

9

Logic Programming

In this chapter, we are going to learn how to write programs using logic programming. We will discuss various programming paradigms and see how programs are constructed with logic programming. We will learn about the building blocks of logic programming and see how to solve problems in this domain. We will implement Python programs to build various solvers that solve a variety of problems.

By the end of this chapter, you will know about the following:

- What is logic programming?
- Understanding the building blocks of logic programming
- Solving problems using logic programming
- Installing Python packages
- Matching mathematical expressions
- Validating primes
- Parsing a family tree
- Analyzing geography
- Building a puzzle solver

What is logic programming?

Logic programming is a programming paradigm, which basically means it is a way to approach programming. Before we talk about what it constitutes and how it is relevant in **Artificial Intelligence (AI)**, let's talk a bit about programming paradigms.

The concept of programming paradigms originates from the need to classify programming languages. It refers to the way computer programs solve problems through code.

Some programming paradigms are primarily concerned with implications or the sequence of operations used to achieve a particular result. Other programming paradigms are concerned about how we organize the code.

Here are some of the more popular programming paradigms:

- **Imperative:** Uses statements to change a program's state, thus allowing for side effects.
- **Functional:** Treats computation as an evaluation of mathematical functions and does not allow changing states or mutable data.
- **Declarative:** A way of programming where programs are written by describing what needs to be done and not how to do it. The logic of the underlying computation is expressed without explicitly describing the control flow.
- **Object oriented:** Groups the code within a program in such a way that each object is responsible for itself. Objects contain data and methods that specify how changes happen.
- **Procedural:** Groups the code into functions and each function is responsible for a series of steps.
- **Symbolic:** Uses a style of syntax and grammar through which the program can modify its own components by treating them as plain data.
- **Logic:** Views computation as automatic reasoning over a database of knowledge consisting of facts and rules.

Logic programming has been around for a while. A language that was quite popular during one of the last heydays of AI was Prolog. It is a language that uses only three constructs:

- Facts
- Rules
- Questions

But with these three constructs you were able to build some powerful systems. One popular usage was in the construction of "expert systems." The idea behind was to interview human experts that had been working in a given field for a long time and codify the interview into AI systems. Some examples of fields for which expert systems were built were:

- **Medicine** – Famous examples include MYCIN, INTERNIST-I, and CADUCEUS
- **Chemical analysis** – DENDRAL is an analysis system used to predict molecular structure

- **Finance** – Advisory programs to assist bankers in making loans
- **Debugging programs** – SAINT, MATLAB, and MACSYMA

In order to understand logic programming, it's necessary to understand the concepts of computation and deduction. To compute something, we start with an expression and a set of rules. This set of rules is basically the program.

Expressions and rules are used to generate the output. For example, let's say we want to compute the sum of 23, 12, and 49:

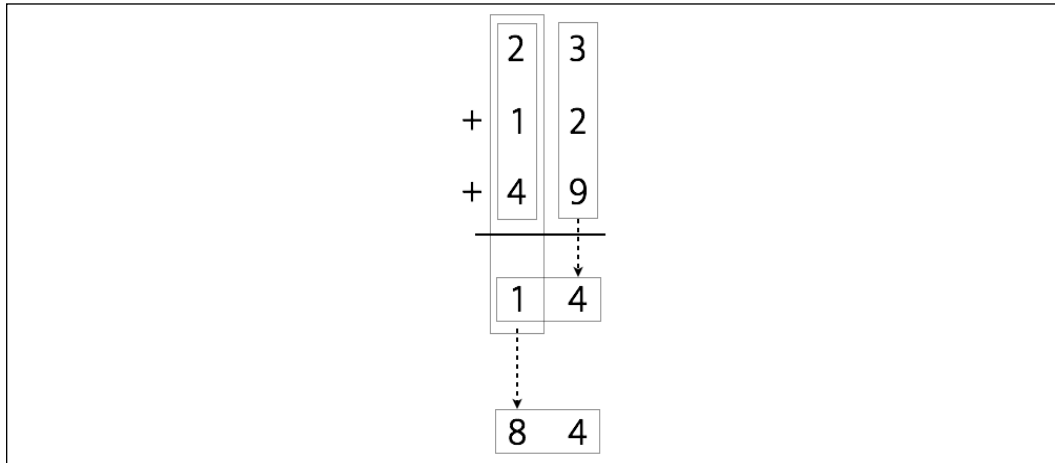


Figure 1: Addition operation mechanics

The procedure to complete the operation would be as follows:

1. Add $3 + 2 + 9 = 14$
2. We need to keep a single digit, which is the 4, and then we carry the 1
3. Add $2 + 1 + 4$ (plus the 1 we carried) = 8
4. Combine 8 and 4. The final result is: 84

On the other hand, to deduce something, we need to start from a conjecture. A proof is constructed according to a set of rules. The process of computation is mechanical, whereas the process of deduction is more creative.

When writing a program using the logic programming paradigm, a set of statements is specified based on facts and rules about the problem domain, and the solver solves it using this information.

Understanding the building blocks of logic programming

In programming object-oriented or imperative paradigms, a variable always needs to be defined. In logic programming, things work a bit differently. An uninstantiated argument can be passed to a function and the interpreter will instantiate these variables by looking at the facts defined by the user. This is a powerful way of approaching the variable matching problem. The process of matching variables with different items is called unification. This is one of the ways logic programming is different. Relations can also be specified in logic programming. Relations are defined by means of clauses called facts and rules.

Facts are just statements that are truths about the program and the data. The syntax is straightforward. For example, *Donald is Allan's son* is a fact, whereas *Who is Allan's son?* is not be a fact. Every logic program needs facts so that it can achieve the given goal based on them.

Rules are the things we have learned in terms of how to express various facts and how to query them. They are the constraints that must be met, and they enable you to make conclusions about the problem domain. For example, let's say you are working on building a chess engine. You need to specify all the rules about how each piece can move on the chessboard.

Solving problems using logic programming

Logic programming looks for solutions by using facts and rules. A goal must be specified for each program. When a logic program and a goal don't contain any variables, the solver comes up with a tree that constitutes the search space for solving the problem and getting to the goal.

One of the most important things about logic programming is how we treat the rules. Rules can be viewed as logical statements. Let's consider the following:

Kathy orders dessert => Kathy is happy

This can be read as an implication that says, *If Kathy is happy, then Kathy orders dessert*. It can also be construed as *Kathy orders dessert whenever she is happy*.

Similarly, let's consider the following rules and facts:

canfly(X) :- bird(X), not abnormal(X).

abnormal(X) :- wounded(X).

bird(john).

bird(mary).

wounded(john).

Here is how to interpret the rules and facts:

- John is wounded
- Mary is a bird
- John is a bird
- Wounded birds are abnormal
- Birds that are not abnormal can fly

From this, we can conclude that Mary can fly, and John cannot fly.

This construction is used in various forms throughout logic programming to solve various types of problems. Let's go ahead and see how to solve these problems in Python.

Installing Python packages

Before we start logic programming in Python, we need to install a couple of packages. The package `logpy` is a Python package that enables logic programming in Python. We will also be using `SymPy` for some of the problems. So, let's go ahead and install `logpy` and `sympy` using `pip`:

```
$ pip3 install logpy Kannan
$ pip3 install sympy
```

If you get an error during the installation process for `logpy`, you can install it from source at <https://github.com/logpy/logpy>. Once you have successfully installed these packages, you can proceed to the next section.

Matching mathematical expressions

We encounter mathematical operations all the time. Logic programming is an efficient way of comparing expressions and finding out unknown values. Let's see how to do that.

