Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Continuous Assessment for Laboratory / Assignment sessions

Academic Year 2023 - 2024

Name: Shashwat Shah SAP ID: 60004220126

Course: Programming Laboratory-II (Python) Course Code: DJ19CEL504

Year: T. Y. B. Tech Sem: VI Batch: C 2-2

Department: Computer Engineering

Performance Indicators (Any no. of Indicators) (Maximum 5 marks per indicator)	1	2	3	4	5	6	7	8	9	10
Course Outcome										
1. Knowledge (Factual/Conceptual/Procedural/ Metacognitive)	2	3	2	2	2	3	3	2	2	2
2. Describe (Factual/Conceptual/Procedural/ Metacognitive)	2	2	2	2	2	2	3	2	2	2
3. Demonstration (Factual/Conceptual/Procedural/ Metacognitive)							3			
4. Strategy (Analyse & / or Evaluate) (Factual/Conceptual/ Procedural/Metacognitive)										
5. Interpret/ Develop (Factual/Conceptual/ Procedural/Metacognitive)	2	2	2	2	2	2	3	3	3	3
6. Attitude towards learning (receiving, attending, responding, valuing, organizing, characterization by value)	3	3	2	2	2	2	2	3	3	3
7. Non-verbal communication skills/ Behaviour or Behavioural skills (motor skills, hand-eye coordination, gross body movements, finely coordinated body movements speech behaviours)	3	3	2	2	2	2	2	3	3	3
Total	13	13	12	12	12	12	12	13	13	13
Signature of the faculty member	LK	LK	LK	LK	LK	LK	LK	LK	LK	LK

Outstanding (5), Excellent (4), Good (3), Fair (2), Needs Improvement (1)

Sign of the Student:

Signature of the Faculty member: Lakshmi
Name of the Faculty member: 6/12/23

Signature of Head of the Department
Date:

Experiment 1

Shashwat Shah
60004220126
C 2-2 Div B

Aim : To study and implement different datatypes and operators in python.

Theory: Datatypes in Python.

① Text type = Strings (str)
Strings in python are either surrounded by single or double quotation marks.

② Numeric type = Variables of numeric type are created when you assign value to them.
a) Integer (int) b) Float (float) c) complex (complex)

③ Seaquence type =
a) List ([]) b) Range (range()) c) Tuple (tuple)

④ Mapping type =
a) Dictionary (dict {}) - These items are ordered, changeable and does not allow duplicates.

⑤ Set types
a) Set (set) - used to store multiple items in a single variable

⑥ Boolean type
a) Boolean (bool) - Boolean represents one of 2 values (True or False)

⑦ Binary Type
a) Bytes (bytes) - Used to manipulate binary data in python

⑧ None Type
None (none) - Used to define null value or no value at all.

Operators in python

① Arithmetic Operators - used with numeric values to perform mathematical operations.

a) Addition (+) b) Subtraction (-) c) Multiplication (*) g) floor division(//)
d) Division (/) e) Modulus (%) f) Exponential (**)

② Assignment Operators - used to assign values to variables

a) Assign (=) d) Multiply & Assign (*=) g) Divide (floor) & assign (//=)
b) Add and Assign (+=) e) Divid & Assign (/=)
c) Subtract and Assign (-=) f) Modulus & assign (%=) h) Exponential & assign (**=)

③ Comparison Operator - used to compare two values

a) Equal (==) d) less than (<)
b) Not Equal (!=) e) Greater than or equal to (>=)
c) Greater than (>) f) less then or equal to (<=)

④ Logical Operators - used to combine conditional statements

a] 'and' operator b] 'or' operator c) 'not' operator

⑤ Identity operators :

a) 'is' operator b) 'is not' operator

⑥ Membership Operators

a) 'in' operator b) 'not in' operator

⑦ Bitwise Operators

a) AND (&) b) OR (|) c) XOR (^) d) NOT (~)

Conclusion : We understood the datatypes and operators in python.

15/7/19

Experiment No. 1
Data Types and Operators in Python

Aim:
To study and implement different data types and operators in python.

Description:
I] Data Types in Python:
In programming, data type is an important concept.
Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

1. **Text Type:**
   a. **String (str):** Strings 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: x = "Hello World"
2. **Numeric Type:**
   Variables of numeric types are created when you assign a value to them:
   a. **Integer (int):** Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
   Example: x = 20
   b. **Float (float):** Float, or "floating point number" is a number, positive or negative, containing one or more decimals. Float can also be scientific numbers with an "e" to indicate the power of 10.
   Example: x = 20.5
   c. **Complex (complex):** Complex numbers are written with a "j" as the imaginary part.
   Example: x = 12 + 67j
3. **Sequence Type:**
   a. **List (list):** Lists are used to store multiple items in a single variable.
   Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
   Lists are created using square brackets.
   List items are ordered, changeable, and allow duplicate values.
   List items are indexed, the first item has index [0], the second item has index [1] etc.
   Example: x = ["apple", "banana", "cherry", 12, 12.0, 44j]

b. **Range (range ()):** The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.
Example: x = range(6)

c. **Tuple (tuple):** Tuples are used to store multiple items in a single variable.
Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
A tuple is a collection which is ordered and unchangeable.
Tuples are written with round brackets.
Tuple items are ordered, unchangeable, and allow duplicate values.
Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
Example: x = ("apple", "banana", "cherry", 12, 4.3, 7j)

4. **Mapping Type:**
   a. **Dictionary (dict):** Dictionaries are used to store data values in key:value pairs.
   A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
   Dictionary items are ordered, changeable, and does not allow duplicates.
   Dictionary items are presented in key:value pairs, and can be referred to by using the key name.
   Keys can be accessed using the keys() method and values can be accessed using the values() method.
   Example: x = {"name" : "John", "age" : 36}
5. **Set Type:**
   a. **Set (set):** Sets are used to store multiple items in a single variable.
   Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
   A set is a collection which is unordered, unchangeable*, and unindexed.
   Set items are unordered, unchangeable, and do not allow duplicate values.
   Example: x = {"apple", "banana", "cherry"}
6. **Boolean Type:**
   a. **Boolean (bool):** Booleans represent one of two values: True or False.
   Example: x = True, x = False
7. **Binary Type:**
   a. **Bytes (bytes):** The byte data type is used to manipulate binary data in python. Bytes is supported by buffer protocol, named memoryview. The memoryview can access the memory of other binary object without copying the actual data.
   The byte literals can be formed by these options.
   Example: x = b"Hello"

8. **None Type:**
    a. **None (None):** The None keyword is used to define a null value, or no value at all. None is not the same as 0, False, or an empty string. None is a data type of its own (NoneType) and only None can be None.
    Example: x = None

II] Operators in Python:

Operators are used to perform operations on variables and values.

In python, there are a total of 7 types of operators as follows:

1. **Arithmetic Operators:**
    Arithmetic operators are used with numeric values to perform common mathematical operations:
    a. **Addition (+):**
    Example: x + y
    b. **Subtraction (-):**
    Example: x - y
    c. **Multiplication (*):**
    Example: x * y
    d. **Division (/):**
    Example: x / y
    e. **Modulus (%):**
    Example: x % y
    f. **Exponentiation (**):**
    Example: x ** y
    g. **Floor Division (//):**
    Example: x // y
2. **Assignment Operators:**
    Assignment operators are used to assign values to variables:
    a. **Assign (=):**
    Assign value of right side of expression to left side operand.
    Example: x = 10
    b. **Add and Assign (+=):**
    Add right side operand with left side operand and then assign to left operand.
    Example: x += 5
    c. **Subtract and Assign (-=):**
    Subtract right operand from left operand and then assign to left operand.
    Example: x -= 5
    d. **Multiply and Assign (*=):**

Multiply right operand with left operand and then assign to left operand.
Example: x *= 5

e. **Divide and Assign (/=):**
Divide left operand with right operand and then assign to left operand.
Example: x /= 5

f. **Modulus and Assign (%=):**
Takes modulus using left and right operands and assign result to left operand.
Example: x %= 5

g. **Exponentiate and Assign (**=):**
Calculate exponent (raise power) value using operands and assign value to left operand.
Example: x **= 5

h. **Divide (Floor) and Assign (//=):**
Divide left operand with right operand and then assign the value(floor) to left operand.
Example: x //= 5

3. **Comparison Operators:**
Comparison operators are used to compare two values:
   a. **Equal (==):**
   Example: x == y
   b. **Not equal (!=):**
   Example: x != y
   c. **Greater than (>):**
   Example: x > y
   d. **Less than (<):**
   Example: x < y
   e. **Greater than or equal to (>=):**
   Example: x >= y
   f. **Less than or equal to (<=):**
   Example: x <= y
4. **Logical Operators:**
Logical operators are used to combine conditional statements:
   a. **'and' operator:**
   Returns True if both statements are true.
   Example: x < 5 and x < 10
   b. **'or' operator:**
   Returns True if one of the statements is true.
   Example: x < 5 or x < 4
   c. **'not' operator:**

Reverse the result, returns False if the result is true and vice-versa.
Example: not(x < 5 and x < 10)

5. **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:
   a. **'is' operator:**
      Returns True if both variables are the same object.
      Example: x is y
   b. **'is not' operator:**
      Returns True if both variables are not the same object.
      Example: x is not y
6. **Membership Operators:**
   Membership operators are used to test if a sequence is present in an object:
   a. **'in' operator:**
      Returns True if a sequence with the specified value is present in the object.
      Example: x in y
   b. **'not in' operator:**
      Returns True if a sequence with the specified value is not present in the object.
      Example: x not in y
7. **Bitwise Operators:**
   Bitwise operators are used to compare (binary) numbers:
   a. **AND (&):**
      Sets each bit to 1 if both bits are 1.
      Example: x & y
   b. **OR (|):**
      Sets each bit to 1 if one of two bits is 1.
      Example: x | y
   c. **XOR (^):**
      Sets each bit to 1 if only one of two bits is 1.
      Example: x ^ y
   d. **NOT (~):**
      Inverts all the bits.
      Example: ~x
   e. **Zero fill left shift (<<):**
      Shift left by pushing zeros in from the right and let the leftmost bits fall off.
      Example: x << 2
   f. **Signed right shift (>>):**

Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.

Example: x >> 2

Implementation and Output:

I] Data Types in Python:

1. String (str):

```
>>> x = "Hello"
>>> x
'Hello'
>>> type(x)
<class 'str'>
```

2. Integer (int):

```
>>> x = 1292
>>> x
1292
>>> type(x)
<class 'int'>
```

3. Float (float):

```
>>> x = 12.3
>>> x
12.3
>>> type(x)
<class 'float'>
```

4. Complex (complex):

```
>>> x = 12 + 45j
>>> x
(12+45j)
>>> type(x)
<class 'complex'>
```

5. List (list):

```
>>> x = [12, 13.00, 'Hello', 4 + 5j]
>>> x
[12, 13.0, 'Hello', (4+5j)]
>>> type(x)
<class 'list'>
```

6. Range (range):

```
>>> x = range(17)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
>>> x
range(0, 17)
>>> type(x)
<class 'range'>
```

7. Tuple (tuple):

```
>>> x = (12 + 7j, 'Hello', 12.0, 55)
>>> x
((12+7j), 'Hello', 12.0, 55)
>>> type(x)
<class 'tuple'>
```

8. Dictionary (dict):

```
>>> x = {'x': 12, 'y': 13, 'z': 'hello'}
>>> x
{'x': 12, 'y': 13, 'z': 'hello'}
>>> type(x)
<class 'dict'>
>>> x.keys()
dict_keys(['x', 'y', 'z'])
>>> x.values()
dict_values([12, 13, 'hello'])
```

9. Set (set):

```
>>> x = {12, 12, 22.3, "Hello", 12j}
>>> x
{'Hello', 12, 12j, 22.3}
>>> type(x)
<class 'set'>
```

10. Boolean (bool):

```
>>> x = True
>>> x
True
>>> type(x)
<class 'bool'>
```

11. Binary (bytes):

```
>>> x = b'hello'
>>> x
b'hello'
>>> type(x)
<class 'bytes'>
```

12. None (NoneType):

```
>>> x = None
>>> x
>>> type(x)
<class 'NoneType'>
```

13. Usage of id() method:

```
>>> x = 100
>>> id(x)
2654159066576
>>> x
100
```

II] Operators in Python:

1. Arithmetic Operators:

```
>>> x = 10
>>> y = 6
>>> x + y
16
>>> x - y
4
>>> x * y
60
>>> x / y
1.6666666666666667
>>> x % y
4
>>> x // y
1
>>> x ** y
1000000
```

2. Assignment Operators:

```
>>> x = 10
>>> x += 7
>>> x
17
>>> x = 10
>>> x
10
>>> x += 7
>>> x
17
>>> x -= 7
>>> x
10
>>> x *= 7
>>> x
70
>>> x /= 7
>>> x
10.0
>>> x %= 7
>>> x
3.0
>>> x //= 7
>>> x
0.0
>>> x **= 7
>>> x
0.0
>>> x = 2
>>> x **= 7
>>> x
128
```

3. Comparison Operators:

```
>>> x = 15
>>> y = 4
>>> x == y
False
>>> x != y
True
>>> x < y
False
>>> x > y
True
>>> x <= y
False
>>> x >= y
True
```

4. Logical Operators:

```
>>> x = 5
>>> y = 7
>>> x < 3 and y > 5
False
>>> x < 3 or y > 5
True
>>> not(x < 3 and y > 5)
True
```

5. Identity Operators:

```
>>> x = 3
>>> y = 7
>>> x is y
False
>>> x is not y
True
```

6. Membership Operators:

```
>>> x = {'Hello', 12, 'Python', 12.0, 46j}
>>> 'Python' in x
True
>>> 'Python' not in x
False
>>> 'DSA' in x
False
>>> 'DSA' not in x
True
```

7. Bitwise Operators:

```
>>> x = 10
>>> y = 7
>>> x & y
2
>>> x | y
15
>>> x ^ y
13
>>> ~x
-11
>>> ~y
-8
>>> x << 2
40
>>> x >> 2
2
>>> y << 2
28
>>> y >> 2
1
```

Conclusion:

From this experiment, we can conclude that python is a versatile language in terms of data handling as it offers a variety of data types and operators for proper handling of the data it receives. There are a total of 8 built-in data types in python – **Text Type (str), Numeric Type (int, float, complex), Sequence Type (list, tuple, range), Mapping Type (dict), Set Type (set), Boolean Type (bool), Binary Type (bytes), None Type (NoneType)**. All of these data types hold (and work with) different kinds of data. Similarly, they are stored differently in memory. Along with a variety of data types, python also offers different operators – **Arithmetic Operators (+, -, *, /, %, **, //), Assignment Operators (=, +=, -=, *=, /=, %=, **=, //=), Comparison Operators (==, !=, >, <, >=, <=), Logical Operators (and, or, not), Identity Operators (is, is not), Membership Operators (in, not in), Bitwise Operators (&, |, ^, ~, <<, >>)** – to work with the available data types.

Experiment 2

Shashwat Shah
60004220126
C-22 Div B

Aim : To study and implement Input - output statements, control and loop statements.

Theory :

① Input - Output Statements.

a) Print() function - prints the specified message to the screen /other standard output device. The message can be string or any other object, the object will be converted to string before written to a screen.

b) Input() function - allows user input.

c) append() function - Inserts a single element into existing list. The element will be added to the end of a list. The length of the list increases by 1.

d) add() function - Adds an element to the set. If the element already exists, then it does not add the element.

e) map() function - Returns a map object of the results after applying the given function to each item of a given iterable list, tuple, etc.

f) split() function - Splits a string into a list, you can specify the seperator, default seperator is any white space.

② Control / Loop statements.

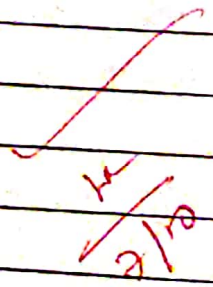
a) While loop - A while loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop is executed.

b) FOR loop - A for loop is used for iterating over a sequence. It is less like the for keyword in other programming language and works more like an iterator method statements, once for each item in list tuple, etc.

c) if-else - If a condition is true it will execute a block of statements and if the condition is false it wont. If the condition is false else statement is executed.

d) if-elif-else - used for decision making i.e the program will evaluate test expression and will execute the remain statements only if the given test expression turns out to be true.

Conclusion : Therefore we understood nuco input - output, control and loop statements work.



Experiment No. 2

Input / Output and Control Statements in Python

Aim:

To study and implement different input / output and control statements, loops and conditions in python.

Description:

I] Input / Output Statements in Python:

Python provides numerous built-in functions that are readily available to us at the Python prompt.

Some of the functions like input() and print() are widely used for standard input and output operations respectively.

Some other functions like append() and add() are also helpful to organize the inputs taken into data structures like lists, sets and tuples.

Functions like map() and split() also aid in taking inputs in the form of strings and processing them for use later.

We will now look at each of these functions in detail:

1. **input():**

   The input() function allows us to take user input. The input is always taken in the form of a string, which can then be type-casted to the required data type.

   Syntax:

   input(*prompt*)

   Here,

   prompt - A String, representing a default message before the input.

   Example:

   x = input('Enter your name: ')

2. **print():**

   The print() function prints the specified message to the screen, or other standard output device.

   The message can be a string, or any other object, the object will be converted into a string before written to the screen.

   Syntax:

   print(*object(s), sep=separator, end=end, file=file, flush=flush*)

   Here,

   object(s) - Any object, and as many as you like. Will be converted to string before printed.

   sep='separator' - Optional. Specify how to separate the objects, if there is more than one. Default is ' '.

   end='end' - Optional. Specify what to print at the end. Default is '\n' (line feed).

   file - Optional. An object with a write method. Default is sys.stdout.

flush - Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False.

Example:

print("Hello", "how are you?", sep="---")

3. **append():**

The append() method appends an element to the end of the list.

Syntax:

list.append(*elmnt*)

Here,

elemnt - Required. An element of any type (string, number, object etc.).

Example:

a = list()
b = 'Hello!'
a.append(b)

4. **add():**

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(*elemnt*)

Here,

elemnt - Required. The element to add to the set.

Example:

a = set()
b = 'Hello!'
a.add(b)

5. **map():**

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax:

map(*function*, *iterables*)

Here,

function – Required. The function to execute for each item.

iterables – Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

Example:

For taking input as a string of integers and storing those integers in a list, we use the following technique:

List = list(map(int, input().split()))

6. **split():**

The split() method splits a string into a list.

