Sister Nivedita University

Department of Computer Science

Lab Manual

for

Paper Name: Artificial Intelligence Lab

Paper Code-

PREFACE

This laboratory manual is prepared by the Department of Computer Science for Artificial Intelligence Lab using Python. This lab manual can be used as an instructional book for students, staff and instructors to assist in performing and understanding the experiments.

INDEX OF THE CONTENTS

SYSTEM REQUIREMENTS

1 HARDWARE REQUIREMENTS:

Intel Pentium 915 GV

80GB SATA II

512MB DDR

2 SOFTWARE REQUIREMENTS:

Python with Jupyter Notebook

ABOUT THE LAB

AIM

The aim of this laboratory is to develop a student's programming skills in Artificial Intelligence lab with Python programming and their applications.

LEARNING OBJECTIVE

The objective of the course is to develop basic programming skills using Python and Artificial Intelligence programming with basic and advanced topics of this course in a way they can solve programming problems related to the real world.

To become proficient with the fundamental tools of program design using Python and Artificial intelligence analysis of algorithms. To develop the ability to design and write programs for implementation of such algorithms.

FORMAT OF THE LAB RECORDS TO BE PREPARED BY THE STUDENTS

The students are required to maintain the lab records as per the instructions:

1. All the record files should have a cover page as per the format.
2. All the record files should have an index as per the format.
3. All the records should have the following :

   I. Problem description
   II. Python Code to solve the problem using Jupyter Notebook
   III. Output of the code

MARKING SCHEME
FOR THE
PRACTICAL EXAMINATION

PRACTICAL EXAMINATION

It is taken by the concerned faculty member of the batch.

THE MARKING SCHEME FOR THE EXAM IS:

Time Allotted: 2 Hours **Full Marks: 100**

Internal Assessment: 50

a) **Participation = 10**
b) **Lab Assignments & Submission = 20**
c) **Continuous evaluation during practical classes = 20**

Lab Exam: 50

a) **Final End-semester assessment on an Experiment = 35**
b) **Viva-Voce = 15**

Total Marks-100

Mandatory instructions for students:

1. Students should report to the concerned labs as per the given timetable.
2. Students should make an entry in the log book whenever they enter the labs during practicals or for their work.
3. When the experiment is completed, students should shut down the computers and makethe counter entry in the logbook.
4. Any damage to the lab computers will be viewed seriously.
5. Students should not leave the lab without the concerned faculty's permission.

PART A

Introduction to Python

Programming

Introduction to Python Programming.

Theory

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

Prerequisites

You should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages is a plus.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features

Python's features include −

- Easy-to-learn − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- Easy-to-read − Python code is more clearly defined and visible to the eyes.
- Easy-to-maintain − Python's source code is fairly easy-to-maintain.
- A broad standard library − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- Interactive Mode − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- Portable − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

- Extendable − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- Databases − Python provides interfaces to all major commercial databases.
- GUI Programming − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- Scalable − Python provides a better structure and support for large programs than shell scripting.
- Apart from the above-mentioned features, Python has a big list of good features, few are listed below −
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

First Python Program:

1. Open notepad and type following program

Print ("Hello World")

2. Save above program with name.py

3. Open command prompt and change path to python program location

4. Type "python name.py" (without quotes) to run the program.

Operators are the constructs which can manipulate the value of operands.
Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Python Variables: Declare, Concatenate, Global & Local

What is a Variable in Python?

A Python variable is a reserved memory location to store values. In other words, a variable in a python program gives data to the computer for processing.

Every value in Python has a datatype. Different data types in Python are Numbers, List, Tuple, Strings, Dictionary, etc. Variables can be declared by any name or even alphabets like a, aa, abc, etc.

How to Declare and use a Variable

Let see an example. We will declare variable "a" and print it.

Python 1 Example

Python 2 Example

List of some different variable types

```
x = 123         # integer
x = 123L        # long integer
x = 3.14        # double float
x = "hello"     # string
x = [0,1,2]     # list
x = (0,1,2)     # tuple
x = open('hello.py', 'r')         # file
```

Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

Non technically, you can think of constant as a bag to store some books and those books cannot be replaced once place inside the bag.

Assigning value to a constant in Python

In Python, constants are usually declared and assigned on a module. Here, the module means a new file containing variables, functions etc which is imported to main file. Inside the module, constants are written in all capital letters and underscores separating the words.

Example 3: Declaring and assigning value to a constant

Create a constant.py

```
1.
2.              PI = 3.14
                GRAVITY = 9.8
```

Create a main.py

```
1.
2.
3.
4.              import constant
```

```
print(constant.PI)
print(constant.GRAVITY)
```

When you run the program, the output will be:

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look on all operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = 10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20

//

Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are 9//2 = 4 and removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) − 9.0//2.0

= 4.0,

11//3 =

-4, - 11.0//3

= -4.0

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then − [Show Example]

< If the value of left operand is less than the value of right operand, then condition becomes true. (a < b) is true.
>= If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. (a >= b) is not true.
<= If the value of left operand is less than or equal to the value of right operand, then condition becomes true. (a <= b) is true.

Python Assignment Operators
Assume variable a holds 10 and variable b holds 20, then − [Show Example]

*= Multiply AND It multiplies right operand with the left operand and assign the result to left operand
c *= a is equivalent
to c = c * a
/= Divide AND It divides left operand with the right operand and assign the result to left operand c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND It takes modulus using two operands and assign the result to left operand c %= a is equivalent to c = c % a
**= Exponent AND Performs exponential (power) calculation on operators and assign value to the left operand c **= a is equivalent to c = c ** a
//= Floor Division It performs floor division on operators and assign value to the left operand c //= a is equivalent to c = c // a
Python Bitwise Operators
Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows − a = 0011 1100 b = 0000 1101

a&b = 0000 1100
a|b = 0011 1101 a^b
= 0011 0001

~a = 1100 0011
There are following Bitwise operators supported by Python language [Show Example]

<< Binary Left Shift The left operands value is moved left by the number of bits specified by the right operand. a << 2 = 240
(means 1111 0000)
>> Binary Right Shift The left operands value is moved right by the number of bits specified by the right operand. a >> 2 = 15
(means 0000 1111)

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

[Show Example]

Operator Description Example

and Logical AND If both the operands are true then condition becomes true. (a and b) is true.

or Logical OR If any of the two operands are non-zero then condition becomes true. (a or b) is true.

not Logical NOT Used to reverse the logical state of its operand. Not(a and b) is false.

Used to reverse the logical state of its operand.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

[Show Example]

Operator Description Example

In Evaluates to true if it finds a variable in the specified sequence and false otherwise. x in y, here in results in a 1 if x is a member of sequence y.

not in Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

[Show Example]

Operator Description Example

Is Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. x is y, here is results in 1 if id(x) equals id(y).

is not Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. x is not y, here is not results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

The following table lists all operators from highest precedence to lowest. [Show Example]

8 <= < > >=
Comparison operators
9 <> == !=
Equality operators
10 = %= /= //= -= += *= **=
Assignment operators
11 Is, is not
Identity operators
12 In, not in
Membership operators
13 not, or, and
Logical operators

Exercise:

1. WAP to add two numbers in python
2. WAP to declare variables and display types of respective variables
3. WAP to demonstrate type casting in python
4. WAP to demonstrate Logical operators

Assignment 2

Aim: To study strings in Python
Theory:
String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

is the same as "hello".

You can display a string literal with the print() function:

Example
print("Hello") print('Hello') Assign String to a Variable
Assigning a string to a variable is done with the variable name followed by an equal sign and the string:
a = "Hello" print(a)
Multiline Strings

You can assign a multiline string to a variable by using three quotes:
a = """Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.""" print(a)
Or three single quotes: Example
a = '''Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.''' print(a)

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length

of 1. Square brackets can be used to access elements of the string.
a = "Hello, World!" print(a[1])

Example

Substring. Get the characters from position 2 to position 5 (not included):

b = "Hello, World!" print(b[2:5])

In-bulit Functions in Python:

1. Strip():Removes all leading whitespace in string.
2. len(string):Returns the length of the string
3. upper():Converts lowercase letters in string to uppercase.
4. Lower(): Vice versa
5. split(str="", num=string.count(str)):Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
6. replace(old, new [, max]):Replaces all occurrences of old in string with new or at most max occurrences if max given.
7. find(str, beg=0 end=len(string)):Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise

Exercise:
1. WAP to Calculate length of string
2. WAP to make string from 1st two and last two characters from given string.
3. WAP to concatenate two strings

Assignment: 3

Aim: To study conditional statements in python
Theory:

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.
Following is the general form of a typical decision making structure found in most of the programming languages −

Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.
Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	if statements

An if statement consists of a boolean expression followed by one or more statements.

2	if...else statements

An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

3	nested if statements

You can use one if or else if statement inside another if or else ifstatement(s).
Let us go through each decision making briefly − Single Statement Suites
If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.
Here is an example of a one-line if clause −

When the above code is executed, it produces the following result −

Exercise:

1. WAP to find out greatest of 3 numbers
2. WAP to find whether given number is odd or even
3. Write a C program to check whether a character is uppercase or lowercasealphabet.
4. WAP to find whether given input is number or character

Assignment: 4

Aim: To study Loops in Python
Theory:

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
Programming languages provide various control structures that allow for more complicated execution paths.
A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −
Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

2	for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

3	nested loops

You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail. Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statementimmediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest itscondition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactacallybut you do not want any command or code to execute.

Exercise:

1. WAP to display even numbers from 1-10
2. WAP to add odd numbers from 1-10
3. Write a Python program to get the Fibonacci series between 0 to 50.
4. Write a Python program to remove the characters which have odd index valuesof a given string.

Assignment 5 Aim: To study python arrays, list, tuples, set, dictionary Theory:

What is an Array?

An array is a special variable, which can hold more than one value at a time. If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

car1 = "Ford" car2
= "Volvo" car3 = "BMW"

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array

You refer to an array element by referring to the index number.

x = cars[0]

cars[0] = "Toyota"

The Length of an Array

Use the len() method to return the length of an array (the number of elements in an array).
x = len(cars)

Looping Array Elements

You can use the for in loop to loop through all the elements of an array.
for x in cars: print(x)

Adding Array Elements

You can use the append() method to add an element to an array.
cars.append("Honda")

Removing Array Elements

You can use the pop() method to remove an element from the array.
cars.pop(1)

You can also use the remove() method to remove an element from the array.
cars.remove("Volvo")

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.
Method
Description

append()

Adds an element at the end of the list

clear()

Removes all the elements from the list

copy()

Returns a copy of the list

count()

Returns the number of elements with the specified value

extend()

Add the elements of a list (or any iterable), to the end of the current list

index()

Returns the index of the first element with the specified value

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

insert()

Adds an element at the specified position

pop()

Removes the element at the specified position

remove()

Removes the first item with the specified value

reverse()

Reverses the order of the list

sort()

Sorts the list

Python List:
The list is a most versatile datatype available in Python which can be written as a list of commaseparated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.
Creating a list is as simple as putting different comma-separated values between square brackets. For example −

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists
To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

When the above code is executed, it produces the following result −

Updating Lists
You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

Note − append() method is discussed in subsequent section. When the above code is executed, it produces the following result −

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

When the above code is executed, it produces following result −

Note − remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

L = ['spam', 'Spam', 'SPAM!']

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods

Python includes the following list functions −

Sr.No.	Function with Description
1 cmp(list1, list2) Compares elements of both lists.	
2 len(list) Gives the total length of the list.	
3 max(list) Returns item from the list with max value.	
4 min(list) Returns item from the list with min value.	
5 list(seq) Converts a tuple into list.	

Python includes following list methods

Sr.No.	Methods with Description
1	list.append(obj)

Appends object obj to list

2

list.count(obj)

Returns count of how many times obj occurs in list

3

list.extend(seq)

Appends the contents of seq to list

4

list.index(obj)

Returns the lowest index in list that obj appears

5

list.insert(index, obj)

Inserts object obj into list at offset index

6

list.pop(obj=list[-1])

Removes and returns last object or obj from list

7

list.remove(obj)

Removes object obj from list

8

list.reverse()

Reverses objects of list in place

9

list.sort([func])

Sorts objects of list, use compare func if given

Python Tuple:

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

The empty tuple is written as two parentheses containing nothing −

To write a tuple containing a single value you have to include a comma, even though there is only one value −

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

When the above code is executed, it produces the following result −

Updating Tuples
Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −
When the above code is executed, it produces the following result −

Delete Tuple Elements
Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
To explicitly remove an entire tuple, just use the del statement. For example −

This produces the following result. Note an exception raised, this is because after del tup tuple does not exist any more −

Basic Tuples Operations
Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.
In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes
Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters
Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

When the above code is executed, it produces the following result −

Built-in Tuple Functions
Python includes the following tuple functions −

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple) Gives the total length of the tuple.

3
max(tuple)
Returns item from the tuple with max value.
4
min(tuple)
Returns item from the tuple with min value.
5
tuple(seq)
Converts a list into tuple.

Python Sets:

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

thisset = {"apple", "banana", "cherry"} print(thisset)

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a for loop, or ask if a specified value is present in a
set, by using thein keyword.
Example

Loop through the set, and print the values:
thisset = {"apple", "banana", "cherry"}

for x in thisset: print(x)

thisset = {"apple", "banana", "cherry"} print("banana" in thisset)
Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

To add one item to a set use the add() method.

