ADITYA

COLLEGE OF ENGINEERING & TECHNOLOGY

An AUTONOMOUS Institution

Approved by AICTE, Permanently Affiliated to JNTUK,
Accredited by NBA & NAAC with A+ Grade
Recognized by UGC under Section 2(f) and 12(B) of UGC Act, 1956
Aditya Nagar, ADB Road, Surampalem, Kakinada District - 533437, A.P.

Department of
COMPUTER SCIENCE AND ENGINEERING

Name : Roll No. : | | | | | | | | | | |

Certified that this is the bonafide record of practical work done by

Mr. /Ms. .....

a student of .....with PIN No. .....

in the ..... Laboratory during the year .....

No. of Practicals Conducted : | | No. of Practicals Attended : | |

Signature - Faculty Incharge **Signature - Head of the Department**

Submitted for the Practical examination held on .....

EXAMINER - 1 **EXAMINER - 2**

ADITYA COLLEGE OF ENGINEERING AND TECHNOLOGY

INSTITUTE VISION AND MISSION

VISION:

To induce higher planes of learning by imparting technical education with

- ✓ International standards
- ✓ Applied research
- ✓ Creative Ability
- ✓ Value based instruction and to emerge as a premiere institute

MISSION:

Achieving academic excellence by providing globally acceptable technical education by forecasting technology through

- ✓ Innovative Research And development
- ✓ Industry Institute Interaction
- ✓ Empowered Manpower

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT VISION AND MISSION

VISION:

To become a center for excellence in Computer Science and Engineering education and innovation.

MISSION:

- Provide state of art infrastructure
- Adapt skill-based learner centric teaching methodology
- Organize socio cultural events for better society
- Undertake collaborative works with academia and industry
- Encourage students and staff self-motivated, problem-solving individuals using Artificial Intelligence
- Encourage entrepreneurship in young minds.

Pointer

S No	Date	Name of the Experiment	Page No	Remark

Pointer

S No	Date	Name of the Experiment	Page No	Remark

Pointer

S No	Date	Name of the Experiment	Page No	Remark

UNIT-1

Parts of Python Programming Language:

Identifiers:

Identifiers are names used to identify variables, functions, classes, or other objects. In Python, they must follow these rules:

- Can consist of letters, digits, and underscores (_).
- Cannot start with a digit.
- Are case-sensitive.

Keywords:

- Keywords are reserved words in Python that have special meaning and cannot be used as identifiers. They are 35 keywords

false	None	From	Global
True	Await	If	Import
Break	Class	In	Is
And	Continue	Lambda	Nonlocal
as	def	Not	Or
Assert	Del	Pass	Raise
Async	Elif	Return	Try
Else	Except	While	With
Finally	For	Yield	

Statements and Expressions:

- **Statements:** Instructions that the Python interpreter executes, like variable assignments or function calls (a = 10).
- **Expressions:** Combinations of variables, operators, and values that are evaluated to produce a result (5 + 2).

Variables:

- Variables store data that can be referenced and manipulated later. They don't need explicit declaration of data types (dynamic typing), e.g., x = 10.

Operators:

Operators are used to perform operations on variables and values. Types of operators include:

- **Arithmetic operators** (+, -, *, /)
- **Comparison operators** (==, !=, <, >)
- **Logical operators** (and, or, not)

Precedence and Associativity:

- Operator precedence determines the order in which operators are evaluated. Associativity defines the direction of evaluation when two operators have the same precedence (usually left to right).

Data Types:

Python has several built-in data types, including:

- **Numeric**: int, float, complex
- **Sequence**: list, tuple, str
- **Mapping**: dict
- **Boolean**: True, False

Indentation:

- Python uses indentation to define code blocks. Unlike other languages that use braces {}, the level of indentation in Python defines the grouping of statements.

Comments:

- Comments help describe code and are ignored by the interpreter:
- Single-line comment: # This is a comment
- Multi-line comment: Can be written using triple quotes ('''...''')

Reading Input:

- Python uses input() to read input from the user, which is always returned as a string.

Print Output:

- The print() function is used to display output on the screen.

Type Conversions:

- Python allows type conversion using functions like int(), float(), str(), etc., to convert between data types.

The type() Function and is Operator:

- **type():** returns the type of an object: type(42) returns <class 'int'>.
- **is operator:** checks whether two objects are identical (i.e., refer to the same memory location).