You can specify the separator. Default separator is any whitespace.

Syntax:

string.split(*separator*, *maxsplit*)

Here,

separator - Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator.

maxsplit - Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences".

Example:

For taking input as a string of integers and storing those integers in a list, we use the following technique:

List = list(map(int, input().split()))

II] Loops, Conditional Statements and Control Statements:

In Python, Loops are used to iterate repeatedly over a block of code. In order to change the way a loop is executed from its usual behavior, control statements are used. Control statements are used to control the flow of the execution of the loop based on a condition.

Decision-making statements in programming languages decide the direction of the flow of program execution.

In Python, if-elif-else statement is used for decision making.

We now discuss some important loops and control statements:

1. **for loop:**

   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.

   Syntax:

   for *iterator_var* in *sequence*:
   statements(s)

   Example:

   for x in "banana":
   print(x)

2. **while loop:**

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

   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.

   Syntax:

   while *expression*:
   statement(s)

   Example:

   i = 1
   while i < 6:

```
    print(i)
    i += 1
```

3. **if statement:**

   if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e., if a certain condition is true then a block of statement is executed otherwise not.

   Syntax:

```
if condition:
    # Statements to execute if
    # condition is true
```

   Example:

```
i = 10
if i > 15:
    print("10 is less than 15")
```

4. **if…else statement:**

   The if statement alone tells us that if a condition is true, it will execute a block of statements and if the condition is false, it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

   Syntax:

```
if condition:
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

   Example:

```
i = 20
if i < 15:
    print("i is smaller than 15")
else:
    print("i is greater than 15")
```

5. **if…elif…else statement:**

   Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

   Syntax:

```
if condition:
    statement
elif condition:
    statement
```

```
else:
    statement
```

Example:

```
i = 20
if i == 10:
    print("i is 10")
elif i == 15:
    print("i is 15")
elif i == 20:
    print("i is 20")
else:
    print("i is not present")
```

Note: There can be multiple elif statements between the initial if statement and the final else statement.

6. **break statement:**

The break statement in Python is used to terminate or abandon the loop containing the statement and brings the control out of the loop. It is used with both the while and the for loops, especially with nested loops (loop within a loop) to quit the loop. It terminates the inner loop and control shifts to the statement in the outer loop.

Example:

```
for letter in 'engineers':
    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break
print 'Current Letter: ', letter
```

7. **continue statement:**

When a program encounters a continue statement in Python, it skips the execution of the current iteration when the condition is met and lets the loop continue to move to the next iteration. It is used to continue running the program even after the program encounters a break during execution.

Example:

```
# Prints all letters except 'e' and 's'
for letter in 'engineers':
    if letter == 'e' or letter == 's':
        continue
    print 'Current Letter: ', letter
```

8. **pass statement:**

The pass statement is a null operator and is used when the programmer wants to do nothing when the condition is satisfied. This control statement in Python does not terminate or skip the execution, it simply passes to the next iteration.

A loop cannot be left empty otherwise the interpreter will throw an error and to avoid this, a programmer can use the pass statement.

Example:

```
# An empty loop
for letter in 'engineers':
   pass
print 'Last Letter: ', letter
```

Implementation and Output:

I] Input / Output Statements in Python:

1. Find square root of number.

Program:

```
n = float(input("Enter the nummber (n): "))
print("Square root of n is:", n ** (0.5))
```

Output:

```
Enter the nummber (n): 81
Square root of n is: 9.0
```

2. Find area and perimeter of rectangle.

Program:

```
l = float(input("Enter length: "))
b = float(input("Enter breadth: "))
print("Area =", l * b, "\nPerimeter =", l + b)
```

Output:

```
Enter length: 15
Enter breadth: 5.5
Area = 82.5 sq. units
Perimeter = 20.5 units
```

3. Swapping of two numbers with and without using third variable.

Program (swapping with third variable):

```
a = float(input("Enter first number (a): "))
b = float(input("Enter second number (b): "))
c = a
a = b
b = c
print("After swapping:")
print("a =", a, "\nb =", b)
```

Output:

```
Enter first number (a): 10
Enter second number (b): 5
After swapping:
a = 5.0
b = 10.0
```

Program (swapping without third variable):

```
a = float(input("Enter first number (a): "))
b = float(input("Enter second number (b): "))
a = a + b
b = a - b
a = a - b
print("After swapping:")
print("a =", a, "\nb =", b)
```

Output:

```
Enter first number (a): 40
Enter second number (b): 15
After swapping:
a = 15.0
b = 40.0
```

4. Adding elements in List, Set and Tuple.

Program (without using map() and split() functions):

```
# We will add elements for list, set and tuple
# for list

List = list()

l = int(input("Size of list: "))

print("Enter list elements:")

for i in range(l):
    List.append(int(input()))

print(List)

# for set

Set = set()

s = int(input("\nSize of set: "))

print("Enter set elements:")

for i in range(s):
```

```
    Set.add(int(input()))

print(Set)

# for tuple

T = (2, 3, 4, 5, 6)

print("\nTuple, before adding new element:")
print(T)


L = list(T)

L.append(int(input("Enter the new element: ")))

T = tuple(L)

print("Tuple, after adding the new elememt:")
print(T)
```

Output:

```
Size of list: 5
Enter list elements:
1
2
3
4
5
[1, 2, 3, 4, 5]

Size of set: 6
Enter set elements:
2
3
4
5
6
7
{2, 3, 4, 5, 6, 7}

Tuple, before adding new element:
(2, 3, 4, 5, 6)
Enter the new element: 8
Tuple, after adding the new elememt:
(2, 3, 4, 5, 6, 8)
```

Program (using map() and split() functions):

```
# Using map and split functions for performing i/o operations on list, set and tuple

l = int(input("Size of list: "))

print("Enter list elements:")
List = list(map(int, input().split()))[:l]

print(List)

s = int(input("\nSize of set: "))

print("Enter set elements:")
Set = set(list(map(int, input().split()))[:s])

print(Set)

t = int(input("\nSize of tuple: "))

print("Enter tuple elements:")
Tuple = tuple(list(map(int, input().split()))[:t])

print(Tuple)

L = list(Tuple)

L.append(int(input("Enter the new element: ")))

Tuple = tuple(L)

print("Tuple, after adding the new elememt:")
print(Tuple)
```

Output:

```
Size of list: 5
Enter list elements:
2 3 4 5 6
[2, 3, 4, 5, 6]

Size of set: 4
Enter set elements:
2 4 6 3
{2, 3, 4, 6}

Size of tuple: 4
```

```
Enter tuple elements:
6 7 2 3
(6, 7, 2, 3)
Enter the new element: 5
Tuple, after adding the new elememt:
(6, 7, 2, 3, 5)
```

II] Loops, Conditional Statements and Control Statements in Python:

1. Print the given triangle pattern using for loop.

Program:

```
h = int(input("Enter height of pyramid: "))

for i in range(1, h + 1):
    for j in range(i):
        print(i, end = " ")
    print()
```

Output:

```
Enter height of pyramid: 6
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
```

2. Find the factorial of a number using for loop.

Program:

```
n = int(input("Enter the number: "))

result = 1

for i in range(1, n + 1):
    result *= i

if (n == 0):
    result = 1

print("Factorial(n) =", result)
```

Output:

```
Enter the number: 6
Factorial(n) = 720
```

3. Print the Fibonacci sequence up to given 'n' value using for loop.

Program:

```
n = int(input("Enter number of terms (n): "))

# print fibonacci series containing n terms
a = 0
b = 1

print("Fibonacci series is:")
if n == 1:
   print(a)
elif n >= 2:
   print(a)
   print(b)
   for i in range(n - 2):
      c = a + b
      a = b
      b = c
      print(c)
```

Output:

```
Enter number of terms (n): 7
Fibonacci series is:
0
1
1
2
3
5
8
```

4. Demonstrate 'while' loop with one example.

Program:

```
n = int(input("Enter the number: "))

tmp = n

result = 1

while n >= 1:
   result *= n
   n -= 1

if tmp == 0:
   result = 1
```

```
print('Factorial(n) =', result)
```

Output:
Enter the number: 7
Factorial(n) = 5040

5. Check whether the input number is even or odd using if...else loop.

Program:

```
# check whether number is even or odd

n = int(input("Enter the number: "))

if n % 2 == 0:
   print("Number is even.")
else:
   print('Number is odd.')
```

Output:
Enter the number: 5
Number is odd.

6. Check whether the input year is leap year or not using nested if.

Program:

```
y = int(input("Enter year: "))

if y % 4 == 0:
   if y % 100 == 0:
      if y % 400 == 0:
         print(y, "is a leap year.")
      else:
         print(y, "is not a leap year.")
   else:
      print(y, "is a leap year.")
else:
   print(y, "is not a leap year.")
```

Output:
Enter year: 2020
2020 is a leap year.

7. Demonstrate 'if...elif...else' loop with one example.

Program:

```
n = 2 + 8j

if type(n) is int:
```

```
    print("Object is int.")
elif type(n) is float:
    print("Object is float.")
elif type(n) is str:
    print("Object is str.")
elif type(n) is set:
    print("Object is set.")
elif type(n) is dict:
    print("Object is dict.")
elif type(n) is tuple:
    print("Object is tuple.")
elif type(n) is list:
    print("Object is list.")
else:
    print("Object is complex.")
```

Output:

```
Object is complex.
```

8. Demonstrate 'continue', 'break' and 'pass' with one example each.

Program:

```
print("Demonstrating 'continue' statement:")
for letter in 'rajendra':
    if letter in ['e', 'a']:
        continue
    print("Current letter:", letter)
print()

print("Demonstrating 'break' statement:")
for letter in 'rajendra':
    if letter in ['e', 'a']:
        break
    print("Current letter:", letter)
print()

print("Demonstrating 'pass' statement:")
for letter in 'rajendra':
    pass
print("Last letter:", letter)
```

Output:

```
Demonstrating 'continue' statement:
Current letter: r
Current letter: j
Current letter: n
Current letter: d
```

```
Current letter: r

Demonstrating 'break' statement:
Current letter: r

Demonstrating 'pass' statement:
Last letter: a
```

Conclusion:

In this experiment, we studied some important input / output techniques (input() and print()) along with different types of loops (for loop and while loop), conditional statements (if, if…else, if…elif…else) and control statements (continue, break, pass). We also studied how functions like append(), add(), map() and split() can be used to take user input in the form of a string, list, set or a tuple. We can thus conclude that python language, along with being extremely versatile and flexible, is also a robust language with a variety of methods for various functionalities like performing input / output operations, performing repetitive and recursive tasks and controlling the flow of the program. In this experiment, we saw various techniques through which python performs the above functionalities, proving its extensive scope.

Experiment 3

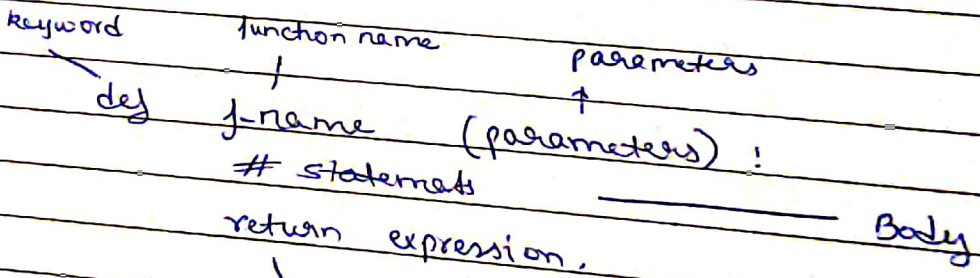
Shashwat Shah
60004220126
Comps B C2-2

Aim: To implement functions in python.

Theory:

A function is a block of code which only runs when it is called. One can pass data, known as parameters, into a function. A function can return data as a result. The main idea to use function is to put some commonly or repeatedly done tasks together and make a function, so that instead of writing the same code again & again for different input we can do the function calls to reuse code contained in it over and over again.

Syntax and structure of a function



Conclusion: In this experiment we understood about functions and their different uses to solve a question in a simple manner.

14/10

Experiment No. 3
Functions in Python

Aim:

To study and implement different functions in python.

Description:

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

Syntax:

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

my_function()
```

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. As many arguments as needed can be added, but they must be separated with a comma.
By default, a function must be called with the correct number of arguments. Meaning that if a function expects 2 arguments, it must be called with 2 arguments, not more, and not less.
In Python, may types of functions exist. Here we will go over a few of them.

I] Built-in Functions:

Python comes with many useful built-in functions. Some of them are as follows:

1. **Type Conversion:**
   You can convert from one type to another with methods like int(), float(), and complex(), etc.
   Example:
   int(5.5) = 5
   float(3) = 3.0
   str(12.3) = '12.3'
   int('3') = 3
   complex('4') = (4 + 0j)
2. **Type Coercion:**
   This is also called as implicit casting. The system casts the value to the default type.
   Example:
   5 / 60.0 = 0.08333333333333333 (a float, although 5 is an integer)
3. **Mathematical Functions:**
   Python provides us with the 'math' module, which consists of various mathematical functions and constants (like pi). This module must be imported before its functions can be used in the program.

Example:

i. math.pi
ii. math.sin(60)
iii. math.log10(10)

4. **Date and Time functions:**

Python Datetime module supplies classes to work with date and time. These classes provide a number of functions to deal with dates, times and time intervals. Date and datetime are an object in Python, so when you manipulate them, you are actually manipulating objects and not string or timestamps.

Example:

i. time.localtime(time.time())
ii. time.asctime(a)
iii. calendar.month(2022, 9)
iv. datetime.date.today()

5. **dir() function:**

The dir() function returns all properties and methods of the specified object, without the values.

This function will return all the properties and methods, even built-in properties which are default for all object.

Example:

dir(Person) (here Person is a user-defined class)

6. **The map() function:**

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax:

map(function, iterables)

Example:

```
def myfunc(a, b):
  return a + b


x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))
```

7. **The filter() function:**

The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.

Syntax:

filter(function, iterable)

Example:

```
seq = [0, 1, 2, 3, 5, 8, 13]
result = filter(lambda x: x % 2 != 0, seq)
```

8. **The help() function:**

The Python help() function is used to display the documentation of modules, functions, classes, keywords, etc.
The help() method calls the built-in Python help system.
The help() method is used for interactive use.

Example:

```
help(list)
help(str)
```

II] User-defined Function:

Functions that we define ourselves to do certain specific task are referred as user-defined functions. The way in which we define and call functions in Python are already discussed.
Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions.
All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions:

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmars working on large project can divide the workload by making different functions.

Syntax:

```
def function_name():
   statements
   .
   .
```

Example:

```
def fun():
   print("Inside function")
fun()
```

III] Anonymous Functions:

In Python, an anonymous function is a function that is defined without a name.
While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
Hence, anonymous functions are also called lambda functions.
We use lambda functions when we require a nameless function for a short period of time.
In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Syntax:

lambda arguments: expression

Example:

```
double = lambda x: x * 2
```

IV] Recursive Functions:

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 mathematically elegant approach to programming.

Syntax:

```
def func(): <--
                |
                | (recursive call)
                |
    func()  ----
```

Example:

```
def factorial(x):
   if x == 1:
      return 1
   else:
      return (x * factorial(x-1))
```

V] Some functions associated with data structures like lists, tuples, sets and dictionaries:

Functions associated with lists:

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

1. **append():** Adds an element at the end of the list
2. **clear():** Removes all the elements from the list
3. **copy():** Returns a copy of the list
4. **count():** Returns the number of elements with the specified value
5. **extend():** Add the elements of a list (or any iterable), to the end of the current list
6. **index():** Returns the index of the first element with the specified value
7. **insert():** Adds an element at the specified position
8. **pop():** Removes the element at the specified position
9. **remove():** Removes the first item with the specified value
10. **reverse():** Reverses the order of the list
11. **sort():** Sorts the list

Functions associated with tuples:

Python has two built-in methods that you can use on tuples.