To add more than one item to a set use the update() method.
thisset = {"apple", "banana", "cherry"} thisset.add("orange")
print(thisset)

thisset = {"apple", "banana", "cherry"} thisset.update(["orange", "mango", "grapes"]) print(thisset)

Get the Length of a Set

To determine how many items a set has, use the len() method.
thisset = {"apple", "banana", "cherry"} print(len(thisset)
Remove Item

To remove an item in a set, use the remove(), or the discard() method.
thisset = {"apple", "banana", "cherry"} thisset.remove("banana") print(thisset)

thisset = {"apple", "banana", "cherry"} thisset.discard("banana") print(thisset)

You can also use thepop(), method to remove an item, but this method will remove item.

the last
removed.

ts are unordered, so you will not know what item that gets

The return value of the pop() method is the removed item.
thisset = {"apple", "banana", "cherry"} x = thisset.pop()
print(x) print(thisset)
Note: Sets are unordered, so when using the gets removed. pop() method, you will not know which item that
Example

The clear() method empties the set:
thisset = {"apple", "banana", "cherry"}

thisset.clear() print(thisset)

thisset = {"apple", "banana", "cherry"} del thisset
print(thisset)

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

thisdict = { "brand": "Ford",
"model": "Mustang", "year": 1964

}
print(thisdict)

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:
x = thisdict["model"]

There is also a method called get() that will give you the same result:

x = thisdict.get("model")

Exercise:
1. Write a Python script to sort (ascending and descending) a dictionary by value
2. Write a Python script to check if a given key already exists in a dictionary.
3. Write a Python script to merge two Python dictionaries
4. Write a Python program to add an item in a tuple.
5. Write a Python program to create a tuple with different data types
6. Write a Python program to sum all the items in a list
7. Write a Python program to get the largest number from a list.
8. Write a Python program to add member(s) in a set.
9. Write a Python program to reverse the order of the items in the array
10. Write a Python program to create an array of 5 integers and display the array items. Accessindividual element through indexes.

Assignment:6
Aim: To study functions in python
Theory:

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example

my_function(): print("Hello from a function")
Calling a Function

To call a function, use the function name followed by parenthesis:

Example

def my_function(): print("Hello from a function") my_function()

Parameters

Information can be passed to functions as parameter. Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example def my_function(fname): print(fname + " Refsnes")
my_function("Emil") my_function("Tobias")
my_function("Linus") Default Parameter

Value

The following example shows how to use a default parameter value. If we call the function without parameter, it uses the default value:

def my_function(country = "Norway"):
print("I am from " + country)

my_function("Sweden") my_function("India") my_function() my_function("Brazil")

Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function. E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

def my_function(food): for x in food: print(x)

fruits = ["apple", "banana", "cherry"] my_function(fruits)
Return Values

To let a function return a value, use the return statement:

Example

def my_function(x):
return 5 * x

print(my_function(3)) print(my_function(5)) print(my_function(9)) Recursion

Python also accepts function recursion, which means a defined function can call itself. Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematicallyelegant

approach to programming.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements-1) every time we recurse. The recursion ends when

the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

```
def tri_recursion(k):
if(k>0):
result = k+tri_recursion(k-1) print(result)
else:           result
= 0
return result

print("\n\nRecursion Example Results") tri_recursion(6)
```

Exercise:

1. Write a Python function to find the Max of three numbers.
2. Write a Python program to reverse a string.

Sample String : "1234abcd"
Expected Output : "dcba4321"

3. Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.

Assignment: 7
Aim: To study mysql commands
Theory: Mysql has two types of languages

a. Data definition language:

i. Create: Used to create database and tables

Syn: create database database_name;
Ex: create database jnec;
Syn: create table table_name(column_name1 type,column_name2 type);
Ex: create table syit(id int, name char(10));

ii. Drop: Used to delete database and table

Syn: drop database/table database_name/table_name;
Ex: drop database/table jnec/syit;

iii. Alter: Used to add, delete or change data type of column of table. Syn: alter table table_name add column column_name type;alter table table_name drop column column name;
alter table table_name alter column column name type; ex: alter table syit add column address varchar(10); alter table syit drop address;
alter table syit alter column name varchar(10);

iv. Truncate: Used to delete contents of table but it will preserve structure of table

Syn: truncate table table_name ;

Ex: truncate table syit;

b. Data manipulation language:
i. Select: used to fetch rows from table
Syn: select * from table_name;(fetches data from all columns) Select column_name from syit;(fetches data from respective column)
Ex: select * from syit; Select name from syit;

ii. Insert: used to insert data into table
Syn: insert into table_name values('value1', value2); (if value is text then include in single quote, if it is number then don't use single quote)
Ex: insert into syit values(1,'amol');

iii. Update: used to update value from particular column based on some contion
Syn: update table_name set column= value where column=value;
Ex: update syit set id=4 where name='amol'; iv. Delete: used to delete row from table
Syn: delete from table_name where col=value;
Ex: delete from syit where name='amol';

Exercise:

1. Create database called mgm.
2. Create table institutes in mgm with columns id, college_name, principal, contact.
3. Insert 5 entries in table.
4. Add column address in institute
5. Drop column contact
6. Modify type of college_name from char to varchar
7. Update contact of Dr H H shinde to 9028885925
8. Delete college of engineering from institutes
9. Truncate institutes
10. Drop institutes and mgm

Assignment: 8 Aim: To connect python with mysql with pymysql
Theory:

What is PyMySQL ?
PyMySQL is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb.
How do I Install PyMySQL?
Before proceeding further, you make sure you have PyMySQL installed on your machine. Just type the following in your Python script and execute it −

If it produces the following result, then it means MySQLdb module is not installed −

The last stable release is available on PyPI and can be installed with pip −

Alternatively (e.g. if pip is not available), a tarball can be downloaded from GitHub and installed

with Setuptools as follows −

Note − Make sure you have root privilege to install the above module.

Database Connection
Before connecting to a MySQL database, make sure of the following points −

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.

- Python module PyMySQL is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Example
Following is an example of connecting with MySQL database "TESTDB" −

```
import pymysql

# Open database connection db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method cursor

= db.cursor()

# execute SQL query using execute() method. cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method. data = cursor.fetchone() print ("Database version : %s
" % data)
# disconnect from server db.close()
```

While running this script, it produces the following result.

If a connection is established with the datasource, then a Connection Object is returned and saved into db for further use, otherwise db is set to None. Next, db object is used to create a cursor object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that the database connection is closed and resources are released.

Creating Database Table
Once a database connection is established, we are ready to create tables or records into the database tables using execute method of the created cursor.

Example
Let us create a Database table EMPLOYEE −

INSERT Operation
The INSERT Operation is required when you want to create your records into a database table.
Example
The following example, executes SQL INSERT statement to create a record in the EMPLOYEE table
–

Example
The following code segment is another form of execution where you can pass parameters directly
–

READ Operation
READ Operation on any database means to fetch some useful information from the database.
Once the database connection is established, you are ready to make a query into this database. You can use either fetchone() method to fetch a single record or fetchall() method to fetch multiple values from a database table.

- fetchone() − It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- fetchall() − It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- rowcount − This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example
The following procedure queries all the records from EMPLOYEE table having salary more than 1000 −

Output
This will produce the following result −

Update Operation
Exercise:

1. Create python application which will accept name and age and store it into mysql
2. Create python application which will display all names and ages from mysql table
3. Write python application to delete all entries with ages 28

Assignment: 9

Aim: To study file handling in python
Theory:
File Handling

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode. There are four different methods (modes) for opening a file:

Syntax

To open a file for reading it is enough to specify the name of the file: f = open("demofile.txt") The code above is the same as: f = open("demofile.txt", "rt")
Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Open a File on the Server
Assume we have the following file, located in the same folder as Python:

Hello! Welcome to demofile.txt This file is for testing purposes. Good Luck!

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

f = open("demofile.txt", "r") print(f.read())

Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:
f = open("demofile.txt", "r") print(f.read(5))

Read Lines

You can return one line by using the readline() method:
f = open("demofile.txt", "r") print(f.readline())

By calling readline() two times, you can read the two first lines:

f = open("demofile.txt", "r") print(f.readline()) print(f.readline())

By looping through the lines of the file, you can read the whole file, line by line:
f = open("demofile.txt", "r") for x in f:
print(x)

Close Files

It is a good practice to always close the file when you are done with it.
f = open("demofile.txt", "r") print(f.readline())
f.close()

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

f = open("demofile2.txt", "a") f.write("Now the file has more content!") f.close()

#open and read the file after the appending: f = open("demofile2.txt", "r") print(f.read())

Example
Open the file "demofile3.txt" and overwrite the content: f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!") f.close()

#open and read the file after the appending: f = open("demofile3.txt", "r") print(f.read())

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

f = open("myfile.txt", "x")

Result: a new empty file is created!

Exercise:

1. Write a Python program to read an entire text file
2. Write a Python program to append text to a file and display the text.
3. Write a Python program to read last n lines of a file

Assignment:10
Aim: To study classes in python
Theory:
Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.
Create a Class
To create a class, use the keyword class:

class MyClass:
x = 5

Create Object

Now we can use the class named myClass to create objects:
p1 = MyClass() print(p1.x)

The init () Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in init () function. All classes have a function called init (), which is always executed when the class is being initiated.

Use the init () function to assign values to object properties, or other operations that are necessary to do when the object is being created:
class Person:
def init (self, name, age): self.name = name
self.age = age

p1 = Person("John", 36) print(p1.name) print(p1.age)

Note: The init () function is called automatically every time the class is being used to create a

new object.
Exercise:
Write a Python class to reverse a string word by word. Input string : 'hello .py'
Expected Output : '.py hello'

PART B

Artificial Intelligence Lab

Using Python

1. Write a program to implement DFS and BFS

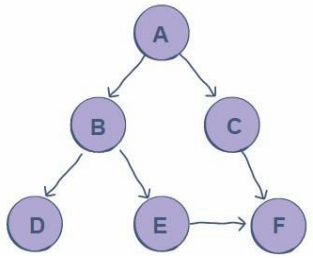
Breadth-first search (BFS) is an algorithm used for tree traversal on graphs or tree data structures. BFS can be easily implemented using recursion and data structures like dictionaries and lists.

The Algorithm

1. Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert it in a queue.
2. If there are no remaining adjacent vertices left, remove the first vertex from the queue.
3. Repeat step 1 and step 2 until the queue is empty or the desired node is found.

Implementation

Consider the below graph, which is to be implemented:



SOURCE CODE:

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
```

```
    visited.append(node)
    queue.append(node)

    while queue:
      s = queue.pop(0)
      print (s, end = " ")

      for neighbour in graph[s]:
        if neighbour not in visited:
          visited.append(neighbour)
          queue.append(neighbour)
bfs(visited, graph, 'A')
```

OUTPUT:

A B C D E F

Explanation

Lines 3-10: The illustrated graph is represented using an adjacency list. An easy way to do this in Python is to use a dictionary data structure, where each vertex has a stored list of its adjacent nodes.

Line 12: visited is a list that is used to keep track of visited nodes.

Line 13: queue is a list that is used to keep track of nodes currently in the queue.

Line 29: The arguments of the bfs function are the visited list, the graph in the form of a dictionary, and the starting node A.

Lines 15-26: bfs follows the algorithm described above:

1. It checks and appends the starting node to the visited list and the queue.
2. Then, while the queue contains elements, it keeps taking out nodes from the queue, appends the neighbors of that node to the queue if they are unvisited, and marks them as visited.

3. This continues until the queue is empty.

Time Complexity

Since all of the nodes and vertices are visited, the time complexity for BFS on a graph is O(V + E)$O(V+E)$; where VV is the number of vertices and EE is the number of edges.

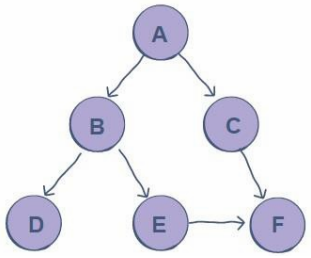
Depth-first search (DFS), is an algorithm for tree traversal on graph or tree data structures. It can be implemented easily using recursion and data structures like dictionaries and sets.

The Algorithm

1. Pick any node. If it is unvisited, mark it as visited and recur on all its adjacent nodes.
2. Repeat until all the nodes are visited, or the node to be searched is found.

Implementation

Consider the below graph, which is to be implemented:



SOURCE CODE:

```
# Using a Python dictionary to act as an adjacency list
        graph = {
           'A' : ['B','C'],
           'B' : ['D', 'E'],
           'C' : ['F'],
           'D' : [],
           'E' : ['F'],
           'F' : []
        }
```

```
visited = set() # Set to keep track of visited nodes.


