GROUP OF INSTITUTIONS

Meerut Institute of Engineering & Technology

Department of Computer Science & Engineering

Dr. A.P.J. Abdul Kalam Technical University, Lucknow

LAB MANUAL

Subject Code: KCS-353

Subject Name: Discrete Structures & Theory of Logic Lab

INDEX

Vision of the Institute

To be an outstanding institution in the country imparting technical education, providing need based, value based and career based programmes and producing self-reliant, self-sufficient technocrats, capable of meeting new challenges.

Mission of the Institute

To educate young aspirants in various technical fields to fulfill global requirement of human resources by providing sustainable quality education, training and invigorating environment, also molding them into skilled competent and socially responsible citizens who will lead the building of a powerful nation.

Vision of Department

To be an excellent department that imparts value based quality education and uplifts innovative research in the ever-changing field of technology.

Mission of Department

1. To fulfill the requirement of skilled human resources with focus on quality education.
2. To create globally competent and socially responsible technocrats by providing value and need based training.
3. To improve Industry-Institution Interaction and encourage the innovative research activities.

Program Educational Objectives

1. Students will have the successful careers in the field of computer science and allied sectors as an innovative engineer.
2. Students will continue to learn and advance their careers through participation in professional activities, attainment of professional certification and seeking advance studies.
3. Students will be able to demonstrate a commitment to life-long learning.
4. Students will be ready to serve society in any manner and become a responsible and aware citizen.
5. Establishing students in a leadership role in any field.

Program Outcomes

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes

1. Ability to apply and analyze computational concepts in the areas related to algorithms, machine learning, cloud computing, web designing and web services.
2. Ability to apply standard practices and methodologies in software development and project management.
3. Ability to employ fundamental concepts and emerging technologies for innovative research activities, carrier opportunities & zeal for higher studies.

Course Outcomes

CO-1	To implement basic discrete structures algorithms.
CO-2	To analyze algebraic techniques and implement algebraic operations.
CO-3	To implement logical problems like Boolean algebra, poker hand problem and birthday problem.
CO-4	To implement closed formula of recursive sequence.

List of Experiments

Exp. No.	Experiment Name	Course Outcome
1	Implementation basic python program related to Data types, operators. a) Evaluate value of $2x^3 - 6x^2 + 2x - 1$ for $x = 3$ b) Write a Python program to find the roots of a quadratic function $ax^2 + bx + c = 0$, where a, b and c are real numbers and $a \neq 0$	CO1
2	Implementation of decision, Loop in python. a) Write a program to calculate factorial of a number. b) Write a program to calculate sum of first n natural numbers where n is finite. **c)** Write a program for cube sum of first n natural numbers where n is finite.	CO1
3	Implementation of various set operations (union, intersection, difference, symmetric difference, Power set, cardinality).	CO2
4	Write program to perform following operation: a) Is the given relation is reflexive? b) Is the given relation is symmetric? **c)** Is the given relation is Transitive?	CO2
5	Write program to generate recursive sequence of a closed formula and also calculate its value at particular non negative integer recursively for the following: a) Polynomial 2^n b) Fibonacci sequence **c)** Factorial of a number	CO4
6	Write program to: a. Perform + m (addition modulo) and x_m (multiplication modulo)for a particular set. b. Check closure property for + m (addition modulo) and x_m (multiplication modulo) for any set you have assumed. c. Find identity element in any given algebraic system if exist. Find inverse of all elements in a given group if identity element is given	CO2
7	Write program for various number systems: a. Decimal to binary, octal & hexadecimal b. Binary to decimal, octal and hexadecimal c. Octal to decimal, binary and hexadecimal d. Hexadecimal to decimal, binary and octal e. Logic gate simulation AND, OR, NOT, EXOR, NOR	CO3

8	Write program to: a. Implement the following Boolean expression: i. A'B+AB' ii. (AB'+C)+ C'A b. Implement full adder and half adder in python.	CO3
Value added Programs		
9.	Write program to implement Birthday Problem.	CO4
10.	Write program to test: b. Given relation is equivalence or not. c. Given algebraic system is Abelian group or not.	CO2

Lab Incharge

Head (CSE)

Lab 1

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Execute Python Syntax

Python syntax can be executed by writing directly in the Command Line:

```
>>> print("Hello, World!")
Hello, World!
```

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

```
C:\Users\Your Name>python myfile.py
```

Python Indentations

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Python Comments

Comments can be used to explain Python code. Comments can be used to make the code more readable. Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Python Variables

Creating Variables

Variables are containers for storing data values. Unlike other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular type and can even change type after they have been set.