1. **count():** Returns the number of times a specified value occurs in a tuple

2. **index():** Searches the tuple for a specified value and returns the position of where it was found

Functions associated with sets:

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

1. **add():** Adds an element to the set
2. **clear():** Removes all the elements from the set
3. **copy():** Returns a copy of the set
4. **difference():** Returns a set containing the difference between two or more sets
5. **discard():** Remove the specified item
6. **intersection():** Returns a set, that is the intersection of two or more sets
7. **union():** Return a set containing the union of sets
8. **pop():** Removes an element from the set

Functions associated with dictionaries:

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

1. **clear():** Removes all the elements from the dictionary
2. **copy():** Returns a copy of the dictionary
3. **keys():** Returns a list containing the dictionary's keys
4. **pop():** Removes the element with the specified key
5. **get():** Returns the value of the specified key
6. **values():** Returns a list of all the values in the dictionary
7. **items():** Returns a list containing a tuple for each key value pair

Implementation and Output:

A] Functions for following data structures:

1. List:

Code:

```
l = ['apple', 'banana', 'mango']
l.append('orange')
print(l)
x = l.copy()
print(x)
print(l.count('apple'))
l2 = ['bmw', 'mercedes', 'ford']
l.extend(l2)
print(l)
print(l.index('banana'))
l.insert(3, 'peach')
print(l)
l.pop(2)
print(l)
l.remove('apple')
print(l)
l.reverse()
```

```
print(l)
l.sort()
print(l)
l.clear()
print(l)
```

Output:

```
['apple', 'banana', 'mango', 'orange']
['apple', 'banana', 'mango', 'orange']
1
['apple', 'banana', 'mango', 'orange', 'bmw', 'mercedes', 'ford']
1
['apple', 'banana', 'mango', 'peach', 'orange', 'bmw', 'mercedes', 'ford']
['apple', 'banana', 'peach', 'orange', 'bmw', 'mercedes', 'ford']
['banana', 'peach', 'orange', 'bmw', 'mercedes', 'ford']
['ford', 'mercedes', 'bmw', 'orange', 'peach', 'banana']
['banana', 'bmw', 'ford', 'mercedes', 'orange', 'peach']
[]
```

2. Tuple:

Code:

```
t = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
print(t.count(7))
print(t.index(8))
```

Output:

```
2
3
```

3. Set:

Code:

```
s = {"apple", "banana", "cherry"}
s.add('orange')
print(s)
x = s.copy()
print(x)
y = {"google", "microsoft", "apple"}
z = x.difference(y)
print(z)
s.discard('banana')
print(s)
z = x.intersection(y)
print(z)
z = x.union(y)
print(z)
s.pop()
```

```
print(s)
s.clear()
print(s)
```

Output:

```
{'apple', 'cherry', 'orange', 'banana'}
{'apple', 'cherry', 'orange', 'banana'}
{'cherry', 'orange', 'banana'}
{'apple', 'cherry', 'orange'}
{'apple'}
{'banana', 'microsoft', 'cherry', 'orange', 'google', 'apple'}
{'cherry', 'orange'}
set()
```

4. Dictionary:

Code:

```
d = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = d.copy()
print(x)
x = d.keys()
print(x)
d.pop('model')
print(d)
x = d.get('brand')
print(x)
x = d.values()
print(x)
x = d.items()
print(x)
d.clear()
print(d)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
dict_keys(['brand', 'model', 'year'])
{'brand': 'Ford', 'year': 1964}
Ford
dict_values(['Ford', 1964])
dict_items([('brand', 'Ford'), ('year', 1964)])
{}
```

B] Write a Python function histogram(l) that takes as input a list of integers with repetitions and returns a list of pairs as follows:

- **for each number n that appears in l, there should be exactly one pair (n,r) in the list returned by the function, where r is the number of repetitions of n in l.**
- **the final list should be sorted in ascending order by r, the number of repetitions. For numbers that occur with the same number of repetitions, arrange the pairs in ascending order of the value of the number.**

For instance:

>>> histogram([13,12,11,13,14,13,7,7,13,14,12])
[(11, 1), (7, 2), (12, 2), (14, 2), (13, 4)]
>>> histogram([7,12,11,13,7,11,13,14,12])
[(14, 1), (7, 2), (11, 2), (12, 2), (13, 2)]
>>> histogram([13,7,12,7,11,13,14,13,7,11,13,14,12,14,14,7])
[(11, 2), (12, 2), (7, 4), (13, 4), (14, 4)]

Code:

```
def histogram(l):
    count = 0
    x = []
    k = []
    for i in range(len(l)):
        index = i
        count = 0
        for j in range(index, len(l)):
            if l[index] == l[j] and l[index] not in k:
                count += 1
        k = k + [l[index]]
        if count != 0:
            x = x + [(l[index], count)]

    x.sort()
    x = sorted(x, key = lambda x: x[1])
    return x

print("Enter the numbers:")
print(histogram(list(map(int, input().split()))))
```

Output:

```
Enter the numbers:
13 7 12 7 11 13 14 14 15 7 11 13 12 14 14 15
```

```
[(11, 2), (12, 2), (15, 2), (7, 3), (13, 3), (14, 4)]
```

C] A positive integer n is said to be perfect if the sum of the factors of n, other than n itself, add up to n. For instance 6 is perfect since the factors of 6 are {1,2,3,6} and 1+2+3=6. Likewise, 28 is perfect because the factors of 28 are {1,2,4,7,14,28} and 1+2+4+7+14=28. Write a Python function perfect(n) that takes a positive integer argument and returns True if the integer is perfect, and False otherwise.

Code:

```
def perfect(num):
    sum = 0
    for i in range(1, num):
        if num % i == 0:
            sum=sum+i
    return sum == num

n = int(input("Enter a number: "))

res = perfect(n)
print(res, ".", sep="")
if res == False:
    print("Entered number is not a perfect number.")
else:
    print("Entered number is a perfect number.")
```

Output:

```
Enter a number: 28
True.
Entered number is a perfect number.

Enter a number: 25
False.
Entered number is not a perfect number.
```

D] Implement a recursive function to solve tower of Hanoi Problem

Code:

```
def hanoi(n, source, auxiliary, target):
    if (n == 1):
        print("Move disk 1 from peg {} to peg {}".format(source, target))
        return
    hanoi(n - 1, source, target, auxiliary)
    print("Move disk {} from peg {} to peg {}".format(n, source, target))
    hanoi(n - 1, auxiliary, source, target)

hanoi(int(input("Enter the number of disks: ")), 'A', 'B', 'C')
```

Output:

```
Enter the number of disks: 4
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 3 from peg A to peg B
Move disk 1 from peg C to peg A
Move disk 2 from peg C to peg B
Move disk 1 from peg A to peg B
Move disk 4 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 2 from peg B to peg A
Move disk 1 from peg C to peg A
Move disk 3 from peg B to peg C
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
```

E] Implement lambda function to find greater of the 2 input numbers

Code:

```
a = int(input("Enter a number(a): "))
b = int(input("Enter a number(b): "))
maximum = lambda a, b: a if a > b else b
print(f"Maximum of {a} and {b} is {maximum(a, b)}.")
```

Output:

```
Enter a number(a): 56
Enter a number(b): 77
Maximum of 56 and 77 is 77.
```

F] Using map function perform element wise addition of elements of two lists.

Code:

```
# creating empty lists
list1 = []
list2 = []

# number of element in each list
n1 = int(input("Enter the number of elements in list 1: "))

print("Enter elements in list 1: ")
list1 = list(map(int, input().split()))[:n1]
print(list1)

n2 = int(input("Enter the number of elements in list 2: "))

print("Enter elements in list 2: ")
```

```
list2 = list(map(int, input().split()))[:n2]
print(list2)

result = list(map(lambda a, b: a + b, list1, list2))

print("The list of element-wise sum of the two lists is: ", str(result))
```

Output:

```
Enter the number of elements in list 1: 5
Enter elements in list 1:
1 2 3 4 5
[1, 2, 3, 4, 5]
Enter the number of elements in list 2: 5
Enter elements in list 2:
6 3 4 2 7
[6, 3, 4, 2, 7]
The list of element-wise sum of the two lists is:  [7, 5, 7, 6, 12]
```

G] Using map and filter find the cube of all the odd numbers from the given input list

Code:

```
arr = []

n = int(input("Enter the number of elements in the array: "))

print("Enter elements in the array:")

arr = list(map(int, input().split()))[:n]

print(arr)

print("The list of cubes of odd numbers in the array is:")

arr2 = list(map(lambda x: x ** 3, filter(lambda x: x % 2 != 0, arr)))
print(arr2)
```

Output:

```
Enter the number of elements in the array: 7
Enter elements in the array:
1 6 5 4 2 8 9
[1, 6, 5, 4, 2, 8, 9]
The list of cubes of odd numbers in the array is:
[1, 125, 729]
```

Conclusion:

In this experiment, we studied and implemented different functions offered in Python. We studied various built-in functions, anonymous function (lambda function), user-defined functions, different functions associated with data structures like lists, sets, tuples and dictionaries and recursive functions. We observed that, depending on the usage of these data structures, some of the functions are common to some data structures, while some are specific to that particular data structure (for example, the functions like union(), intersection() and difference() are specifically for the set data structure). We implemented different programs like creation of histogram, solution to the Hanoi Tower problem, detection of perfect number, etc. After studying the various types of functions offered by Python, we can safely conclude that functions are very powerful tools which help code to be reused multiple times and provide a convenient way to program difficult problems and model complex environments. Thus, we have seen the robustness and versatility of Python Language through functions provided by it.

Experiment 4

Shashwat Shah
60004220126
C22 Div B

Aim : To implement classes, objects and inheritance

Theory :

① Classes

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates new types of objects, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods for modifying their state.

② Objects

An object is an instance of a class. A class is like a blueprint while an instance is a copy of a class with actual values. Its not an idea any more.

③ Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. It represents real world relationships well. It provides the reusability of code. It allows us to add more features to a class without modifying it. It is transitive in nature, which means that if class B inherits from class A, then all the subclasses of B would automatically inherit from B.

Conclusion : In this experiment we learnt about class objects and inheritance

Experiment No. 4

Classes, Objects and Inheritance in Python

Aim:

To study and implement objects, classes and inheritance in Python.

Description:

I] Classes and Objects:

Classes:

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.

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

Objects:

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of:

1. State: It is represented by the attributes of an object. It also reflects the properties of an object.
2. Behaviour: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. Identity: It gives a unique name to an object and enables one object to interact with other objects.

Syntax (class definition):

```
class ClassName:
   # Statement
```

Syntax (object definition):

```
obj = ClassName()
print(obj.atrr)
```

Example of a class and object:

```
class Dog: (defining class Dog)
   pass
rodger = Dog() (defining object of class Dog)
```

II] __init__() Function:

The __init__ function is similar to constructors in C++ and Java.

All classes have a function called __init__(), which is always executed when the class is being initiated.

The __init__() function is used to assign values to object properties, or other operations that are necessary to do when the object is being created.

Syntax:

```
def __init__(self, … ):
   # statements
```

Example:

```
class ComplexNumber:
   def __init__(self, r=0, i=0):
      self.real = r
      self.imag = i
```

III] Object Methods:

Objects can also contain methods. Methods in objects are functions that belong to the object.

Syntax:

```
def method(self, … )
   # statements
```

Example:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)
```

IV] Modify Object Properties:

Object properties can be modified using the dot operator.

Example:

```
p1.age = 40
```

V] Delete Object Properties and Objects:

Objects and object properties can be deleted by using the del keyword.

Example:

```
del p1.age
del p1
```

VI] Inheritance:

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Creating Parent Class:

Any class can be a parent class, so the syntax is the same as creating any other class.

Creating Child Class:

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class.

Example:

```
class Student(Person):
  pass
```

Now the Student class has the same properties and methods as the Person class.

Adding the __init__() function:

To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function in the child's __init__() function.

Example:

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Using the super() function:

Python also has a super() function that will make the child class inherit all the methods and properties from its parent.

Example:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Adding methods and properties:

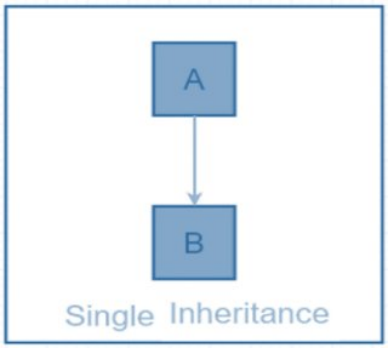
Methods and properties can be added according to the syntax discussed previously.

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

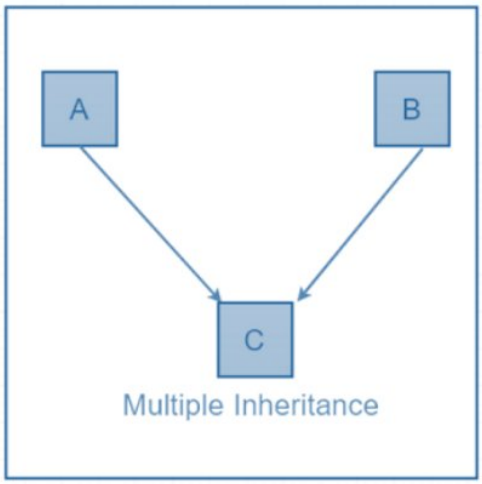
VII] Types of Inheritance:

The types of inheritance depend on the number of children and parents involved. There are four kinds of inheritance available in Python:

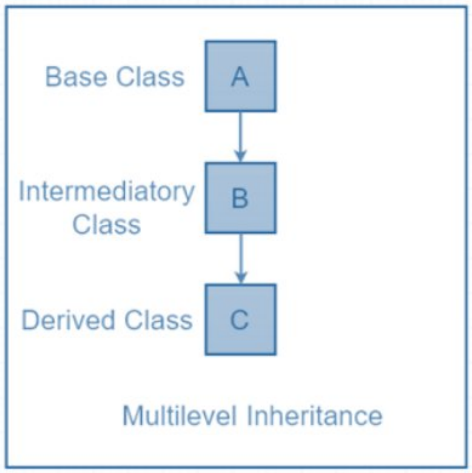
1. Single: Single inheritance allows a derivate class to inherit properties of one parent class, and this allows code reuse and the introduction of additional features in existing code.



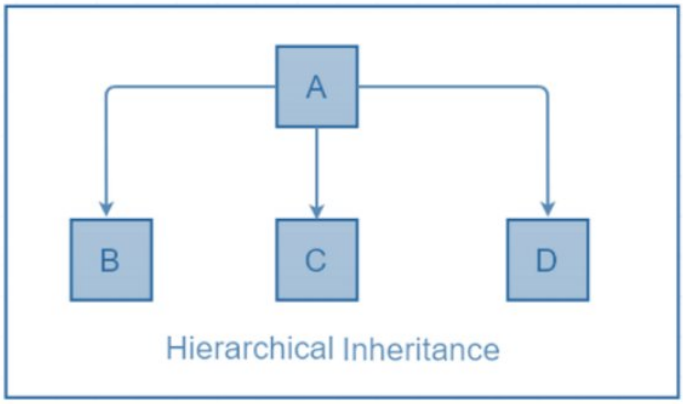
2. Multiple Inheritance: If a class is able to be created from multiple base classes, this kind of Inheritance is known as multiple Inheritance. When there is multiple Inheritance, each of the attributes that are present in the classes of the base has been passed on to the class that is derived from it.



3. Multi-level Inheritance: The features that are part of the original class, as well as the class that is derived from it, are passed on to the new class. It is similar to a relationship involving grandparents and children.



4. Hierarchical Inheritance: If multiple derived classes are created from the same base, this kind of Inheritance is known as hierarchical inheritance. In this instance, we have two base classes as a parent (base) class as well as two children (derived) classes.



Implementation and Output:

A] Classes and Objects:

Code:
```
class MyClass:
    a = 10

p1 = MyClass()
print(p1.a)
```

Output:
```
<__main__.MyClass object at 0x0000018549E919C0>
10
```

B] Use of __init__() function in class:

Code:
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Output:
```
John
36
```

C] Object Methods, Modification of Object Properties, Deleting Object Properties and Objects:

Code:

```
class Person:
    def __init__(self, fname, lname, age):
        self.firstname = fname
        self.lastname = lname
        self.age = age

    def printName(self):
        print(self.firstname, self.lastname)

p1 = Person(input("Enter first name: "), input("Enter last name: "), input("Enter age: "))
p1.printName()
p1.firstname = "Devraj"
p1.printName()
print("Age is:", p1.age)
del p1.age # deleting object property
del p1 # deleting object
```

Output:

```
Enter first name: Vijay
Enter last name: Harkare
Enter age: 20
Vijay Harkare
Devraj Harkare
Age is: 20
```

D] Python Inheritance:

Code:

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printName(self):
        print(self.firstname, self.lastname)

class Student(Person):
    def __init__(self, fname, lname):
        # Person.__init__(self, fname, lname)
        super().__init__(fname, lname) # Here self isn't required
```

```
x = Student(input("Enter first name: "), input("Enter last name: "))
x.printName()
```

Output:

```
Enter first name: Vijay
Enter last name: Harkare
Vijay Harkare
```

E] Python Code to implement following inheritance example:
Classes: Employee, Developer, Tester, Manager.
Developer, tester, Manager inherit Employee.
Manager handles Developer, tester.
Manager class: implement functions to add Developer/Tester and Remove Developer/ Tester.
Display to see the list of employees he manages.

Code:

```
class Employee():
    emp_list = { }
    manager = { }
    def __init__(self, fname, lname, eid, type):
        self.emp_list[eid] = self
    def info(self, obj):
        return (obj.fname, obj.lname, obj.eid, obj.type)

class Manager(Employee):
    def __init__(self, fname, lname, eid):
        Employee.__init__(self, fname, lname, eid, 'Manager')
        super().manager[eid] = self
        self.eid = eid
        self.fname = fname
        self.lname = lname
        self.type = 'Manager'

    test = { }
    dev = { }
    man_emp = { }

    def add(self, obj):
        super().emp_list[obj.eid] = obj
        if (obj.type == 'Developer'):
            self.dev[obj.eid] = obj
        else:
            self.test[obj.eid] = obj
        self.man_emp[obj.eid] = obj
```

```
    def remove(self, eid, type):
        del super().emp_list[eid]
        if (type == 'Developer'):
            del self.dev[eid]
        else:
            del self.test[eid]
        del self.man_emp[eid]

    def display(self):
        for i in self.man_emp:
            print(i, ":", str(super().info(self.man_emp[i])))

    def manager_list(self):
        return super().manager

class Developer(Employee):
    def __init__(self, fname, lname, eid):
        Employee.__init__(self, fname, lname, eid, 'Developer')
        self.eid = eid
        self.fname = fname
        self.lname = lname
        self.type = 'Developer'

class Tester(Employee):
    def __init__(self, fname, lname, eid):
        Employee.__init__(self, fname, lname, eid, 'Tester')
        self.eid = eid
        self.fname = fname
        self.lname = lname
        self.type = 'Tester'

manager1 = Manager('Vijay', 'Harkare', 1)
print("Data corresponding to the manager:", manager1.manager[manager1.eid].fname,
manager1.manager[manager1.eid].lname)
print()

print("List entry structure for employees is:")
print("<ID>: (<First Name>, <Last Name>, <ID>, ,<Type>)")
print()

print('List of employees before adding developer:')

manager1.display()
print()

print("List of employees after adding developer:")
```

```
manager1.add(Developer('Anaida', 'Lewis', 2))

manager1.display()
print()

manager1.add(Tester('Vidhi', 'Kansara', 3))

print('List of employees after adding tester:')
manager1.display()
print()

print('List of employees after removing tester:')
manager1.remove(3, 'Tester')
manager1.display()
```

Output:

```
Data corresponding to the manager: Vijay Harkare

List entry structure for employees is:
<ID>: (<First Name>, <Last Name>, <ID>, ,<Type>)

List of employees before adding developer:

List of employees after adding developer:
2 : ('Anaida', 'Lewis', 2, 'Developer')

List of employees after adding tester:
2 : ('Anaida', 'Lewis', 2, 'Developer')
3 : ('Vidhi', 'Kansara', 3, 'Tester')

List of employees after removing tester:
2 : ('Anaida', 'Lewis', 2, 'Developer')
```

Conclusion:

In this experiment, we explored the object-oriented properties of Python, given that Python is an object-oriented language. We observed that Python packs a plethora of functionalities when it comes to object-oriented programming. Here, we studied and implemented some of the most important concepts in object-oriented programming, like classes, objects (their creation and usage), and inheritance. We also studied and implemented various functions like __init__() and super(), which help to define the classes and establish and leverage the relationship between classes involved in inheritance. We also studied the concepts related to methods and attributes contained within classes, and how they behave when involved in inheritance. We also studied and implemented modification of objects and their properties, and finally deleting them when no longer needed. Thus, we can conclude from our observations and experimentation, that Python offers robust and extensive object-oriented programming support, which can be effectively used to model and simulate the complex world around us, along with the intricate and implicit relations among various objects present.