def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
dfs(visited, graph, 'A')
```

OUTPUT:

A B C D E F

Explanation

- **Lines 2-9**: The illustrated graph is represented using an **adjacency list** - an easy way to do it in Python is to use a dictionary data structure. Each vertex has a list of its adjacent nodes stored.
- **Line 11**: `visited` is a set that is used to keep track of visited nodes.
- **Line 21**: The `dfs` function is called and is passed the `visited` set, the `graph` in the form of a dictionary, and `A`, which is the starting node.
- **Lines 13-18**: `dfs` follows the algorithm described above:
    1. It first checks if the current node is unvisited - if yes, it is appended in the `visited` set.
    2. Then for each neighbor of the current node, the `dfs` function is invoked again.
    3. The base case is invoked when all the nodes are visited. The function then returns.

Time Complexity

- Since all the nodes and vertices are visited, the average time complexity for DFS on a graph is O(V + E)$O(V+E)$, where VV is the number of vertices and EE is the number of edges. In case of DFS on a tree, the time complexity is O(V)$O(V)$, where VV is the number of nodes.

Note: We say average time complexity because a set's in operation has an average time complexity of O(1)O(1). If we used a list, the complexity would be higher.

2. Write a Program to find the solution for travelling salesman Problem.

What is a Travelling Salesperson Problem?

The travelling salesperson problem (TSP) is a classic optimization problem where the goal is to determine the shortest tour of a collection of n "cities" (i.e. nodes), starting and ending in the same city and visiting all of the other cities exactly once.

In such a situation, a solution can be represented by a vector of n integers, each in the range 0 to n-1, specifying the order in which the cities should be visited.

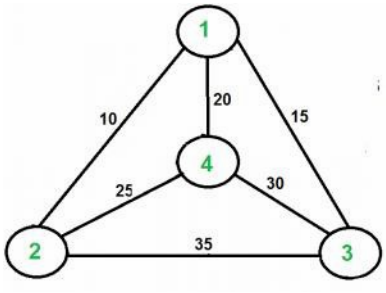
TSP is an NP-hard problem, meaning that, for larger values of n, it is not feasible to evaluate every possible problem solution within a reasonable period of time.

Consequently, TSPs are well suited to solving using randomized optimization algorithms.

Traveling Salesman Problem (TSP) Implementation

Travelling Salesman Problem (TSP) : Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



Explanation

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.

3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

SOURCE CODE

```
# Python3 program to implement traveling salesman problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        # store current Path weight(cost)
        current_pathweight = 0
        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        # update minimum
        min_path = min(min_path, current_pathweight)
    return min_path
if__name__== "__main__":
```

```
# matrix representation of graph
graph = [[0, 10, 15, 20], [10, 0, 35, 25],
        [15, 35, 0, 30], [20, 25, 30, 0]]
s = 0
print(travellingSalesmanProblem(graph, s))
```

OUTPUT

80

3. Write a program to implement Simulated Annealing Algorithm

SOURCE CODE

```
# convex unimodal optimization function
from numpy import arange
from matplotlib import pyplot

# objective function
def objective(x):
        return  x[0]**2.0
# define range for input
r_min, r_max = -5.0, 5.0
# sample input range uniformly at 0.1 increments
inputs = arange(r_min, r_max, 0.1)
# compute targets
results = [objective([x]) for x in inputs]
# create a line plot of input vs result
pyplot.plot(inputs, results)
# define optimal input value
x_optima = 0.0
# draw a vertical line at the optimal input
pyplot.axvline(x=x_optima, ls='--', color='red')
# show the plot
```

```
pyplot.show()
```

SOURCE CODE

```
# explore temperature vs algorithm iteration for simulated annealing
from matplotlib import pyplot
# total iterations of algorithm
iterations = 100
# initial temperature
initial_temp = 10
# array of iterations from 0 to iterations - 1
iterations = [i for i in range(iterations)]
# temperatures for each iterations
temperatures = [initial_temp/float(i + 1) for i in iterations]
# plot iterations vs temperatures
pyplot.plot(iterations, temperatures)
pyplot.xlabel('Iteration')
pyplot.ylabel('Temperature')
pyplot.show()
```

SOURCE CODE

```
# explore metropolis acceptance criterion for simulated annealing
from math import exp
from matplotlib import pyplot
# total iterations of algorithm
iterations = 100
# initial temperature
initial_temp = 10
# array of iterations from 0 to iterations - 1
iterations = [i for i in range(iterations)]
# temperatures for each iterations
```

```
temperatures = [initial_temp/float(i + 1) for i in iterations]
# metropolis acceptance criterion
differences = [0.01, 0.1, 1.0]
for d in differences:
        metropolis = [exp(-d/t) for t in temperatures]
        # plot iterations vs metropolis
        label = 'diff=%.2f' % d
        pyplot.plot(iterations, metropolis, label=label)
# inalize plot
pyplot.xlabel('Iteration')
pyplot.ylabel('Metropolis Criterion')
pyplot.legend()
pyplot.show()
```

4. Write a program to find the solution for wampus world problem

SOURCE CODE

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']
# Data generation
train_num = 200
test_num = 100
config = {
   'Corn': [[150, 190], [40, 700], [20,4]],
   'Potato': [[300, 600], [70, 10], [10, 20]],
   'grass': [[100, 40], [10, 40], [505, 1]]
}
plants = list(config.keys())
dataset = pd.DataFrame(columns=['height(cm)', 'Leaf length(cm)', 'Stem diameter(cm)', 'type'])
index = 0
```

```
# Natural
for p in config:
    for i in range(int(train_num/3-3)):
        row = []
        for  j, [min_val, max_val] in enumerate(config[p]):
            v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
            while v in dataset[dataset.columns[j]]:
                v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
            row.append(v)
        row.append(p)
        dataset.loc[index] = row
        index += 1
# Wrong data
for i in range(train_num - index):
    k = np.random.randint(3)
    p = plants[k]
    row = []
    for  j, [min_val, max_val] in enumerate(config[p]):
        v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
        while v in dataset[dataset.columns[j]]:
            v = round(np.random.rand()*(max_val-min_val)+min_val, 2)
        row.append(v)
    row.append(plants[(k+1)%3])
    dataset.loc[index] = row
    index+=1
# dataset = dataset.infer_objects()
dataset = dataset.reindex(np.random.permutation(len(dataset)))
dataset.reset_index(drop=True, inplace=True)
dataset.iloc[:int(train_num), :-1].to_csv('potato_train_data.csv', index=False)
dataset.iloc[:int(train_num):, [-1]].to_csv('potato_train_label.csv', index=False)
"""Here, only the training data set is generated, and the test data is similar to this
```

Data visualization

We can see the distribution of data points by drawing a scatter diagram of the data of two dimensions."""

```
def visualize(dataset, labels, features, classes, fig_size=(10, 10), layout=None):
    plt.figure(figsize=fig_size)
    index = 1
    if layout == None:
        layout = [len(features), 1]
    for i in range(len(features)):
        for j in range(i+1, len(features)):
            p = plt.subplot(layout[0], layout[1], index)
            plt.subplots_adjust(hspace=0.4)
            p.set_title(features[i]+'&'+features[j])
            p.set_xlabel(features[i])
            p.set_ylabel(features[j])
            for k in range(len(classes)):
                p.scatter(dataset[labels==k, i], dataset[labels==k, j], label=classes[k])
            p.legend()
            index += 1
    plt.show()

dataset = pd.read_csv('potato_train_data.csv')
labels = pd.read_csv('potato_train_label.csv')
features = list(dataset.keys())
classes = np.array(['Corn', 'Potato', 'grass'])
for i in range(3):
    labels.loc[labels['type']==classes[i], 'type'] = i
dataset = dataset.values
labels = labels['type'].values
visualize(dataset, labels, features, classes)
```

5. Write a program to implement 8 puzzle problem

SOURCE CODE

```
import copy
from heapq import heappush, heappop
n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, k):
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

class node:

    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):
        self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
    def __lt__(self, nxt):
        return self.cost < nxt.cost
def calculateCost(mat, final) -> int:
    count = 0
```

```
        for i in range(n):
                for j in range(n):
                        if ((mat[i][j]) and
                                (mat[i][j] != final[i][j])):
                                count += 1
        return count
def newNode(mat, empty_tile_pos, new_empty_tile_pos,
                        level, parent, final) -> node:
        new_mat = copy.deepcopy(mat)
        x1 = empty_tile_pos[0]
        y1 = empty_tile_pos[1]
        x2 = new_empty_tile_pos[0]
        y2 = new_empty_tile_pos[1]
        new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
        cost = calculateCost(new_mat, final)
        new_node = node(parent, new_mat, new_empty_tile_pos,
                                        cost, level)
        return new_node
def printMatrix(mat):
        for i in range(n):
                for j in range(n):
                        print("%d " % (mat[i][j]), end = " ")
                print()
def isSafe(x, y):
        return x >= 0 and x < n and y >= 0 and y < n
def printPath(root):
        if root == None:
                return
        printPath(root.parent)
        printMatrix(root.mat)
        print()
```

```
def solve(initial, empty_tile_pos, final):
        pq = priorityQueue()
        cost = calculateCost(initial, final)
        root = node(None, initial,
                                empty_tile_pos, cost, 0)
        pq.push(root)
        while not pq.empty():
                minimum = pq.pop()
                if minimum.cost == 0:
                        printPath(minimum)
                        return
                for i in range(n):
                        new_tile_pos = [
                                minimum.empty_tile_pos[0] + row[i],
                                minimum.empty_tile_pos[1] + col[i], ]
                        if isSafe(new_tile_pos[0], new_tile_pos[1]):
                                child = newNode(minimum.mat,
                                                        minimum.empty_tile_pos,
                                                        new_tile_pos,
                                                        minimum.level + 1,
                                                        minimum, final,)
                                pq.push(child)
initial = [ [ 1, 2, 3 ],
                        [ 5, 6, 0 ],
                        [ 7, 8, 4 ] ]
final = [ [ 1, 2, 3 ],
                [ 5, 8, 6 ],
                [ 0, 7, 4 ] ]
empty_tile_pos = [ 1, 2 ]
solve(initial, empty_tile_pos, final)
```

OUTPUT:

```
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4
```

6. Write a program to implement Towers of Hanoi problem

SOURCE CODE

```
class Tower:
  def__init_(self):
    self.terminate = 1
  def printMove(self, source, destination):
    print("{} -> {}".format(source, destination))
  def move(self, disc, source, destination, auxiliary):
    if disc == self.terminate:
      self.printMove(source, destination)
    else:
      self.move(disc - 1, source, auxiliary, destination)
      self.move(1, source, destination, auxiliary)
      self.move(disc - 1, auxiliary, destination, source)
t = Tower();
t.move(3, 'A', 'B', 'C')
```

OUTPUT

```
A -> B
A -> C
B -> C
```

A -> B

C -> A

C -> B

A -> B

7. Write a program to implement A* Algorithm

SOURCE CODE

```
from queue import PriorityQueue

#Creating Base Class
class State(object):
    def__init_(self, value, parent, start = 0, goal = 0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.start = parent.start
            self.goal = parent.goal
            self.path = parent.path[:]
            self.path.append(value)

        else:
            self.path = [value]
            self.start = start
            self.goal = goal

    def GetDistance(self):
        pass
    def CreateChildren(self):
        pass
 # Creating subclass
```

```
class State_String(State):
    def__init_(self, value, parent, start = 0, goal = 0 ):
        super(State_String, self)._init_(value, parent, start, goal)
        self.dist = self.GetDistance()

    def GetDistance(self):
            if self.value == self.goal:
                return 0
            dist = 0
            for i in range(len(self.goal)):
                letter = self.goal[i]
                dist += abs(i - self.value.index(letter))
            return dist

    def CreateChildren(self):
            if not self.children:
                for i in range(len(self.goal)-1):
                    val = self.value
                    val = val[:i] + val[i+1] + val[i] + val[i+2:]
                    child = State_String(val, self)
                    self.children.append(child)

# Creating a class that hold the final magic
class A_Star_Solver:
    def__init_(self, start, goal):
        self.path = []
        self.vistedQueue =[]
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal
```

```
    def Solve(self):
        startState = State_String(self.start,0,self.start,self.goal)

        count = 0
        self.priorityQueue.put((0,count, startState))
        while(not self.path and self.priorityQueue.qsize()):
            closesetChild = self.priorityQueue.get()[2]
            closesetChild.CreateChildren()
            self.vistedQueue.append(closesetChild.value)
            for child in closesetChild.children:
                if child.value not in self.vistedQueue:
                    count += 1
                    if not child.dist:
                        self.path = child.path
                        break
                    self.priorityQueue.put((child.dist,count,child))
        if not self.path:
            print("Goal Of  is not possible !" + self.goal )
        return self.path

# Calling all the existing stuffs
if__name__== "_main_":
    start1 = "BHANU"
    goal1 = "NHUBA"
    print("Starting.  ")
    a = A_Star_Solver(start1,goal1)
    a.Solve()
    for      i      in      range(len(a.path)):
        print("{0}){1}".format(i,a.path[i]))
```

8. Write a program to implement Hill Climbing Algorithm

SOURCE CODE

```
# hill climbing search of the ackley objective function
from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function
def objective(v):
	x, y = v
	return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi * x) + cos(2 * pi * y))) + e + 20

# check if a point is within the bounds of the search
def in_bounds(point, bounds):
	# enumerate all dimensions of the point
	for d in range(len(bounds)):
		# check if out of bounds for this dimension
		if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
			return False
	return True

# hill climbing local search algorithm
def hillclimbing(objective, bounds, n_iterations, step_size):
	# generate an initial point
	solution = None
```

```
        while solution is None or not in_bounds(solution, bounds):
                solution = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
        # evaluate the initial point
        solution_eval = objective(solution)
        # run the hill climb
        for i in range(n_iterations):
                # take a step
                candidate = None
                while candidate is None or not in_bounds(candidate, bounds):
                        candidate = solution + randn(len(bounds)) * step_size
                # evaluate candidate point
                candidte_eval = objective(candidate)
                # check if we should keep the new point
                if candidte_eval <= solution_eval:
                        # store the new point
                        solution, solution_eval = candidate, candidte_eval
                        # report progress
                        print('>%d f(%s) = %.5f' % (i, solution, solution_eval))
        return [solution, solution_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.05
# perform the hill climbing search
best, score = hillclimbing(objective, bounds, n_iterations, step_size)
print('Done!')
```

```
print('f(%s) = %f' % (best, score))
```

9. Build a Chatbot using AWS Lex, Pandora bots.

A **Chatbot Python** is an intelligent piece of software that is capable of communicating and performing actions similar to a human. **Chatbot In Python Project Report** are used a lot in customer interaction, marketing on social network sites and instantly messaging the client. This **Chatbot In Python Tutorial** also includes the downloadable **Python Chatbot Code Download** source code for free.