Dynamic and Strongly Typed Language:

- Python is **dynamic** because the type of a variable is determined at runtime. It is **strongly typed** because it doesn't automatically convert between data types without explicit instruction

Control Flow Statements:

if Statement:

- Executes a block of code if a condition is true.

Syntax:

```
if condition:
    # Code block
```

if-else statement:

- Executes one block if the condition is true, another if it's false.

Syntax:

```
if condition:
    # Code block
else:
    # Another code block
```

if-elif-else:

- Checks multiple conditions, if the condition is true it will execute the if block statement otherwise the elif condition will execute otherwise else block will execute.

Syntax:

```
if condition1:
   # Code block
elif condition2:
   # Another code block
else:
   # Final block
```

Nested if statement:

- One if statement inside another if statement.

Syntax:

```
if condition1:
   # Block of code to execute if condition1 is True
   if condition2:
      # Block of code to execute if condition2 is also True
   else:
      # Block of code to execute if condition2 is False
else:
   # Block of code to execute if condition1 is False
```

while loop:

- Repeats a block of code as long as the condition is true.

Syntax:

```
while condition:
# Loop body
```

for loop:

- Iterates over a sequence (like a list or string).

Syntax:

```
for item in sequence:
   # Loop body
```

continue and break Statements:

- **continue**: Skips the current iteration and moves to the next.

Syntax:

```
for element in sequence:
   if condition:
      continue
```

- **break**: Exits the loop immediately.

Syntax:

```
for element in sequence:
   if condition:
      break
```

Catching Exceptions using try and except statement:

- Used for handling exceptions (errors) in a program

Syntax:

```
try:
    # Code that might raise an exception
except SomeException:
    # Handle the exception
```

Experiment 1

1. Write a program to find the largest element among three Numbers.

Aim: To find the largest element among three Numbers

Program:

```
#static input
num1, num2, num3 = 10 , 30 , 20
if num1 >= num2 and num1 >= num3:
            print(num1)
elif num2 >= num1 and num2 >= num3:
           print(num2)
else:
            print(num3)

#Dynamic input
num1=int(input("Enter First Number: "))
num2=int(input("Enter Second Number: "))
num3=int(input("Enter Third Number: "))
if num1 >= num2 and num1 >= num3:
  print(num1)
elif num2 >= num1 and num2 >= num3:
  print(num2)
else:
  print(num3)
```

Output:

```
30
Enter First Number: 55
Enter Second Number: 11
Enter Third Number: 99
99
```

Experiment 2

2. Write a Program to display all prime numbers within an interval

Aim: To display all prime numbers within an interval

Program:

```
lower = int(input("Enter lower limit: "))
upper = int(input("Enter upper Number: "))
print("Prime numbers between", lower, "and", upper, "are:")
for num in range(lower, upper + 1):
  # all prime numbers are greater than 1
  if num > 1:
     for i in range(2, num):
        if (num % i) == 0:
           break
     else:
        print(num)
```

Output:

```
Enter lower limit: 60
Enter upper Number: 99
Prime numbers between 60 and 99 are:
61
67
71
73
79
83
89
97
```

Experiment 3

3. Write a program to swap two numbers without using a temporary variable

Aim: To swap two numbers without using a temporary variable

Program:

```
# Initial values
a = 5
b = 10
# Display original values
print("Before swapping:")
print("a =", a)
print("b =", b)
# Swapping using tuple unpacking
a, b = b, a
# Display swapped values
print("After swapping:")
print("a =", a)
print("b =", b)
```

Output:

```
Before swapping:
a = 5
b = 10
After swapping:
a = 10
b = 5
```

Experiment 4

4. Demonstrate the following Operators in Python with suitable examples. i) Arithmetic Operators ii) Relational Operators iii) Assignment Operators iv) Logical Operators v) Bit wise Operators vi) Ternary Operator vii) Membership Operators viii) Identity Operators

Aim: Demonstrate the following operators i) Arithmetic Operators ii) Relational Operators iii) Assignment Operators iv) Logical Operators v) Bit wise Operators vi) Ternary Operator vii) Membership Operators viii) Identity Operators