Create a new Python file and import the following packages:

```
from logpy import run, var, fact
import logpy.assoccomm as la
```

Define a couple of mathematical operations:

```
# Define mathematical operations
add = 'addition'
mul = 'multiplication'
```

Both addition and multiplication are commutative operations (meaning the operands can be flipped without changing the result). Let's specify that:

```
# Declare that these operations are commutative
# using the facts system
fact(la.commutative, mul)
fact(la.commutative, add)
fact(la.associative, mul)
fact(la.associative, add)
```

Let's define some variables:

```
# Define some variables
a, b, c = var('a'), var('b'), var('c')
```

Consider the following expression:

```
expression_orig = 3 x (-2) + (1 + 2 x 3) x (-1)
```

Let's generate this expression with masked variables. The first expression would be:

$$expression1 = (1 + 2 \times a) \times b + 3 \times c$$

The second expression would be:

$$expression2 = c \times 3 + b \times (2 \times a + 1)$$

The third expression would be:

$$expression3 = (((2 \times a) \times b) + b) + 3 \times c$$

If you observe carefully, all three expressions represent the same basic expression. The goal is to match these expressions with the original expression to extract the unknown values:

```
# Generate expressions
expression_orig = (add, (mul, 3, -2), (mul, (add, 1, (mul, 2, 3)),
-1))
```

```
expression1 = (add, (mul, (add, 1, (mul, 2, a)), b), (mul, 3, c))
expression2 = (add, (mul, c, 3), (mul, b, (add, (mul, 2, a), 1)))
expression3 = (add, (add, (mul, (mul, 2, a), b), b), (mul, 3, c))
```

Compare the expressions with the original expression. The method `run` is commonly used in `logpy`. This method takes the input arguments and runs the expression. The first argument is the number of values, the second argument is a variable, and the third argument is a function:

```
# Compare expressions
print(run(0, (a, b, c), la.eq_assoccomm(expression1, expression_
orig)))
print(run(0, (a, b, c), la.eq_assoccomm(expression2, expression_
orig)))
print(run(0, (a, b, c), la.eq_assoccomm(expression3, expression_
orig)))
```

The full code is given in `expression_matcher.py`. If you run the code, you will see the following output:

```
((3, -1, -2),)
((3, -1, -2),)
()
```

The three values in the first two lines represent the values for `a`, `b`, and `c`. The first two expressions matched with the original expression, whereas the third one returned nothing. This is because, even though the third expression is mathematically the same, it is structurally different. Pattern comparison works by comparing the structure of the expressions.

Validating primes

Let's see how to use logic programming to check for prime numbers. We will use the constructs available in `logpy` to determine which numbers in the given list are prime, as well as finding out if a given number is a prime or not.

Create a new Python file and import the following packages:

```
import itertools as it
import logpy.core as lc
from sympy.ntheory.generate import prime, isprime
```

Next, define a function that checks if the given number is prime depending on the type of data. If it's a number, then it's straightforward. If it's a variable, then we must run the sequential operation. To give a bit of background, the method `conde` is a goal constructor that provides logical AND and OR operations.

The method `condeseq` is like `conde`, but it supports the generic iteration of goals:

```
# Check if the elements of x are prime
def check_prime(x):
    if lc.isvar(x):
        return lc.condeseq([(lc.eq, x, p)] for p in map(prime,
it.count(1)))
    else:
        return lc.success if isprime(x) else lc.fail
```

Declare the variable `x` that will be used:

```
# Declate the variable
x = lc.var()
```

Define a set of numbers and check which numbers are prime. The method `membero` checks if a given number is a member of the list of numbers specified in the input argument:

```
# Check if an element in the list is a prime number
list_nums = (23, 4, 27, 17, 13, 10, 21, 29, 3, 32, 11, 19)
print('\nList of primes in the list:')
print(set(lc.run(0, x, (lc.membero, x, list_nums), (check_prime, x))))
```

Let's use the function in a slightly different way now by printing the first 7 prime numbers:

```
# Print first 7 prime numbers
print('\nList of first 7 prime numbers:')
print(lc.run(7, x, check_prime(x)))
```

The full code is given in `prime.py`. If you run the code, you will see the following output:

```
List of primes in the list:
{3, 11, 13, 17, 19, 23, 29}
List of first 7 prime numbers: (2, 3, 5, 7, 11, 13, 17)
```

You can confirm that the output values are correct.

