**Artificial Intelligence Lab**

**LAB 7 – Unification and Resolution**

**Aditya Kumar**

**RA1911030010100**

**O2 Section**

**Date – 15/3/22**

**Problem Statement :**

Implementation of resolution

**Algorithm :**

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form.**

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

**Code :**

import copy

import time

class Parameter:

variable\_count = 1

def \_\_init\_\_(self, name=None):

if name:

self.type = "Constant"

self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type\_, name):

self.type = type\_

self.name = name

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_str\_\_(self):

return self.name

class Predicate:

def \_\_init\_\_(self, name, params):

self.name = name

self.params = params

def \_\_eq\_\_(self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def \_\_str\_\_(self):

return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence:

sentence\_count = 0

def \_\_init\_\_(self, string):

self.sentence\_index = Sentence.sentence\_count

Sentence.sentence\_count += 1

self.predicates = []

self.variable\_map = {}

local = {}

for predicate in string.split("|"):

name = predicate[:predicate.find("(")]

params = []

for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):

if param[0].islower():

if param not in local: # Variable

local[param] = Parameter()

self.variable\_map[local[param].name] = local[param]

new\_param = local[param]

else:

new\_param = Parameter(param)

self.variable\_map[param] = new\_param

params.append(new\_param)

self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):

self.predicates.remove(predicate)

for key, val in self.variable\_map.items():

if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def \_\_eq\_\_(self, other):

if len(self.predicates) == 1 and self.predicates[0] == other:

return True

return False

def \_\_str\_\_(self):

return "".join([str(predicate) for predicate in self.predicates])

class KB:

def \_\_init\_\_(self, inputSentences):

self.inputSentences = [x.replace(" ", "") for x in inputSentences]

self.sentences = []

self.sentence\_map = {}

def prepareKB(self):

self.convertSentencesToCNF()

for sentence\_string in self.inputSentences:

sentence = Sentence(sentence\_string)

for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get(

predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal

if "=>" in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent(

self.inputSentences[sentenceIdx])

def askQueries(self, queryList):

results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))

negatedPredicate = negatedQuery.predicates[0]

prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get(

negatedPredicate.name, []) + [negatedQuery]

self.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [

False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map

if result:

results.append("TRUE")

else:

results.append("FALSE")

return results

def resolve(self, queryStack, visited, depth=0):

if time.time() > self.timeLimit:

raise Exception

if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate()

queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map:

return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]:

if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName):

canUnify, substitution = performUnification(

copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence)

newSentence.removePredicate(kbPredicate)

newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old]

newSentence.variable\_map.pop(old)

parameter.unify(

"Variable" if new[0].islower() else "Constant", new)

newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params):

if param.name in substitution:

new = substitution[param.name]

predicate.params[index].unify(

"Variable" if new[0].islower() else "Constant", new)

for predicate in newSentence.predicates:

newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited)

if kb\_sentence.containsVariable() and len(kb\_sentence.predicates) > 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1):

return True

return False

return True

def performUnification(queryPredicate, kbPredicate):

substitution = {}

if queryPredicate == kbPredicate:

return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params):

if query == kb:

continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution:

substitution[query.name] = kb.name

elif substitution[query.name] != kb.name:

return False, {}

query.unify("Constant", kb.name)

else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

kb.unify("Variable", query.name)

else:

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find("=>")]

premise = []

for predicate in antecedent.split("&"):

premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find("=>") + 2:])

return "|".join(premise)

def getInput(filename):

with open(filename, "r") as file:

noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)]

noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip()

for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results):

print(results)

with open(filename, "w") as file:

for line in results:

file.write(line)

file.write("\n")

file.close()

if \_\_name\_\_ == '\_\_main\_\_':

inputQueries\_, inputSentences\_ = getInput(‘/home/ubuntu/environment/RA1911030010100/input.txt')