Program:

```
# Arithmetic Operators
a, b = 10, 5
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)

# Relational Operators
x, y = 15, 10
print("Equal to:", x == y)
print("Not equal to:", x != y)
print("Greater than:", x > y)
print("Less than:", x < y)
print("Greater than or equal to:", x >= y)
print("Less than or equal to:", x <= y)

# Assignment Operators
a = 10
a += 5
print("Add and assign (a += 5):", a)
a -= 3
print("Subtract and assign (a -= 3):", a)
a *= 2
print("Multiply and assign (a *= 2):", a)
a /= 4
print("Divide and assign (a /= 4):", a)
a %= 2
print("Modulus and assign (a %= 2):", a)
# Logical Operators
a, b = True, False
print("AND (a and b):", a and b)
print("OR (a or b):", a or b)
print("NOT (not a):", not a)
```

```
# Bitwise Operators
x, y = 10, 4
print("Bitwise AND (x & y):", x & y)
print("Bitwise OR (x | y):", x | y)
print("Bitwise XOR (x ^ y):", x ^ y)
print("Bitwise NOT (~x):", ~x)
print("Left Shift (x << 1):", x << 1)
print("Right Shift (x >> 1):", x >> 1)

# Ternary Operator
x, y = 10, 20
result = "x is greater" if x > y else "y is greater or equal"
print("Ternary Operator result:", result)

# Membership Operators
list = [1, 2, 3, 4, 5]
print("5 in list:", 5 in list)
print("10 not in list:", 10 not in list)

# Identity Operators
a, b = [1, 2, 3], [1, 2, 3]
print("a is b:", a is b)
print("a is not b:", a is not b)
```

Output:

```
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
Floor Division: 2
Modulus: 0
Exponentiation: 100000
Equal to: False
Not equal to: True
Greater than: True
Less than: False
Greater than or equal to: True
Less than or equal to: False
Add and assign (a += 5): 15
Subtract and assign (a -= 3): 12
Multiply and assign (a *= 2): 24
Divide and assign (a /= 4): 6.0
Modulus and assign (a %= 2): 0.0
AND (a and b): False
OR (a or b): True
NOT (not a): False
Bitwise AND (x & y): 0
```

```
Bitwise OR (x | y): 14
Bitwise XOR (x ^ y): 14
Bitwise NOT (~x): -11
Left Shift (x << 1): 20
Right Shift (x >> 1): 5
Ternary Operator result: y is greater or equal
5 in list: True
10 not in list: True
a is b: False
a is not b: True
```

Experiment 5

5. Write a program to add and multiply complex numbers

Aim: To add and multiply complex numbers(Enter first complex number i.e. 3+4j not 3+4i)

Program:

```
first = complex(input('Enter first complex number: '))
second = complex(input('Enter first complex number: '))
# Addition of complex number
addition = first + second
# Displaying Sum
print('SUM = ', addition)
# Multiplication of complex number
product = first * second
# Displaying Product
print('PRODUCT = ', product)
```

Output:

```
Enter first complex number: 3+4j
Enter first complex number: 5+8j
SUM =  (8+12j)
PRODUCT =  (-17+44j)
```

Experiment 6

6. Write a program to print multiplication table of a given number

Aim: To print multiplication table of a given number

Program:

```
num = int(input(" Enter a number : "))
# using the for loop to generate the multiplication tables
print("Table of: ",num,"is")
for a in range(1,21):
   print(num,'x',a,'=',num*a)
```

Output:

```
Enter a number : 5
Table of:  5 is
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
5 x 11 = 55
5 x 12 = 60
5 x 13 = 65
5 x 14 = 70
5 x 15 = 75
5 x 16 = 80
5 x 17 = 85
5 x 18 = 90
5 x 19 = 95
5 x 20 = 100
```

Experiment 7

7. Write how to Handle specific exceptions – like division of a number by zero in python

Aim: To Handle specific exceptions – like division of a number by zero

Program:

```
try:
    # Prompt the user to enter two numbers
    numerator = float(input("Enter the numerator: "))
    denominator = float(input("Enter the denominator: "))
    # Perform division
    result = numerator / denominator

except ZeroDivisionError:
    # Handle the specific exception for division by zero
    print("Error: Cannot divide by zero!")

except ValueError:
    # Handle a different specific exception for invalid input
    print("Error: Please enter valid numbers!")

else:
    # This block runs if no exceptions are raised
    print("Result:", result)

finally:
    # This block always executes, regardless of an exception
    print("Execution complete.")
```

Output:

```
Enter the numerator: 10
Enter the denominator: 0
Error: Cannot divide by zero!
Execution complete.
```

UNIT-2

Functions:

Built-In Functions:
Python provides several built-in functions like:

- **len():** Returns the length of a sequence.
- **print():** Prints the output to the screen.
- **type():** Returns the data type of an object.
- **int(), float(), str():** Convert values to integers, floats, and strings respectively.
- **sum(), max(), min():** Perform arithmetic on lists or ranges.

