Propositional Logic Operations in Python

1. AND Operation (Logical Conjunction) The AND operation returns True if both operands are True, otherwise it returns False. python def and_operation(p, q): return p and q

2. OR Operation (Logical Disjunction) The OR operation returns True if at least one of the operands is True. It only returns False if both operands are False. python def or_operation(p, q): return p or q

3. NOT Operation (Logical Negation) The NOT operation inverts the truth value of its operand. If the operand is True, it returns False, and vice versa. python def not_operation(p): return not p

4. IMPLIES Operation (Logical Implication) The IMPLIES operation returns False only if the first operand is True and the second operand is False. In all other cases, it returns True. python def implies_operation(p, q): return not p or q

Example Usage Here's how you can use these functions: python

Example values

p = True q = False

Performing operations

print("AND Operation (p ∧ q):", and_operation(p, q)) # Output: False print("OR Operation (p ∨ q):", or_operation(p, q)) # Output: True print("NOT Operation (¬p):", not_operation(p)) # Output: False print("IMPLIES Operation (p → q):", implies_operation(p, q)) # Output: False

Summary These functions provide a straightforward implementation of basic propositional logic operations in Python. You can easily extend or modify them for more complex logical expressions as needed. Enjoy experimenting with logical operations!

2. Evaluate Logical Statements: o Create an Evaluation Function: Develop a function that takes logical statements as input and evaluates their truth value. o Example: A function evaluate(statement, values) where statement is a logical expression and values is a dictionary mapping propositions to their truth values. Answer To evaluate logical statements based on given truth values, we can create a function that parses and evaluates the logical expression. Below is an implementation of the evaluate function that takes a logical statement and a dictionary of truth values for the propositions. Logical Statement Evaluation in Python

3. Evaluation Function The evaluate function will use Python's built-in eval function to compute the truth value of the logical expression. To ensure safety and proper evaluation, we will define our logical operations using their corresponding symbols. Implementation python def evaluate(statement, values):

Define logical operations

Replace logical symbols with Python equivalents

statement = statement.replace('∧', ' and ') statement = statement.replace('∨', ' or ') statement = statement.replace('¬', ' not ') statement = statement.replace('→', ' <= ') # p → q is equivalent to not p or q

Create a local scope for eval

local_scope = {key: values[key] for key in values}

Evaluate the statement

try: return eval(statement, {"builtins": None}, local_scope) except Exception as e: raise ValueError(f"Error in evaluating the statement: {e}")

Example Usage Here's how you can use the evaluate function: python

Define truth values for propositions

truth_values = { 'p': True, 'q': False, 'r': True }

Logical statements

statement1 = 'p ∧ q' # Should evaluate to False statement2 = 'p ∨ r' # Should evaluate to True statement3 = '¬p → q' # Should evaluate to False statement4 = 'p → (q ∨ r)' # Should evaluate to True

Evaluating statements

print("Evaluating statement 1 (p ∧ q):", evaluate(statement1, truth_values)) # Output: False print("Evaluating statement 2 (p ∨ r):", evaluate(statement2, truth_values)) # Output: True print("Evaluating statement 3 (¬p → q):", evaluate(statement3, truth_values)) # Output: False print("Evaluating statement 4 (p → (q ∨ r)):", evaluate(statement4, truth_values)) # Output: True

Summary The evaluate function allows you to input logical statements using logical symbols and a dictionary of truth values. It then evaluates the expression and returns the corresponding truth value. This implementation provides a flexible way to handle logical expressions in Python. Feel free to test and expand upon it for more complex logical evaluations!

To extend the logical operations to support predicate logic with quantifiers, we can create functions for the universal quantifier (∀) and existential quantifier (∃). These functions will take a predicate function and a domain (set of elements) as input, and return the result of applying the respective quantifier. Predicate Logic Operations in Python

1. Universal Quantifier (∀) The universal quantifier returns True if the given predicate is True for all elements in the domain, and False otherwise. python def forall(predicate, domain): return all(predicate(x) for x in domain)

2. Existential Quantifier (∃) The existential quantifier returns True if there exists at least one element in the domain for which the predicate is True, and False otherwise. python def exists(predicate, domain): return any(predicate(x) for x in domain)

Example Usage Let's consider a simple example to illustrate the usage of these functions. Suppose we have a domain of numbers D = {1, 2, 3, 4, 5} and a predicate is_even(x) that checks if a number is even. python D = [1, 2, 3, 4, 5]

def is_even(x): return x % 2 == 0

We can use the forall and exists functions to evaluate universal and existential quantifications over this domain and predicate. python

Universal quantification: ∀x ∈ D, is_even(x)

print("∀x ∈ D, is_even(x):", forall(is_even, D)) # Output: False

Existential quantification: ∃x ∈ D, is_even(x)

print("∃x ∈ D, is_even(x):", exists(is_even, D)) # Output: True

The output shows that it is not true for all numbers in the domain D that they are even (∀x ∈ D, is_even(x) is False), but there exists at least one number in D that is even (∃x ∈ D, is_even(x) is True). Summary The forall and exists functions provide a way to express and evaluate universal and existential quantifications in predicate logic. By combining these functions with predicate definitions, you can handle a wide range of logical statements involving quantifiers. Feel free to explore and expand upon these functions as needed for your specific use cases.

To create a simple AI agent that uses logic to make decisions, let's consider a scenario where the agent needs to determine the best action based on environmental conditions. We'll define the scenario, implement the decision logic using the logical operations, and demonstrate the agent's decision-making process. Scenario: TESLA AUTOPILOT CAR Imagine an autonomous car navigating through a highway. The machine needs to decide the best action based on the presence of obstacles and the desired destination. Propositions p: There is an obstacle in front of the car. q: The desired destination is to th e left of the car r: The desired destination is to the right of the car Decision Logic The car's decision logic will be based on the following rules: If there is an obstacle in front of the car and the desired destination is to the right, the robot should turn left. If there is an obstacle in front of the car and the desired destination is to the left, the robot should turn right. If there is no obstacle in front of the car, the robot should move forward. We can express these rules using logical operations: python def decide_action(p, q, r): if and_operation(p, q): return "Turn left" elif and_operation(p, r): return "Turn right" else: return "Move forward"

Example Usage Let's consider a few scenarios and see how the AI agent makes decisions: python

Scenario 1: Obstacle in front, destination to the right

print("Scenario 1:", decide_action(True, True, False)) # Output: Turn left

Scenario 2: Obstacle in front, destination to the left

print("Scenario 2:", decide_action(True, False, True)) # Output: Turn right

Scenario 3: No obstacle, destination to the right

print("Scenario 3:", decide_action(False, True, False)) # Output: Move forward

In the first scenario, the car detects an obstacle in front (p) and the desired destination is to the left (q), so it decides to turn left. In the second scenario, the car detects an obstacle in front (p) and the desired destination is to the right (r), so it decides to turn right. In the third scenario, there is no obstacle in front of the robot (¬p), so it decides to move forward. Summary By defining the scenario, propositions, and decision logic using logical operations, we have created a simple AI agent that can make decisions based

on environmental conditions. This example demonstrates how logical reasoning can be applied to guid e an agent's decision-making process in a specific scenario. You can further expand upon this concept by introducing more complex scenarios, additional propositions, and more sophisticated decision logic using the logical operations presented earlier.