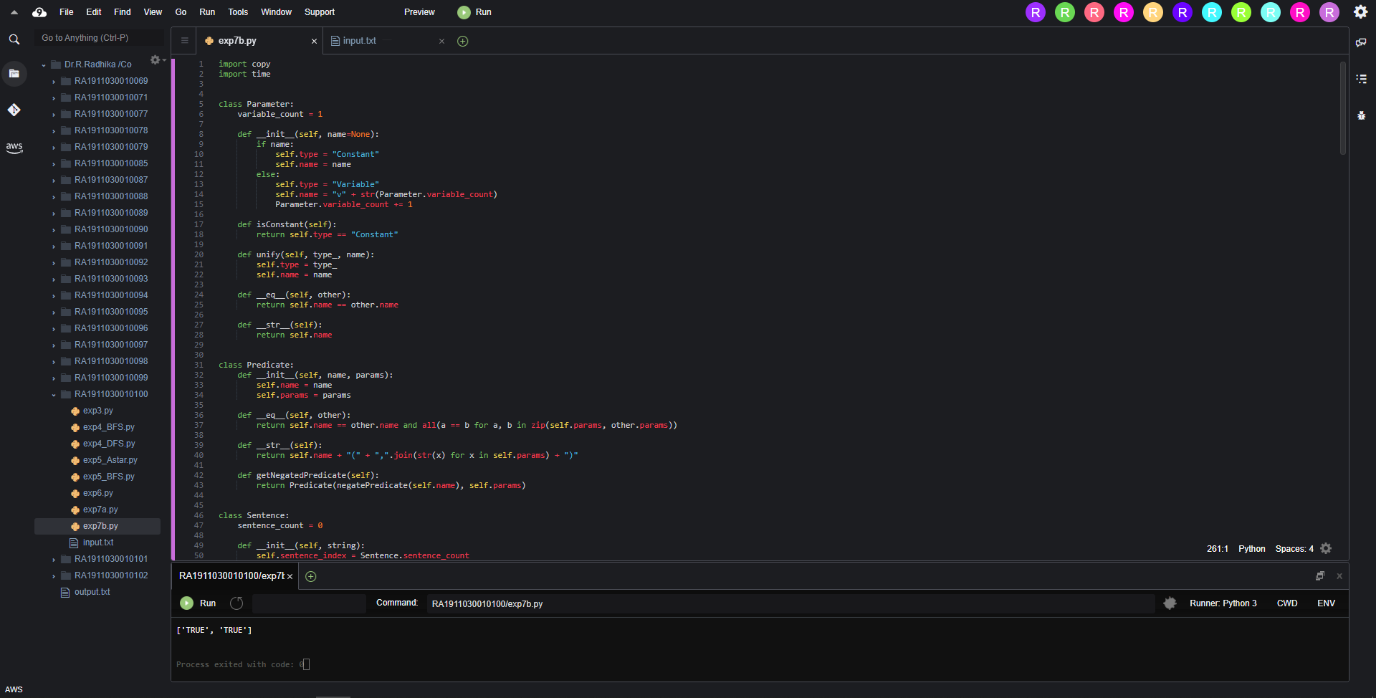
knowledgeBase = KB(inputSentences\_)

