first string: SSD
Enter the second string: DDS
Concatenated string: SSDDDS
```

Text_File.txt

Hello Hello how are you!

Char_Count_Mr.py

```
from mrjob.job import MRJob
class MRCharCount(MRJob):
    def mapper(self, _, line):
        for char in line.strip():
            yield char, 1


    def reducer(self, char, counts):
        yield char, sum(counts)


if _name__== '_main':
    MRCharCount.run()
```

OUTPUT

```
" "     4
"!"     1
"H"     2
"a"     1
"e"     3
"h"     1
"l"     4
"o"     4
"r"     1
"u"     1
"w"     1
"y"     1
```

Word_Count_Mr.py

```
from mrjob.job import MRJob

import re

WORD_REGEXP = re.compile(r"[\w']+")

class MRWordCount(MRJob):

    def mapper(self, _, line):

        for word in WORD_REGEXP.findall(line):

            yield word.lower(), 1


    def reducer(self, word, counts):

        yield word, sum(counts)


if __name__== '__main__':

    MRWordCount.run()
```

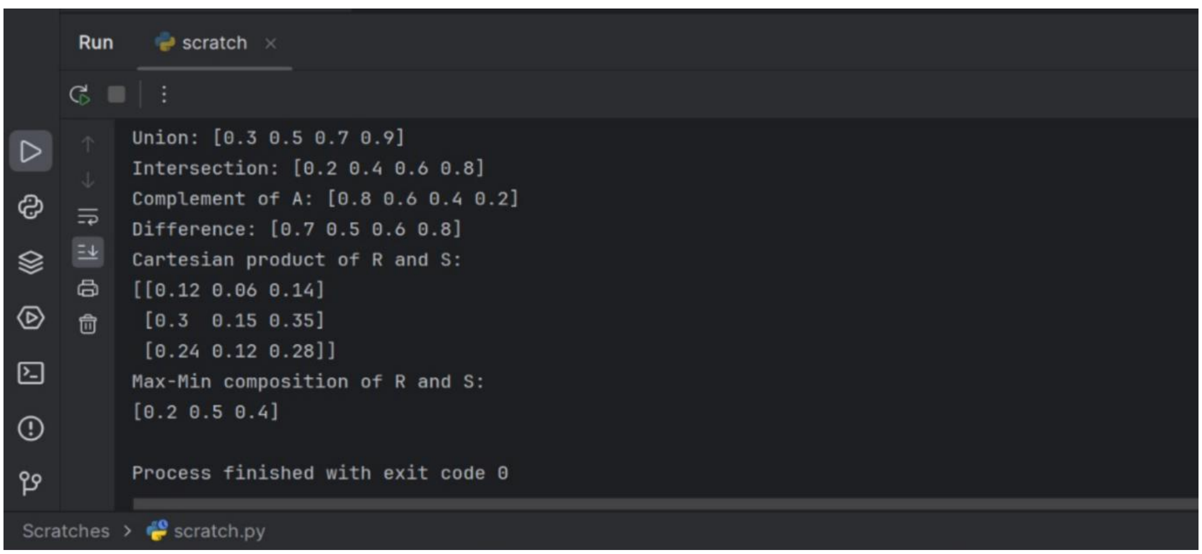
OUTPUT

```
"are"   1
"hello" 2
"how"   1
"you"   1
```

```
import numpy as np

# Function to perform Union operation on fuzzy sets
def fuzzy_union(A, B):
    return np.maximum(A, B)

# Function to perform Intersection operation on fuzzy sets
def fuzzy_intersection(A, B):
    return np.minimum(A, B)

# Function to perform Complement operation on a fuzzy set
def fuzzy_complement(A):
    return 1 - A

# Function to perform Difference operation on fuzzy sets
def fuzzy_difference(A, B):
    return np.maximum(A, 1 - B)

# Function to create fuzzy relation by Cartesian product of two fuzzy sets
def cartesian_product(A, B):
    return np.outer(A, B)

# Function to perform Max-Min composition on two fuzzy relations
def max_min_composition(R, S):
    return np.max(np.minimum.outer(R, S), axis=1)