Experiment 5

Shashwat Shah
60004220126
C2-2 Div B

Aim: To understand and implement exception handling

Theory:

Errors are the problems in a program due to which the program will stop the execution, on the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program. Two types of errors occur in python:

(1) Syntax Errors: When in the runtime an error that has a language error

(2) Logical Errors: When in the runtime an error occurs after passing the syntax test is called exception are logical type. They are also known as exceptions are the unusual events that occur during the execution of the program that interrupts the normal flow of the program. Generally exception is raised the program stops the execution and thus the further code is not executed, therefore an exception is a python object that represents a runtime error. An exception object represents an error.

a) Value error - It gets raised when the built in function for a datatype has the valid type of arguments, but the arguments have invalid values specified.

b) Arithmetic error - It is the base class for all errors related to numeric calculations.

c) Import error - It gets raised when an import statement gets raised.

d) Lookup error - It is the base class for all lookup

e) Keyboard interrupt - This gets raised when the progra... execution by pressing ctrl + c.

Conclusion: Therefore, we have implemented Exception handling using python.

28/10

Experiment No. 5
Exception Handling in Python

Aim:
To study and implement Exception Handling in Python.

Description:
Exception Handling:
When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
These exceptions can be handled using the try statement.
A total of four blocks are involved in exception handling.

Try and Except block:
The **try** block lets you test a block of code for errors.
The **except** block lets you handle the error.
Example:
```
try:
  print(x)
except:
  print("An exception occurred")
```

Else block:
The **else** block lets you execute code when there is no error.
Example:
```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally block:
The **finally** block lets you execute code, regardless of the result of the try- and except blocks.
Since the try block raises an error, the except block will be executed.
Without the try block, the program will crash and raise an error.
Example:
```
try:
  print(x)
except:
  print("Something went wrong")
```

```
finally:
  print("The 'try except' is finished")
```

Raising an exception:

As a Python developer one can choose to throw an exception if a condition occurs.
To throw (or raise) an exception, we use the raise keyword.
The **raise** keyword is used to raise an exception.
One can define what kind of error to raise, and the text to print to the user.

Example:

```
raise Exception("Sorry, no numbers below zero")
```

User-defined exceptions:

Exceptions need to be derived from the Exception class, either directly or indirectly.
Although not mandatory, most of the exceptions are named as names that end in "Error" similar to the naming of the standard exceptions in python.

Syntax:

```
class CustomError(Exception):
   pass

raise CustomError("Example of Custom Exceptions in Python")
```

Example:

```
class MyError(Exception):

   # Constructor or Initializer
   def __init__(self, value):
      self.value = value

   # __str__ is to print() the value
   def __str__(self):
      return(repr(self.value))

try:
   raise(MyError(3*2))
```

Some important built-in Exceptions in Python:

1. **ImportError:**
   Raised when the imported module is not found.
2. **IndexError:**
   Raised when the index of a sequence is out of range.
3. **KeyError:**
   Raised when a key is not found in a dictionary.
4. **KeyboardInterrupt:**

Raised when the user hits the interrupt key (Ctrl+C or Delete).

5. **RuntimeError:**
   Raised when an error does not fall under any other category.
6. **StopIteration:**
   Raised by next() function to indicate that there is no further item to be returned by iterator.
7. **SyntaxError:**
   Raised by parser when syntax error is encountered.
8. **TypeError:**
   Raised when a function or operation is applied to an object of incorrect type.
9. **ValueError:**
   Raised when a function gets an argument of correct type but improper value.
10. **NameError:**
    Raised when a variable is not found in local or global scope.
11. **AttributeError:**
    Raised when attribute assignment or reference fails.
12. **ZeroDivisionError:**
    Raised when the second operand of division or modulo operation is zero.

Implementation and Output:

1. Display built-in exceptions:

Code and Output:

```
1. ZeroDivisionError:
>>> n = 9
>>> m = 0
>>> n / m
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

2. ImportError / ModuleNotFoundError:
>>> import xyz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'xyz'

3. IndexError:
>>> list_data = [1, 2, 3, 4, 5]
>>> x = list_data[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

4. KeyboardInterrupt:

```
>>> name = input("Enter your name: ")
Enter your name: Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

5. KeyError:

```
>>> dict_data = {'2':'Two', '4':'Four', '6':'Six'}
>>> dict_data['5']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '5'
```

6. NameError:

```
>>> Names = ['Vijay', 'Vidhi', 'Rohan']
>>> names.lower()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'names' is not defined. Did you mean: 'Names'?
```

7. AttributeError:

```
>>> Names = ['Vijay', 'Vidhi', 'Rohan']
>>> Names.lower
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'lower'
```

8. TypeError:

```
>>> Names = ['Vijay', 'Vidhi', 'Rohan']
>>> Names / 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

9. StopIteration:

```
>>> list_data = iter([1, 2, 3, 4])
>>> print(next(list_data))
1
>>> print(next(list_data))
2
>>> print(next(list_data))
3
>>> print(next(list_data))
4
>>> print(next(list_data))
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
StopIteration


10. SyntaxError:
>>> n1 = 10
>>> n2 = 20
>>> if n1 > n2
  File "<stdin>", line 1
    if n1 > n2
                ^
SyntaxError: expected ':'
```

2. Catch and handle built-in exceptions:

Code:

```
import math

try:
    imp = "import "+ input("Enter the module to be imported: ")
    exec(imp)
    n = {}
    n1, n2, k1, k2 = eval(input("Enter two numbers and their keys separated by comma (n1, n2, k1, k2):
"))
    n[k1] = n1
    n[k2] = n2
    k1, k2 = eval(input("Enter the keys of numbers separated by comma: "))
    result = n[k1] / n[k2]
    m = int(input("Enter one more number: "))
    print("Result is:", result / math.sqrt(m))
    if (n[k1] == 0):
        raise RuntimeError
except ValueError:
    print("Invalid literal.")
except ImportError:
    print("Module not found.")
except ZeroDivisionError:
    print("Division by zero.")
except SyntaxError:
    print("Comma missing.")
except RuntimeError:
    print("May be meaningless.")
except KeyboardInterrupt:
    print()
    print("Program was interrupted.")
except KeyError:
    print("The requested key wasn't found.")
```

```
except:
   print("Something wrong in input.")
else:
   print("No exceptions.")
finally:
   print("Finally call is executed.")
```

Output:

1. Normal execution without exceptions:

```
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): 2, 3, 0, 1
Enter the keys of numbers separated by comma: 0, 1
Enter one more number: 10
Result is: 0.21081851067789192
No exceptions.
Finally call is executed.
```

2. ValueError:

```
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): 2, 3, 0, 1
Enter the keys of numbers separated by comma: 0, 1
Enter one more number: -10
Invalid literal.
Finally call is executed.
```

3. ImportError:

```
Enter the module to be imported: xyz
Module not found.
Finally call is executed.
```

4. ZeroDivisionError:

```
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): 12, 0, 0, 1
Enter the keys of numbers separated by comma: 0, 1
Division by zero.
Finally call is executed.
```

5. SyntaxError:

```
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): 2 3 0 1
Comma missing.
Finally call is executed.
```

6. RuntimeError:

```
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): 0, 2, 0, 1
```

```
Enter the keys of numbers separated by comma: 0, 1
Enter one more number: 10
Result is: 0.0
May be meaningless.
Finally call is executed.

7. KeyboardInterrupt:
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2):
Program was interrupted.
Finally call is executed.

8. KeyError:
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): 2, 3, 0, 1
Enter the keys of numbers separated by comma: 4, 5
The requested key wasn't found.
Finally call is executed.

9. Other error:
Enter the module to be imported: math
Enter two numbers and their keys separated by comma (n1, n2, k1, k2): Vijay, Vidhi, 0, 1
Something wrong in input.
Finally call is executed.
```

3. Creating a user-defined Exception Handling mechanism:

Code:

```
class BaseError(Exception):
   pass

class HighValueError(BaseError):
   pass

class LowValueError(BaseError):
   pass

value = 29

while(1):
   try:
      n = int(input("Enter a number: "))
      if n > value:
         raise HighValueError
      elif n < value:
         raise LowValueError
   except LowValueError:
```

```
        print("Very low value, give i/p again.")
        print()
    except HighValueError:
        print("Very high value, give i/p again.")
        print()
    else:
        print("Nice! Correct answer!")
        break
```

Output:

```
Enter a number: 35
Very high value, give i/p again.

Enter a number: 15
Very low value, give i/p again.

Enter a number: 29
Nice! Correct answer!
```

Conclusion:

In this experiment, we studied the Exception Handling mechanism provided by Python. Exception Handling is an important part in any program. It helps to ensure proper working of the program so that accurate outputs are achieved, without errors, after executing the program. We also observed that Python offers a way to determine exactly which error has occurred, and also the block in which it occurs. It does this through the try…except…else…finally block. This helps is quick debugging of the program. We also observed that, in Python, custom exceptions can be defined and used by the users according to the functionality of their program. We also studied various built-in exceptions like ValueError, KeyError, NameError, KeyboardInterrupt, ZeroDivisionError, IndexError, ImportError, TypeError, AttributeError, RuntimeError, SyntaxError, StopIteration, etc., and we also observed that such built-in exceptions can be raised in the program when necessary, using the raise keyword. Hence, we can conclude that Python offers a reliable and efficient Exception Handling mechanism for effective exception detection, handling and debugging.

Experiment 6

Shashwat Shah
60004220126
C2-2 Div B

Aim : To implement file handling

Theory : Python supports file handling and allows users to handle files i.e to read and write files, along with many other file handling options, to operate files. The concept of file handling has streched over various other languages. But the implementations is either complicated or lengthy, but like other concepts of python, this concepts of python is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters and they form a text file. Each line of a file is terminated with a special character called the EOL or the end of line characters like comma (,) or newline character. It ends the current line and tells the interpreter a new one has begun. Lets start with reading and writing files. These are various modules supported in python file handling features

(1) r : Open an existing file for a read operation.

(2) w : Open an existing file for a write operation. If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.

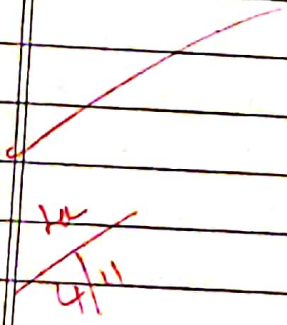
(3) a : Open an existing file or append operation. It won't override existing data.

(4) r+ : To read and write data into the file. The previous data in the file will be overridden

(5) w+ : To write and read data. It will override existing data

(6) a+ : To append and read data from a file. It won't override existing data

Conclusion: Therefore, we learned file handling and how to manipulate files in python.



Experiment No. 6
File Handling in Python

Aim:
To study and implement File Handling in Python.

Description:
File Handling:
Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but like other concepts of Python, this concept here is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters, and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let's start with the reading and writing files.

File open() method:
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:

1. "r" - Read - Default value. Opens a file for reading, error if the file does not exist
2. "a" - Append - Opens a file for appending, creates the file if it does not exist
3. "w" - Write - Opens a file for writing, creates the file if it does not exist
4. "x" - Create - Creates the specified file, returns an error if the file exists

In addition, one can specify if the file should be handled as binary or text mode:

1. "t" - Text - Default value. Text mode
2. "b" - Binary - Binary mode (e.g., images)

Example:
```
f = open("demofile.txt", "rt")
```

Open a file on the server:
To open the file, we 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
Example:
```
f = open("demofile.txt", "r")
print(f.read())
```
If the file is located in a different location, one will have to specify the file path.
Example:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read only parts of the file:

By default the read() method returns the whole text, but one can also specify how many characters one wants to return.

Example:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read lines:

A single line can be entered by using the readline() method.

Example:

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

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

Example:

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

By looping through the lines of the file, one can read the whole file, line by line.

Example:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close files:

It is a good practice to always close the file when one is done with it.

Example:

```
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

Example:

```
f = open("demofile2.txt", "a"):
f = open("demofile2.txt", "r")
```

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

Example:

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

Delete a file:

To delete a file, one must import the OS module, and run its os.remove() function.

Example:

```
import os
os.remove("demofile.txt")
```

Check if file exists:

To avoid getting an error, one might want to check if the file exists before one tries to delete it.

Example:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete folder:

To delete an entire folder, use the os.rmdir() method.

Example:

```
import os
os.rmdir("myfolder")
```

Implementation and Output:

A] Take 10 numbers from the user. Add it to a file (lets say T1.txt). Read the contents of the file and sort the data. Put the sorted data in a different file (T2.txt)

Code:

```
n = int(input("Enter the count of numbers: "))

print("Enter", n, "numbers:")

with open("T1.txt", "w") as f:
    for i in range(n - 1):
        f.write(str(input()) + "\n")
    f.write(str(input()))
```

```
with open("T1.txt", "r") as f1, open("T2.txt", "w") as f2:
    l = list()
    for line in f1:
        l.append(int(line))
    l.sort()
    for i in range(n - 1):
        f2.write(str(l[i]) + "\n")
    f2.write(str(l[n - 1]))
```

Output:

Output in the terminal:

```
Enter the count of numbers: 10
Enter 10 numbers:
5
4
7
6
9
8
0
1
3
2

Contents of T1.txt before sorting:
5
4
7
6
9
8
0
1
3
2

Contents of T2.txt after sorting:
0
1
2
3
4
5
6
7
8
9
```

B] Take a T1.txt file with several words in it. Sort the words lexicographically and put the sorted words in a different file T2.txt.

Code:

```
fileRead = open("T1.txt", "r")
List = list()
f = fileRead.read()
f = f.split()

for line in f:
    List.append(str(line))
    List.sort(key=lambda item: (item, len(item)))
print("List of sorted words is:")
print(List)

fileWrite = open("T2.txt", "w")
for i in range(len(List)):
    fileWrite.write(str(List[i]))
    fileWrite.write("\n")
```

Output:

Content of T1.txt (original text):
Hello I am Vijay Harkare a musician with 13 years of experience

Output in the terminal:

```
List of sorted words is:
['13', 'Harkare', 'Hello', 'I', 'Vijay', 'a', 'am', 'experience', 'musician', 'of', 'with', 'years']
```

Content of T2.txt after sorting:

```
13
Harkare
Hello
I
Vijay
a
am
experience
musician
of
with
years
```

C] Take a T1.txt file with several words in it. Reverse each word and put the reversed words in order in a different file T2.txt.

Code:

```
fileRead = open("T1.txt", "r")
List = list()
f = fileRead.read()
f = f.split()

for line in f:
    List.append(str(line)[::-1])
    List.sort(key=lambda item: (item, len(item)))

print("List of reversed words in order is:")
print(List)

fileWrite = open("T2.txt", "w")
for i in range(len(List)):
    fileWrite.write(str(List[i]))
    fileWrite.write("\n")
```

Output:

Content of T1.txt before reversing the words and ordering them:
Hello I am Vijay Harkare a musician with 13 years of experience

Output in the terminal:

```
List of reversed words in order is:
['31', 'I', 'a', 'ecneirepxe', 'erakraH', 'fo', 'htiw', 'ma', 'naicisum', 'olleH', 'sraey', 'yajiV']
```

Content of T2.txt after reversing the words and ordering them:

```
31
I
a
ecneirepxe
erakraH
fo
htiw
ma
naicisum
olleH
sraey
yajiV
```

Conclusion:

In this experiment, we studied and implemented File Handling in Python. Many operations in real-world applications require the handling and manipulation of files. Therefore, File Handling plays an important role in many applications. We observed that Python offers a wide range of classes and functions to work with files. Some of the common functions used were open(), read(), readline() and close(). We observed that these functions are characterized by a number of parameters which specify the mode in which a particular file is to be worked with, which facilitates working with multiple files in different modes. Some of these modes include read (r), write (w), append (a), etc. Also, a single file can be opened in different modes simultaneously, which can be achieved using the r+, a+ and w+ modes. We can also perform operations on the content of the file by reading the content of the file by read() or readline() functions. Thus, we can conclude that Python offers a versatile and effective File Handling mechanism, which helps in working with different types of files, and manipulating the content in them as needed.

Experiment 7

Shashwat Shah
60004220126
C 2-2 Div B

Aim: To understand and implement regular expressions in python.

Theory: A regular expression is a special sequence of characters that uses a search pattern to find a string or set of strings. It can detect the presence or absence of a text by matching it with a particular pattern, and also can split a pattern into one or more sub patterns.

(1) re module - Supports the use of regex in python. Its primary function is to offer a search, where it takes a regular expression and a string

(2) re.findall(): Return all non-overlapping matches of a pattern in string as a list of string. The string is scanned left to right and matches are returned in the order found.

(3) re.compile(): Regular expressions are compiled into pattern objects which have methods for various operations such as searching for pattern matches or performing string substitutions

(4) re.split(): Split string by the occurances of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

P.T.O

Special Sequence	Description	
① \A	Matches if the string begins with the given character	\Afor
② \b	Matches if the word begins or ends with the given characters. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word	\bge
③ \B	It is the opposite of \b i.e. the string should not start or end with the given regex.	\Bge
④ \d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d
⑤ \D	Matches any non-digit character, this is equivalence to set class [0-9]	\D
⑥ \s	Matches any whitespace character.	\s
⑦ \S	Matches any non-whitespace character.	\S
⑧ \w	Matches any alpha numeric character	\w
⑨ \W	Matches any non alpha numeric character	\W
⑩ \z	Matches if the string ends with given regex.	\z

Conclusion: Therefore we have implemented regular expressions in Python.

14/8/24

Experiment No. 7
Regular Expression in Python

Aim:
To study and implement Regular Expression in Python.

Description:
Regular Expression (RegEx):
A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
RegEx can be used to check if a string contains the specified search pattern.

RegEx Module:
Python has a built-in package called re, which can be used to work with Regular Expressions.
The module needs to be imported before use.
Syntax:
import re

RegEx in Python:
After importing the re module, we can start using regular expressions.
Example:
```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

RegEx Functions:
The re module offers a set of functions that allows us to search a string for a match:

1. **findall()**: Returns a list containing all matches
2. **search()**: Returns a Match object if there is a match anywhere in the string
3. **split()**: Returns a list where the string has been split at each match
4. **sub()**: Replaces one or many matches with a string

Metacharacters:
Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"

+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"
()	Capture and group	"()"

Special Sequences:

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Sets:

A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Descripton
[arn]	Returns a match where one of the specified characters (a, r, or n) is present
[a-n]	Returns a match for any lower case character, alphabetically between a and n

[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

The findall() function:

The findall() function returns a list containing all matches.