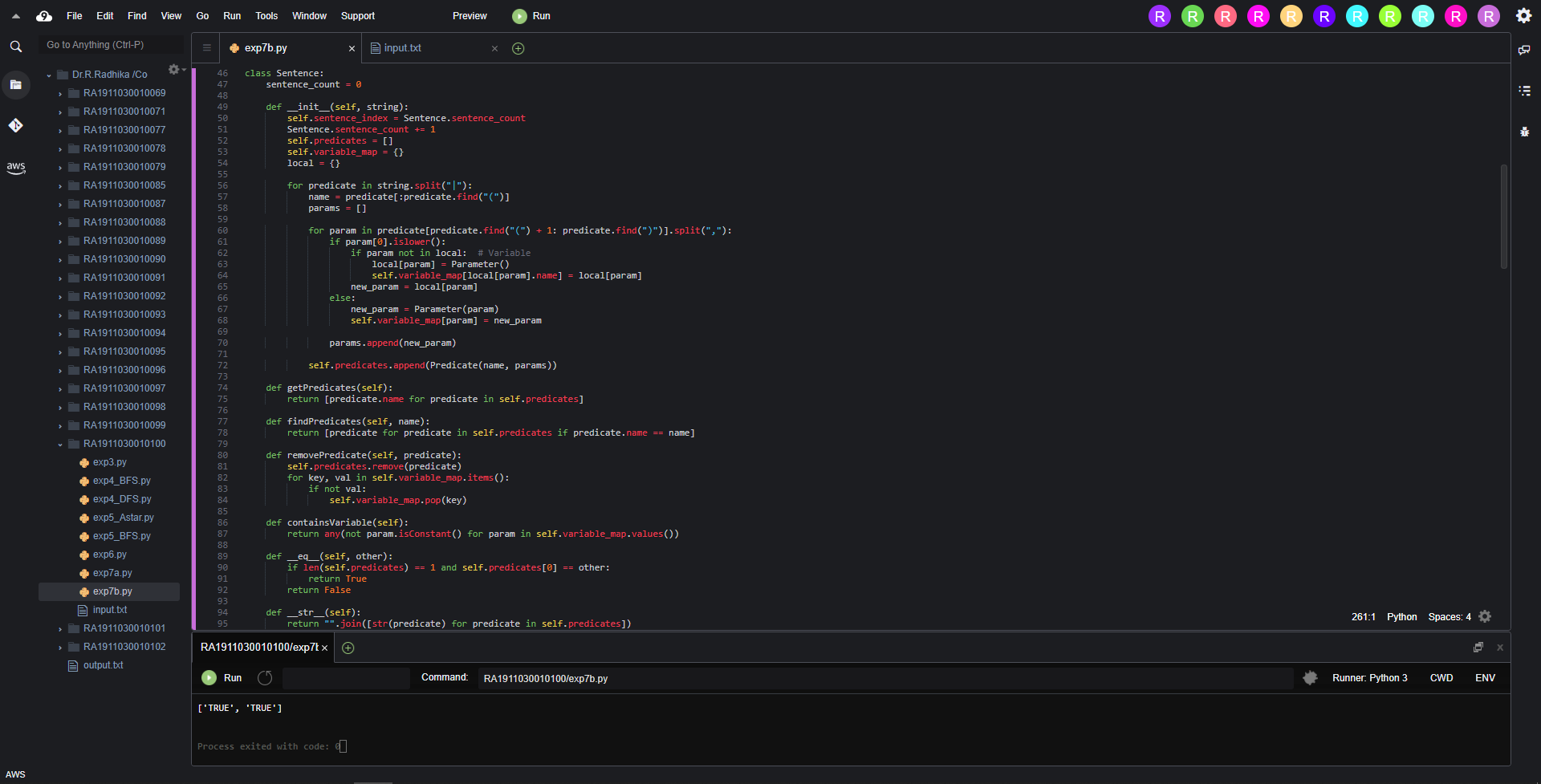
knowledgeBase.prepareKB()