By the way I have here a simple Live **Chat** System in PHP Free Source Code maybe you are looking for this source code too.

To start creating this **Chatbot In Python Tutorial**, make sure that you have **PyCharm IDE** installed in your computer.

By the way if you are new to python programming and you don't know what would be the the Python IDE to use, I have here a list of Best Python IDE for Windows, Linux, Mac OS that will suit for you.

Steps on how to create a **Chatbot In Python**

Chatbot In Python Tutorial With Source Code

Step 1: Create a project name.

First when you finished installed the **Pycharm IDE** in your computer, open it and then create a "**project name**" after creating a project name click the "**create**" button.

Step 2: Create a python file.

Second after creating a project name, "**right click**" your project name and then click "**new**" after that click the "**python file**".

Step 3: Name your python file.

Third after creating a python file, Name your python file after that click "**enter**".

Step 4: The actual code.

This is the actual coding on how to create **Chatbot In Python**, and you are free to copy this code and download the full source code given below.

SOURCE CODE

```
def send():
    send = "You:"+ e.get()
```

```
    text.insert(END,"\n" + send)
    if(e.get()=='hi'):
        text.insert(END, "\n" + "Bot: hello")
    elif(e.get()=='hello'):
        text.insert(END, "\n" + "Bot: hi")
    elif (e.get() == 'how are you?'):
        text.insert(END, "\n" + "Bot: i'm fine and you?")
    elif (e.get() == "i'm fine too"):
        text.insert(END, "\n" + "Bot: nice to hear that")
    else:
        text.insert(END, "\n" + "Bot: Sorry I didnt get it.")
text = Text(root,bg='light blue')
text.grid(row=0,column=0,columnspan=2)
e = Entry(root,width=80)
send = Button(root,text='Send',bg='blue',width=20,command=send).grid(row=1,column=1)
e.grid(row=1,column=0)
root = Tk()
root.title('IT SOURCCODE SIMPLE CHATBOT')
root.mainloop()
```

10. Build a bot which provides all the information related to your college.

SOURCE CODE

SOURCE CODE

```
def send():
    send = "You:"+ e.get()
    text.insert(END,"\n" + send)
    if(e.get()=='hi'):
        text.insert(END, "\n" + "Bot: hello")
    elif(e.get()=='hello'):
        text.insert(END, "\n" + "Bot: hi")
    elif (e.get() == 'how are you?'):
```

```
        text.insert(END, "\n" + "Bot: i'm fine and you?")
    elif (e.get() == "i'm fine too"):
        text.insert(END, "\n" + "Bot: nice to hear that")
    else:
        text.insert(END, "\n" + "Bot: Sorry I didnt get it.")
text = Text(root,bg='light blue')
text.grid(row=0,column=0,columnspan=2)
e = Entry(root,width=80)
send = Button(root,text='Send',bg='blue',width=20,command=send).grid(row=1,column=1)
e.grid(row=1,column=0)
root = Tk()
root.title('IT SOURCCODE SIMPLE CHATBOT')
root.mainloop()
```

11. Build a virtual assistant for Wikipedia using Wolfram Alpha and Python.

SOURCE CODE

```
import wolframalpha

# Taking input from user
question = input('Question: ')

# App id obtained by the above steps
app_id =('U5HXGG-79KT69H58Q')

# Instance of wolf ram alpha
# client class
client = wolframalpha.Client(app_id)

# Stores the response from
# wolf ram alpha
res = client.query(question)
```

```
# Includes only text from the response
answer = next(res.results).text

print(answer)
```

12. The following is a function that counts the number of times a string occurs in another string:

```
# Count the number of times string s1 is found in string s2
def countsubstring(s1,s2):
count = 0
for i in range(0,len(s2)-len(s1)+1):
if s1 == s2[i:i+len(s1)]:
count += 1
return count
```

For instance, countsubstring('ab','cabalaba') returns 2.

Write a recursive version of the above function. To get the rest of a string (i.e. everything but the first character).

SOURCE CODE

```
# Python3 program to count occurrences of pattern in a text.
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    # Create lps[] that will hold the longest prefix suffix values for pattern
    lps = [None] * M
    j = 0  # index for pat[]

    # Preprocess the pattern (calculate lps[]  array)
    computeLPSArray(pat, M, lps)

    i = 0  # index for txt[]
    res = 0
    next_i = 0

    while (i < N):
        if pat[j] == txt[i]:
```

```
            j = j + 1
            i = i + 1
        if j == M:

            # When we find pattern first time,we iterate again to check if there exists more pattern
            j = lps[j - 1]
            res = res + 1

            # We start i to check for more than once appearance of pattern, we will reset i to previous
start+1
            if lps[j] != 0:
                next_i = next_i + 1
                i = next_i
                j = 0



        # Mismatch after j matches
        elif ((i < N) and (pat[j] != txt[i])):

            # Do not match lps[0..lps[j-1]] characters, they will match anyway
            if (j != 0):
                j = lps[j - 1]
            else:
                i = i + 1

    return res

def computeLPSArray(pat, M, lps):

    # Length of the previous longest
    # prefix suffix
    len = 0
    i = 1
    lps[0] = 0  # lps[0] is always 0

    # The loop calculates lps[i] for
    # i = 1 to M-1
    while (i < M):
        if pat[i] == pat[len]:
            len = len + 1
            lps[i] = len
            i = i + 1

        else:  # (pat[i] != pat[len])

            # This is tricky. Consider the example.
            # AAACAAAA and i = 7. The idea is similar
```

```
            # to search step.
            if len != 0:
                len = lps[len - 1]

                # Also, note that we do not increment
                # i here

            else:  # if (len == 0)
                lps[i] = len
                i = i + 1

# Driver code
if__name__== "__main__":

    txt = "WELCOME TO PYTHON WORLD PYTHON PYTHON"
    pat = "THON"
    ans = KMPSearch(pat, txt)

    print(ans)
```

13. Higher order functions. Write a higher-order function count that counts the number of elements in a list that satisfy a given test. For instance: count(lambda x: x>2, [1,2,3,4,5]) should return 3, as there are three elements in the list larger than 2. Solve this task without using any existing higher-order function.

SOURCE CODE

```
# Python3 program to count occurrences of an element if x is present in arr[] thenreturns the
count of occurrencesof x, otherwise returns -1.
def count(arr, x, n):


    # get the index of first occurrence of x
    i = first(arr, 0, n-1, x, n)

    # If x doesn't exist in arr[] then return -1
    if i == -1:
        return i
```

```
    # Else get the index of last occurrence of x. Note that we are only looking in the subarray after first occurrence
    j = last(arr, i, n-1, x, n);

    # return count
    return j-i+1;

# if x is present in arr[] then return the index of FIRST occurrence of x in arr[0..n-1], otherwise returns -1
def first(arr, low, high, x, n):
    if high >= low:

        # low + (high - low)/2
        mid = (low + high)//2

        if (mid == 0 or x > arr[mid-1]) and arr[mid] == x:
            return mid
        elif x > arr[mid]:
            return first(arr, (mid + 1), high, x, n)
        else:
            return first(arr, low, (mid -1), x, n)
    return -1;

# if x is present in arr[] then return the index of LAST occurrence of x in arr[0..n-1], otherwise returns -1
def last(arr, low, high, x, n):
    if high >= low:

        # low + (high - low)/2
        mid = (low + high)//2;
```

```
    if(mid == n-1 or x < arr[mid+1]) and arr[mid] == x :
        return mid
    elif x < arr[mid]:
        return last(arr, low, (mid -1), x, n)
    else:
        return last(arr, (mid + 1), high, x, n)
  return -1

# driver program to test above functions
arr = [1, 2, 2, 3, 3, 3, 3]
x = 3  # Element to be counted in arr[]
n = len(arr)
c = count(arr, x, n)
print ("%d occurs %d times "%(x, c))
```

OUTPUT

3 occurs 4 times

14. Brute force solution to the Knapsack problem. Write a function that allows you to generate random problem instances for the knapsack program. This function should generate a list of items containing N items that each have a unique name, a random size in the range 1 ..... 5 and a random value in the range 1 10.

Next, you should perform performance measurements to see how long the given knapsack solver take to solve different problem sizes. You should peform atleast 10 runs with different randomly generated problem instances for the problem sizes 10,12,14,16,18,20 and 22. Use a

backpack size of 2:5 x N for each value problem size N. Please note that the method used to generate random numbers can also affect performance, since different distributions of values can make the initial conditions of the problem slightly more or less demanding. How much longer time does it take to run this program when we increase the number of items? Does the backpack size affect the answer?

Try running the above tests again with a backpack size of 1 x N and with 4:0 x N.

SOURCE CODE

```
# Python3 program to solve fractional Knapsack Problem

class ItemValue:

    """Item Value DataClass"""

    def __init_(self, wt, val, ind):
        self.wt = wt
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost

# Greedy Approach

class FractionalKnapSack:

    """Time Complexity O(n log n)"""
    @staticmethod
    def getMaxValue(wt, val, capacity):
        """function to get maximum value """
        iVal = []
        for i in range(len(wt)):
            iVal.append(ItemValue(wt[i], val[i], i))

        # sorting items by value
        iVal.sort(reverse=True)

        totalValue = 0
```

```
        for i in iVal:
            curWt = int(i.wt)
            curVal = int(i.val)
            if capacity - curWt >= 0:
                capacity -= curWt
                totalValue += curVal
            else:
                fraction = capacity / curWt
                totalValue += curVal * fraction
                capacity = int(capacity - (curWt * fraction))
                break
        return totalValue


# Driver Code
if__name__== "_main_":
    wt = [10, 40, 20, 30]
    val = [60, 40, 100, 120]
    capacity = 50

    # Function call
    maxValue = FractionalKnapSack.getMaxValue(wt, val, capacity)
    print("Maximum value in Knapsack =", maxValue)
```

OUTPUT:-

Maximum value in Knapsack = 240.0

15. Assume that you are organising a party for N people and have been given a list L of people who, for social reasons, should not sit at the same table. Furthermore, assume that you have C tables (that are infinitly large).

Write a function layout(N,C,L) that can give a table placement (ie. a number from 0 : : :C - 1) for each guest such that there will be no social mishaps.

For simplicity we assume that you have a unique number 0 N-1 for each guest and that the list of restrictions is of the form [(X,Y),] denoting guests X, Y that are not allowed to sit together. Answer with a dictionary mapping each guest into a table assignment, if there are no possible layouts of the guests you should answer False.

SOURCE CODE

```
# Python3 program to solve fractional Knapsack Problem

class ItemValue:

    """Item Value DataClass"""

    def __init_(self, wt, val, ind):
        self.wt = wt
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost

# Greedy Approach
class FractionalKnapSack:

    """Time Complexity O(n log n)"""
    @staticmethod
    def getMaxValue(wt, val, capacity):
        """function to get maximum value """
        iVal = []
        for i in range(len(wt)):
            iVal.append(ItemValue(wt[i], val[i], i))
```

```
        # sorting items by value
        iVal.sort(reverse=True)

        totalValue = 0
        for i in iVal:
            curWt = int(i.wt)
            curVal = int(i.val)
            if capacity - curWt >= 0:
                capacity -= curWt
                totalValue += curVal
            else:
                fraction = capacity / curWt
                totalValue += curVal * fraction
                capacity = int(capacity - (curWt * fraction))
                break
        return totalValue


# Driver Code
if__name__== "_main_":
    wt = [10, 40, 20, 30]
    val = [60, 40, 100, 120]
    capacity = 50

    # Function call
    maxValue = FractionalKnapSack.getMaxValue(wt, val, capacity)
    print("Maximum value in Knapsack =", maxValue)
```

OUTPUT:-

```
Maximum value in Knapsack = 240.0
```

PART C

Artificial Intelligence Lab

Using Prolog

2 Prolog and programming languages

2.1 Symbol manipulation

AI programs can be written in any decent, sufficiently expressive programming language, including typical imperative programming languages such as C, C++, Java and Pascal. However, many developers prefer to use programming languages such as PROLOG and LISP. Both languages have a basis which cannot be traced back to the early ideas of computing as state change of variables, as holds for the imperative languages, but instead to more abstract, mathematical notions of computing. It is this formal foundation which explains why these two languages are enormously expressive, much more than the imperative languages. As a consequence, it is straightforward to create and manipulate complex data structures in these languages, which is not only convenient but really essential for AI, as AI people tend to solve complicated problems. Although much of what will be said in this introduction also holds for LISP, we will not pay further attention to this latter programming language.