Example:

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned.

Example:

```
import re

txt = "The rain in Spain"
x = re.findall("Portugal", txt)
print(x)
```

The search() function:

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned.

Example:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())
```

If no matches are found, the value None is returned.

Example:

```
import re

txt = "The rain in Spain"
```

```
x = re.search("Portugal", txt)
print(x)
```

The split() function:

The split() function returns a list where the string has been split at each match.

Example:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

One can control the number of occurrences by specifying the maxsplit parameter.

Example:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

The sub() function:

The sub() function replaces the matches with the text of your choice.

Example:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

One can control the number of replacements by specifying the count parameter.

Example:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

Match Object:

A Match Object is an object containing information about the search and the result.
If there is no match, the value None will be returned, instead of the Match Object.

Example:

```
import re
```

```
txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) #this will print an object
```

The Match object has properties and methods used to retrieve information about the search, and the result:

1. .span() returns a tuple containing the start-, and end positions of the match.
2. .string returns the string passed into the function
3. .group() returns the part of the string where there was a match

Example:

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
print(x.string)
print(x.group())
```

Implementation and Output:

Conisder a Text File consisting of following data:

Mr. Anderson
Ms. Thareja
Mrs. Morris
Mr. Roy
Ms. Gandhi
Mrs. Modi
https://www.google.com
http://www.udemy.com
www.udacity.com
https://www.stackoverflow.com
http://www.djsce.ac.in
https://plus.google.com
rishit.grover@gmail.com
kapeesh.grover@yahoo.co.in
abhishek.shah@gmail.com
shahp98@gmail.com
demo_user@gmail.com
rolflmoa@yahoo.co.in
27777647
233*333*88
455-78-888
022-240-93836

02642*221*381

Use regular expression for the above text to find:

- **Names of the User.**
- **Website name excluding http/s**
- **Identify email ids**
- **Identify Phone numbers**

Code:

```
import re

file = open("samples.txt")
text = file.read()

# for names
pattern_names = r'M\w*.\s*\w*\s*\w*[\n]'
pattern_email = r'[a-zA-Z0-9\.\-+_]+@[a-zA-Z0-9\.\-+_]+\.[a-zA-Z]+'

names1 = re.findall(pattern_names, text)
names = []
for i in names1:
    names.append(i[:-1])

print("Names are:")
for i in names:
    print(i)
print()

# for website addresses
pattern_web = r"(?i)\b((?:https?://|www\d{0,3}[.]|[a-z0-9.\-]+[.][a-
z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\)))*\))+(?:\(([^\s()<>]+|(\([^\s()<>]+\)))*\)|[^\s`!()\[\]
{};:'\".,<>?«»“”‘’]))"

websites = re.findall(pattern_web, text)

print("Websites are:")
for i in websites:
    if 'https://' in i[0]:
        print(i[0][8:])
    elif 'http://' in i[0]:
        print(i[0][7:])
    else:
        print(i[0])
print()

# for email addresses
```

```
emails = re.findall(pattern_email, text)

print("Email addresses are:")
for i in emails:
    print(i)
print()

pattern_username = r'\S+.@'
pattern_domain = r'@\S+.'

usernames = re.findall(pattern_username, ' '.join(emails))
domains = re.findall(pattern_domain, ' '.join(emails))

print("Usernames and domains of each of the email addresses are:")
for i, j in zip(usernames, domains):
    print("User Id.: ", i[:-1], "; Domain: ", j[1:], sep="")
print()

# for numbers
pattern_number = r'[0-9]+[#\-*]*[0-9]+[#\-*]*[0-9]+'
numbers = re.findall(pattern_number, text)

print("Phone numbers are:")
for i in numbers:
    print(i)
```

Output:

```
Names are:
Mr. Anderson
Ms. Thareja
Mrs. Morris
Mr. Roy
Ms. Gandhi
Mrs. Modi

Websites are:
www.google.com
www.udemy.com
www.udacity.com
www.stackoverflow.com
www.djsce.ac.in
plus.google.com

Email addresses are:
rishit.grover@gmail.com
kapeesh.grover@yahoo.co.in
```

```
abhishek.shah@gmail.com
shahp98@gmail.com
demo_user@gmail.com
rolflmoa@yahoo.co.in

Usernames and domains of each of the email addresses are:
User Id.: rishit.grover; Domain: gmail.com
User Id.: kapeesh.grover; Domain: yahoo.co.in
User Id.: abhishek.shah; Domain: gmail.com
User Id.: shahp98; Domain: gmail.com
User Id.: demo_user; Domain: gmail.com
User Id.: rolflmoa; Domain: yahoo.co.in

Phone numbers are:
27777647
233*333*88
455-78-888
022-240-93836
02642*221*381
```

Conclusion:

In this experiment, we studied and implemented Regular Expressions in Python. Regular Expressions, i.e., RegEx in Python allow the user to detect certain patterns in text and are also useful in pattern matching problems. Regular Expressions have wide applications in the fields of authorization, natural language processing, data validation, data scraping (especially web scraping), etc. We observed that Python offers different functions like findall(), search(), split() and sub() which help in pattern recognition and matching using Regular Expressions. We also observed that Python offers a variety of metacharacters and special sequences along with set and logical functionalities, which help in matching diverse and complex patterns as per the need of the user. We tested Regular Expressions to identify names, email addresses, website names and phone numbers, and we observed that, using the above-mentioned functionalities of Regular Expressions, we were able to identify each of the examples accurately and completely. Thus, we can conclude that Regular Expression in Python is an effective and efficient tool which helps to identify and match complex patterns in the provided text.

Experiment 8

Shashwat Shah
60004220126
C 2-2 Div B

Aim: To implement database connectivity in python

Theory:

MySQL connector/python enables python programs to access MySQL database, using an API that is compliant with the python database API specification V2.0 (PEP 249). It is written in pure python and does not have any dependencies except for the python standard libraries

Steps to create connection of python application to database.

1. Import my SQL - connector module.
2. Create the connection object.
3. Create the cursor object.
4. Execute the query.

Creating cursor objects.

The cursor objects can be defined as an abstraction specified in the python DB-API 2.0. It facilitates us to have multiple separate working environments through the same connection to the database. We can create the cursor object by calling the 'cursor' function of the connection object. The cursor object is an important aspect of executing queries to the databases.

Conclusion: Therefore, we have implemented database connectivity using Python.

4 2/12

Experiment No. 8
Database Connectivity in Python

Aim:
To study and implement Database Connectivity in Python.

Description:
Database Connectivity in Python using MySQL:
Python can be used in database applications.
One of the most popular databases is MySQL.
Using the MySQL Connector in Python, there are the following steps to connect a Python application to our database:

1. Import mysql.connector module
2. Create the connection object.
3. Create the cursor object
4. Execute the query

Creating Connection:
To create a connection between the MySQL database and the python application, the connect() method of mysql.connector module is used.
Pass the database details like HostName, username, and the database password in the method call. The method returns the connection object.
The syntax to use the connect() is given below.
Syntax:
Connection-Object= mysql.connector.connect(host = <host-name> , user = <username> , passwd = <password>)

Creating a Cursor Object:
The cursor object can be defined as an abstraction specified in the Python DB-API 2.0. It facilitates us to have multiple separate working environments through the same connection to the database. We can create the cursor object by calling the 'cursor' function of the connection object. The cursor object is an important aspect of executing queries to the databases.
The syntax to create the cursor object is given below.
Syntax:
<my_cur> = conn.cursor()

Creating Database:
To create a database in MySQL, use the "CREATE DATABASE" statement.
Syntax:
mycursor.execute("CREATE DATABASE mydatabase")

Checking if Database exists:

We can check if a database exists by listing all databases in our system by using the "SHOW DATABASES" statement:

Syntax:

mycursor.execute("SHOW DATABASES")

We can also try to access the database when making the connection. If the database does not exist, we will get an error.

Creating a Table:

To create a table in MySQL, we need to use the "CREATE TABLE" statement.

We need to make sure that we define the name of the database when we create the connection.

Syntax:

mycursor.execute("CREATE TABLE <table_name> (column1 type1, column2 type2, …)")

Checking if Table exists:

We can check if a table exist by listing all tables in our database with the "SHOW TABLES" statement.

Syntax:

mycursor.execute("SHOW TABLES")

Inserting into Table:

To fill a table in MySQL, use the "INSERT INTO" statement.

Example:

```
sql = "INSERT INTO <table_name> (column1, column2, …) VALUES (%s, %s, …)"
val = (value1, value2, …)
mycursor.execute(sql, val)
```

Inserting Multiple Rows:

To insert multiple rows into a table, use the executemany() method.

The second parameter of the executemany() method is a list of tuples, containing the data we want to insert.

Syntax:

```
sql = "INSERT INTO <table_name> (column1, column2, …) VALUES (%s, %s, …)"
val = [
        (value11, value12, …),
        (value21, value22, …),
        (value31, value32, …),
        …
]
mycursor.executemany(sql, val)
```

Selecting from a Table:
To select from a table in MySQL, use the "SELECT" statement.
Syntax:
mycursor.execute("SELECT * FROM <table_name>")
myresult = mycursor.fetchall()

Here, we use the fetchall() method, which fetches all rows from the last executed statement.

Selecting Columns:
To select only some of the columns in a table, use the "SELECT" statement followed by the column name(s).
Syntax:
mycursor.execute("SELECT column1, column2, … FROM <table_name>")
myresult = mycursor.fetchall()

Using the fetchone() method:
If we are only interested in one row, we can use the fetchone() method.
The fetchone() method will return the first row of the result:
Syntax:
myresult = mycursor.fetchone()

Selecting with a Filter:
When selecting records from a table, we can filter the selection by using the "WHERE" statement.
Syntax:
sql = "SELECT * FROM <table_name> WHERE <condition>"
mycursor.execute(sql)
myresult = mycursor.fetchall()

Using Wildcard Characters:
We can also select the records that starts, includes, or ends with a given letter or phrase.
Use the % to represent wildcard characters.
Syntax:
sql = "SELECT * FROM tab_name WHERE column_name LIKE 'X%'"
OR,
sql = "SELECT * FROM tab_name WHERE column_name LIKE '%X'"
OR,
sql = "SELECT * FROM tab_name WHERE column_name LIKE 'X%X'"

Deleting a Record:

We can delete records from an existing table by using the "DELETE FROM" statement.

Syntax:

sql = "DELETE FROM <table_name> WHERE <condition>"
mycursor.execute(sql)
mydb.commit()

Updating Table:

We can update existing records in a table by using the "UPDATE" statement.

Syntax:

sql = "UPDATE <table_name> SET <column> = <value> WHERE <condition>"
mycursor.execute(sql)
mydb.commit()

Altering Table:

We can alter the existing table by using the "ALTER" statement.

Syntax:

mycursor.execute("ALTER TABLE <table_name> [alter option [, alter option] ...]")
mydb.commit()

Deleting a Table:

We can delete an existing table by using the "DROP TABLE" statement.

Syntax:

sql = "DROP TABLE <table_name>"
mycursor.execute(sql)
mydb.commit()

Deleting a Database:

We can delete an existing table by using the "DROP DATABASE" statement.

Syntax:

sql = "DROP DATABASE <database_name>"
mycursor.execute(sql)
mydb.commit()

Implementation and Output:

1. Create Database:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
)

mycursor = mydb.cursor()
mycursor.execute("create database furniture")
```

Output:

(No visible output. Database gets created.)

2. Check if Database exists:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021'
)

mycursor = mydb.cursor()
mycursor.execute("show databases")
for x in mycursor:
   print(x)
```

Output:

```
('company',)
('furniture',)
('information_schema',)
('music_library',)
('music_library1',)
('music_library2',)
('mysql',)
('part',)
('performance_schema',)
('sakila',)
('sys',)
('test_db',)
```

```
('world',)
```

3. Create Table:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = "furniture"
)

mycursor = mydb.cursor()
mycursor.execute("""create table cupboard(
   id int(15) not null,
   name varchar(75) not null,
   price float not null,
   count int not null,
   primary key(id)
)""")
```

Output:

(No visible output. Table gets created.)

4. Check if Table exists:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("show tables")
for x in mycursor:
   print(x)
```

Output:

```
('cupboard',)
```

5. Insert values (Show examples for Single Row and Multiple Rows):

Code:

1. Single Row:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("""insert into cupboard(id, name, price, count)
values (6, 'cb31', 7500, 8)""")
mydb.commit()
print(mycursor.rowcount, 'record inserted.')
```

2. Multiple Rows:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()

sql = """insert into cupboard(id, name, price, count) values (%s, %s, %s, %s)"""

val = [
   (2, 'cb2', 15000, 5),
   (3, 'cb3', 7000, 15),
   (4, 'cb4', 20000, 3),
   (5, 'cb5', 40000, 5)
]

mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, 'record(s) inserted.')
```

Output:

1. Single Row:

```
1 record inserted.
```

```
2. Multiple Rows:
4 record(s) inserted.


Table after insertion:
(2, 'cb2', 15000.0, 5)
(3, 'cb3', 7000.0, 15)
(4, 'cb4', 20000.0, 3)
(5, 'cb5', 40000.0, 5)
(6, 'cb31', 7500.0, 8)
```

6. Delete a row based on values:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
    host = 'localhost',
    user = 'root',
    password = 'decViz@2021',
    database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("delete from cupboard where name = 'cb31'")
mydb.commit()
print(mycursor.rowcount, 'record(s) deleted.')
```

Output:

```
1 record(s) deleted.

Table after deletion:
(2, 'cb2', 15000.0, 5)
(3, 'cb3', 7000.0, 15)
(4, 'cb4', 20000.0, 3)
(5, 'cb5', 40000.0, 5)
```

7. Display the rows of the table:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
    host = 'localhost',
    user = 'root',
    password = 'decViz@2021',
    database = 'furniture'
```

```
)

mycursor = mydb.cursor()
mycursor.execute("select * from cupboard")

myresult = mycursor.fetchall()

for x in myresult:
   print(x)a
```

Output:

```
(2, 'cb2', 15000.0, 5)
(3, 'cb3', 7000.0, 15)
(4, 'cb4', 20000.0, 3)
(5, 'cb5', 40000.0, 5)
```

8. Select specific Columns:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("select id, price from cupboard")

myresult = mycursor.fetchall()

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

Output:

```
(2, 15000.0)
(3, 7000.0)
(4, 20000.0)
(5, 40000.0)
```

9. Use the fetchone() method:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
```

```
  host = 'localhost',
  user = 'root',
  password = 'decViz@2021',
  database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("select id, price from cupboard")

myresult = mycursor.fetchone()

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

Output:
2
15000.0

10. Update the values of a specific row:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
  host = 'localhost',
  user = 'root',
  password = 'decViz@2021',
  database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("update cupboard set price = 8500 where name = 'cb3'")
mydb.commit()
print(mycursor.rowcount, 'record(s) updated.')
```

Output:
1 record(s) updated.

Table after the update:
(2, 'cb2', 15000.0, 5)
(3, 'cb3', 8500.0, 15)
(4, 'cb4', 20000.0, 3)
(5, 'cb5', 40000.0, 5)

11. Search whether a particular record is present in the table or not:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("select * from cupboard where name = 'cb4'")

myresult = mycursor.fetchall()

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

Output:

```
(4, 'cb4', 20000.0, 3)
```

12. Use Wildcard Characters:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("select * from cupboard where name like '%3%'")

myresult = mycursor.fetchall()

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

Output:

```
(3, 'cb3', 8500.0, 15)
```

13. Alter the table by adding new column:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("alter table cupboard add year int(4)")
mydb.commit()
```

Output:

```
(No visible output. Column added.)
```

Table after addition of Column (None value is for the new column, year):

```
(2, 'cb2', 15000.0, 5, None)
(3, 'cb3', 8500.0, 15, None)
(4, 'cb4', 20000.0, 3, None)
(5, 'cb5', 40000.0, 5, None)
```

14. Delete the table:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("drop table cupboard")
mydb.commit()
```

Output:

```
(No visible output. Table deleted.)
```

List of Tables after deletion:

```
(No output, since the database contained only one table.)
```

15. Delete the database:

Code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host = 'localhost',
   user = 'root',
   password = 'decViz@2021',
   database = 'furniture'
)

mycursor = mydb.cursor()
mycursor.execute("drop database furniture")
mydb.commit()
```

Output:

(No visible output. Database deleted.)

List of Databases after deletion:

```
('company',)
('information_schema',)
('music_library',)
('music_library1',)
('music_library2',)
('mysql',)
('part',)
('performance_schema',)
('sakila',)
('sys',)
('test_db',)
('world',)
```

Conclusion:

In this experiment, we studied and implemented Database Connectivity in Python. We used the MySQL database for demonstrating above-mentioned task. We observed how conveniently Python scripts could be written with the help of the mysql.connector module, which could interact with the user's databases on localhost. We studied and observed the effect of various MySQL queries for creating database and table, inserting records in the table, updating the records in the table, altering the table, displaying the records in the table with and without filters or wildcard characters, deleting the table and database, etc. We observed that, in Python, to execute MySQL queries using mysql.connector module, it is needed to first establish connection with the host, and define the connection and the cursor objects, which then enable us to work with our databases. Thus, we can conclude that the mysql.connector module in Python is a simple, convenient and robust tool for implementing Database Connectivity in Python using MySQL.

Experiment 9

Shashwat Shah
60004220126
C2-2 Div B

Aim: Design a GUI application to show input and output operations using Tkinter. (Calculator)

Theory:

Tkinter is the most commonly used library for developing GUI (Graphical user interface) in Python. It is a standard python interface to the Tk GUI toolkit shipped with python. As Tk and Tkinter are available on most of the Unix platforms as well as on the windows system, developing GUI applications with Tkinter becomes the fastest and easiest.

Steps to create tkinter module.

1. Import tkinter module.
2. Create the GUI application main window
3. Add one or more of the above mentioned widgets to the GUI application.
4. Enter the main event loop to make action against each event triggered by the user.

Tkinter widgets

a) Button - Display buttons in your applications
b) Label - Provide a single line caption for other widgets
c) Text - Display text in multiple lines.
d) Frame - Used as a container widget to organize other widgets.

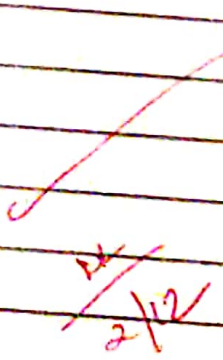
Geometry Management.

a) pack() method - organises widgets in blocks before placing them in the parent widget.

b) grid() method - organizes widgets in a table-like structure in the parent widget.

c) place() method - organise widgets by placing them in a specific in parent widget.