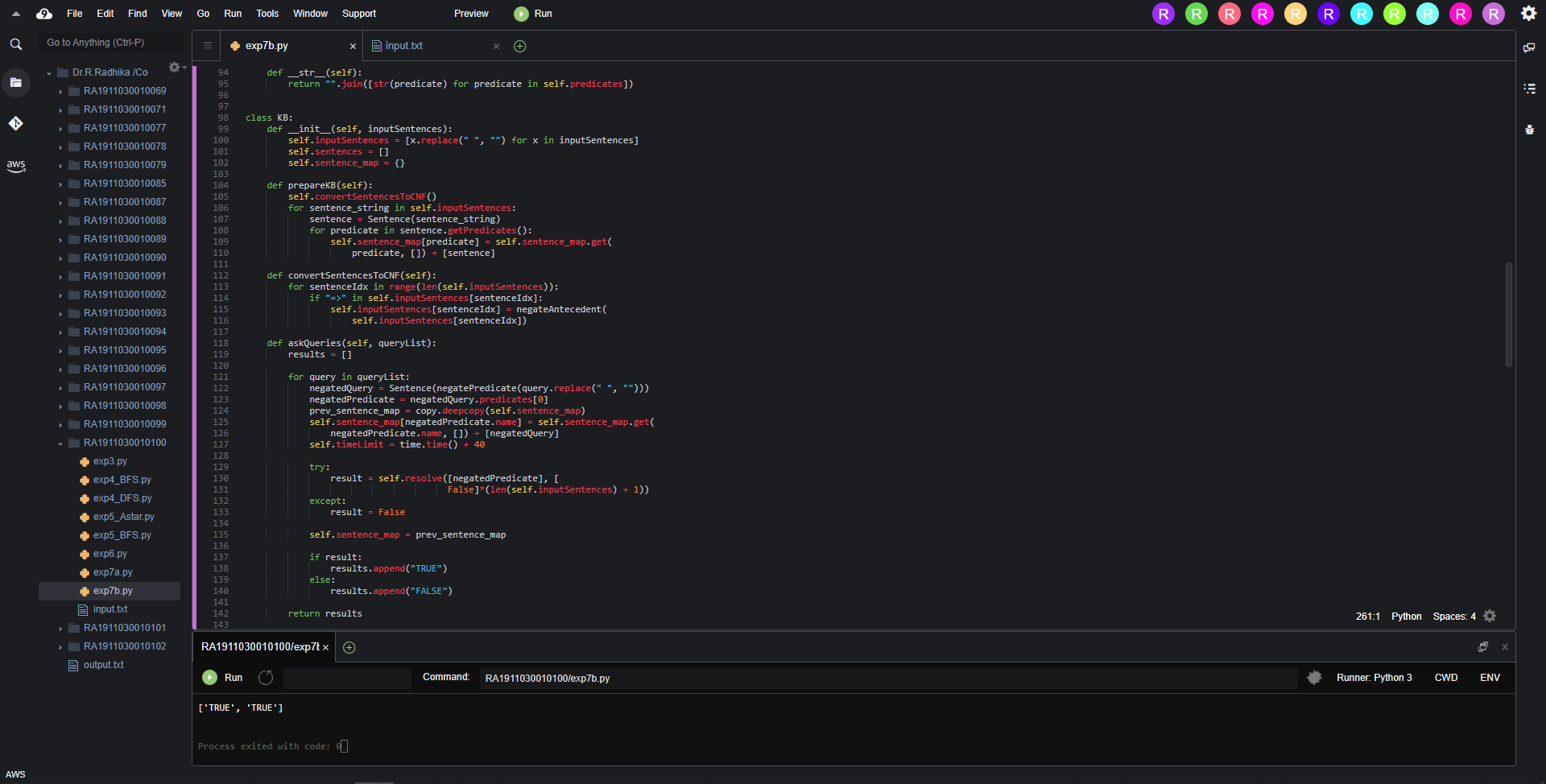
results\_ = knowledgeBase.askQueries(inputQueries\_)