Commonly Used Modules:
Python has a rich set of modules that you can import to extend functionality:

- **math**: For mathematical operations (math.sqrt(), math.pi)
- **random**: For generating random numbers (random.random(), random.randint())
- **datetime**: For working with dates and times (datetime.datetime.now())
- **os**: For interacting with the operating system (os.getcwd(), os.listdir())
- **sys**: For system-specific parameters and functions (sys.argv, sys.exit())

Function Definition and Calling the Function:
Defining a function:

- A function is defined using the def keyword, followed by the function name, parentheses containing any parameters, and a colon. The code block within the function is indented.

Syntax:

```
def function_name(parameters):
    # Code block
return value  # Optional
```

Calling a function:

- Calling a function in Python involves using the function's name followed by parentheses. If the function requires parameters, you need to pass the appropriate arguments inside the parentheses.

  Syntax:

```
result = function_name(arguments)
```

return Statement and void Function:

- **return:** The return statement is used to return a value from a function.

  Syntax:

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

- **void:** A void function does not return a value.

  Syntax:

```
def greet():
print("Hello!")
```

Scope and Lifetime of Variables:

- **Scope**: Refers to the region where a variable can be accessed.
- **Local scope**: Variables defined inside a function.
- **Global scope**: Variables defined outside all functions.
- **Lifetime**: Variables inside functions exist only as long as the function is executing.

Default Parameters:

- You can specify default values for function parameters.

Syntax:

```
def greet(name="Guest"):
print("Hello", name)
```

Keyword Arguments:

- When calling a function, you can specify arguments by name.

Syntax:

```
def greet(first, last):
  print("Hello", first, last)

greet(last="Doe", first="John")
```

***args and **kwargs**:

- ***args:** allows you to pass a variable number of non-keyword arguments.

  Syntax:

```
def add(*numbers):
  return sum(numbers)
```

- ****kwargs:** allows you to pass a variable number of keyword arguments.

  Syntax:

```
def greet(**info):
  print("Hello", info["name"])
```

Command Line Arguments:

- You can use the sys module to capture arguments passed via the command line.

Syntax:

```
import sys
print(sys.argv)  # List of command line arguments
```

Strings:

Creating and Storing Strings:

- Strings can be created using single ('...') or double ("...") quotes.

Syntax:

```
s = "Hello, World!" [or]
s = ‘Hello, World!’
```

Basic String Operations:

Concatenation:

- Joining two or more strings together using the + operator.

Syntax:

```
str1 = "Hello"
```

```
str2 = "World"
result = str1 + " " + str2  # "Hello World"
```

Repetition:

- Repeating a string multiple times using the * operator

Syntax:

```
str1 = "Hi"
result = str1 * 3  # "HiHiHi"
```

Membership:

- Checking if a substring exists within a string using the in keyword

Syntax:

```
s = "Hello, World!"
result = "World" in s  # True
result = "World" not in s  # False
```

String Length:

- Finding the number of characters in a string using the len() function.

Syntax:

```
s = "Hello"
length = len(s)  # 5
```

Accessing Characters in String by Index Number,:

- Accessing individual characters in a string using indexing

Syntax:

```
s = "Hello"
first_char = s[0]  # 'H'
last_char = s[-1]  # 'o'
```

String Slicing:

- Extracting a substring from a string using slice notation.

Syntax:

```
substring = string[start:stop:step]
```

Example:

```
s = "Hello, World!"
substring = s[7:12]  # "World"
```

String Joining:

- Joining strings in Python combines elements from an iterable (like a list) into a single string, using a specified separator.

Syntax:

```
separator.join(iterable)
```

Changing Case:

Changing the case of a string using built-in methods:

- **upper()**: Converts all characters to uppercase.
- **lower()**: Converts all characters to lowercase.
- **title()**: Converts the first character of each word to uppercase.