Example

```
x = 4 # x is of type int
x = "Sally" # x is now of type str
print(x)
```

String variables can be declared either by using single or double quotes:

Example

```
x = "John"
# is the same as
x = 'John'
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assign Value to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z
```

Output Variables

The Python `print` statement is often used to output variables. To combine text and a variable, Python uses the `+` character:

Example

```
x = "awesome"
print("Python is " + x)
```

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

For numbers, the + character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

If you try to combine a string and a number, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)
```

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

To verify the type of any object in Python, use the `type()` function:

Example

```
print(type(x))
print(type(y))
print(type(z))
```

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

Integers:

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z)
```

Float

Float or "floating point number" is a number, positive or negative, containing one or more decimals.

Floats:

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example

Floats:

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

Complex

Complex numbers are written with a "j" as the imaginary part:

Example

Complex:

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

Type Conversion

You can convert from one type to another with the int(), float(), and complex() methods:

Example: Convert from one type to another:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)
```

```
#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Random Number

Python does not have a random() function to make a random number, but Python has a built-in module called random that can be used to make random numbers:

Example

Import the random module, and display a random number between 1 and 9:

```
import random

print(random.randrange(1,10))
```

Python Strings

String Literals

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

'hello' 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:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double 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.

Example: Get the character at position 1 (remember that the first character has the position 0):

```
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])
```

Example: The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Example: The len() method returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Example: The lower() method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Example: The upper() method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Example: The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Example: The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

String Format

Example

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using the format() method! The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

Example

Use the format() method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	**Name**	**Example**
+	Addition	x + y

-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3

^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	**Name**	**Example**
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits

<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

LAB 2

Implementation of decision, Loop in python.

a) **Write a program to calculate factorial of a number.**
b) **Write a program to calculate sum of first n natural numbers where n is finite.**
c) **Write a program for cube sum of first n natural numbers where n is finite.**

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops. An "if statement" is written by using the if keyword.

Example

If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation, using whitespace, to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
```

```
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else

The else keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example a is greater to b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
print("A") if a > b else print("=") if a == b else print("B")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, AND if c is greater than a:

```
if a > b and c > a:
  print("Both conditions are True")
```

Or

The or keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, OR if a is greater than c:

```
if a > b or a > c:
  print("At least one of the conditions is True")
```

Python Loops

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
```

```
  continue
print(i)
```

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

The for loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

Example

Exit the loop when x is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):
  print(x)
```

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example Using the start parameter:

```
for x in range(2, 6):
  print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
  print(x)
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

LAB 3

Implementation of various set operations (union, intersection, difference, symmetric difference, Power set, cardinality).

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

Example

Create a Set:

```
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 the in keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Example

Check if "banana" is present in the set:

```
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.

Example

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Example

Add multiple items to a set, using the `update()` method:

```
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.

Example

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

Example

Remove "banana" by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

You can also use the `pop()`, method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the `pop()` method is the removed item.

Example

Remove the last item by using the `pop()` method:

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

Example

The clear() method empties the set:

```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

Example

The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

The set() Constructor

It is also possible to use the set() constructor to make a set.

Example

Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

Set Methods

Python has a set of built-in methods that you can use on sets.

Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets

intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Python Set add() Method

The add() method adds an element to the set. If the element already exists, the add() method does not add the element.

Syntax:

set.add(*elmnt*)

Example

Add an element to the fruits set:

```
fruits = {"apple", "banana", "cherry"}

fruits.add("orange")

print(fruits)
```

Python Set clear() Method

The clear() method removes all elements in a set.

Syntax:

set.clear()

Example

Remove all elements from the fruits set:

```
fruits = {"apple", "banana", "cherry"}

fruits.clear()

print(fruits)
```

Python Set copy() Method

The copy() method copies the set.

Syntax:

set.copy()

Example

Copy the fruits set:

```
fruits = {"apple", "banana", "cherry"}

x = fruits.copy()

print(x)
```

Python Set difference() Method

The difference() method returns a set that contains the difference between two sets.

Meaning: The returned set contains items that exist only in the first set, and not in both sets.

Syntax:

set.difference(*set*)

Example

Return a set that contains the items that only exist in set x, and not in set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.difference(y)

print(z)
```

Python Set difference_update() Method

The difference_update() method removes the items that exist in both sets. The difference_update() method is different from the difference() method, because the difference() method ***returns a new set***, without the unwanted items, and the difference_update() method ***removes*** the unwanted items from the original set.

Syntax: ***set***.difference_update(***set***)

Example

Remove the items that exist in both sets:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.difference_update(y)
print(x)
```

Python Set discard() Method

The discard() method removes the specified item from the set. This method is different from the remove() method, because the remove() method *will raise an error* if the specified item does not exist, and the discard() method *will not*.

Syntax:

set.discard(*value*)

Example

Remove "banana" from the set:

```
fruits = {"apple", "banana", "cherry"}

fruits.discard("banana")

print(fruits)
```

Python Set intersection() Method

The intersection() method returns a set that contains the similarity between two or more sets.

Meaning: The returned set contains only items that exist in both sets, or in all sets if the comparison is done with more than two sets.

Syntax

set.intersection(*set1, set2 ... etc*)

Example

Return a set that contains the items that exist in both set x, and set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z)
```

Example

Compare 3 sets, and return a set with items that is present in all 3 sets:

```
x = {"a", "b", "c"}
y = {"c", "d", "e"}
z = {"f", "g", "c"}

result = x.intersection(y, z)

print(result)
```

Python Set intersection_update() Method

The intersection_update() method removes the items that is not present in both sets (or in all sets if the comparison is done between more than two sets). The intersection_update() method is different from the intersection() method, because the intersection() method *returns a new set*, without the unwanted items, and the intersection_update() method *removes* the unwanted items from the original set.

Syntax

set.intersection_update(*set1, set2 ... etc*)

Example

Remove the items that is not present in both x, and set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.intersection_update(y)

print(x)
```

Python Set isdisjoint() Method

The isdisjoint() method returns True if none of the items are present in both sets, otherwise it returns False.

Syntax

set.isdisjoint(*set*)

Example

Return True if no items in set x is present in set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "facebook"}

z = x.isdisjoint(y)

print(z)
```

Python Set issubset() Method

The issubset() method returns True if all items in the set exists in the specified set, otherwise it retuns False.

Syntax

set.issubset(*set*)

Example

Return True if all items set x are present in set y:

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b", "a"}

z = x.issubset(y)
```

```
print(z)
```

Python Set issuperset() Method

The issuperset() method returns True if all items in the specified set exists in the original set, otherwise it retuns False.

Syntax: ***set***.issuperset(***set***)

Example

Return True if all items set y are present in set x:

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}
z = x.issuperset(y)

print(z)
```

Python Set pop() Method

The pop() method removes a random item from the set. This method returns the removed item.

Syntax

set.pop()

Example

Remove a random item from the set:

```
fruits = {"apple", "banana", "cherry"}

fruits.pop()

print(fruits)
```

Python Set remove() Method

The remove() method removes the specified element from the set. This method is different from the discard() method, because the remove() method *will raise an error* if the specified item does not exist, and the discard() method *will not*.

Syntax

set.remove(*item*)

Example

Remove "banana" from the set:

```
fruits = {"apple", "banana", "cherry"}

fruits.remove("banana")

print(fruits)
```

Python Set symmetric_difference() Method

The symmetric_difference() method returns a set that contains all items from both set, but not the items that are present in both sets.

Meaning: The returned set contains a mix of items that are not present in both sets.