Parsing a family tree

Now that we are more familiar with logic programming, let's use it to solve an interesting problem. Consider the following family tree:

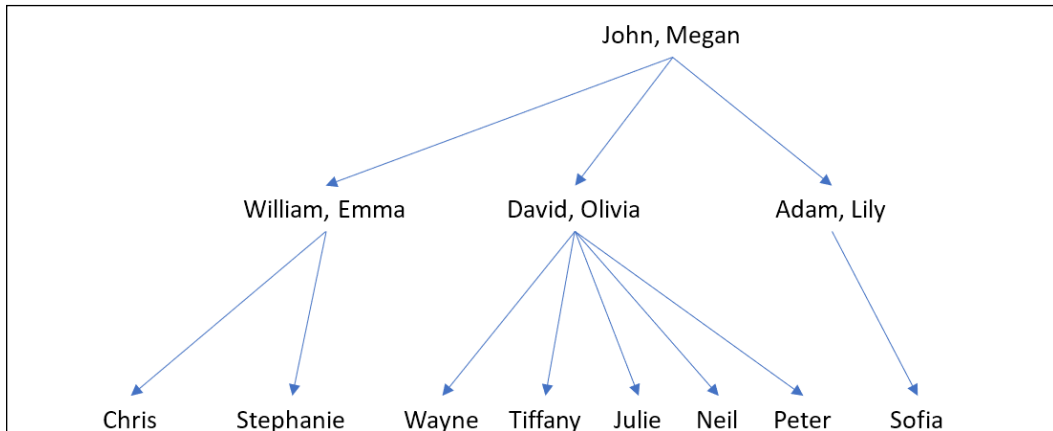


Figure 2: Sample family tree

John and Megan have three sons – William, David, and Adam. The wives of William, David, and Adam are Emma, Olivia, and Lily respectively. William and Emma have two children – Chris and Stephanie. David and Olivia have five children – Wayne, Tiffany, Julie, Neil, and Peter. Adam and Lily have one child – Sophia. Based on these facts, we can create a program that can tell us the name of Wayne's grandfather or Sophia's uncles. Even though we have not explicitly specified anything about the grandparent or uncle relationships, logic programming can infer them.

These relationships are specified in a file called `relationships.json` provided for you. The file looks like the following:

```

{
  "father":
  [
    {"John": "William"},
    {"John": "David"},
    {"John": "Adam"},
    {"William": "Chris"},
    {"William": "Stephanie"},
    {"David": "Wayne"},
    {"David": "Tiffany"},
    {"David": "Julie"},
    {"David": "Neil"},
    {"David": "Peter"},
    {"Adam": "Sophia"}
  ],
  "mother":
  [
    {"Megan": "William"},
  ]
}
```

```

        {"Megan": "David"},
        {"Megan": "Adam"},
        {"Emma": "Stephanie"},
        {"Emma": "Chris"},
        {"Olivia": "Tiffany"},
        {"Olivia": "Julie"},
        {"Olivia": "Neil"},
        {"Olivia": "Peter"},
        {"Lily": "Sophia"}
    ]
}
```

It is a simple JSON file that specifies the father and mother relationships. Note that we haven't specified anything about the husbands and wives, grandparents, or uncles.

Create a new Python file and import the following packages:

```

import json
from logpy import Relation, facts, run, conde, var, eq
```

Define a function to check if x is the parent of y . We will use the logic that if x is the parent of y , then x is either the father or the mother. We have already defined "father" and "mother" in the fact base:

```

# Check if 'x' is the parent of 'y'
def parent(x, y):
    return conde([father(x, y)], [mother(x, y)])
```

Define a function to check if x is the grandparent of y . We will use the logic that if x is the grandparent of y , then the offspring of x will be the parent of y :

```

# Check if 'x' is the grandparent of 'y'
def grandparent(x, y):
    temp = var()
    return conde((parent(x, temp), parent(temp, y)))
```