PROLOG is frequently used for the development of interpreters of particular (formal) lan-

guages, such as other programming languages, algebraic or logical languages, and knowledge-representation languages in general. Languages are defined in terms of preserved (key)words,

i.e. symbolic names with a fixed meaning. In Pascal, for instance, the keywords program, begin, end are used. It is rather straightforward to develop a PROLOG program that is able to identify such keywords, and to interpreter these keywords in their correct semantic con- text. Of course, this does not mean that it is not possible to develop such interpreters in imperative languages, but almost all necessary facilities to do so are already included in any PROLOG system and really constitute the core of the language. Compare this to the situation in imperative programming languages, where one needs to resort to preprocessors and class libraries, making it often far from easy to understand what is happening. In other words, the level of abstraction offered by PROLOG is much closer to the actual problem one is trying to solve than to the machine or software environment being used. Using abstraction is preferred by most people when developing a program for an actual problem, as it means: to ignore as

many irrelevant details as possible, and to focus on the problem issues that really matter. When you are sufficiently versed in PROLOG, the languages allows you to develop significant programs with only limited effort.

Although PROLOG and AI are often mentioned in one phrase, this does not mean that PROLOG is only suitable for the development of AI programs. In fact, the language has and is being used as a vehicle for database, programming language, medical informatics and bioinformatics research. And finally, the language is used by many people simply to solve a problem they have to deal with, resulting in systems that from the user's point of view are indistinguishable from other systems, as in particular commercial PROLOG systems offer extra language facilities to develop attractive GUIs.2

2.2 Characteristics of PROLOG

There are a number of reasons why PROLOG is so suitable for the development of advanced software systems:

- Logic programming. PROLOG is a logical language; the meaning of a significant frac-

tion of the PROLOG language can be completely described and understood in terms of the Horn subset of first-order predicate logica (Horn-clause logic). So, this is the mathematical foundation of the language, explaining its expressiveness.

- A single data structure as the foundation of the language. PROLOG offers the term as

the basic data structure to implement any other data structure (lists, arrays, trees, records, queues, etc.). The entire language is tailored to the manipulation of terms.

- Simple syntax. The syntax of PROLOG is much simpler than that of the imperative programming languages. A PROLOG program is actually from a syntactical point of view simply a sequence of terms. For example, the following PROLOG program

```
p(X) :- q(X).
```

corresponds to the term :-(p(X),q(X)). Whereas the definition of the formal syntax of a language such as Java or Pascal requires many pages, PROLOG's syntax can be described in a few lines. Finally, the syntax of data is exactly the same as the syntax of programs!

- Program–data equivalence. Since in PROLOG, programs and data conform to the same syntax, it is straightforward to interpret programs as data of other programs, and also to take data as programs. This feature is of course very handy when developing interpreters for languages.

- Weak typing. Types of variables in PROLOG do not have to be declared explicitly.

Whereas in C or Java, the type of any object (for example int, real, char) needs to be known before the compiler is able to compile the program, the type of a variable

in PROLOG only becomes relevant when particular operations are performed on the

variable. This so-called weak typing of program objects has as a major advantage that program design decisions can be postponed to the very last moment. This eases the

programmer's task. However, weak typing not only comes with advantages, as it may make finding program bugs more difficult. Expert programmers are usually able to cope with these problems, while in addition most PROLOG systems come with tools for the analysis of programs which may be equally useful for finding bugs.

- Incremental program development. Normally, a C or Java program needs to be developed

almost fully before it can be executed. A PROLOG program can be developed and

tested incrementally. Instead of generating a new executable, only the new program fragments need to be interpreted or compiled. It is even possible to modify a program during its execution. This offers the programmer the possibility of incremental program development, a very powerful and flexible approach to program development. It is mostly used in research environments and in the context of prototyping.

- Extensibility. A PROLOG system can be extended and modified. It is even possible to modify the PROLOG language, and to adapt both its syntax and semantics to the needs. For example, although PROLOG is not an object-oriented language, it can be extended quite easily into an object-oriented language, including primitives for inheritance and message passing. Compare this to the task of modifying a Java interpreter into a PROLOG compiler, something which would require an almost complete redesign and re-implementation of the original Java system.

3 Introductory remarks about PROLOG

By means of a number of small PROLOG programs, which can be found in the directory

/home/student/courses/cs3510/prolog, the most important features of the PROLOG lan- guage

will be studied in this practical. The programs are available in the following files:

exercise0.pl (facts, queries and rules) exercise1.pl (lists and recursion) exercise2.pl (backtracking)

exercise3.pl (parameters) exercise4.pl (order of clauses) exercise5.pl (fail and cut) exercise6.pl (finite automaton) exercise7.pl (sorting) exercise8.pl (relational database)

exercise9.pl (natural language processing)

Copy these files to your own directory.

3.1 PROLOG under Windows and Unix

Under Windows SWI-PROLOG is activated and reads in a PROLOG program by clicking on the

program (which should have the .pl extension in order to be recognised). is also available under Windows.

One can quit the system by the command:

halt.

SICSTUS-PROLOG

The SWI-PROLOG system can be activated under Unix by typing in:

pl

The SICSTUS-PROLOG system can be activated under Unix by typing in:

sicstus

A program stored in a file, for example the file exercise0.pl, can be read into the PROLOG

interpreter by typing (assuming you are in the directory where the file resides):

?- [exercise0].

This is called consulting a file; the symbol ?- is the system prompt. If a file has the extension .pl, the extension need not be typed in, as the file name is automatically com- pleted. Hence, normally files with extension .pl indicate PROLOG programs (this convention is unfortunately also used for Perl programs). If one prefers to use another extension than

.pl, the full name of the file in between single quotes must be supplied. For example,

?- [’prog.pro’].

causes the program in the file prog.pro to be loaded. The above also works for the Windows environment.

It is also possible, although not very convenient, to type in a program in the PROLOG

system itself. This is accomplished by ‘consulting’ the file user.

?- [user].

|

The file user represents your keyboard. Its input can be terminated by <cntrl>d. In most cases, it

is better to use an editor, such as emacs or some other editor, as these offer editing possibility far superior to the user interface. There is normally a special PROLOG mode available for emacs users. Note that a program that has been typed in at user is lost when leaving the system.

In the next sections, it is assumed that the meaning of most PROLOG predicates is known. The handout 'Introduction to PROLOG' and any book about PROLOG contain descriptions of the standard predicates.

3.2 PROLOG programs

A PROLOG program consists of a sequence of clauses; a PROLOG clause is either:

- a fact,
- a query or goal, or
- a rule.

Either or all of these may be empty (so, the simplest PROLOG program is nothing!). Let us see what response we get from PROLOG when trying to execute the empty program:

knuth-1 > pl

Welcome to SWI-Prolog (Version 3.2.9)

Copyright (c) 1993-1999 University of Amsterdam. All rights reserved. For help, use ?- help(Topic). or ?- apropos(Word).

?- nothing.

[WARNING: Undefined predicate: 'nothing/0'] No

?-

The PROLOG prompt ?- basically asks the user to type in a query. What is typed in, is

shown in italised, bold face, and includes the terminating dot. PROLOG issues a warning, as

it does not recogise the query ?- nothing. However, this warning is not part of PROLOG, but a feature of SWI-PROLOG to help in program debugging. The response of PROLOG to this empty program is No, which means that this program has failed.3

Above, we have only employed the declarative terminology of PROLOG, which views PRO-LOG as a logical language. There also exists a procedural terminology, where facts and rules are called procedure (entries), and queries or goals are called (procedure) calls. In addition, we speak of the head and the body of a clause, which again is terminology that derives from the procedural interpretation of PROLOG. Hence, using this terminology, a PROLOG clause is defined as:

clause ::= head :- body.

where body is defined as:

body ::= disjunction,. . .,disjunction

disjunction ::= condition | (condition; · · · ;condition)

However, even when using the procedural terminology, it is difficult to ignore the logical basis of PROLOG, as the colon ',' has the meaning of conjunction ∧ (AND), and the semicolon ';' has the meaning of disjunction ∨ (OR). We will use both terminologies in this practical to conform to the

practice in the PROLOG and logic programming community.

4 PROLOG exercises

It is assumed you have copied the files to your directory as indicated in Section 3.

4.1 Meet The Dutch Royal Family: facts, queries and rules

Consider the following facts concerning the Dutch Royal Family. This family is so large, that I declare myself unable to unravel the details of the relationships among the members of the family, hence only the most obvious facts have been represented (but you are invited to exend the program!):

3Deep Philosophers will note that we have not really studied the effects of the empty program, as the tested program actually consisted of a single clause: the query ?- nothing. When we type in nothing at all we get the real response to the empty program from PROLOG, which is ?-. So, this is the real nothing!

```
% The facts about the Dutch Royal Family

mother(wilhelmina,juliana). mother(juliana,beatrix). mother(juliana,christina).
mother(juliana,irene). mother(juliana,margriet). mother(beatrix,friso). mother(beatrix,alexander).
mother(beatrix,constantijn). mother(emma,wilhelmina).

father(hendrik,juliana). father(bernard,beatrix). father(bernard,christina). father(bernard,irene).
father(bernard,margriet). father(claus,friso). father(claus,alexander). father(claus,constantijn).
father(willem,wilhelmina).

queen(beatrix). queen(juliana). queen(wilhelmina). queen(emma). king(willem).
```

In these facts, mother and father are called predicates or functors; elements such as beatrix are called constants or atoms (an atom is a constant that is not a number). The meaning of the fact mother(juliana, beatrix) is that Princess Juliana, the previous queen, is the mother of Queen Beatrix, the present queen. The meanings of the other mother and father facts are similar.

Now, if you are either a mother or father of a person, you are also seen as the parent of that person. This is a general rule, which even applies to members of the Royal Family. A rule, however, which uniquely applies to the Royal family is the one which defines kings and queens as rulers. These two principles can be formulated as four PROLOG rules:

```
parent(X, Y) :- mother(X,Y).

parent(X, Y) :- father(X,Y).

ruler(X) :- queen(X).

ruler(X) :- king(X).
```

Note that the bodies of these rules consist of a single condition. X and Y in these rules are variables. They seem similar to atoms, but are required to start with an upper-case letter or underscore. Variables can be bound or instantiated to constants or other variables by a process called matching or unification (see below, Section 4.3).

Surely, one of the most popular questions asked by American tourists visiting the Nether- lands is who predecesses who.4 Now, this can be easily formulated into another two rules:

predecessor(X,Y) :- parent(X,Y),

ruler(X),

ruler(Y). predecessor(X,Y) :-

ruler(X),

parent(X,Z), predecessor(Z,Y).

Note that these rules have multiple conditions constituting the body that must be satisfied in order for a rule to be satisfied. Also note the indentation of the body in comparison to the head; it makes PROLOG programs more readable.

The facts and rules above constitute a

PROLOG program. The clauses of a PROLOG program are loaded into a part of the PROLOG system called the PROLOG database, which is rather confusing terminology, as a PROLOG system is not a database system. It can be invoked by means of queries or goals.

► The complete PROLOG program is included in the file exercise0.pl. Consult this file, and query the program. If the PROLOG system returns with a response, you can enter a semicolon ';', and PROLOG will give an alternative solution. If you simply enter <return > PROLOG stops looking for alternatives. For example, enter the following queries:

?- parent(beatrix,alexander).

?- parent(beatrix,X).

?- parent(beatrix,emma).

?- parent(X,Y), ruler(Y).

?- predecessor(X,beatrix).

Try to understand the answers returned by PROLOG by studying the program. (But be careful not to turn into a royalist!)

4.2 Lists and recursion

The list is one of the most popular data structures in PROLOG. A list is simply a sequence of elements, separated by commas and embraced by brackets:

[e1, e2,. . ., en]

Each element ei, $1 \leq i \leq n$, $n \geq 0$, can be an arbitrary PROLOG term, including a list.

Consider now the following list:

4Another one is how much their possessions are worth.

[a,b,c,d]

It consists of four elements, where each element in this case is an atom (recall that this is a constant symbol that is not a number).

The following list consists of three elements,

[e1,e2,e3]

but is actually defined as

[e1|[e2|[e3|[]]]]

where e1 is called the head of the list, and [e2|[e3|[]]] is called its tail. Thus, as you already suspected, a list is a term of which the functor (from a logical point of view, the functor is now not a predicate, but a function symbol) has two arguments: the head and the tail. A list containing a single element [e] is really the following term: [e|[]]. A simplified notation as [e1,e2,e3] is called syntactic sugar, as it makes dealing with lists easier. (Syntactic sugar makes your programs more tasty!)

Note that the following five lists

[a,b,c]

[a|[b,c]]

[a|[b|[c]]]

[a|[b|[c|[]]]] [a,b|[c]]

are different representations of what essentially is the same list.

The handout 'Introduction to PROLOG' includes an example of a simple PROLOG pro- gram. This program identifies whether a particular element is a member of a set of elements, represented as a list. The following clauses are given (the predicate 'member' was replaced by 'element', as 'member' is already included in most PROLOG systems):

```
/*1*/ element(X,[X|_]).

/*2*/ element(X,[_|Y]) :-

element(X,Y).
```

Examples of queries to this program are:

```
?- element(d,[a,b,c,d,e,f]).
```

Note that the predicate element both occurs in the body of clause 2 and also in its head. Hence, this is an example of a recursive rule. In fact, you have seen recursion before when we considered Dutch royalty.

► The two PROLOG clauses are included in the file exercise1.pl. Consult this file, and query the system. For example, try:

```
?- element(a,[b,a,c]).

?- element(d,[b,a,c]).

?- element(X,[a,b,c,d]).
```

Again, try to understand what is happening.