Syntax

set.symmetric_difference(*set*)

Example

Return a set that contains all items from both sets, except items that are present in both sets:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
```

```
z = x.symmetric_difference(y)

print(z)
```

Python Set symmetric_difference_update() Method

The symmetric_difference_update() method updates the original set by removing items that are present in both sets, and inserting the other items.

Syntax

set.symmetric_difference_update(*set*)

Example

Remove the items that are present in both sets, AND insert the items that is not present in both sets:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.symmetric_difference_update(y)

print(x)
```

Python Set union() Method

The union() method returns a set that contains all items from the original set, and all items from the specified sets. You can specify as many sets you want, separated by commas. If an item is present in more than one set, the result will contain only one appearance of this item.

Syntax

set.union(*set1, set2...*)

Example

Return a set that contains all items from both sets, duplicates are excluded:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.union(y)

print(z)
```

Python Set update() Method

The update() method updates the current set, by adding items from another set. If an item is present in both sets, only one appearance of this item will be present in the updated set.

Syntax

set.update(*set*)

Example

Insert the items from set y into set x:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.update(y)

print(x)
```

LAB 4

Write program to perform following operation:

a. **Is the given relation is reflexive?**
b. **Is the given relation is symmetric?**
c. **Is the given relation is Transitive?**

a.
```
def is_reflexive(Set, Relation):
    newSet = {(a, b) for a in Set for b in Set if a == b}
    if Relation >= newSet:
        return True
    return False
```

b.
```
def is_symmetric(Relation):
    if all(tup[::-1] in Relation for tup in Relation):
        return True
    return False
```

c.
```
def is_transitive(relation):
    for a,b in relation:
        for c,d in relation:
            if b == c and ((a,d) not in relation):
                # print (a,b),(c,d) # uncomment for tests...
                return False
    return True
```

LAB 5

Write program to generate recursive sequence of a closed formula and also calculate its value at particular non negative integer recursively for the following:

a. Polynomial 2^n
b. Fibonacci sequence
c. Factorial

Example (a): Calculation of polynomial 2^n

i. Closed form- Let $f(n)=2^n$ |n>=0
ii. Sequence: 1,2,4,8,16…….
iii. Recursive definition: Basic Step- f(0)=1
Recursive step: f(n)=2* f(n-1)

Example (b): Calculation of Fibonacci sequence

i. Closed Form F(n)=
ii. Sequence: 0,1,1,2,3,5……..
iii. Recursive definition: Basic Step- f(0)=0, f(1)=1
Recursive Step: f(n)= f(n-1)+ f(n-2)

Program

(a) $f(n)=2^n$ (Closed form)

Generation of sequence

```
>>> n=input('enter your number')
>>> for j in range(n):
        print(int(math.pow(2,j+1)))
```

Calculate value at particular non-negative integer recursively

```
>>> def power(n):
        if n == 0:
            return 1 else:
                return 2 * power(n - 1)
```

```
>>> power(3)
```

(b) F(n)= (Closed form)

Generation of sequence

```
>>> n=input('enter your number')
>>> for i in range(n):
print(int((math.pow((1+math.sqrt(5)),i)- math.pow((1-math.sqrt(5)),i))/((math.pow(2,i)*math.sqrt(5)))))
```

```
>>> def fibonacci(n):
        if n == 0:
            return 0 else:
                if n==1:
                return 1
            else:
                return fibonacci(n-1)+fibonacci(n-2)

>>> n=input ('enter number of sequence')
>>> for i in range(n):
print(fibonacci(i))
```

(c) F(n)=n!(closed form does not exist)

Generation of sequence:

```
>>>import math
>>> a=input('enter your number')
>>> for i in range(a):
        print(math.factorial(i)) print("\n")
```

Calculate value at particular non-negative integer recursively

```
>>> def factorial(n): if n == 0:
        return 1 else:
            return n * factorial(n - 1)

>>> factorial(input(' enter your number'))
```

LAB 6

Write program to:

a. Perform + m (addition modulo) and x_m (multiplication modulo) for a particular set.
b. Check closure property for + m (addition modulo) and x_m (multiplication modulo) for any set you have assumed.
c. Find identity element in any given algebraic system if exist.
d. Find inverse of all elements in a given group if identity element is given.