Define a function to check if x is the sibling of y . We will use the logic that if x is the sibling of y , then x and y will have the same parents. Notice that there is a slight modification needed here because when we list out all the siblings of x , x will be listed as well because x satisfies these conditions. So, when we print the output, we will have to remove x from the list. We will discuss this in the main function:

```

# Check for sibling relationship between 'a' and 'b'
def sibling(x, y):
    temp = var()
    return conde((parent(temp, x), parent(temp, y)))
```

Define a function to check if x is y 's uncle. We will use the logic that if x is y 's uncle, then x grandparents will be the same as y 's parents. Notice that there is a slight modification needed here because when we list out all the uncles of x , x 's father will be listed as well because x 's father satisfies these conditions. So, when we print the output, we will have to remove x 's father from the list. We will discuss this in the main function:

```
# Check if x is y's uncle
def uncle(x, y):
    temp = var()
    return conde((father(temp, x), grandparent(temp, y)))
```

Define the main function and initialize the relations `father` and `mother`:

```
if __name__ == '__main__':
    father = Relation()
    mother = Relation()
```

Load the data from the `relationships.json` file:

```
with open('relationships.json') as f:
    d = json.loads(f.read())
```

Read the data and add it to the fact base:

```
for item in d['father']:
    facts(father, (list(item.keys())[0], list(item.values())[0]))

for item in d['mother']:
    facts(mother, (list(item.keys())[0], list(item.values())[0]))
```

Define the variable x :

```
x = var()
```

We are now ready to ask some questions and see if the solver can come up with the right answers. Let's ask who John's children are:

```
# John's children
name = 'John'
output = run(0, x, father(name, x))
print("\nList of " + name + "'s children:")
for item in output:
    print(item)
```

Who is William's mother?

```
# William's mother
```

```
name = 'William'
output = run(0, x, mother(x, name))[0]
print("\n" + name + "'s mother:\n" + output)
```

Who are Adam's parents?

```
# Adam's parents name = 'Adam'
output = run(0, x, parent(x, name))
print("\nList of " + name + "'s parents:")
for item in output:
    print(item)
```

Who are Wayne's grandparents?

```
# Wayne's grandparents name = 'Wayne'
output = run(0, x, grandparent(x, name))
print("\nList of " + name + "'s grandparents:")
for item in output:
    print(item)
```

Who are Megan's grandchildren?

```
# Megan's grandchildren
name = 'Megan'
output = run(0, x, grandparent(name, x))
print("\nList of " + name + "'s grandchildren:")
for item in output:
    print(item)
```

Who are David's siblings?

```
# David's siblings
name = 'David'
output = run(0, x, sibling(x, name))
siblings = [x for x in output if x != name]
print("\nList of " + name + "'s siblings:")
for item in siblings:
    print(item)
```

Who are Tiffany's uncles?

```
# Tiffany's uncles
name = 'Tiffany'
name_father = run(0, x, father(x, name))[0]
output = run(0, x, uncle(x, name))
output = [x for x in output if x != name_father]
print("\nList of " + name + "'s uncles:")
for item in output:
    print(item)
```

List all of the spouses in the family:

```
# All spouses
a, b, c = var(), var(), var()
output = run(0, (a, b), (father, a, c), (mother, b, c))
print("\nList of all spouses:")
for item in output:
    print('Husband:', item[0], '<==> Wife:', item[1])
```

The full code is given in `family.py`. If you run the code, you will see some outputs. The first half looks like the following:

```
List of John's children:
David
William
Adam

William's mother:
Megan

List of Adam's parents:
John
Megan

List of Wayne's grandparents:
John
Megan
```

Figure 3: Family tree example output

The second half looks like the following:

```
List of Megan's grandchildren:
Chris
Sophia
Peter
Stephanie
Julie
Tiffany
Neil
Wayne

List of David's siblings:
William
Adam

List of Tiffany's uncles:
William
Adam

List of all spouses:
Husband: Adam <==> Wife: Lily
Husband: David <==> Wife: Olivia
Husband: John <==> Wife: Megan
Husband: William <==> Wife: Emma
```

Figure 4: Family tree example output

You can compare the output with the family tree to ensure that the answers are indeed correct.