4.3 Unification and matching

An important mechanism underlying every PROLOG-interpreter is the unification of terms. Unification basically attempts to make two terms equal by substituting appropriate terms for the variables occurring in the two terms. In all cases, the so called most general unifier (mgu) is determined. A term can both be a condition and a head of a clause, but also a structure deeply nested into a condition or head.

To answer a query, the PROLOG interpreter starts with the first condition in the query, taking it as a procedure call. The PROLOG database is subsequently searched for a suitable entry to the called procedure; the search starts with the first clause in the database, and continues until a clause has been found which has a head that can be matched with the procedure call. A match between a head and a procedure call is obtained, if there exists a substitution for the variables occurring both in the head and in the procedure call, such that the two become (syntactically) equal after the substitution has been applied to them. Such a match exists:

- if the head and the procedure call contain the same predicate, and
- if the terms in corresponding argument positions after substitution of the variables are equal; one then also speaks of a match for argument positions.

Applying a substitution to a variable is called instantiating or binding the variable to a term. For example, the query:

?- parent(X,Y).

yields the following instantiations given the program in the file exercise0.pl discussed in Section 4.1:

X = wilhelmina Y = juliana

A user can force unifying two terms by means of the infix predicate (functor) '='. As an example, consider the following two terms:

p(X,a,f(Y,g(b))) and

p(f(b),a,f(c,g(b)))

The following query

?- p(X,a,f(Y,g(b))) = p(f(b),a,f(c,g(b))).

leads to the following substitutions:

X = f(b) Y = c

because:

- the two functors of the two terms are equal (both p);
- to make the first arguments syntactically equal, the term f(b) has to be substituted for the variable X;
- the second arguments are already equal;
- in order to make the third arguments equal, we have to unify the terms f(Y,g(b)) and f(c,g(b)). The functors of these two arguments are already equal, as are the second arguments of the terms. If we substitute the term c for the variable Y, the second arguments will also have become equal.

If two terms have been successfully unified, it is said that a match has been found; otherwise, there is no match.

► Investigate whether the following pairs of terms match, and, if they do, which substitutions do make them equal?

p(f(X,g(b)),Y) and p(f(h(q),Y),g(b))

p(X,a,g(X)) and p(f(b),a,g(b)) [[a]|[b]] and [X,Y]

p(X,f(X)) and p(Y,Y)

It is not possible to find a match for the last pair of terms. Yet, PROLOG is not able to

recognise this fact. First, the matching algorithm substitutes Y for X:

X = Y

The second arguments of p would become equal if

Y = f(X)

This creates a cyclic binding between the variables X and Y; hence, it is not possible to unify these two terms. Many PROLOG interpreters, however, are not able to detect this fact, and get stuck in an infinite loop. What is missing in the implementation of the unification algorithm is known as the occur-check. It is often left out for efficiency reasons. To make a distinction between unification that includes the occur-check, and unification without, the latter is often called matching.

4.4 Backtracking

Backtracking is a mechanism in PROLOG that is offered to systematically search for solutions of a problem specified in terms of PROLOG clauses. A particular PROLOG specification may have more than one solution, and PROLOG normally tries to find one of those. When it is not possible to prove a subgoal given an already partially determined solution to a problem, PROLOG does undo all substitutions which were made to reach this subgoal, and alternative substitutions are tried.

The search space that is examined by the PROLOG system has the form of a tree. More insight into the structure of this space is obtained by a PROLOG program that itself defines a tree data structure:

```
/*1*/ branch(a,b).

/*2*/ branch(a,c).

/*3*/ branch(c,d).

/*4*/ branch(c,e).
```

Figure 1: A binary tree.

```
/*5*/ path(X,X).

/*6*/ path(X,Y) :-

branch(X,Z),

path(Z,Y).
```

► Load the file exercise2.pl into the PROLOG interpreter, and pose the following queries to the

PROLOG database. Check the answers returned by PROLOG.

```
?- path(a,d).
```

Figure 1 shows the tree corresponding to the program. This figure is useful for understanding the execution steps carried out by the PROLOG interpreter. In the next section, we shall consider a number of facilities offered by most PROLOG systems which assist in debugging programs.

4.5 Tracing and spying

PROLOG systems offer a number of facilities to 'debug' a program. Two of those techniques will be considered in this section. They are probably quite useful when working through the remainder of the exercises. A program can be debugged by using:

- the trace predicate, which makes sure that every execution step is shown;
- selective tracing of a program, which is made possible by so-called spy points. This is accomplished by the spy predicate.

By means of an example, we illustrate below how debugging with those two facilities works in practice.

After consulting a program, the trace facility can be switched on by the query:

?- trace.

For example, if after loading the file exercise2.pl as described in Section 4.4 into PROLOG, and after switching on the trace facility, and querying PROLOG:

| ?- trace.

Yes

| ?- path(a,e).

the following is shown:

Call: (7) path(a,e) ?

The question marks indicates that the user may enter an option, to change the behaviour of the interpreter. A list of all possible options is obtained by entering the letter 'h'. Each next execution step is shown by entering <return >:

?- path(a,e).

Call: (7) path(a, e) ? creep

Call: (8) branch(a, _L155) ? creep Exit: (8) branch(a, b) ? creep Call: (8) path(b, e) ? creep

Call: (9) branch(b, _L178) ? creep Fail: (9) branch(b, _L178) ? creep

Fail: (8) path(b, e) ? creep

Redo: (8) branch(a, _L155) ? creep

Exit: (8) branch(a, c) ? creep

Call: (8) path(c, e) ? creep

Call: (9) branch(c, _L167) ? creep

Exit: (9) branch(c, d) ? creep

Call: (9) path(d, e) ? creep

Call: (10) branch(d, _L190) ? creep

Fail: (10) branch(d, _L190) ? creep

Fail: (9) path(d, e) ? creep

Redo: (9) branch(c, _L167) ? creep

Exit: (9) branch(c, e) ? creep

Call: (9) path(e, e) ? creep

Exit: (9) path(e, e) ? creep

Exit: (8) path(c, e) ? creep

Exit: (7) path(a, e) ? creep

Yes

?-

This information can be interpreted as follows. The number between parentheses represents the recursion level of the call (7 as a starting level); Call is followed by the subgoal that has to be proven; Exit indicates that the subgoal has been proven, en Fail indicates that proving the subgoal failed. If the program, as a consequence of a failed subgoal, backtracks to a previously considered subgoal, then this is indicated by Redo. Note that variables are renamed to unique internal names, such as L155.

► The above program is available in the file exercise2.pl. Now, experiment with the trace facility so that you understand it thoroughly.

Switch off debug mode by nodebug.

Selective information of the execution is obtained by means of spy points. A spy point is indicated as follows:

spy(<specification >)

where <specification > is the name of a predicate or a name follows by the arity (number of arguments) of the predicate. For example:

spy(path). spy(branch/2).

A spy point can be removed by the predicate: nospy:

nospy(path). nospy(branch/2).

The structure of the information generated by the PROLOG interpreter in the case of spy points is almost identical to the information that is generated by using trace.

► Experiment with spy points using the tree search algorithm.

4.6 Variation of input-output parameters

An argument of a query can both function as an input and an output parameter, depending on the context of the query. This ensures that a program can be used for sometimes completely different, even opposite purposes, like construction and analysis. Consider the the following program, which is available in the file exercise3.pl:5

conc([],L,L). conc([X|L1],L2,[X|L3]) :-

conc(L1,L2,L3).

This program can be used to concatenate two lists, but, in fact, it can be used as well to split up a list into its parts.

► Check the statement above by consulting the file exercise3.pl,; ask the following questions to the PROLOG interpreter (when the interpreter returns with a response, enter ';' in order to enforce backtracking, so that PROLOG will attempt to find alternative solutions.

?- conc(X,Y,[a,b,c,d,e]).

?- conc([a,b],[c,d],X).

The fact that it is possible to use the same program for such different purposes follows from the fact that PROLOG is essentially a declarative language based on a mathematical (logical) notion of computing.

4.7 The order of clauses and conditions

According to the principle of logic programming, neither the order of PROLOG clauses nor the order of conditions in the bodies of PROLOG clauses really matters. However, as a consequence of its fixed backtracking scheme this is not entirely the case for PROLOG. For example, it is possible that there does exist a solution to a problem according to the logical interpretation of

5The predicate conc is known in PROLOG as append, but as this is often a built-in, we have renamed append to conc, a PROLOG program, whereas PROLOG is not able to find it. As a consequence, any PROLOG programmer requires a good understanding of where and when order in and among clauses matters, and when not.

The program we are going to study in order to get a better understanding of the order issues in PROLOG is available in the file exercise4.pl.

The following facts concern the relation parent(X,Y), which expresses that X is a parent of Y:

parent(pam,bob). parent(tom,bob). parent(tom,liz). parent(bob,ann). parent(bob,pat). parent(pat,jim).

► Pose a number of queries to the program. For example, how do you find out what the names are of Tom's children? And, who are Bob's parents?

The following two clauses recursively define that X is a predecessor of Z, predecessor1(X,Z), if X is a parent of Z or if there is a person Y, such that X is a parent of Y and Y is a predecessor of Z.

/*1*/ predecessor1(X,Z) :-

parent(X,Z).

/*2*/ predecessor1(X,Z) :-

parent(X,Y), predecessor1(Y,Z).

► Query this program, and find out how PROLOG is able to give answers to your queries using

PROLOG's debug facilities.

There are three other declaratively correct specifications of the problem possible, which only differ with respect to order of clauses and conditions. Firstly, the order of the two clauses can be reversed. This results in the following program:

/*3*/ predecessor2(X,Z) :-

```
parent(X,Y), predecessor2(Y,Z).

/*4*/ predecessor2(X,Z) :-

parent(X,Z).
```

► Ask again a number of queries, and compare the results with those obtained for the first program. Can you explain the differences?

It is also possible to interchange the two conditions in the second clause, yielding the following program:

```
/*5*/ predecessor3(X,Z) :-

parent(X,Z).

/*6*/ predecessor3(X,Z) :-

predecessor3(X,Y), parent(Y,Z).
```

► Which queries will this program be unable to answer?

Finally, we can interchange clause 1 as well interchange both conditions in clause 2¿ The following result is then obtained:

```
/*7*/ predecessor4(X,Z) :-

predecessor4(X,Y), parent(Y,Z).

/*8*/ predecessor4(X,Z) :-

parent(X,Z).
```

► Investigate which queries cannot be answered by this program, and try to explain why they cannot.

4.8 Fail and cut

Some of the predicates which PROLOG offers can be used to control PROLOG's execution method in particular ways. Two important ones are the fail and the cut predicate. If the fail predicate is used as a condition, then this condition will always fail, regardless of anything else (hence, of course, the name of this predicate). Usually, the fail predicate is used to enforce backtracking in order to obtain or consider alternative solutions for a problem. Now, consider the program given in the file exercise5.pl:

```
element1(X,[X|_]).

element1(X,[_|Y]) :- element1(X,Y).

all_elements1(L) :- element1(X,L), write(X),

nl, fail.

all_elements1(_).
```

By means of the clauses all elements1, the elements of a list L are printed one by one. Note that the element1 predicate is called in all elements1.

► Experiment with the all elements1 program. Do you understand why there is also a second

all elements1 clause included?

Next, the effects of the cut predicate on program execution are considered. Whereas the fail predicate enforces backtracking, the cut tries to prevent it. So, these two predicates are more or less complementary. Consider the following use of the cut:

element2(X,[X|_]) :- !.

element2(X,[_|Y]) :- element2(X,Y).

all_elements2(L) :- element2(X,L),

write(X), nl,

fail. all_elements2(_).

► Study the difference in behaviour between all elements1 and all elements2, and try to explain this difference.

4.9 A finite automaton

A finite automaton is a mathematical object, which is used to determine whether a particular string of symbols has the right structure or syntax. Finite automata are for example used to build compilers of programming languages, but also to build simulators.

An automaton is always in a one of a given set of states. One of those states is the initial state, another one is the final state. By examining one of the symbols in the given string, the automaton may go to another state, as specified by means of a transition function. If the automaton is in the final state after processing the string, it is said that the string is accepted ; otherwise, it is said that the string has been rejected.

For the automaton that is considered here, the following set is defined:

S = {s1, s2, s3, s4}

Assume that s1 is the initial state and that s3 is the final state. The transition function is defined by means of the following productions:

s1 → as1 s1 → as2 s1 → bs1 s2 → bs3 s3 → bs4 s4 → s3

The last transition is a silent transition: the automaton simply moves to another state without reading a symbol.

The transition function can be easily specified in PROLOG; for example the predicate

transition(s1,c,s2) says that the automaton changes from state s1 to s2 after reading the symbol ‘c’. The condition silent(s1,s2) represents a silent transition. The fact that s3 is a silent transition is indicated by the predicate end. The transition function as defined above can therefore be defined in PROLOG as follows:

end(s3).

transition(s1,a,s1). transition(s1,a,s2). transition(s1,b,s1). transition(s2,b,s3). transition(s3,b,s4).

silent(s4,s3).

The following rules investigate whether a string is either or not accepted:

accept(S,[]) :- end(S).

accept(S,[X|Rest]) :- transition(S,X,S1), accept(S1,Rest).

accept(S,String) :- silent(S,S1), accept(S1,String).