Conclusion - Therefore, we have implemented / designed a calculator using tkinter



Experiment No. 9
GUI using Tkinter in Python

Aim:
To study and implement GUI using Tkinter in Python.

Description:
GUI in Python:
Modern computer applications are user-friendly. User interaction is not restricted to console-based 1/0. They have a more ergonomic graphical user interface (GUI) thanks to high-speed processors and powerful graphics hardware. These applications can receive inputs through mouse clicks and can enable the user to choose from alternatives with the help of radio buttons, dropdown lists, and other GUI elements (or widgets).

Such applications are developed using one of various graphics libraries available. A graphics library is a software toolkit having a collection of classes that define a functionality of various GUI elements. These graphics libraries are generally written in *CIC++*. Many of them have been ported to Python in the form of importable modules. Some of them are listed below:

1. Tkinter is the Python port for Tcl-Tk GUI toolkit developed by Fredrik Lundh. This module is bundled with standard distributions of Python for all platforms.
2. PyQtis, the Python interface to Qt, is a very popular cross-platform GUI framework.
3. PyGTK is the module that ports Python to another popular GUI widget toolkit called GTK.
4. WxPython is a Python wrapper around WxWidgets, another cross-platform graphics library.

Basic GUI Application using Tkinter:
GUI elements and their functionality are defined in the Tkinter module.

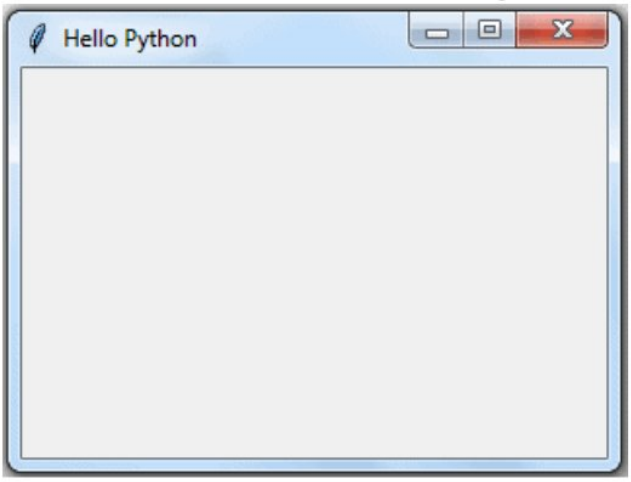
Example:
The following code demonstrates the steps in creating a UI:

```
from tkinter import *
window=Tk()
# add widgets here

window.title('Hello Python')
window.geometry("300x200+10+20")
window.mainloop()
```

1. First of all, import the TKinter module.
2. After importing, setup the application object by calling the Tk() function. This will create a top-level window (root) having a frame with a title bar, control box with the minimize and close buttons, and a client area to hold other widgets.

3. The geometry() method defines the width, height and coordinates of the top left corner of the frame as below (all values are in pixels):
   window.geometry("widthxheight+XPOS+YPOS")
4. The application object then enters an event listening loop by calling the mainloop() method.
5. The application is now constantly waiting for any event generated on the elements in it.
6. The event could be text entered in a text field, a selection made from the dropdown or radio button, single/double click actions of mouse, etc.
7. The application's functionality involves executing appropriate callback functions in response to a particular type of event.
8. The event loop will terminate as and when the close button on the title bar is clicked.

The above code will create the following window:



All Tkinter widget classes are inherited from the Widget class.

Buttons:
The button can be created using the Button class. The Button class constructor requires a reference to the main window and to the options.

Signature:
Button(window, attributes)

We can set the following important properties to customize a button:

1. **text:** caption of the button
2. **bg:** background colour
3. **fg:** foreground colour
4. **font:** font name and size
5. **image:** to be displayed instead of text
6. **command:** function to be called when clicked

Example:

```
from tkinter import *
window=Tk()
btn=Button(window, text="This is Button widget", fg='blue')
btn.place(x=80, y=100)
window.title('Hello Python')
window.geometry("300x200+10+10")
window.mainloop()
```

Label:

A label can be created in the UI in Python using the Label class. The Label constructor requires the top-level window object and options parameters. Option parameters are similar to the Button object.

Example:

The following adds a label in the window:

```
from tkinter import *
window=Tk()
lbl=Label(window, text="This is Label widget", fg='red', font=("Helvetica", 16))
lbl.place(x=60, y=50)
window.title('Hello Python')
window.geometry("300x200+10+10")
window.mainloop()
```

Here, the label's caption will be displayed in red colour using Helvetica font of 16-point size.

Entry:

This widget renders a single-line text box for accepting the user input. For multi-line text input use the Text widget. Apart from the properties already mentioned, the Entry class constructor accepts the following:

1. **bd:** border size of the text box; default is 2 pixels.
2. **show:** to convert the text box into a password field, set show property to "*".

Syntax:

The following code adds the text field:

```
txtfld=Entry(window, text="This is Entry Widget", bg='black',fg='white', bd=5)
```

Example:

The following example creates a window with a button, label and entry field:

```
from tkinter import *
window=Tk()
btn=Button(window, text="This is Button widget", fg='blue')
```

```
btn.place(x=80, y=100)
lbl=Label(window, text="This is Label widget", fg='red', font=("Helvetica", 16))
lbl.place(x=60, y=50)
txtfld=Entry(window, text="This is Entry Widget", bd=5)
txtfld.place(x=80, y=150)
window.title('Hello Python')
window.geometry("300x200+10+10")
window.mainloop()
```

The above example will create the following window:

Selection Widgets:

Various selection widgets available in Tkinter module are as follows:

1. **Radiobutton:** This widget displays a toggle button having an ON/OFF state. There may be more than one button, but only one of them will be ON at a given time.
2. **Checkbutton:** This is also a toggle button. A rectangular check box appears before its caption. Its ON state is displayed by the tick mark in the box which disappears when it is clicked to OFF.
3. **Combobox:** This class is defined in the ttk module of tkinter package. It populates drop down data from a collection data type, such as a tuple or a list as values parameter.
4. **Listbox:** Unlike Combobox, this widget displays the entire collection of string items. The user can select one or multiple items.

Many more widgets are available in Tkinter, which are listed below:

1. Canvas
2. Entry
3. Frame
4. Label
5. Menubutton
6. Menu
7. Message

8. Scale
9. Scrollbar
10. Text
11. Toplevel
12. Spinbox
13. PanedWindow
14. LabelFrame
15. MessageBox

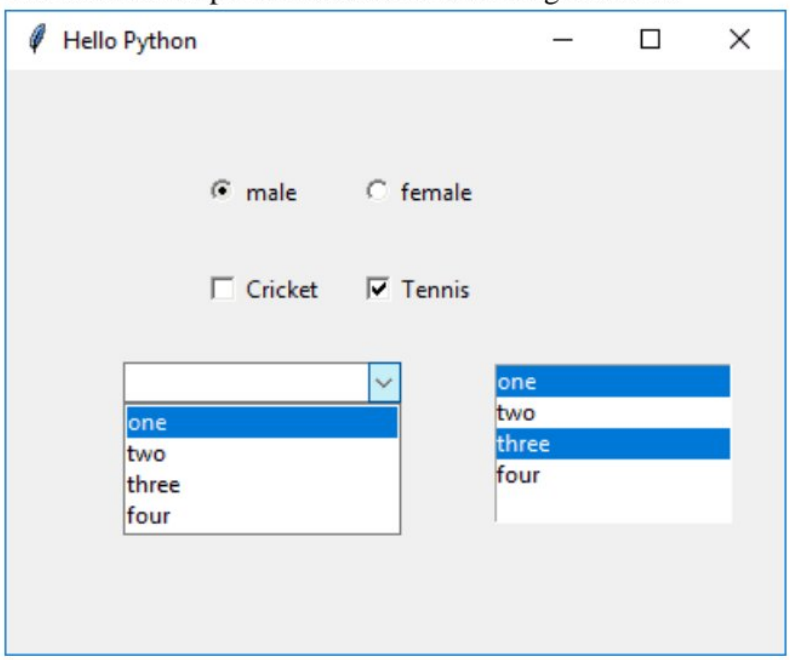
Example:

The following example demonstrates the window with the selection widgets: Radiobutton, Checkbutton, Listbox and Combobox:

```
from tkinter import *
from tkinter.ttk import Combobox
window=Tk()
var = StringVar()
var.set("one")
data=("one", "two", "three", "four")
cb=Combobox(window, values=data)
cb.place(x=60, y=150)

lb=Listbox(window, height=5, selectmode='multiple')
for num in data:
   lb.insert(END,num)
lb.place(x=250, y=150)

v0=IntVar()
v0.set(1)
r1=Radiobutton(window, text="male", variable=v0,value=1)
r2=Radiobutton(window, text="female", variable=v0,value=2)
r1.place(x=100,y=50)
r2.place(x=180, y=50)

v1 = IntVar()
v2 = IntVar()
C1 = Checkbutton(window, text = "Cricket", variable = v1)
C2 = Checkbutton(window, text = "Tennis", variable = v2)
C1.place(x=100, y=100)
C2.place(x=180, y=100)

window.title('Hello Python')
window.geometry("400x300+10+10")
```

```
window.mainloop()
```

The above example will create the following window:



Geometry Management:

Tkinter offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

1. **pack() method:** It organizes the widgets in blocks before placing in the parent widget.
2. **grid() method:** It organizes the widgets in grid (table-like structure) before placing in the parent widget.
3. **place() method:** It organizes the widgets by placing them on specific positions directed by the programmer.

Event Handling:

An event is a notification received by the application object from various GUI widgets as a result of user interaction. The Application object is always anticipating events as it runs an event listening loop. User's actions include mouse button click or double click, keyboard key pressed while control is inside the text box, certain element gains or goes out of focus etc.

Events are expressed as strings in <modifier-type-qualifier> format.

Many events are represented just as qualifier. The type defines the class of the event.

The following table shows how the Tkinter recognizes different events:

Event	Modifier	Type	Qualifier	Action
<Button-1>		Button	1	Left mouse button click.
<Button-2>		Button	2	Middle mouse button click.
<Destroy>		Destroy		Window is being destroyed.
<Double-Button-1>	Double	Button	1	Double-click first mouse button 1.
<Enter>	Enter			Cursor enters window.
<Expose>		Expose		Window fully or partially exposed.
<KeyPress-a>		KeyPress	a	Any key has been pressed.
<KeyRelease>		KeyRelease		Any key has been released.
<Leave>		Leave		Cursor leaves window.
<Print>			Print	PRINT key has been pressed.
<FocusIn>		FocusIn		Widget gains focus.
<FocusOut>		FocusOut		Widget loses focus.

An event should be registered with one or more GUI widgets in the application. If it's not, it will be ignored. In Tkinter, there are two ways to register an event with a widget. First way is by using the bind() method and the second way is by using the command parameter in the widget constructor.

Bind() Method:
The bind() method associates an event to a callback function so that, when the even occurs, the function is called.

Syntax:
Widget.bind(event, callback)

Example:
For example, to invoke the MyButtonClicked() function on left button click, we can use the following code:

```
from tkinter import *
window=Tk()
btn = Button(window, text='OK')
btn.bind('<Button-1>', MyButtonClicked)
```

The event object is characterized by many properties such as source widget, position coordinates, mouse button number and event type. These can be passed to the callback function if required.

Command Parameter:

Each widget primarily responds to a particular type. For example, Button is a source of the Button event. So, it is by default bound to it. Constructor methods of many widget classes have an optional parameter called command. This command parameter is set to callback the function which will be invoked whenever its bound event occurs. This method is more convenient than the bind() method.

Example for Button:

btn = Button(window, text='OK', command=myEventHandlerFunction)

In the example given below, the application window has two text input fields and another one to display the result. There are two button objects with the captions Add and Subtract. The user is expected to enter the number in the two Entry widgets. Their addition or subtraction is displayed in the third.

The first button (Add) is configured using the command parameter. Its value is the add() method in the class. The second button uses the bind() method to register the left button click with the sub() method. Both methods read the contents of the text fields by the get() method of the Entry widget, parse to numbers, perform the addition/subtraction and display the result in third text field using the insert() method.

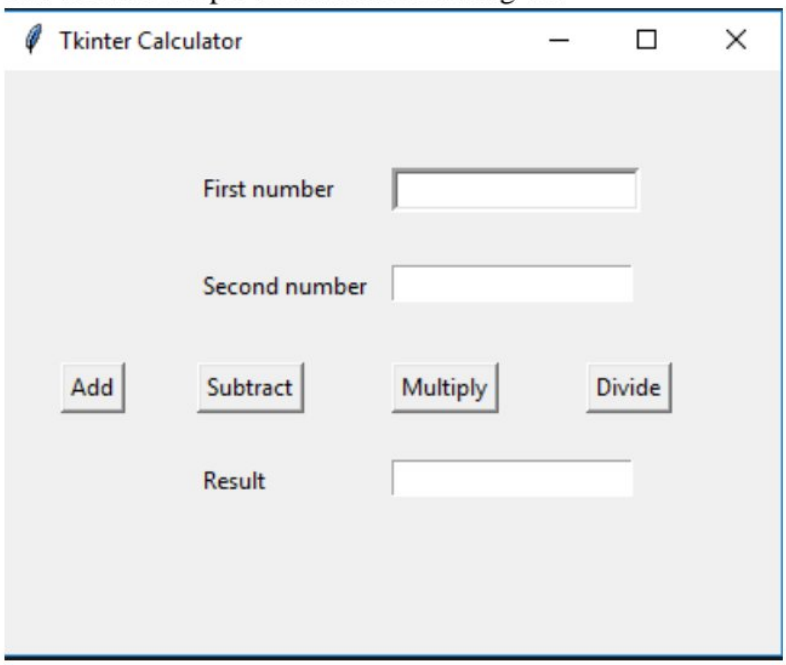
Example:

```
from tkinter import *
class MyWindow:
   def __init__(self, win):
      self.lbl1=Label(win, text='First number')
      self.lbl2=Label(win, text='Second number')
      self.lbl3=Label(win, text='Result')
      self.t1=Entry(bd=3)
      self.t2=Entry()
      self.t3=Entry()
      self.btn1 = Button(win, text='Add')
      self.btn2=Button(win, text='Subtract')
      self.btn3=Button(win, text='Multiply')
      self.btn4=Button(win, text='Divide')
      self.lbl1.place(x=100, y=50)
      self.t1.place(x=200, y=50)
      self.lbl2.place(x=100, y=100)
```

```
        self.t2.place(x=200, y=100)
        self.b1=Button(win, text='Add')
        self.b1.bind('<Button-1>', self.add)
        self.b2=Button(win, text='Subtract')
        self.b2.bind('<Button-1>', self.sub)
        self.b3=Button(win, text='Multiply')
        self.b3.bind('<Button-1>', self.mul)
        self.b4=Button(win, text='Divide')
        self.b4.bind('<Button-1>', self.div)
        self.b1.place(x=30, y=150)
        self.b2.place(x=100, y=150)
        self.b3.place(x=200, y=150)
        self.b4.place(x=300, y=150)
        self.lbl3.place(x=100, y=200)
        self.t3.place(x=200, y=200)
    def add(self,event):
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=num1+num2
        self.t3.insert(END, str(result))
    def sub(self, event):
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=num1-num2
        self.t3.insert(END, str(result))
    def mul(self, event):
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=num1*num2
        self.t3.insert(END, str(result))
    def div(self, event):
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=num1/num2
        self.t3.insert(END, str(result))

window=Tk()
mywin=MyWindow(window)
```

```
window.title('Tkinter Calculator')
window.geometry("400x300+10+10")
window.mainloop()
```

The above example creates the following UI:



Implementation and Output:

Design a GUI application to show input and output operations using Tkinter. (Here we have implemented a calculator with multiple functionalities)

Code:

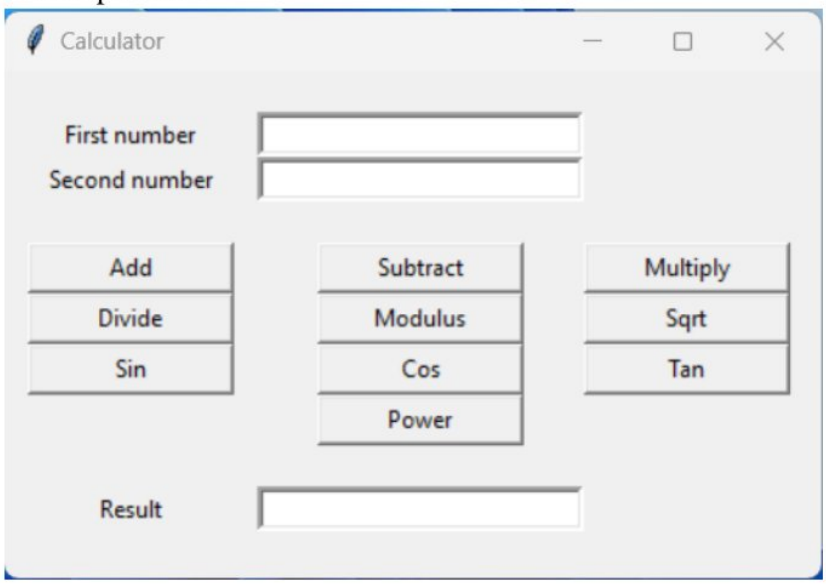
```
from tkinter import *
import math

class MyWindow:
    def __init__(self, win):
        self.lbl1=Label(win, text='First number')
        self.lbl2=Label(win, text='Second number')
        self.lbl3=Label(win, text='Result')
        self.lbl4=Label(win, text="")
        self.lbl6=Label(win, text="")
        self.t1=Entry(bd=3)
        self.t2=Entry(bd=3)
        self.t3=Entry(bd=3)
        self.lbl6.grid(row=0, column=0,columnspan=3, ipadx=20)
        self.lbl1.grid(row=1, column=0,columnspan=1, ipadx=20)
        self.t1.grid(row=1, column=1,columnspan=1, ipadx=20)
        self.lbl2.grid(row=2, column=0,columnspan=1, ipadx=20)
        self.t2.grid(row=2, column=1,columnspan=1, ipadx=20)
        self.lbl4.grid(row=3, column=0, columnspan=3, ipadx=20)
        self.b1=Button(win, text='Add', width=8)
        self.b1.bind('<Button-1>', self.add)
        self.b2=Button(win, text='Subtract', width=8)
        self.b2.bind('<Button-1>', self.sub)
        self.b3=Button(win, text='Multiply', width=8)
        self.b3.bind('<Button-1>', self.mul)
        self.b4=Button(win, text='Divide', width=8)
        self.b4.bind('<Button-1>', self.div)
        self.b5=Button(win, text='Modulus', width=8)
        self.b5.bind('<Button-1>', self.mod)
        self.b6=Button(win, text='Sqrt', width=8)
        self.b6.bind('<Button-1>', self.sqroot)
        self.b7=Button(win, text='Sin', width=8)
        self.b7.bind('<Button-1>', self.sine)
        self.b8=Button(win, text='Cos', width=8)
        self.b8.bind('<Button-1>', self.cosine)
        self.b9=Button(win, text='Tan', width=8)
        self.b9.bind('<Button-1>', self.tangent)
        self.b10=Button(win, text='Power', width=8)
        self.b10.bind('<Button-1>', self.power)
        self.b1.grid(row=4, column=0,columnspan=1, ipadx=20)
        self.b2.grid(row=4, column=1,columnspan=1, ipadx=20)
        self.b3.grid(row=4, column=2,columnspan=1, ipadx=20)
        self.b4.grid(row=5, column=0,columnspan=1, ipadx=20)
```

```
    self.b5.grid(row=5, column=1,columnspan=1, ipadx=20)
    self.b6.grid(row=5, column=2,columnspan=1, ipadx=20)
    self.b7.grid(row=6, column=0,columnspan=1, ipadx=20)
    self.b8.grid(row=6, column=1,columnspan=1, ipadx=20)
    self.b9.grid(row=6, column=2,columnspan=1, ipadx=20)
    self.b10.grid(row=7, column=1,columnspan=1, ipadx=20)
    self.lbl5=Label(win, text="")
    self.lbl5.grid(row=8, column=0,columnspan=1, ipadx=20)
    self.lbl3.grid(row=9, column=0,columnspan=1, ipadx=20)
    self.t3.grid(row=9, column=1,columnspan=1, ipadx=20)
  def add(self,event):
    self.t2.config(state='normal')
    self.t3.delete(0, 'end')
    num1=int(self.t1.get())
    num2=int(self.t2.get())
    result=num1+num2
    self.t3.insert(END, str(result))
  def sub(self, event):
    self.t2.config(state='normal')
    self.t3.delete(0, 'end')
    num1=int(self.t1.get())
    num2=int(self.t2.get())
    result=num1-num2
    self.t3.insert(END, str(result))
  def mul(self, event):
    self.t2.config(state='normal')
    self.t3.delete(0, 'end')
    num1=int(self.t1.get())
    num2=int(self.t2.get())
    result=num1*num2
    self.t3.insert(END, str(result))
  def div(self, event):
    self.t2.config(state='normal')
    self.t3.delete(0, 'end')
    num1=int(self.t1.get())
    num2=int(self.t2.get())
    result=num1/num2
    self.t3.insert(END, str(result))
  def mod(self, event):
    self.t2.config(state='normal')
    self.t3.delete(0, 'end')
    num1=int(self.t1.get())
    num2=int(self.t2.get())
    result=num1%num2
    self.t3.insert(END, str(result))
  def sqroot(self, event):
```

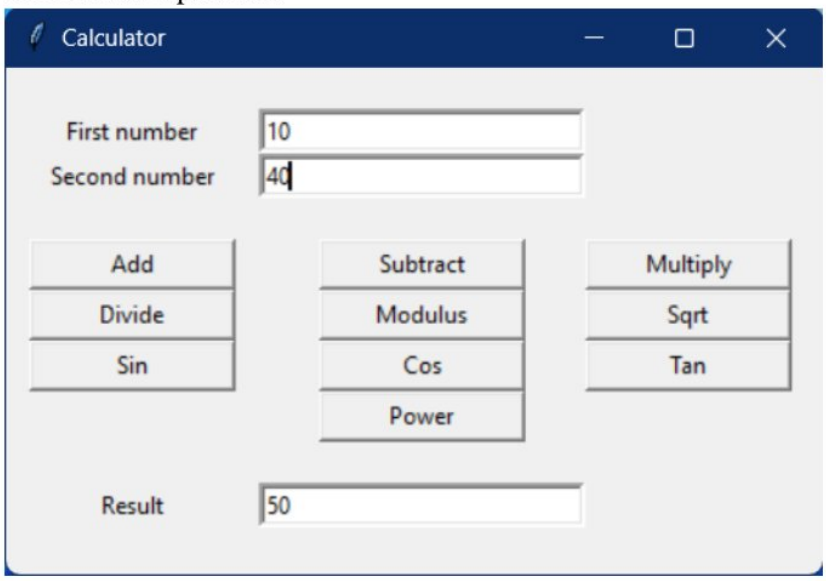
```
        self.t2.config(state='disabled')
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        result=math.sqrt(num1)
        self.t3.insert(END, str(round(result, 4)))
        self.t3.insert(END, str())
    def sine(self, event):
        self.t2.config(state='disabled')
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        result=math.sin(num1*(math.pi/180))
        self.t3.insert(END, str(round(result, 4)))
    def cosine(self, event):
        self.t2.config(state='disabled')
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        result=math.cos(num1*(math.pi/180))
        self.t3.insert(END, str(round(result, 4)))
    def tangent(self, event):
        self.t2.config(state='disabled')
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        result=math.tan(num1*(math.pi/180))
        self.t3.insert(END, str(round(result, 4)))
    def power(self, event):
        self.t2.config(state='normal')
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=pow(num1, num2)
        self.t3.insert(END, str(result))

window=Tk()
mywin=MyWindow(window)
window.title('Calculator')
window.geometry("420x260+10+10")
window.mainloop()
```

Output:

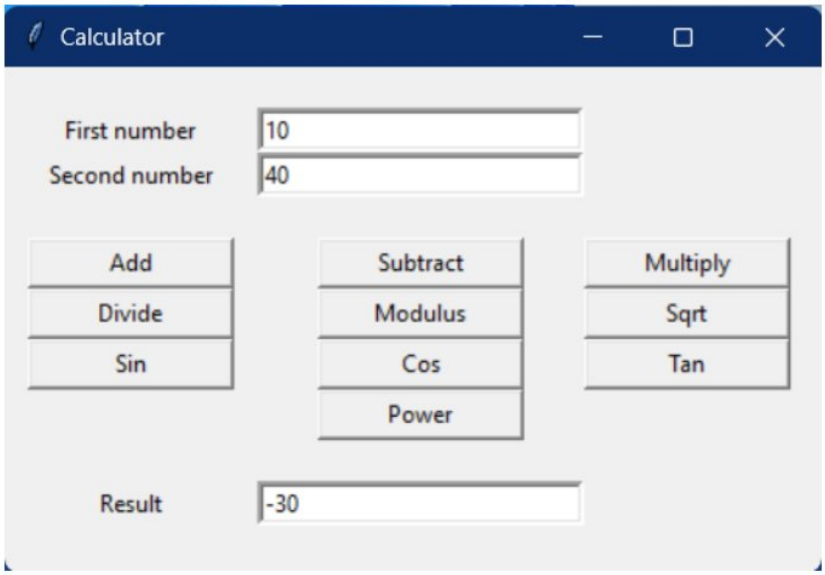
1. Complete UI:



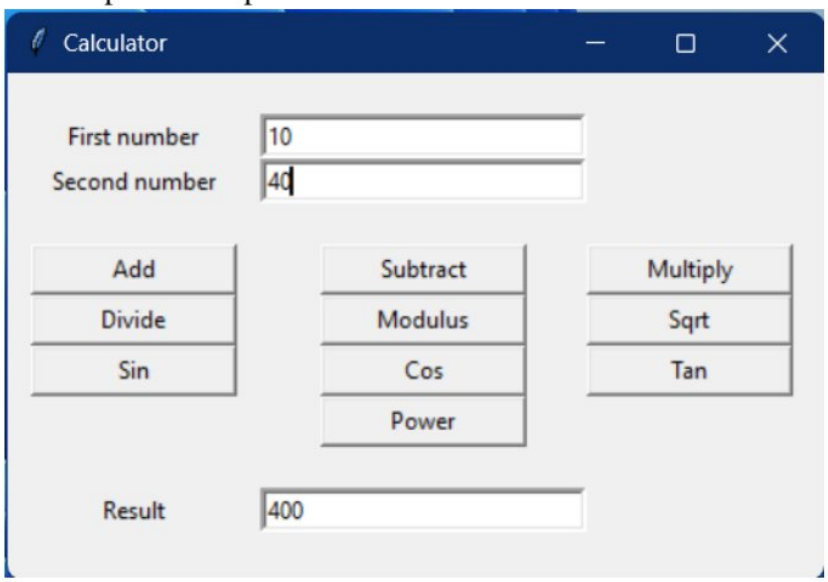
2. Addition Operation:



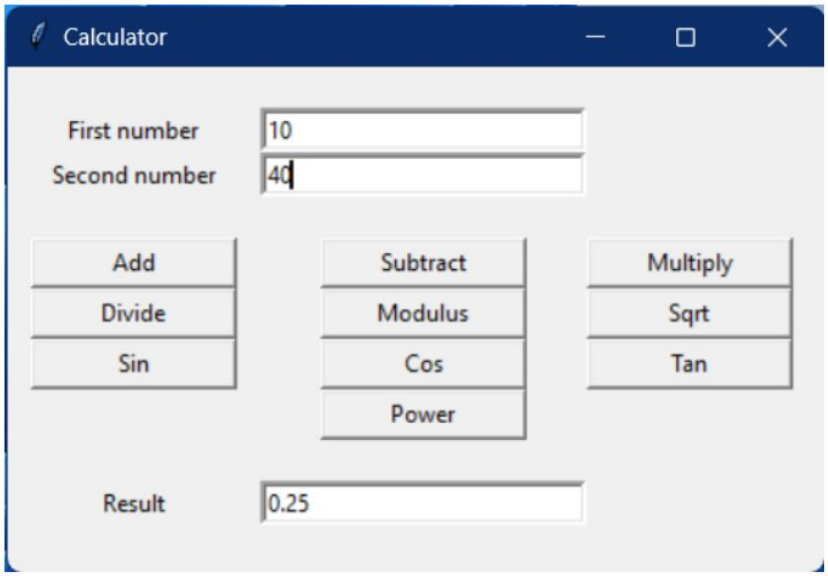
3. Subtraction Operation:



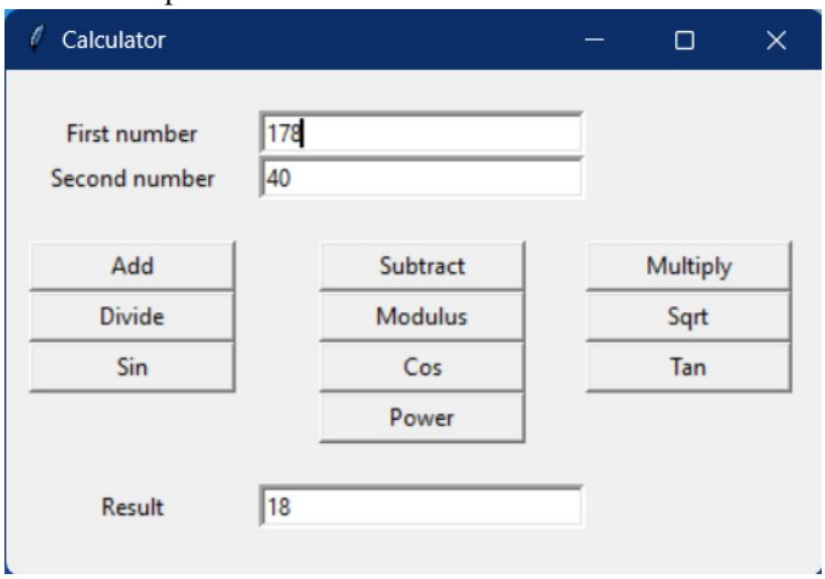
4. Multiplication Operation:



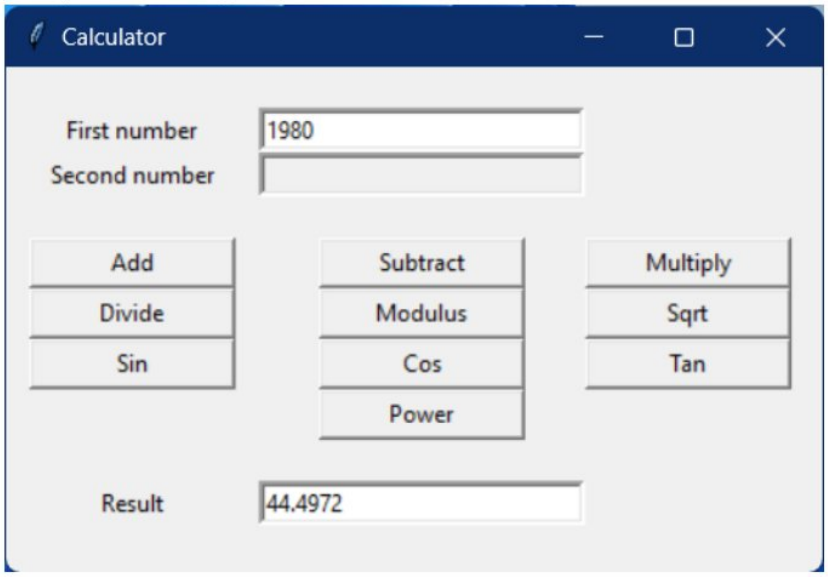
5. Division Operation:



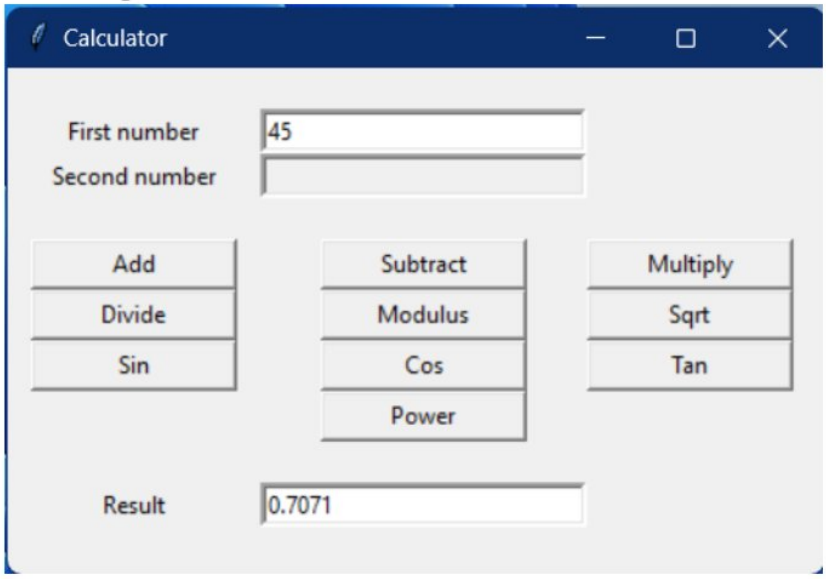
6. Modulo Operation:



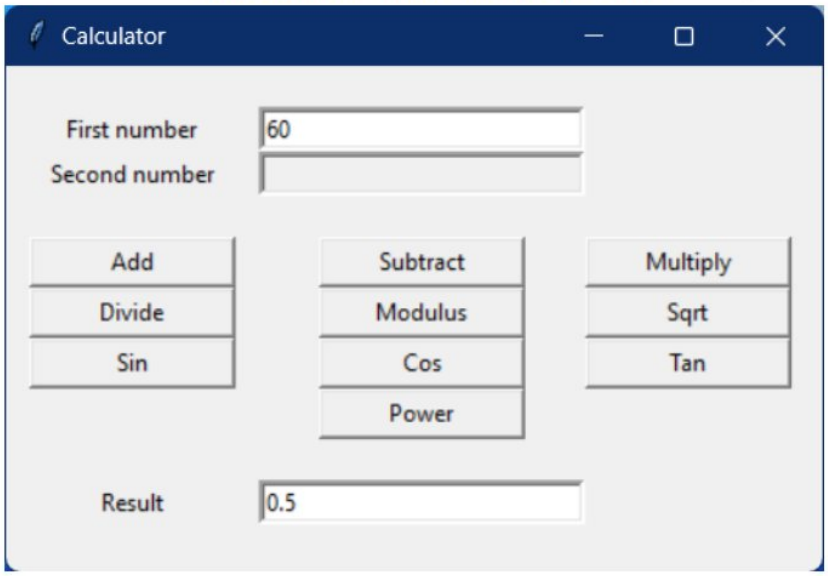
7. Square Root Operation:



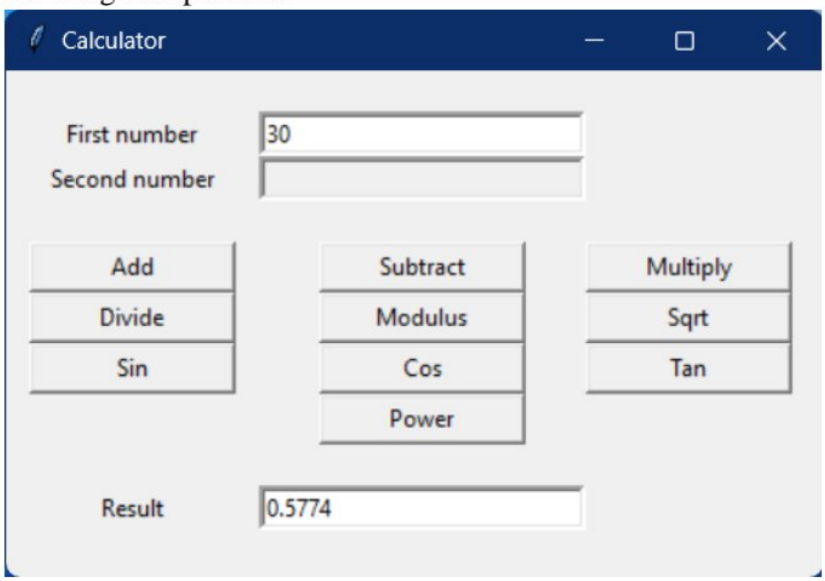
8. Sine Operation:



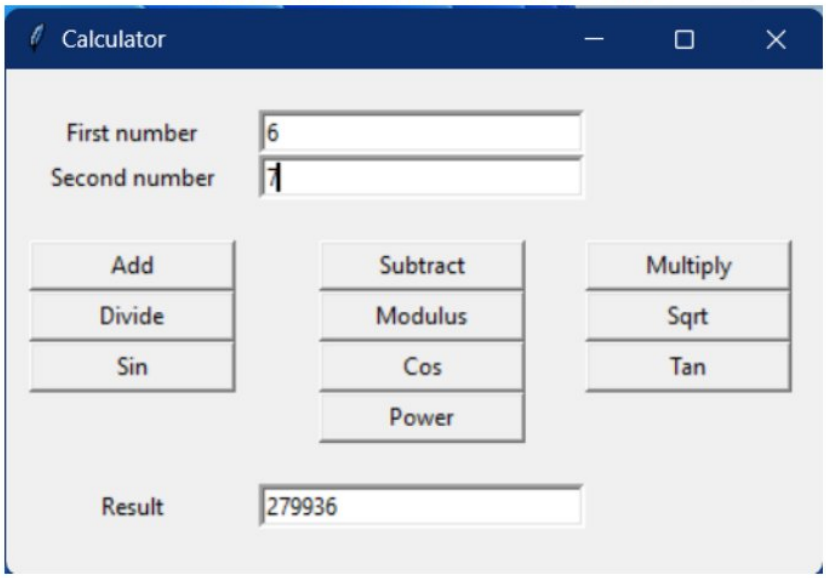
9. Cosine Operation:



10. Tangent Operation:



11. Power Operation:



Conclusion:

In this experiment, we implemented GUI application using Tkinter module in Python. We studied and implemented various tools and functionalities offered by the Tkinter module. We observed that Tkinter provides a simple and easy-to-use set of tools which integrate with each other seamlessly to form the final GUI. We studied the process to build a good GUI application using the functionalities offered by Tkinter viz., Window (using the Tk() method), mainloop() method, geometry() method etc. We studied various widgets (which are at the center of Tkinter module) like Label, Entry, Button, Combobox, Listbox, Radiobutton, Checkbutton, etc. We also observed that Tkinter offers the user to define the geometry of their GUI application using the Geometry Manager, and studied various methods provided by it, viz. pack(), grid() and place(). We observed that Tkinter provides us with a way of making our GUI applications dynamic by providing robust Event Handling mechanisms for the available widgets. Thus, we can conclude that Tkinter provides us with simple, efficient, robust and easy-to-use tools and functionalities which enable the user to build impactful GUI applications.