- **s.strip():** Remove whitespace from both ends.
- **s.replace(old, new):** Replace occurrences of a substring.
- **s.split(delimiter):** Split the string into a list.

Syntax:

```
s = "   Hello, World!   "
# 1. Convert to uppercase
uppercase = s.upper()         # "   HELLO, WORLD!   "
# 2. Convert to lowercase
lowercase = s.lower()         # "   hello, world!   "
# 3. Remove whitespace from both ends
stripped = s.strip()          # "Hello, World!"
# 4. Replace occurrences of a substring
replaced = s.replace("World", "Python")  # "   Hello, Python!   "
# 5. Split the string into a list
split_list = s.split(", ")    # ['   Hello', 'World!   ']
# Print results
print("Uppercase:", uppercase)
print("Lowercase:", lowercase)
print("Stripped:", stripped)
print("Replaced:", replaced)
print("Split List:", split_list)
```

Stripping Whitespace:

- Removing leading and trailing whitespace using the strip() method.

Syntax:

```
s = "   Hello, World!   "
stripped_s = s.strip()  # "Hello, World!"
```

Replacing Substrings:

- Replacing parts of a string with another substring using the replace() method.

Syntax:

```
s = "Hello, World!"
new_s = s.replace("World", "Python")  # "Hello, Python!"
```

Splitting and Joining Strings:

- **split()**: Splits a string into a list based on a delimiter.
- **join()**: Joins elements of a list into a string using a specified delimiter.

Syntax:

```
s = "Hello, World!"
words = s.split(", ")  # ['Hello', 'World!']

joined_string = " ".join(words)  # "Hello World!"
```

Finding Substrings:

- Finding the position of a substring using the find() method. Returns the index or -1 if not found.

Syntax:

```
s = "Hello, World!"
```

```
index = s.find("World")  # 7
```

Formatting Strings:

- Formatting strings using f-strings or the format() method.

Syntax:

```
name = "Alice"
age = 25
formatted_string = f"My name is {name} and I am {age} years old."  # "My name is Alice and I am 25 years old."
```

Lists:

Creating Lists:

- Lists are created using square brackets

Syntax:

```
list = [0,1,0,1] #list of numbers provided with commas
```

Basic List Operations:

Concatenation:

- Combining two or more lists into one using the + operator.

Syntax:

```
list1 + list2
```

Repetition:

- Repeating the elements of a list a specified number of times using the * operator

Syntax:

```
list * n
```

Membership:

- Checking if a value exists in a list using the in keyword.

Syntax:

```
element in list # if we use “in” and the element is shown then it is true
```

Indexing and Slicing in Lists:

Indexing:

- Indexing allows you to access individual elements in a list by their position.
- Python uses zero-based indexing, meaning the first element has an index of 0, the second has an index of 1, and so on.
- Positive indices start from 0 and go up (left to right).
- Negative indices start from -1 and go backward (right to left).

Syntax: list[index]

Slicing:

- Slicing is used to access a range or subset of elements from a list. The syntax for slicing is list[start:stop:step].
- **start:** The index to begin the slice (inclusive). Defaults to 0 if not specified.
- **stop:** The index to end the slice (exclusive). The slice goes up to, but does not include, this index.

- **step:** Optional, determines the stride (i.e., how many elements to skip). Defaults to 1.
- If step is omitted, the default value is 1 (i.e., elements are accessed consecutively).
- Negative values for step reverse the direction of the slice.

Syntax: list[start:stop:step]

Built-in Functions Used on Lists:

- **len(list):** Returns the number of elements in the list.
  Syntax:
  len(list)
- **max(list):** Returns the largest element.
  Syntax:
  max(list)
- **min(list):** Returns the smallest element.
  Syntax:
  min(list)
- **sum(list):** Returns the sum of all elements.
  Syntax:
  sum(list)
- **sorted(list):** Returns a new list containing all elements of the input list in sorted order.
  Syntax:
  sorted(list, key=None, reverse=False)
- **reversed(list):** Returns an iterator that accesses the list elements in reverse order.
  Syntax:
  reversed(list)
- **enumerate(list):** Adds a counter to each element in the list and returns it as an enumerate object
  Syntax:
  enumerate(list, start=0)
- **all(list):** Returns True if all elements in the list are True or if the list is empty.
  Syntax:
  all(list)
- **any(list):** Returns True if at least one element in the list is True.
  Syntax:
  any(list)
- **Copy():** Returns a shallow copy of the list
  Syntax:
  list.copy()
- **list():** Converts an iterable into a list.
  Syntax:
  list(iterable)
- **filter():** Filters elements in the list based on a function condition, returning as iterator.
  Syntax:
  filter(function, list)
- **map():** Applies a function to all elements in a list, returning an iterator.
  Syntax:
  map(function, list)

List Methods:

- **list.append(x):** Adds an element to the end of the list.