F (P.1) Investigate using this program which input string with 3 and 4 symbols are accepted by the finite automaton. The program is included in the file exercise6.pl. The first argument of accept is the initial state; the second argument is a string, represented as a list of symbols.

4.10 Sorting

As you already know, there are quite a number of different algorithms for sorting elements of a set (or multiset), assuming that a total order is defined on these elements. One of the simplest is the well-known bubble sort algorithm. In the bubble sort, two successive elements in a sequence of (here) numbers are examined. If they are in the right order, the algorithm proceeds to the next pair; otherwise, the two numbers are interchanged, and the algorithm proceeds by considering the next pair. This goes on until there are no elements left which need to be interchanged. Both worst and average case time complexity of the bubble sort is quadratic.

It is quite straightforward to express this algorithm in terms of a PROLOG program.

Assuming that a set of elements is represented as a list, one possible program is:

```
bubsort(L,S) :-
conc(X,[A,B|Y],L), B < A, conc(X,[B,A|Y],M),
!,
bubsort(M,S).
bubsort(L,L).
conc([],L,L). conc([H|T],L,[H|V]) :-
conc(T,L,V).
```

► Investigate the workings of this program; it is available in file exercise7.pl.

4.11 A relational database in PROLOG

Basically, PROLOG programs are about defining relation among objects in a domain. As

storing data as instances of relations is the essential feature, you are probably not surprised to learn the PROLOG is very good in performing the sort of operations on data, which many people normally associate with a relational database management system.

For example, consider salary(scale,amount) which is the sort of relation defined in a relational data model. Recall that scale and amount are referred to as attributes, whereas specific instances of this relation are called tuples. A relational database contains tuples defined according to the data model. Retrieving data stored in the database according to particular constraints on the attributes is called querying the database.

The file exercise8.pl includes an example of a database, with the following data model (Note that the data model is really implicit in this case, which is because we simply make use of standard PROLOG facilities without implementing anything extra):

- employee(name,department number ,scale)
- department(department number,name department)
- salary(scale,amount)

The database (PROLOG database in this case) included the following tuples (facts), again included in exercise8.pl:

employee(mcardon,1,5). employee(treeman,2,3). employee(chapman,1,2).
employee(claessen,4,1). employee(petersen,5,8). employee(cohn,1,7). employee(duffy,1,9).

department(1,board). department(2,human_resources). department(3,production).
department(4,technical_services). department(5,administration).

salary(1,1000). salary(2,1500). salary(3,2000). salary(4,2500). salary(5,3000). salary(6,3500).
salary(7,4000). salary(8,4500). salary(9,5000).

Relational database theory offers a number of useful operations which can be carried out to extract information from a given database. The result of these operations is a new relation,

i.e. a new set of tuples. Typical examples of such database operations are selection, projec- tion and the join. Below we shall discuss simple PROLOG implementations of the selection, projection and join, just to give you an impression of how such programs may look like. The implementation of a more complete relational database package would, of course, be more involving than would be realistic for this practical. We will make use of an SQL-like notation (as you know, SQL is the most popular query language in relational databases).

The purpose of a selection is to request tuples which fulfil certain conditions with respect to their attribute values. Consider, for example, the query: 'Select all employees from de- partment 1 with a salary higher than scale 2.' In SQL-like syntax, this could be stated as follows:

SELECT * FROM employee WHERE department_number = 1 AND scale > 2.

Expressed as a PROLOG query, the purpose is to select tuples satisfying the following condi- tion:

?- employee(Name,Department_N,Scale), Department_N = 1,Scale > 2.

► Consult the file exercise8.pl and enter the PROLOG query given above.

The following output is produced:

Name = mcardon Department_N = 1

Scale = 5.

By entering the ';' operator (the logical ∨ connective) is it again possible to find out whether there are any alternative solutions, as PROLOG will then backtrack, and will check whether there are any other employees satisfying the conditions. Note that, in fact, we have not yet produced a functionally complete implementation of the selection operator, as only a single tuple is selected from the database, but we are nearly there. Recall the trick we considered before, where we made use of the fail predicate in order to enforce backtracking. Here we use this trick again. This gives us the following implementation of the selection operator:

selection(X,Y) :- call(X),

call(Y),

write(X), nl,

fail. selection(_,_).

The predicate call generates a subgoal determined by the binding of its variable argument. It is assumed that the first argument of selection represents the relation from which tuples are to be selected; the second argument contains the set of selection conditions which should be satisfied

by the selected tuples. For example, all employees in department 1 with a salary scale higher than 2 are selected by the following query:

?- selection(employee(Name, Department_N, Scale), (Departments_N = 1,Scale > 2)).

► Design a PROLOG query to select those employees who earn between £3,000 and £4,500 per month? Also select all employees who are in salary scale 1, and working in department 1.

The projection operator in relational database theory is used to select some attributes (or columns in the corresponding tables) from a relation. For example, the query 'give for all employees only the name and scale', in SQL-like notation:

SELECT name,scale FROM employee.

However, the PROLOG query does again give rise to the selection of a single tuple:

?- employee(Name,_,Scale).

But, as you already know, by using the ';' operator the other tuples can be selected as well.

► Type in the above-given PROLOG query. Can you explain why we use here the anonymous variable ?

As you may have guessed, the resulting PROLOG implementation of the projection operator is quite similar to the implementation of the selection operator discussed above.

```
projection(X,Y) :- call(X),
write(Y), nl,
fail. projection(_,_).
```

The first argument of projection should be the relation that is being projected; the second argument lists the attributes on which the relationship must be projected. For example, name and scale of employees are obtained as follows:

?- projection(employee(Name, Department_N, Scale), (Name, Scale)).

Note that this is still not a complete implementation of the projection, as the resulting relation may not include doubles. As incorporating this in the program as well would make it quite a bit more complicated, hardly worth the effort for small databases, we leave the program as it is.

► Experiment with the projection program until you fully understand how it works.

It is now quite straightforward to combine selection and projection. This would imply not only specifying the attributes used in projecting a relation, but also giving the conditions on the attributes which need to be satisfied by the selected tuples. Consider, for example, the following query 'Print the name and scale of those employees in department 1, with a salary higher than scale 2'. In SQL-like notation:

SELECT name,scale FROM employee WHERE department_number = 1 AND scale > 2.

► Define yourself the predicate sel pro which is meant to combine selection and projection.

The final database operation which we will consider is the join. The join simply merges tuples in relations based on a join condition. Suppose that one would like to obtain a list of employees, with for every employee the salary included. For this purpose, we need information from two different relations: employee and salary. For every employee we need to find the salary corresponding to the salary scale. Hence, the join condition in this case is equality of tuples in the two relations concerning the attribute scale. In SQL-like notation:

JOIN employee WITH salary WHERE employee.scale = salary.scale

The following two clauses offer an implementation of the join operator:

```
join(X,Y,Z) :-
call(X),
call(Y),
call(Z),
write(X),
write(Y), nl,
fail. join(_,_,_).
```

The first and second arguments represent relations; the third argument is used to specify join conditions. For example:

```
?- join(employee(Name,Department_N,Scale1), salary(Scale2,Amount),
(Scale1 = Scale2)).
```

will yield the requested result.

F (P.2) Express in PROLOG a join between the relations employee and department, and inspect the result.

4.12 Natural language processing

From its inception, PROLOG has been enormously popular with researchers in the field of natural language processing. Due to its declarative nature, the language is both suitable for the development of programs that analyse sentences as well as for language generation. In this case, we only consider a very simplistic example of a program, as natural language processing is a field in its own right, involving many special issues.

The following program is given in the file exercise9.pl:

```
sentence(X) :- noun_phrase(X,Y),
verb_phrase(Y,[]).
noun_phrase([X|Y],Y) :-
noun(X). noun_phrase([X1,X2,X3|Y],Y) :-
article(X1),
 adjective(X2), noun(X3).
verb_phrase([X|Y],Z) :-
verb(X), noun_phrase(Y,Z).
```

► Develop a small dictionary that can be used to generate sentences.

► Why are the predicate noun_phrase and verb_phrase supplied with a second argument? Is this program also capable of generating sentences? We could also have made use of the conc predicate in developing this program (see exercise3.pl). How would this have affected the program?

20 Sample Programs for Self-Paced Learning

1. Family Relationships

- Create a knowledge base representing a family tree. Write queries to determine relationships like sibling, cousin, uncle, etc.

```
% Define family relationships

parent(john, mary).
parent(mary, susan).
parent(mary, tom).
parent(jane, mary).
parent(jane, peter).
parent(peter, ann).

% Define rules for relationships
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
uncle(X, Y) :- brother(X, Z), parent(Z, Y).

% Additional rules
brother(X, Y) :- parent(Z, X), parent(Z, Y), male(X), X \= Y.
sister(X, Y) :- parent(Z, X), parent(Z, Y), female(X), X \= Y.

% Facts about gender
male(john).
male(tom).
male(peter).
female(mary).
female(jane).
female(susan).
female(ann).

% Example Queries
% ?- sibling(susan, tom).
% ?- grandparent(john, susan).
% ?- uncle(peter, susan).
```

2. Animal Classification

- Develop a simple expert system that classifies animals based on their characteristics. Implement rules to infer the type of animal.

```
% Facts about animals
animal(dog).
animal(cat).
animal(tiger).
animal(elephant).

% Facts about characteristics
has_fur(dog).
has_fur(cat).
has_stripes(tiger).
has_trunk(elephant).

% Classification rules
mammal(X) :- animal(X), has_fur(X).
wild_animal(X) :- animal(X), has_stripes(X).
domestic_animal(X) :- animal(X), \+ has_stripes(X), \+ has_trunk(X).

% Example Queries
% ?- mammal(dog).
% ?- wild_animal(tiger).
% ?- domestic_animal(cat).
```

3. Arithmetic Operations

- Write Prolog predicates to perform basic arithmetic operations (addition, subtraction, multiplication, division) and query them.

```
% Arithmetic operations
add(X, Y, Z) :- Z is X + Y.
subtract(X, Y, Z) :- Z is X - Y.
multiply(X, Y, Z) :- Z is X * Y.
divide(X, Y, Z) :- Y \= 0, Z is X / Y.

% Example Queries
% ?- add(2, 3, Z).
% ?- subtract(5, 3, Z).
% ?- multiply(4, 2, Z).
% ?- divide(10, 2, Z).
```

4. Pathfinding in a Graph

- Implement depth-first search (DFS) and breadth-first search (BFS) algorithms to find a path between two nodes in a graph.

```
% Graph edges
edge(a, b).
edge(a, c).
```

```
edge(b, d).
edge(c, e).
edge(d, f).
edge(e, f).

% Depth-first search (DFS)
dfs(Start, Goal, Path) :-
dfs_helper(Start, Goal, [Start], Path).

dfs_helper(Goal, Goal, Path, Path).
dfs_helper(Start, Goal, Visited, Path) :-
edge(Start, Next),
\+ member(Next, Visited),
dfs_helper(Next, Goal, [Next|Visited], Path).

% Breadth-first search (BFS)
bfs(Start, Goal, Path) :-
bfs_helper([[Start]], Goal, Path).

bfs_helper([[Goal|T]|_], Goal, Path) :-
reverse([Goal|T], Path).
bfs_helper([CurrentPath|Paths], Goal, Path) :-
CurrentPath = [CurrentNode|_],
findall([Next,CurrentNode|CurrentPath],
        (edge(CurrentNode, Next), \+ member(Next, CurrentPath)),
        NewPaths),
append(Paths, NewPaths, UpdatedPaths),
bfs_helper(UpdatedPaths, Goal, Path).

% Example Queries
% ?- dfs(a, f, Path).
% ?- bfs(a, f, Path). Create a Prolog program to play Tic-Tac-Toe. Implement the game logic
and create rules to determine the winner.
```

6. Solving Puzzles

- Solve a given puzzle (e.g., the 8-puzzle or Sudoku) using Prolog. Implement the rules and logic to find the solution.

```
% Initial board state
initial_board([
   [_, _, _],
   [_, _, _],
   [_, _, _]
]).

% Display board
display_board(Board) :-
   maplist(writeln, Board).

% Make a move
```

```
make_move(Board, Row, Col, Player, NewBoard) :-
    nth1(Row, Board, RowList),
    nth1(Col, RowList, _),
    replace(Board, Row, Col, Player, NewBoard).

% Replace element in board
replace([Row|Rest], 1, Col, Player, [NewRow|Rest]) :-
    replace_row(Row, Col, Player, NewRow).
replace([Row|Rest], RowNum, Col, Player, [Row|NewRest]) :-
    RowNum > 1,
    RowNum1 is RowNum - 1,
    replace(Rest, RowNum1, Col, Player, NewRest).

% Replace element in row
replace_row([_|Rest], 1, Player, [Player|Rest]).
replace_row([Elem|Rest], Col, Player, [Elem|NewRest]) :-
    Col > 1,
    Col1 is Col - 1,
    replace_row(Rest, Col1, Player, NewRest).

% Check winner
winner(Board, Player) :-
    (row_win(Board, Player) ; col_win(Board, Player) ; diag_win(Board, Player)).

row_win(Board, Player) :-
    member(Row, Board),
    all_equal(Row, Player).

col_win(Board, Player) :-
    transpose(Board, Transposed),
    row_win(Transposed, Player).

diag_win(Board, Player) :-
    diag1(Board, Diag1),
    all_equal(Diag1, Player) ;
    diag2(Board, Diag2),
    all_equal(Diag2, Player).

diag1([[A,_,C],[_,B,_],[C,_,A]], [A,B,C]).
diag2([[_,_,C],[_,B,_],[C,_,_]], [C,B,C]).

% Check if all elements are equal
all_equal([X], X).
all_equal([X,X|Rest], X) :- all_equal([X|Rest], X).

% Transpose matrix
transpose([[]|_], []).
transpose(Matrix, [Row|Rows]) :-
    maplist(nth1(1), Matrix, Row),
    maplist(tl, Matrix, Rest),
    transpose(Rest, Rows).
```

```
tl([_|T], T).

% Example Queries
% ?- initial_board(Board), make_move(Board, 1, 1, x, NewBoard), display_board(NewBoard).
% ?- initial_board(Board), make_move(Board, 1, 1, x, NewBoard), make_move(NewBoard, 1, 2,
o, NewBoard2), display_board(NewBoard2).
```