a. Addition modulo

```
for i in range(len(a)):
    for j in range(len(a)):
        b=(a[i]+a[j])%6
        print(b)
    print('\n')
```

Multiplication modulo

```
for i in range(len(a)):
    for j in range(len(a)):
        b=(a[i]*a[j])%6
        print(b)
    print('\n')
```

b.
```
def closure(a):
    for i in range(len(a)):
        for j in range(len(a)):
            if (((a[i]*a[j])%6) not in a): return False
        return True
```

c.
```
def identity(a):
    for i in range(len(a)):
        for j in range(len(a)-1):
            if (((a[i]+a[j])==a[j])and a[i]+a[j+1]==a[j+1]): return a[i]
```

```
                return(" identity not exixt")
```

d.
```
# this is a function so return only one inverse if exist
def inverse(a, e): #a is the set and e is the identity element
        for i in range(len(a)):
                for j in range(len(a)):
                        if (((a[i]+a[j])==e)):
                                return a[i], "is inverse of",a[j]
                        return "inverse not exist"
```

```
# this program return all inverse if exist. It print nothing if no inverse exist
def inverse(a,e,k): #a is the set and e is the identity element,k is counter to check for any identity
        for i in range(len(a)):
                for j in range(len(a)):

                        if (((a[i]*a[j])==e)):
                                k=k+1;print a[i], "is inverse of",a[j]


        if(k==0):
                print "inverse not exist"
```

LAB 7

Write program for various number systems:

a. **Decimal to binary, octal & hexadecimal**
b. **Binary to decimal, octal and hexadecimal**
c. **Octal to decimal, binary and hexadecimal**
d. **Hexadecimal to decimal, binary and octal**
e. **Logic gate simulation AND, OR, NOT, EXOR, NOR**

a. Decimal to binary, octal & hexadecimal

```
>>> a=6

>>> bin(a) '0b110'

>>> oct(a) '06'

>>> hex(a) '0x6'
```

b. Binary to decimal, octal and hexadecimal

```
>>> c='0b11010' or

>>> c='11010'

>>> int(c,2) 26

>>> oct(int(c,2)) '032'

>>> hex(int(c,2)) '0x1a'
```

c. Octal to decimal, binary and hexadecimal

```
>>> c='032'

>>> int(c,8) 26

>>> bin(int(c,8)) '0b11010'

>>> hex(int(c,8)) '0x1a'
```

d. Hexadecimal to decimal, binary and octal

```
#  Convert Hexadecimal to Decimal

print("Enter 'x' for exit.");

hexdec = input("Enter number in Hexadecimal Format: ");

if hexdec == 'x':

   exit();

else:

   dec = int(hexdec, 16);

   print(hexdec,"in Decimal =",str(dec))
```

```
# Convert Hexadecimal to Octal
print("Enter 'x' for exit.");
hexdec = input("Enter a number in Hexadecimal Format: ");
if hexdec == 'x':
   exit();
else:
   dec = int(hexdec, 16);
   print(hexdec,"in Octal =",oct(dec));
```

e. Logic gate simulation AND, OR, NOT, EXOR, NOR

```
def AND (a,b):
        if a == 1 and b == 1:
                return 1
        else:
                return 0

def NAND (a,b):
        if a == 1 and b == 1:
                return 0
        else:
                return 1

def OR(a,b):
        if a == 1:
                return 1
        elif b == 1:
                return 1
        else:
                return 0
```

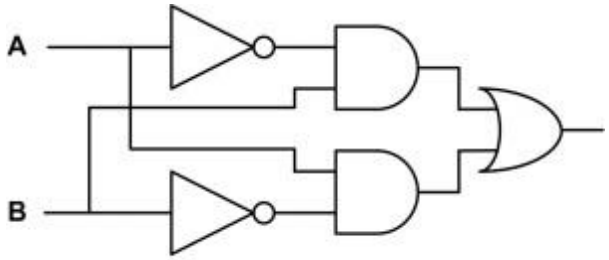
```
def NOR (a,b):
        if a != b:
                return 1
        else:

                return 0
```