# Example usage
A = np.array([0.2, 0.4, 0.6, 0.8]) # Fuzzy set A
B = np.array([0.3, 0.5, 0.7, 0.9]) # Fuzzy set B

# Operations on fuzzy sets
union_result = fuzzy_union(A, B)
intersection_result = fuzzy_intersection(A, B)
complement_A = fuzzy_complement(A)
difference_result = fuzzy_difference(A, B)

print("Union:", union_result)
print("Intersection:", intersection_result)
print("Complement of A:", complement_A)
print("Difference:", difference_result)
```

```
# Fuzzy relations
R = np.array([0.2, 0.5, 0.4]) # Fuzzy relation R
S = np.array([0.6, 0.3, 0.7]) # Fuzzy relation S

# Cartesian product of fuzzy relations
cartesian_result = cartesian_product(R, S)

# Max-Min composition of fuzzy relations
composition_result = max_min_composition(R, S)

print("Cartesian product of R and S:")
print(cartesian_result)

print("Max-Min composition of R and S:")
print(composition_result)
```

OUTPUT:



```
Union: [0.3 0.5 0.7 0.9]
Intersection: [0.2 0.4 0.6 0.8]
Complement of A: [0.8 0.6 0.4 0.2]
Difference: [0.7 0.5 0.6 0.8]
Cartesian product of R and S:
[[0.12 0.06 0.14]
 [0.3  0.15 0.35]
 [0.24 0.12 0.28]]
Max-Min composition of R and S:
[0.2 0.5 0.4]

Process finished with exit code 0
```

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

# Define evaluation function (this is a mock function, replace this with your actual evaluation
# function)
def evaluate(individual):
   # Here 'individual' represents the parameters for the neural network
   # You'll need to replace this with your actual evaluation function that trains the neural
network
   # and evaluates its performance
   # Return a fitness value (here, a random number is used as an example)
   return random.random(),

# Define genetic algorithm parameters
POPULATION_SIZE = 10
GENERATIONS = 5

# Create types for fitness and individuals in the genetic algorithm
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

# Initialize toolbox
toolbox = base.Toolbox()

# Define attributes and individuals
toolbox.register("attr_neurons", random.randint, 1, 100)  # Example: number of neurons
toolbox.register("attr_layers", random.randint, 1, 5)  # Example: number of layers
toolbox.register("individual", tools.initCycle, creator.Individual, (toolbox.attr_neurons,
                                                  toolbox.attr_layers), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Genetic operators
toolbox.register("evaluate", evaluate)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=1, up=100, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Create initial population
population = toolbox.population(n=POPULATION_SIZE)

# Run the genetic algorithm
```

```
for gen in range(GENERATIONS):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)

    fitnesses = toolbox.map(toolbox.evaluate, offspring)
    for ind, fit in zip(offspring, fitnesses):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

# Get the best individual from the final population
best_individual = tools.selBest(population, k=1)[0]
best_params = best_individual

# Print the best parameters found
print("Best Parameters:", best_params)
```

OUTPUT:

```
Best Parameters: [54, 5]
```

```
import random
# Define the objective function
def objective_function(x):
    return -x**2 + 4

# Generate initial population
def generate_initial_population(pop_size=10):
    return [random.uniform(-10, 10) for _ in range(pop_size)]

# Calculate fitness of each antibody
def calculate_fitness(population):
    return [objective_function(x) for x in population]

# Clone and mutate
def clone_and_mutate(antibody, clone_factor=1):
    # Simple mutation: slight random change
    return antibody + random.uniform(-clone_factor, clone_factor)

# The Clonal Selection Algorithm
def clonal_selection(iterations=100, pop_size=10):
    population = generate_initial_population(pop_size)
    for _ in range(iterations):
        # Calculate fitness
        fitness = calculate_fitness(population)

        # Select the best half of the population
        sorted_pop = [x for _, x in sorted(zip(fitness, population), reverse=True)]
```

```
        selected = sorted_pop[:len(sorted_pop)//2]

        # Clone and mutate the selected antibodies
        clones = [clone_and_mutate(x) for x in selected for _ in range(2)]  # Each selected antibody generates 2 clones

        # Form new population with clones and calculate new fitness
        population = clones
        fitness = calculate_fitness(population)

        # Keep the best solution
        best_index = fitness.index(max(fitness))

    return population[best_index]

# Run the algorithm
best_solution = clonal_selection()
print(f"Best solution: {best_solution}")
print(f"Maximum value: {objective_function(best_solution)}")
```

OUTPUT:

```
Best solution: 0.14758564550161957
Maximum value: 3.9782184772418705
```