7. Natural Language Processing

- Write a simple Prolog program to parse and understand basic English sentences. Create grammar rules and a lexicon.

```
% Simple grammar rules
sentence --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun.
verb_phrase --> verb, noun_phrase.

% Lexicon
determiner --> [the].
determiner --> [a].
noun --> [cat].
noun --> [dog].
verb --> [chases].
verb --> [sees].

% Example Queries
% ?- phrase(sentence, [the, cat, chases, a, dog]).
% ?- phrase(sentence, X).
```

8. Blocks World Problem

- Implement the blocks world problem in Prolog. Write rules to move blocks from one configuration to another.

```
% Initial state
initial_state([
    on(a, b),
    on(b, table),
    on(c, table),
    clear(a),
    clear(c)
]).

% Goal state
goal_state([
    on(c, b),
    on(b, a),
    on(a, table)
]).
```

```
% Move rules
move([clear(X), clear(Y), on(X, Z)|Rest], [clear(Z), on(X, Y), clear(X)|Rest]) :-
    \+ member(clear(Y), Rest),
    \+ member(on(Y, _), Rest).

% Plan to reach goal state
plan(State, Goal, Moves) :-
    plan(State, Goal, [State], Moves).

plan(State, Goal, _, []) :-
    subset(Goal, State).
plan(State, Goal, Visited, [Move|Moves]) :-
    move(State, NewState),
    \+ member(NewState, Visited),
    plan(NewState, Goal, [NewState|Visited], Moves).

% Example Queries
% ?- initial_state(State), goal_state(Goal), plan(State, Goal, Moves).
% ?- initial_state(State), goal_state(Goal), plan(State, Goal, Moves), writeln(Moves).
```

9. Family Relations with Inheritance

- Extend the family relationships knowledge base to include inheritance of properties and traits.

```
% Define family relationships
parent(john, mary).
parent(mary, susan).
parent(mary, tom).
parent(jane, mary).
parent(jane, peter).
parent(peter, ann).

% Define rules for relationships
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
uncle(X, Y) :- brother(X, Z), parent(Z, Y).

% Additional rules
brother(X, Y) :- parent(Z, X), parent(Z, Y), male(X), X \= Y.
sister(X, Y) :- parent(Z, X), parent(Z, Y), female(X), X \= Y.

% Facts about gender
male(john).
male(tom).
male(peter).
female(mary).
female(jane).
female(susan).
female(ann).
```

```
% Inheritance
inheritance(X, Y, Z) :- parent(X, Y), inherited_trait(X, Z).

% Example inherited traits
inherited_trait(john, blue_eyes).
inherited_trait(jane, brown_hair).

% Example Queries
% ?- sibling(susan, tom).
% ?- grandparent(john, susan).
% ?- uncle(peter, susan).
% ?- inheritance(john, mary, Trait).
```

10. Prolog Maze Solver

- Develop a Prolog program to solve a maze. Implement rules to navigate through the maze and find the exit.

```
% Define maze edges
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(e, f).
edge(b, g).
edge(g, h).
edge(h, i).
edge(i, j).
edge(j, k).

% Solve maze
solve_maze(Start, Goal, Path) :-
    dfs_maze(Start, Goal, [Start], Path).

dfs_maze(Goal, Goal, Path, Path).
dfs_maze(Start, Goal, Visited, Path) :-
    edge(Start, Next),
    \+ member(Next, Visited),
    dfs_maze(Next, Goal, [Next|Visited], Path).

% Example Queries
% ?- solve_maze(a, k, Path).
% ?- solve_maze(a, f, Path).
```

11. AI Planning

- Write a Prolog program to implement a simple AI planner. Create a set of actions and goals, and find a sequence of actions to achieve the goals.

```
% Define actions
```

```
action(move(b, table, a), [clear(b), clear(a)], [clear(table), clear(b), on(a, table)], [on(a, b)]).
action(move(a, b, table), [clear(a), clear(table)], [clear(b), on(a, b)], [on(a, table), clear(table)]).

% Plan to achieve goals
plan(State, Goals, Plan) :-
    plan(State, Goals, [], Plan).

plan(State, Goals, _, []) :-
    subset(Goals, State).
plan(State, Goals, Visited, [Action|Actions]) :-
    action(Action, Preconds, AddList, DelList),
    subset(Preconds, State),
    subtract(State, DelList, TempState),
    union(AddList, TempState, NewState),
    \+ member(NewState, Visited),
    plan(NewState, Goals, [NewState|Visited], Actions).

% Example Queries
% ?- plan([clear(a), clear(b), on(a, table), clear(table)], [on(a, b)], Plan).
% ?- plan([clear(a), clear(table), on(a, b)], [on(a, table)], Plan).
```

12. Prolog Chess

- Implement a simple version of a chess game in Prolog. Write rules for legal moves and basic game mechanics.

```
% Initial board state
initial_board([
    [r, n, b, q, k, b, n, r],
    [p, p, p, p, p, p, p, p],
    [_, _, _, _, _, _, _, _],
    [_, _, _, _, _, _, _, _],
    [_, _, _, _, _, _, _, _],
    [_, _, _, _, _, _, _, _],
    [P, P, P, P, P, P, P, P],
    [R, N, B, Q, K, B, N, R]
]).

% Legal moves (simplified)
legal_move(Board, [X1, Y1], [X2, Y2]) :-
    nth1(Y1, Board, Row),
    nth1(X1, Row, Piece),
    Piece \= _,
    nth1(Y2, Board, NewRow),
    nth1(X2, NewRow, Target),
    Target = _.

% Example Queries
% ?- initial_board(Board), legal_move(Board, [1, 2], [1, 4]).
% ?- initial_board(Board), legal_move(Board, [5, 7], [5, 5]).
```

13. Expert System for Medical Diagnosis

- Develop an expert system that can diagnose diseases based on symptoms provided by the user.

```
% Symptoms and diseases
symptom(fever, flu).
symptom(cough, flu).
symptom(sore_throat, flu).
symptom(fever, covid).
symptom(cough, covid).
symptom(shortness_of_breath, covid).

% Diagnosis rules
diagnose(Disease) :-
    findall(Symptom, symptom(Symptom, Disease), Symptoms),
    ask_symptoms(Symptoms).

% Ask about symptoms
ask_symptoms([]).
ask_symptoms([Symptom|Symptoms]) :-
    format('Do you have ~w? (yes/no): ', [Symptom]),
    read(yes),
    ask_symptoms(Symptoms).

% Example Queries
% ?- diagnose(flu).
% ?- diagnose(covid).
```

14. Prolog Interpreter for Arithmetic Expressions

- Write a Prolog program to evaluate arithmetic expressions given as lists (e.g., [2, +, 3, *, 4]).

```
% Interpreter for arithmetic expressions
evaluate([X], X).
evaluate([X, +, Y | Rest], Result) :-
    evaluate([Y | Rest], SubResult),
    Result is X + SubResult.
evaluate([X, -, Y | Rest], Result) :-
    evaluate([Y | Rest], SubResult),
    Result is X - SubResult.
evaluate([X, *, Y | Rest], Result) :-
    evaluate([Y | Rest], SubResult),
    Result is X * SubResult.
evaluate([X, /, Y | Rest], Result) :-
    evaluate([Y | Rest], SubResult),
    Result is X / SubResult.
```

```
% Example Queries
% ?- evaluate([2, +, 3, *, 4], Result).
% ?- evaluate([10, /, 2, -, 1], Result).
```

15. Traveling Salesman Problem (TSP)

- Implement a Prolog solution for the Traveling Salesman Problem using a heuristic or exact algorithm.

```
% Cities and distances
distance(a, b, 10).
distance(a, c, 15).
distance(b, c, 20).
distance(b, d, 25).
distance(c, d, 30).

% Find shortest path
tsp(Start, Path, Cost) :-
    findall([Path, Cost], tsp_helper(Start, Path, Cost), Paths),
    sort(2, @=<, Paths, SortedPaths),
    nth1(1, SortedPaths, [Path, Cost]).

tsp_helper(Start, [Start|Rest], Cost) :-
    tsp_visit(Start, [Start], Rest, Cost).

tsp_visit(Start, Visited, [Next|Rest], Cost) :-
    distance(Start, Next, D),
    \+ member(Next, Visited),
    tsp_visit(Next, [Next|Visited], Rest, SubCost),
    Cost is D + SubCost.
tsp_visit(Start, Visited, [], 0) :-
    length(Visited, L),
    L > 1.

% Example Queries
% ?- tsp(a, Path, Cost).
% ?- tsp(b, Path, Cost).
```

16. Logic Puzzles

- Solve classic logic puzzles (e.g., Einstein's puzzle) using Prolog. Implement the rules and infer the solution.

```
% Solve Einstein's Puzzle
solve_puzzle(Solution) :-
    Solution = [house(red, _, _, _, _),
                house(green, _, _, _, _),
                house(blue, _, _, _, _),
                house(yellow, _, _, _, _),
                house(white, _, _, _, _)],
    member(house(_, norwegian, _, _, _), Solution),
```

```
    member(house(blue, _, _, _, _), Solution),
    member(house(_, _, dog, _, _), Solution),
    member(house(_, _, _, tea, _), Solution),
    member(house(_, _, _, _, marlboro), Solution),
    member(house(_, _, bird, _, _), Solution),
    member(house(_, _, _, beer, _), Solution).

% Example Queries
% ?- solve_puzzle(Solution).
% ?- solve_puzzle(Solution), member(house(Color, norwegian, Pet, Drink, Smoke), Solution).
```

17. Scheduling

- Create a Prolog program to solve a scheduling problem, such as scheduling classes for a set of students and professors.

```
% Define classes, professors, and students
class(math).
class(science).
class(history).

professor(john).
professor(mary).
professor(susan).

student(alice).
student(bob).
student(charlie).

% Availability
available(john, [monday, wednesday]).
available(mary, [tuesday, thursday]).
available(susan, [monday, friday]).

% Schedule rules
schedule(Class, Professor, Day) :-
    class(Class),
    professor(Professor),
    available(Professor, Days),
    member(Day, Days).

% Example Queries
% ?- schedule(math, john, Day).
% ?- schedule(history, mary, Day).
```

18. Game of Nim

- Implement the Game of Nim in Prolog. Write the rules and create a strategy for the AI to play against a human.

```
% Initial state
initial_state([3, 4, 5]).

% Move rules
move([A,B,C], [A2,B,C]) :- A > 0, A2 is A - 1.
move([A,B,C], [A,B2,C]) :- B > 0, B2 is B - 1.
move([A,B,C], [A,B,C2]) :- C > 0, C2 is C - 1.

% Win condition
win([0,0,0]).

% Play game
play(State) :- win(State), write('Game Over!'), nl.
play(State) :-
    \+ win(State),
    move(State, NewState),
    write('Move: '), write(NewState), nl,
    play(NewState).

% Example Queries
% ?- initial_state(State), play(State).
% ?- play([2, 3, 4]).
```

19. Genealogy Tree

- Develop a genealogy tree and implement queries to find ancestors, descendants, and relationships.

```
% Define family relationships
parent(john, mary).
parent(mary, susan).
parent(mary, tom).
parent(jane, mary).
parent(jane, peter).
parent(peter, ann).

% Ancestry rules
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

% Descendants
descendant(X, Y) :- ancestor(Y, X).

% Example Queries
% ?- ancestor(john, susan).
% ?- descendant(susan, john).
% ?- ancestor(jane, tom).
```

20. Symbolic Differentiation

- Write a Prolog program to perform symbolic differentiation of mathematical expressions.

```
% Differentiation rules
diff(X, X, 1) :- !.
diff(C, X, 0) :- atomic(C), C \= X.
diff(U+V, X, DU+DV) :- diff(U, X, DU), diff(V, X, DV).
diff(U-V, X, DU-DV) :- diff(U, X, DU), diff(V, X, DV).
diff(U*V, X, DU*V+U*DV) :- diff(U, X, DU), diff(V, X, DV).
diff(U/V, X, (DU*V-U*DV)/(V*V)) :- diff(U, X, DU), diff(V, X, DV).
diff(sin(U), X, cos(U)*DU) :- diff(U, X, DU).
diff(cos(U), X, -sin(U)*DU) :- diff(U, X, DU).
diff(exp(U), X, exp(U)*DU) :- diff(U, X, DU).
diff(log(U), X, DU/U) :- diff(U, X, DU).

% Example Queries
% ?- diff(x*x + 2*x + 1, x, Result).
% ?- diff(sin(x) + cos(x), x, Result).
```