LAB 8

Write program to:

a. **Implement the following Boolean expression:**
   i. **A'B+AB'**
   ii. **(AB'+C)+ C'A**
b. **Implement full adder and half adder in python.**

a. Implementation diagram

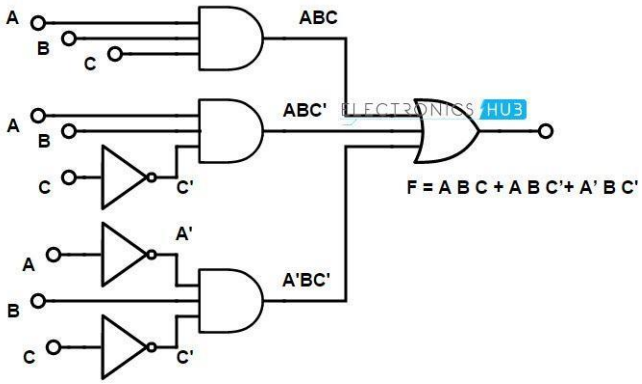


A xor B = A'B + AB'

First execute AND,NOT and OR Then

>>>OR(AND(NOT(a),b),AND(a,NOT(b)))

Implementation diagram

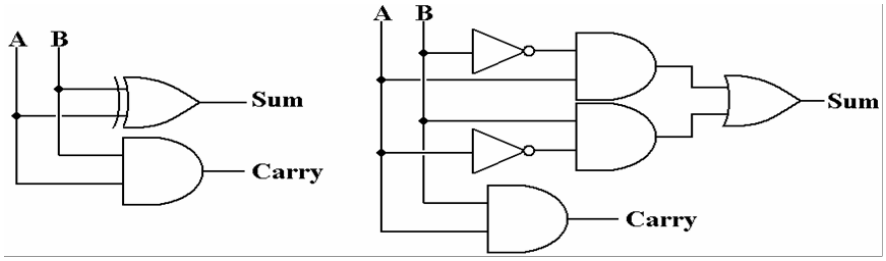



```
>>> def AND (a,b,c):
        if a == 1 and b == 1 and c==1:
                return 1
        else:
```

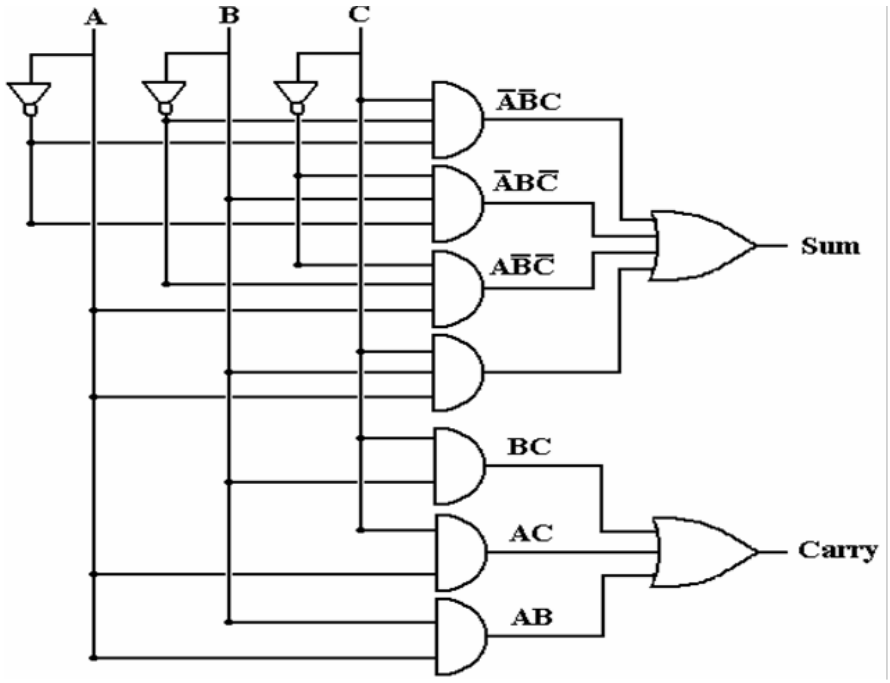
```
        return 0
```

```
>>> def OR(a,b,c):
        if (a == 1 or b==1 or c==1):
                return 1
        else:
                return 0
>>>OR(AND(a,b,c),AND(a,b,NOT(c)),AND(NOT(a),b,NOT(c)))
```