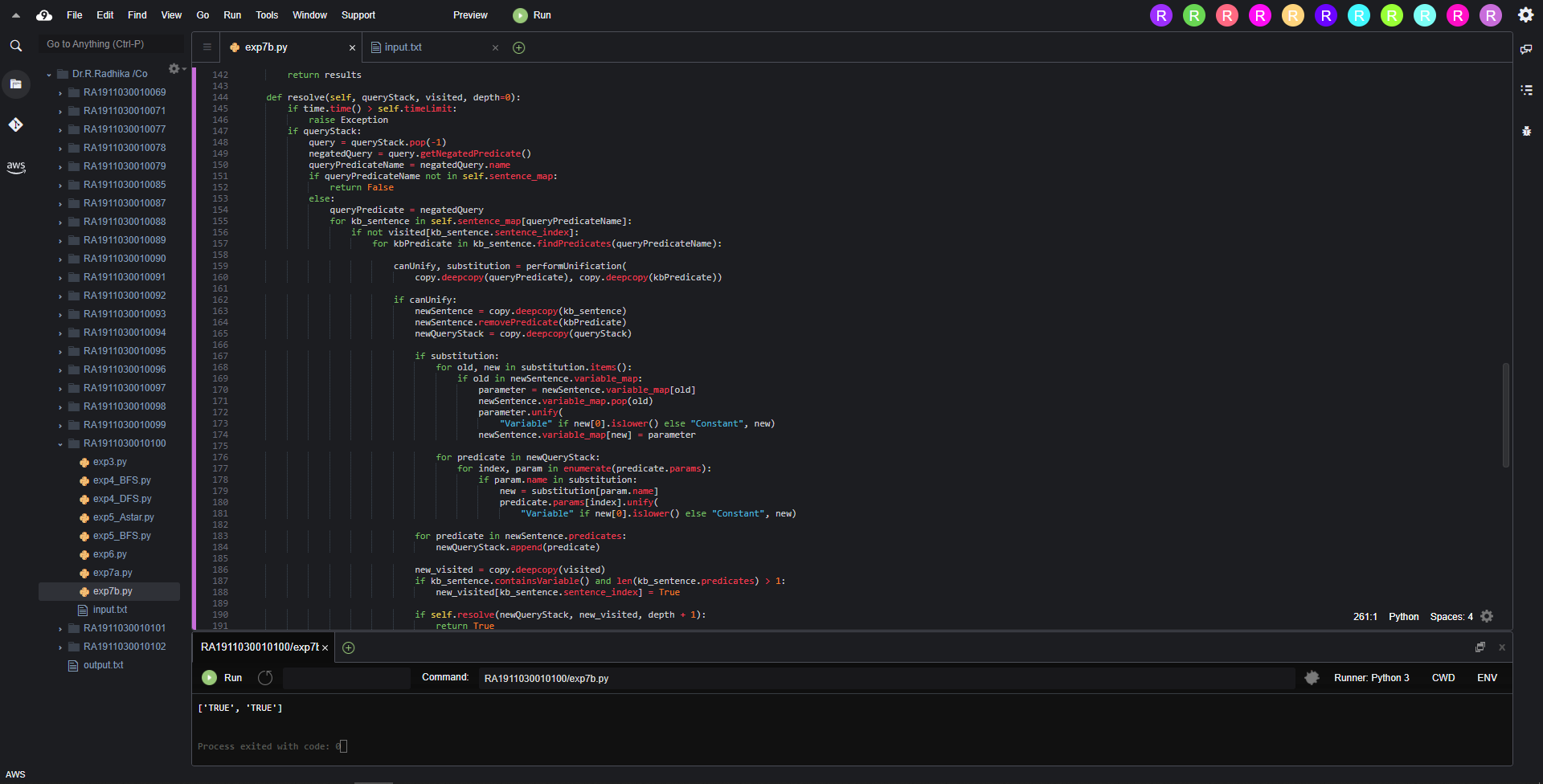
printOutput("output.txt", results\_)