Experiment 10

Shashwat Shah
60004220126
C2-2 Div B

Aim : To perform Pandas operations on data set in Python

Theory : Pandas is a python library used for working with the datasets. It has functions for analyzing, cleaning, exploring, and manipulating data from datasets.

The name Pandas has reference to both panel data and also python data analysis. It was created by Wes McKinney in 2008. Pandas allows us to analyze big data and make conclusions based on statistical theory. Pandas can also help cleaning messy datasets and make them readable and relevant.

Relevant data is very important in data science. Pandas gives you answers about the data, like is there a correlation between two or more columns, what is average value, max value, minimum value? Pandas are also able to delete rows that are not relevant, or contain wrong values, like empty or null values.

Conclusion : Thus from this experiment we learnt about how datasets are managed and its various operations using Pandas.

Experiment No. 10

Pandas and Matplotlib in Python

Aim:

To study and demonstrate the use of pandas and matplotlib modules in Python.

Description:

Pandas:

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- **Min** value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

Many useful functions exist in pandas module, which are demonstrated in the experiment.

Pandas Dataframe:

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional airay, or a table with rows and columns.

Syntax:

df = pd.DataFrame(data)

Matplotlib:

Matplotlib is a low level graph plotting library in python that serves as a visualization utility.

Matplotlib was created by John D. Hunter.

Matplotlib is open source and we can use it freely.

Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible.

- Create publication quality plots.
- Make interactive figures that can zoom, pan, update.
- Customize visual style and layout.
- Export to many file formats.
- Embed in JupyterLab and Graphical User Interfaces.
- Use a rich array of third-party packages built on Matplotlib.

Different functionalities of matplotlib like plotting scatter plots, plotting barcharts, piecharts, histograms, etc. are demonstrated in this experiment.

Implementation and Output:

I] Pandas:

Create series, create own dataframe:

1. Creating Series

Code:

```
# Creating series
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)
```

Output:

```
0  1
1  7
2  2
dtype: int64
```

2. Creating Series with labels:

Code:

```
# Creating series with Labels
import pandas as pd
a = [1, 7, 2]
```

```
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
```

Output:

x 1

y 7

z 2

dtype: int64

3. Creating series with key/value pairs:

Code:

```
# Creating series with key/value pairs
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

Output:

day1 420

day2 380

day3 390

dtype: int64

4. Creating Dataframe from series:

Code:

```
# Creating Dataframe from series
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
```

```
print(myvar)
#refer to the row index:
print(df.loc[0])
#use a list of indexes:
print(df.loc[[0, 1]])
```

Output:

```
   calories  duration
0       420        50
1       380        40
2       390        45
calories    420
duration     50
Name: 0, dtype: int64
   calories  duration
0       420        50
1       380        40
```

5. Using named indexes:

Code:

```
# Using named indexes
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
#refer to the named index:
print(df.loc["day2"])
```

Output:

```
      calories  duration
day1       420        50
day2       380        40
day3       390        45
calories   380
duration    40
Name: day2, dtype: int64
```

Read CSV:

1. Load the CSV into a Dataframe:

Code:

```
# Load the csv into a Dataframe
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

Output:

```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
..        ...    ...       ...       ...
164        60    105       140     290.8
165        60    110       145     300.0
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4

[169 rows x 4 columns]
```

Data Cleaning:

Dataset used:

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	NaN
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	NaN
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

Code:

```
# Info about data
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head())
print(df.tail())
print(df.info())
```

Output:

```
   Duration        Date  Pulse  Maxpulse  Calories
0        60  '2020/12/01'    110       130     409.1
1        60  '2020/12/02'    117       145     479.0
2        60  '2020/12/03'    103       135     340.0
3        45  '2020/12/04'    109       175     282.4
4        45  '2020/12/05'    117       148     406.0
    Duration        Date  Pulse  Maxpulse  Calories
27       60  '2020/12/27'     92       118     241.0
28       60  '2020/12/28'    103       132       NaN
29       60  '2020/12/29'    100       132     280.0
30       60  '2020/12/30'    102       129     380.3
31       60  '2020/12/31'     92       115     243.0
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  32 non-null     int64
 1   Date      31 non-null     object
 2   Pulse     32 non-null     int64
 3   Maxpulse  32 non-null     int64
 4   Calories  30 non-null     float64
dtypes: float64(1), int64(3), object(1)
memory usage: 1.4+ KB
None
```

1. Delete NA values from the dataframe(all NA and NA values of specific columns)

Code:

```
a) For all columns:
# Removing null values
import pandas as pd
df = pd.read_csv('data.csv')
new_df = df.dropna(subset=["Calories"])
print(new_df.to_string())
b) For one column:
# Removing null values
import pandas as pd
df = pd.read_csv('data.csv')
new_df = df.dropna()
print(new_df.to_string())
```

Output:

a) For all columns:

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3

14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

b) For specific column:

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7

13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

2. Fill NA values with random values, mean, median:

Code:

a) With random values:

```
# Replacing with random value for one column
import pandas as pd
df = pd.read_csv('data.csv')
df["Calories"].fillna(130, inplace = True)
print(df)
```

b) With mean:

```
# with mean
import pandas as pd
df = pd.read_csv('data.csv')
```

```
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace = True)
print(df)
```

c) With median:

```
# with median
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].median()
df["Calories"].fillna(x, inplace = True)
print(df)
```

Output:

a) With random values:

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0

16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	130.0
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	130.0
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

b) With mean:

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.10
1	60	'2020/12/02'	117	145	479.00
2	60	'2020/12/03'	103	135	340.00
3	45	'2020/12/04'	109	175	282.40
4	45	'2020/12/05'	117	148	406.00
5	60	'2020/12/06'	102	127	300.00
6	60	'2020/12/07'	110	136	374.00
7	450	'2020/12/08'	104	134	253.30
8	30	'2020/12/09'	109	133	195.10
9	60	'2020/12/10'	98	124	269.00
10	60	'2020/12/11'	103	147	329.30
11	60	'2020/12/12'	100	120	250.70

12	60	'2020/12/12'	100	120	250.70
13	60	'2020/12/13'	106	128	345.30
14	60	'2020/12/14'	104	132	379.30
15	60	'2020/12/15'	98	123	275.00
16	60	'2020/12/16'	98	120	215.20
17	60	'2020/12/17'	100	120	300.00
18	45	'2020/12/18'	90	112	304.68
19	60	'2020/12/19'	103	123	323.00
20	45	'2020/12/20'	97	125	243.00
21	60	'2020/12/21'	108	131	364.20
22	45	NaN	100	119	282.00
23	60	'2020/12/23'	130	101	300.00
24	45	'2020/12/24'	105	132	246.00
25	60	'2020/12/25'	102	126	334.50
26	60	2020/12/26	100	120	250.00
27	60	'2020/12/27'	92	118	241.00
28	60	'2020/12/28'	103	132	304.68
29	60	'2020/12/29'	100	132	280.00
30	60	'2020/12/30'	102	129	380.30
31	60	'2020/12/31'	92	115	243.00

c) With median:

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3

8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	291.2
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	291.2
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

Display statistical information of the data frame:

Code:

```
# statistical information
import pandas as pd
df = pd.read_csv('data.csv')
```

```
print(df.describe())
```

Output:

	Duration	Pulse	Maxpulse	Calories
count	32.000000	32.000000	32.000000	30.000000
mean	68.437500	103.500000	128.500000	304.680000
std	70.039591	7.832933	12.998759	66.003779
min	30.000000	90.000000	101.000000	195.100000
25%	60.000000	100.000000	120.000000	250.700000
50%	60.000000	102.500000	127.500000	291.200000
75%	60.000000	106.500000	132.250000	343.975000
max	450.000000	130.000000	175.000000	479.000000

Establish relationship between the columns of the data frame:

Code:

```
# Establish Relationship
import pandas as pd
df = pd.read_csv('data.csv')
print(df.corr())
```

Output:

	Duration	Pulse	Maxpulse	Calories
Duration	1.000000	0.004410	0.049959	-0.114169
Pulse	0.004410	1.000000	0.276583	0.513186
Maxpulse	0.049959	0.276583	1.000000	0.357460
Calories	-0.114169	0.513186	0.357460	1.000000

II] Matplotlib:

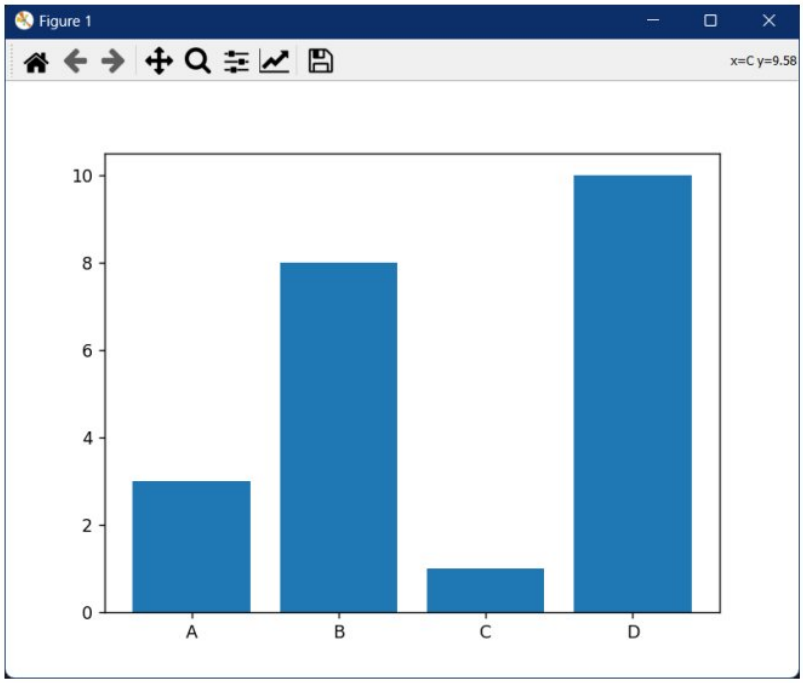
1. Plot barchart:

Code:

```
# Barchart
import matplotlib.pyplot as plt
import numpy as np
```

```
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
plt.bar(x,y)
plt.show()
```

Output:

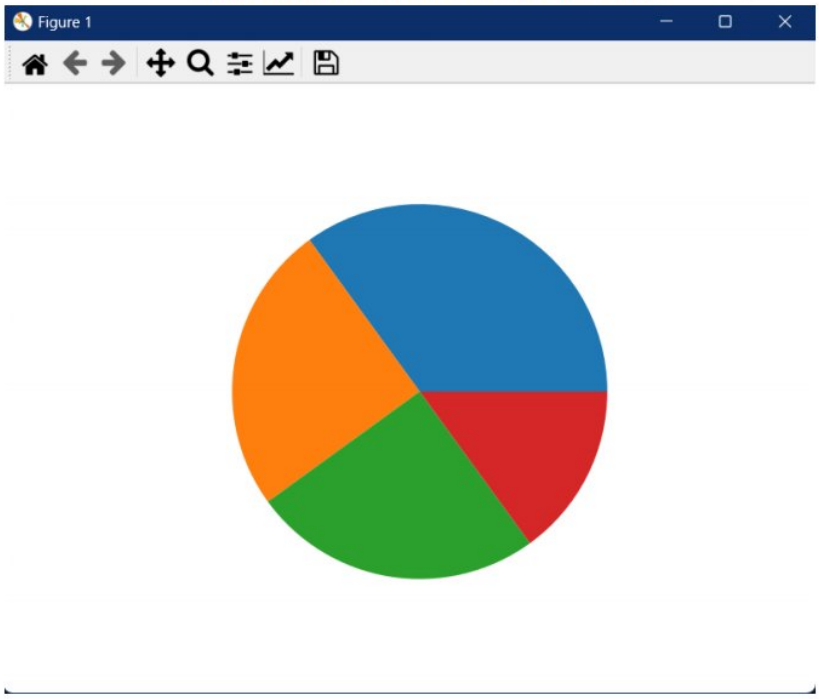


2. Plot piechart:

Code:

```
# Piechart
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
plt.pie(y)
plt.show()
```

Output:

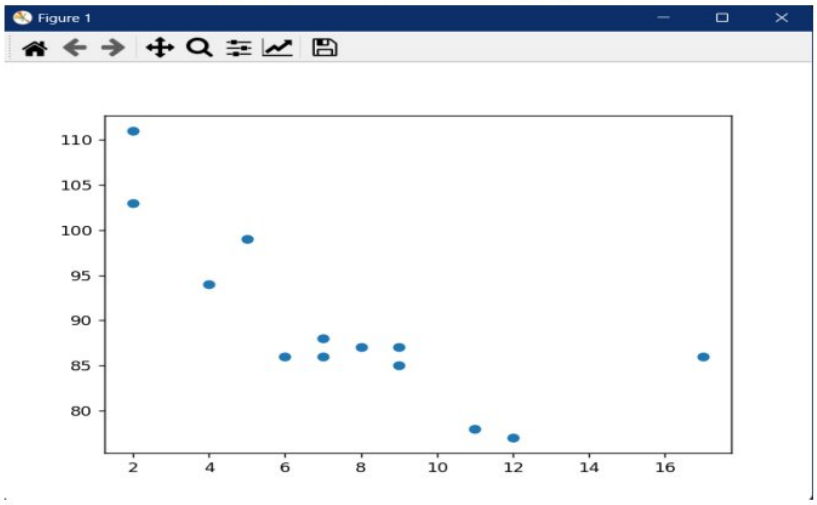


3. Scatter Plot:

Code:

```
# Scatter Plot
import matplotlib.pyplot as plt
import numpy as np
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)
plt.show()
```

Output:

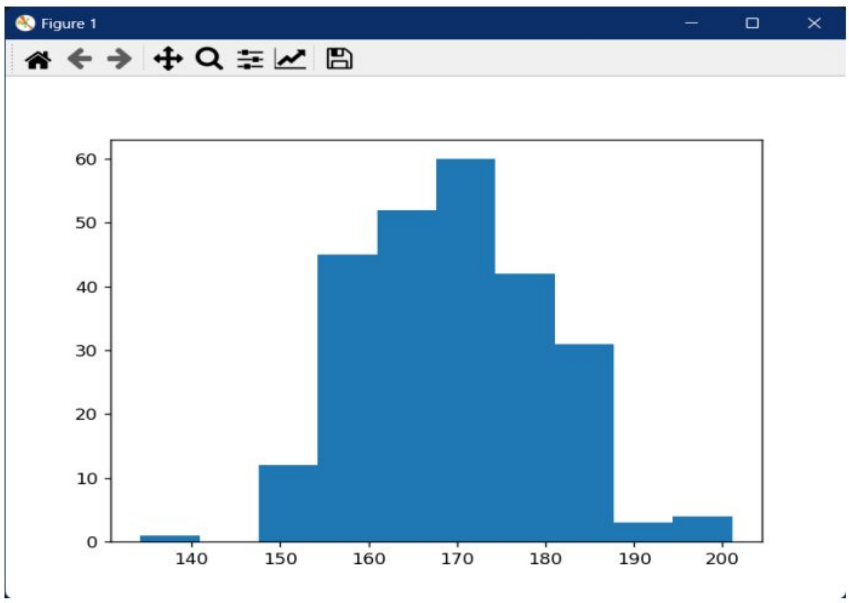


4. Histogram:

Code:

```
# Histogram
import matplotlib.pyplot as plt
import numpy as np
x = np.random.normal(170, 10, 250)
plt.hist(x)
plt.show()
```

Output:



Conclusion:

In this experiment, we implemented various functionalities offered by the pandas and the matplotlib modules. We observed that, pandas is an extremely useful module for handlining datasets (which are read in the form of Dataframes). Different operations like analysing, cleaning, etc. can be performed on the datasets using the pandas module. On the other hand, we observed that the relationships between different attributes, and the distribution of data in the datasets can be effectively visualized using the matplotlib module using scatter plots, histograms, etc. Thus, we can conclude that pandas and matplotlib are powerful and robust tools which can prove to be extremely useful in data science and analytics.