Analyzing geography

Let's use logic programming to build a solver to analyze geography. In this problem, we will specify information about the location of various states in the US and then query the program to answer various questions based on those facts and rules. The following is a map of the US:



Figure 5: Adjacent and coastal states example map

You have been provided with two text files named `adjacent_states.txt` and `coastal_states.txt`. These files contain the details about which states are adjacent to each other and which states are coastal. Based on this, we can get interesting information like "What states are adjacent to both Oklahoma and Texas?" or "Which coastal state is adjacent to both New Mexico and Louisiana?"

Create a new Python file and import the following:

```
from logpy import run, fact, eq, Relation, var
```

Initialize the relations:

```
adjacent = Relation()
coastal = Relation()
```

Define the input files to load the data from:

```
file_coastal = 'coastal_states.txt'
file_adjacent = 'adjacent_states.txt'
```

Load the data:

```
# Read the file containing the coastal states
with open(file_coastal, 'r') as f:
    line = f.read()
    coastal_states = line.split(',')
```

Add the information to the fact base:

```
# Add the info to the fact base
for state in coastal_states:
    fact(coastal, state)
```

Read the adjacent data:

```
# Read the file containing the coastal states
with open(file_adjacent, 'r') as f:
    adjlist = [line.strip().split(',') for line in f if line and
               line[0].isalpha()]
```

Add the adjacency information to the fact base:

```
# Add the info to the fact base
for L in adjlist:
    head, tail = L[0], L[1:]
    for state in tail:
        fact(adjacent, head, state)
```

Initialize the variables x and y:

```
# Initialize the variables
x = var()
y = var()
```

We are now ready to ask some questions. Check if Nevada is adjacent to Louisiana:

```
# Is Nevada adjacent to Louisiana?
output = run(0, x, adjacent('Nevada', 'Louisiana'))
print('\nIs Nevada adjacent to Louisiana?:')
print('Yes' if len(output) else 'No')
```

Print out all the states that are adjacent to Oregon:

```
# States adjacent to Oregon
```

```
output = run(0, x, adjacent('Oregon', x))
print('\nList of states adjacent to Oregon:')
for item in output:
    print(item)
```

List all the coastal states that are adjacent to Mississippi:

```
# States adjacent to Mississippi that are coastal
output = run(0, x, adjacent('Mississippi', x), coastal(x))
print('\nList of coastal states adjacent to Mississippi:')
for item in output:
    print(item)
```


List seven states that border a coastal state:

```
# List of 'n' states that border a coastal state n = 7
output = run(n, x, coastal(y), adjacent(x, y))
print('\nList of ' + str(n) + ' states that border a coastal state:')
for item in output:
    print(item)
```

List states that are adjacent to both Arkansas and Kentucky:

```
# List of states that adjacent to the two given states
output = run(0, x, adjacent('Arkansas', x), adjacent('Kentucky', x))
print('\nList of states that are adjacent to Arkansas and Kentucky:')
for item in output:
    print(item)
```

The full code is given in `states.py`. If you run the code, you will see the following output:



```
Is Nevada adjacent to Louisiana?:
No

List of states adjacent to Oregon:
Washington
California
Nevada
Idaho

List of coastal states adjacent to Mississippi:
Alabama
Louisiana

List of 7 states that border a coastal state:
Georgia
Pennsylvania
Massachusetts
Wisconsin
Maine
Oregon
Ohio

List of states that are adjacent to Arkansas and Kentucky:
Missouri
Tennessee
```

Figure 6: Adjacent and coastal state example output

You can cross-check the output with the US map to verify whether the answers are right. You can also add more questions to the program to see if it can answer them.