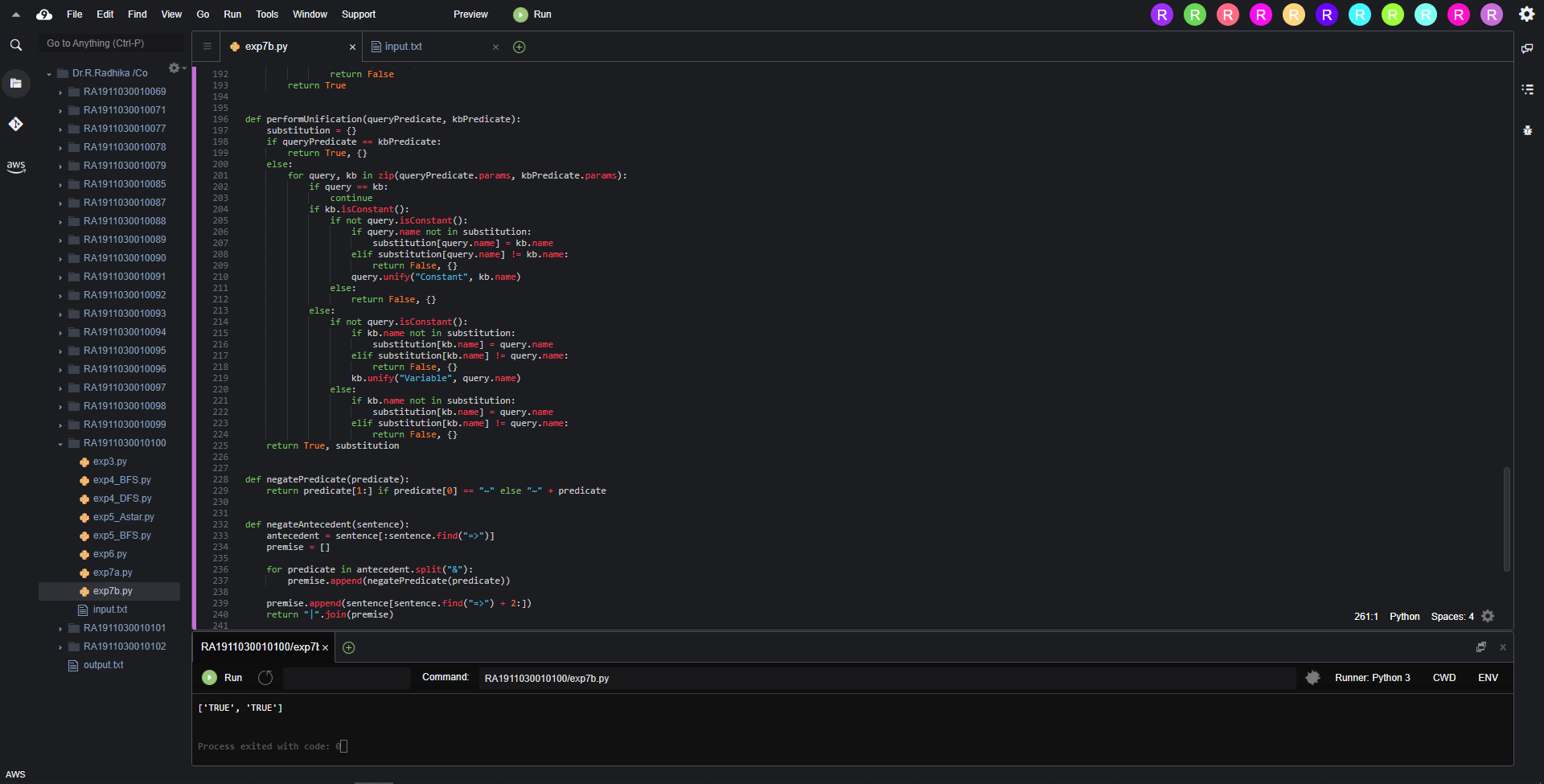
**Output :**

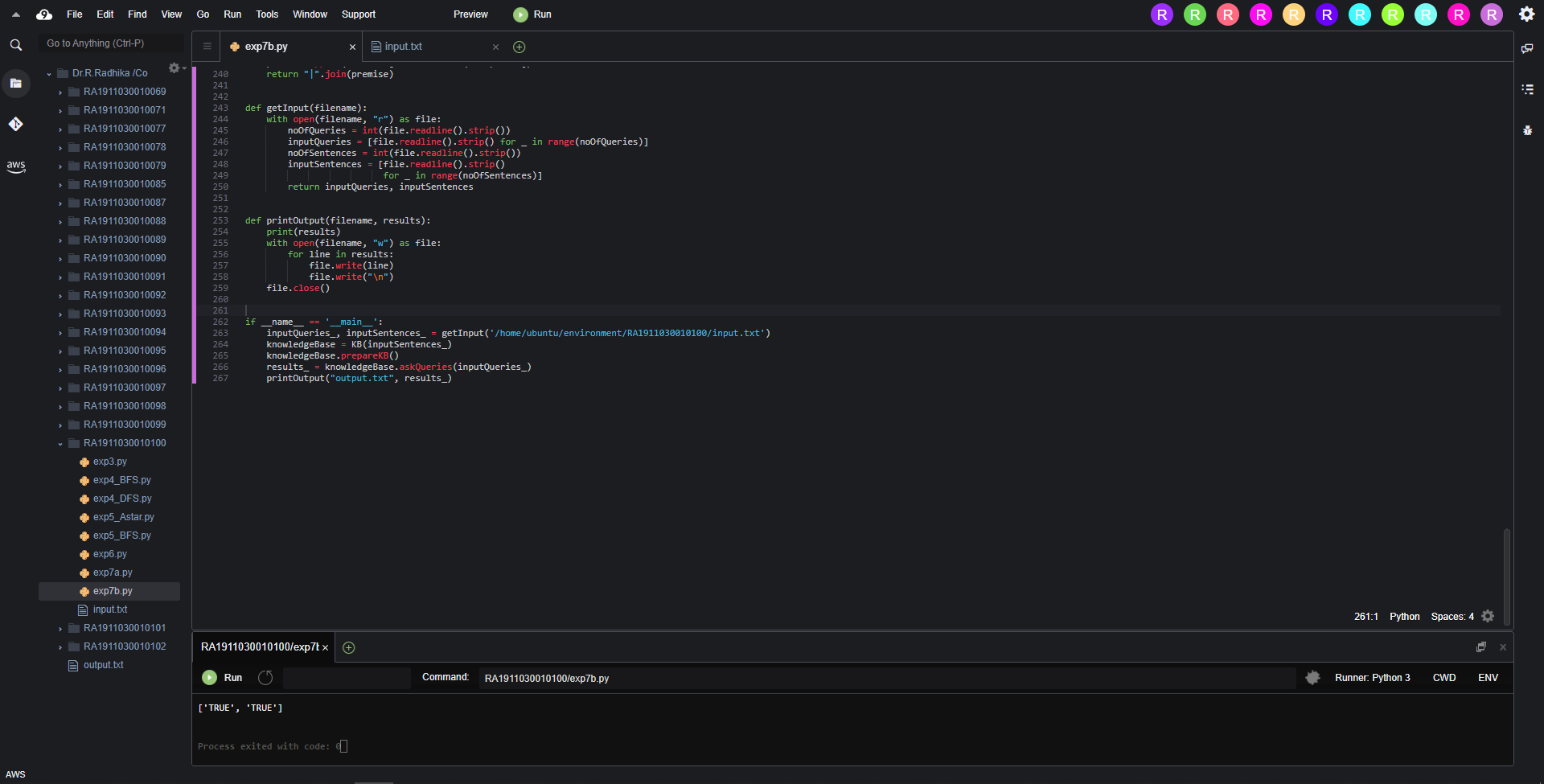