Syntax:

list.append(x)

- **list.insert(i, x):** Inserts an element at index i.

  Syntax:

  list.insert(i, x)

- **list.remove(x):** Removes the first occurrence of element x.

  Syntax:

  list.remove(x)

- **list.pop(i):** Removes the element at index i.

  Syntax:

  list.pop(i)

- **list.sort():** Sorts the list.

  Syntax:

  list.sort()
  list.sort(reverse=True) # For descending order

del Statement:

- The del statement is used to delete list elements.

Syntax:

del list[0] # Deletes the first element
del list[:] # Deletes all elements in the list

Experiment 8

8. Write a program to define a function with multiple return values

Aim: To define a function with multiple return values

Program:

```
def fun():
        str = "Aditya college of Engineering & Technology"
        x = 20
        return str, x
str, x = fun()
print(str)
print(x)
```

Output:

```
Aditya college of Engineering & Technology
20
```

Experiment 9

9. Write a program to define a function using default arguments

Aim: To define a function using default arguments

Program:

```
def add_numbers( a = 7,  b = 8):
   sum = a + b
   print('Sum:', sum)
add_numbers(2, 3)
add_numbers(a = 2)
add_numbers()
```

Output:

```
Sum: 5
Sum: 10
Sum: 15
```

Experiment 10

10. Write a program to find the length of the string without using any library functions

Aim: To find the length of the string without using any library functions

Program:

```
string = 'Hello'
count = 0
for i in string:
    count+=1
print(count)
```

Output: 5

Experiment 11

10. Write a program to check if the substring is present in a given string or not

Aim: To check if the substring is present in a given string or not

Program:

```
def sub_string(substring, mainstring):
    if substring in mainstring:
        print(F"{substring} is present in {mainstring}")
    else:
        print(F"{substring} is not present in {mainstring}")
sub_string("New", "NewYork")
```

Output:

```
New is present in NewYork
```

Experiment 12

12. Write a program to perform the given operations on a list: i. Addition ii. Insertion iii. Slicing

Aim: To perform the given operations on a list: i. Addition ii. Insertion iii. Slicing

Program:

```
list = [1,2,3,4,5]
print("list", list)

#Addition
list.append(6)
print(F" After appending of 6, list is {list}")

#insertion
list.insert(1, 10)
print(F" After insertion 10 in 1 index, list is {list}")

#slicing
print(F" Slicing of list is {list[3:6]}")
```

Output:

```
list [1, 2, 3, 4, 5]
After appending of 6, list is [1, 2, 3, 4, 5, 6]
After insertion 10 in 1 index, list is [1, 10, 2, 3, 4, 5, 6]
Slicing of list is [3, 4, 5]
```

Experiment 13

13. Write a program to perform any 5 built-in functions by taking any list

Aim: To perform any 5 built-in functions by taking any list

Program:

```
# Taking list input from the user
numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))

# 1. Using `min()` to find the smallest element
smallest = min(numbers)
print("Smallest element:", smallest)

# 2. Using `max()` to find the largest element
largest = max(numbers)
print("Largest element:", largest)

# 3. Using `sum()` to find the sum of all elements
total = sum(numbers)
print("Sum of elements:", total)

# 4. Using `len()` to get the number of elements
count = len(numbers)
print("Number of elements:", count)

# 5. Using `sorted()` to sort the list in ascending order
sorted_list = sorted(numbers)
print("Sorted list:", sorted_list)
```

Output:

```
Enter numbers separated by spaces: 5 2 9 1 7
Smallest element: 1
Largest element: 9
Sum of elements: 24
Number of elements: 5
Sorted list: [1, 2, 5, 7, 9]
```

UNIT-3

Dictionaries:

Creating Dictionary:

- A dictionary is a collection of key-value pairs, where keys are unique and values can be of any data type.