Building a puzzle solver

Another interesting application of logic programming is solving puzzles. We can specify the conditions of a puzzle and the program will come up with a solution. In this section, we will specify various bits and pieces of information about four people and ask for the missing piece of information.

In the logic program, we specify the puzzle as follows:

- Steve has a blue car.
- The person who owns a cat lives in Canada. Matthew lives in the USA.
- The person with a black car lives in Australia.
- Jack has a cat.
- Alfred lives in Australia.
- The person who has a dog lives in France.
- Who has a rabbit?

The goal is to find the person who has a rabbit. Here are the full details about the four people:

		Pet	Car Color	Country
	Steve	dog	blue	France
	Jack	cat	green	Canada
	Matthew	rabbit	yellow	USA
	Alfred	parrot	black	Australia

Figure 7: Puzzle solver input data

Create a new Python file and import the following packages:

```
from logpy import *
from logpy.core import lall
```

Declare the variable `people`:

```
# Declare the variable people
people = var()
```

Define all the rules using `lall`. The first rule is that there are four people:

```
# Define the rules
rules = lall(
    # There are 4 people
    (eq, (var(), var(), var(), var()), people),
```

The person named Steve has a blue car:

```
# Steve's car is blue
(membero, ('Steve', var(), 'blue', var()), people),
```

The person who has a cat lives in Canada:

```
# Person who has a cat lives in Canada
(membero, (var(), 'cat', var(), 'Canada'), people),
```

The person named Matthew lives in the USA:

```
# Matthew lives in USA
(membero, ('Matthew', var(), var(), 'USA'), people),
```

The person who has a black car lives in Australia:

```
# The person who has a black car lives in Australia
(membero, (var(), var(), 'black', 'Australia'), people),
```

The person named Jack has a cat:

```
# Jack has a cat
(membero, ('Jack', 'cat', var(), var()), people),
```

The person named Alfred lives in Australia:

```
# Alfred lives in Australia
(membero, ('Alfred', var(), var(), 'Australia'), people),
```

The person who has a dog lives in France:

```
# Person who owns the dog lives in France
(membero, (var(), 'dog', var(), 'France'), people),
```

One of the people in this group has a rabbit. Who is that person?

```
# Who has a rabbit?
(membero, (var(), 'rabbit', var(), var()), people)
)
```

Run the solver with the preceding constraints:

```
# Run the solver
solutions = run(0, people, rules)
```

Extract the output from the solution:

```
# Extract the output
output = [house for house in solutions[0] if 'rabbit' in house][0][0]
```

Print the full matrix obtained from the solver:

```
# Print the output
print('\n' + output + ' is the owner of the rabbit')
print('\nHere are all the details:')
attribs = ['Name', 'Pet', 'Color', 'Country']
print('\n' + '\t\t'.join(attribs))
print('=' * 57)
for item in solutions[0]:
    print('')
    print('\t\t'.join([str(x) for x in item]))
```

The full code is given in `puzzle.py`. If you run the code, you will see the following output:

```
Matthew is the owner of the rabbit
Here are all the details:
Name          Pet          Color          Country
=====
Steve         dog          blue          France
Jack          cat          ~_9           Canada
Matthew       rabbit       ~_11          USA
Alfred        ~_13        black         Australia
```

Figure 8: Puzzle solver output

The preceding figure shows all the values obtained using the solver. Some of them are still unknown as indicated by the numbered names. Even though the information was incomplete, the solver was able to answer the question. But in order to answer every single question, you may need to add more rules. This program was to demonstrate how to solve a puzzle with incomplete information. You can play around with it and see how you can build puzzle solvers for various scenarios.

Summary

In this chapter, we learned how to write Python programs using logic programming. We discussed how various programming paradigms deal with building programs. We understood how programs are built in logic programming. We learned about various building blocks of logic programming and discussed how to solve problems in this domain. We implemented various Python programs to solve interesting problems and puzzles.

In the next chapter, we will learn about heuristic search techniques and use those algorithms to solve real-world problems.