b. Implement Half Adder and Full Adder



SUM = A XOR B = A ⊕B CARRY = A AND B = A.B



LAB 9

Write program to implement Birthday Problem.

In a group of n people some may be born on the same day of the year, and others may be born on different days of the year. We can compute the probability that everybody in the group was born on a different day, and subtract that from 1 to get the probability that at least two people in the group have the same birthday.

With 366 people in a 365-day year, we are 100% sure that at least two have the same birthday, but we only need to be 50% sure. Simulation gives us an elegant way of solving this problem.

```
from random import randint


def equal_birthdays(sharers=2, groupsize=23, rep=100000):
    'Note: 4 sharing common birthday may have 2 dates shared between two people each'
    g = range(groupsize)
    sh = sharers - 1
    eq = sum((groupsize - len(set(randint(1,365) for i in g)) >= sh)
             for j in range(rep))
    return (eq * 100.) / rep


def equal_birthdays(sharers=2, groupsize=23, rep=100000):
    'Note: 4 sharing common birthday must all share same common day'
    g = range(groupsize)
    sh = sharers - 1
    eq = 0
    for j in range(rep):
        group = [randint(1,365) for i in g]
        if (groupsize - len(set(group)) >= sh and
            any( group.count(member) >= sharers for member in set(group))):
            eq += 1
    return (eq * 100.) / rep
```

```
group_est = [2]
for sharers in (2, 3, 4, 5):
   groupsize = group_est[-1]+1
   while equal_birthdays(sharers, groupsize, 100) < 50.:
      # Coarse
      groupsize += 1

 for groupsize in range(int(groupsize - (groupsize - group_est[-1])/4.), groupsize + 999):
      # Finer
      eq = equal_birthdays(sharers, groupsize, 250)
      if eq > 50.:
         break
for groupsize in range(groupsize - 1, groupsize +999):
      # Finest
      eq = equal_birthdays(sharers, groupsize, 50000)
      if eq > 50.:
         break
group_est.append(groupsize)
   print("%i independent people in a group of %s share a common birthday. (%5.1f)" % (sharers, groupsize, eq))
```

Output:

2 independent people in a group of 23 share a common birthday. (50.9)

3 independent people in a group of 87 share a common birthday. (50.0)

4 independent people in a group of 188 share a common birthday. (50.9)

5 independent people in a group of 314 share a common birthday. (50.6)

LAB 10

Write program to test:

a. **Given relation is equivalence or not.**
b. **Given algebraic system is Abelian group or not.**

Equivalence Relation

An equivalence relation on a set X is a subset of $X \times X$, i.e., a collection R of ordered pairs of elements of X, satisfying certain properties. Write "$x \, R \, y$" to mean (x, y) is an element of R, and we say "x is related to y," then the properties are

1. Reflexive: $a \, R \, a$ for all $a \in X$,

2. Symmetric: $a \, R \, b$ implies $b \, R \, a$ for all $a, b \in X$

3. Transitive: $a \, R \, b$ and $b \, R \, c$ imply $a \, R \, c$ for all $a, b, c \in X$,

where these three properties are completely independent. Other notations are often used to indicate a relation, e.g., $a \equiv b$ or $a \sim b$.

Algebraic Structure

A non empty set S is called an algebraic structure w.r.t binary operation (*) if it follows following axioms:

- **Closure:**(a*b) belongs to S for all a,b ∈ S.

Ex : S = {1,-1} is algebraic structure under *
As 1*1 = 1, 1*-1 = -1, -1*-1 = 1 all results belongs to S.

But above is not algebraic structure under + as 1+(-1) = 0 not belongs to S.