Syntax:

dict_name = {key1: value1, key2: value2, ...}

Accessing and Modifying Key-Value Pairs:

Accessing Key-Value Pairs:

- You can access the value associated with a specific key using square brackets [] or the .get() method.
- **Using Square Brackets []:**
    1. If the key exists, it returns the value.
    2. If the key does not exist, it raises a KeyError

  Syntax:

  value = dict_name[key]
- **Using .get() Method**:
    1. If the key exists, it returns the value.
    2. If the key does not exist, it returns None or a default value

  Syntax:

  value = dict_name.get(key, default_value)

Modifying Key-Value Pairs:

- You can modify the value associated with a specific key in a dictionary by assigning a new value to the key.
- **Modifying Existing Keys**:
    1. If the key exists, its value will be updated.
    2. If the key does not exist, a new key-value pair will be added.

  Syntax:

  dict_name[key] = new_value

Built-In Functions Used on Dictionaries:

- **len():** Returns the number of key-value pairs in the dictionary.
  Syntax:

  len(dictionary)
- **dict.keys():** Returns a view object that displays a list of all the keys in the dictionary.
  Syntax:

  dictionary.keys()
- **dict.values():** Returns a view object that displays a list of all the values in the dictionary.
  Syntax:

  dictionary.values()
- **dict.items():** Returns a view object that displays a list of the dictionary's key-value tuple pairs.
  Syntax:

  dictionary.items()

- **dict.get(key[, default]):** Returns the value for the specified key. If the key does not exist, it returns default (if provided) or None.
  Syntax:
  dictionary.get(key, default)
- **dict.update([other]):** Updates the dictionary with the key-value pairs from another dictionary or from an iterable of key-value pairs. Existing keys will have their values updated.
  Syntax:
  dictionary.update(other)
- **dict.copy():** Returns a shallow copy of the dictionary.
  Syntax:
  new_dict = dictionary.copy()
- **dict.clear():** Removes all items from the dictionary.
  Syntax:
  dictionary.clear()
- **dict.popitem():** Removes and returns the last inserted key-value pair as a tuple. Raises KeyError if the dictionary is empty.
  Syntax:
  key, value = dictionary.popitem()

Dictionary Methods:

- **dict.get(key):** Returns the value associated with the key.
  Syntax:
  value = dict_name.get(key)
  value = dict_name.get(key, default_value)

- **dict.update(other_dict):** Updates the dictionary with key-value pairs from another dictionary.
  Syntax:
  dict_name.update(other_dict)
- **dict.pop(key):** Removes the specified key and returns its value.
  Syntax:
  value = dict_name.pop(key)
  value = dict_name.pop(key, default_value)

- **dict.clear():** Removes all key-value pairs from the dictionary
  Syntax:
  dict_name.clear()

Dictionary Methods

- **dict.get(key)**: Returns the value associated with the key.
  Syntax:
  value = dict_name.get(key)
  value = dict_name.get(key, default_value)
- **dict.update(other_dict)**: Updates the dictionary with key-value pairs from another dictionary.
  Syntax:
  dict_name.update(other_dict)

- **dict.pop(key)**: Removes the specified key and returns its value.
  Syntax:
  value = dict_name.pop(key)
  value = dict_name.pop(key, default_value)
- **dict.clear()**: Removes all key-value pairs from the dictionary.
  Syntax:
  dict_name.clear()

del Statement:

- **del dict[key]**: Deletes a key-value pair from the dictionary.
- **del dict**: Deletes the entire dictionary

Syntax:
del dict_name

Tuples:

Creating Tuples:

- Tuples are immutable sequences, typically used to store related pieces of data.

Syntax:
tuple_name = (value1, value2, ...)

Basic Tuple Operations:

Concatenation:

- You can concatenate two tuples using the + operator.

Syntax:
new_tuple = tuple1 + tuple2

Repetition:

- You can repeat a tuple using the * operator.

Syntax:
new_tuple = tuple_name * n

Membership Test

- You can check if an element exists in a tuple using the in keyword.

Syntax:
if element in tuple_name:
# Do something

Unpacking a Tuple:

- You can unpack a tuple into individual variables.

Syntax:
a, b, c = tuple_name

tuple() Function:

- Converts other data types (like lists) to tuples.