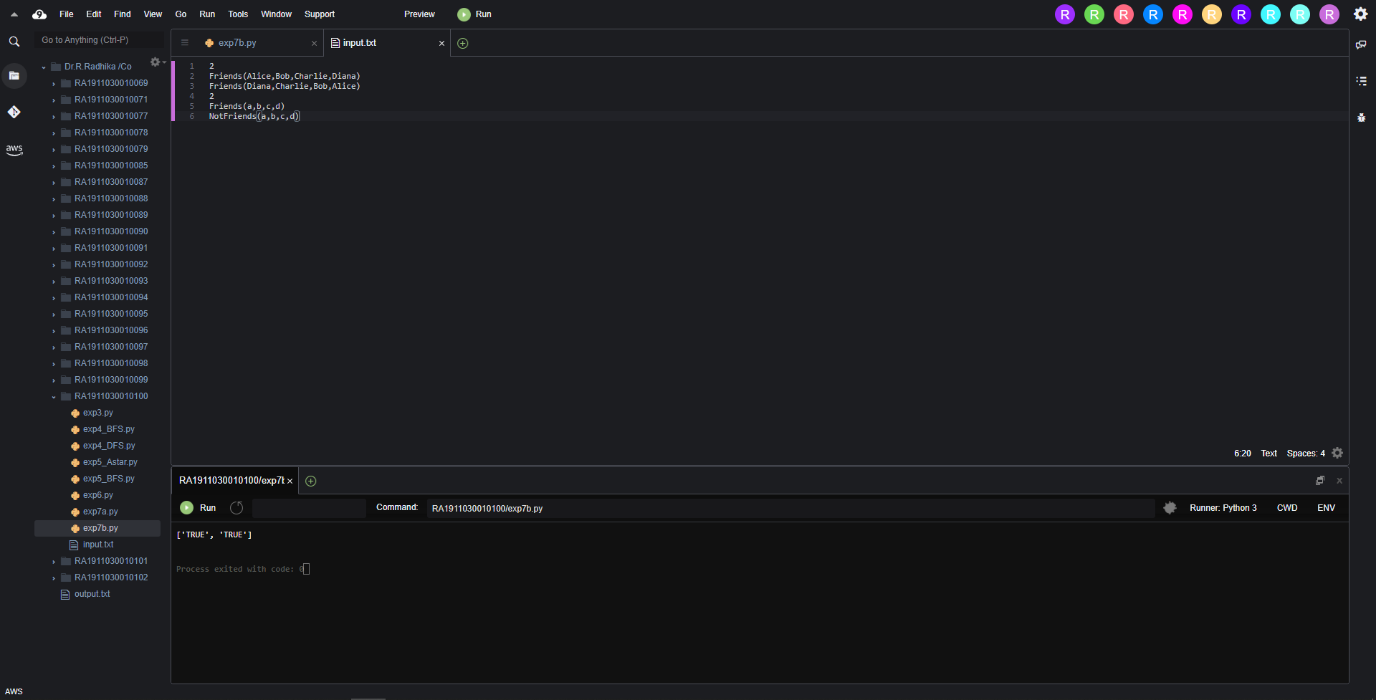












**Result :**

Hence we performed Resolution.