Syntax:

tuple_name = tuple(sequence)

Indexing and Slicing in Tuples

Indexing:

- You can access individual elements in a tuple using their index. Tuples are zero-indexed, meaning the first element has an index of 0.

Syntax:

element = tuple_name[index]

Slicing:

- Slicing allows you to access a subset of elements from a tuple. You can specify the start index, end index, and optionally the step.
- **start:** The index to start slicing from (inclusive).
- **end:** The index to slice up to (exclusive).
- **step:** The interval of the slicing (optional).

Syntax:

sub_tuple = tuple_name[start:end:step]

Negative Indexing:

- You can also use negative indexing to access elements from the end of the tuple.

Syntax:

element = tuple_name[-index]

Built-In Functions Used on Tuples:

- **len():** Returns the number of elements in the tuple.
  Syntax:
  length = len(tuple_name)
- **max():** Returns the largest element in the tuple.
  Syntax:
  largest = max(tuple_name)
- **min():** Returns the smallest element in the tuple.
  Syntax:
  smallest = min(tuple_name)
- **sum():** Returns the sum of all elements in the tuple. This works only if the elements are numbers.
  Syntax:
  total = sum(tuple_name)
- **count():** Return the number of times a specified element appears in the tuple.
  syntax:
  count = tuple_name.count(element)
- **index():** Returns the index of the first occurrence of a specified element in the tuple.
  Syntax:
  index = tuple_name.index(element)

Relation Between Tuples and Lists:

- Tuples are immutable, whereas lists are mutable.
- You can convert between tuples and lists using tuple() and list() functions

Syntax:

Lists: []

Tuples: ()

Similarities:

1. **Ordered**: Both tuples and lists maintain the order of items. The items are indexed, and you can access them using their index.
2. **Allow Duplicates**: Both data structures can contain duplicate elements. For example, you can have multiple instances of the same value in both tuples and lists.
3. **Heterogeneous**: Both can store elements of different data types. You can have integers, strings, floats, and even other lists or tuples within them.
4. **Slicing and Indexing**: Both tuples and lists support slicing and indexing to access elements or sub-sections.

Relation Between Tuples and Dictionaries:

- Tuples can be used as keys in dictionaries because they are immutable, while lists cannot be used as dictionary keys.

Syntax:

```
# Creating a tuple
my_tuple = (element1, element2, element3, ...)
# Creating a dictionary
my_dict = {key1: value1, key2: value2, key3: value3, ...}
```

Similarities Between Tuples and Dictionaries:

1. **Heterogeneous Data**: Both tuples and dictionaries can store a mixture of different data types, such as integers, strings, floats, lists, and even other tuples or dictionaries.
2. **Ordered**:
   a. **Tuples**: They maintain the order of elements based on their index.
   b. **Dictionaries**: As of Python 3.7, dictionaries maintain the insertion order of key-value pairs.
3. **Accessing Elements**:
   a. You can access elements in a tuple by index.
   b. You can access values in a dictionary using keys.
4. **Iteration**: Both tuples and dictionaries can be iterated over using loops.

zip() Function:

- Combines two sequences into a sequence of tuples.

Syntax:

```
zipped = zip(sequence1, sequence2)
```

Sets:

Creating Sets:

- Sets are unordered collections of unique elements.

Syntax:

```
set_name = {value1, value2, ...}
```

Set Methods:

- **set.add(x)**: Adds an element x to the set.
  Syntax:
  ```
  set_name.add(element)
  ```
- **set.remove(x)**: Removes element x from the set. Raises a KeyError if x is not found.

Syntax:

set_name.remove(element)

- **set.union(other_set)**: Returns a new set with elements from both sets.

  Syntax:

  new_set = set_name.union(other_set)

- **set.intersection(other_set)**: Returns a new set with elements common to both sets.

  Syntax:

  new_set = set_name.intersection(other_set)

- **set.difference(other_set)**: Returns a new set with elements in the first set but not in the second.

  Syntax:

  new_set = set_name.difference(other_set)

Frozenset:

- A frozenset is an immutable set

Syntax:

frozen = frozenset(iterable)

Experiment 14

14. Write a program to create tuples (name, age, address, college) for at least two members and concatenate the tuples and print the concatenated tuples

Aim: To create tuples (name, age, address, college) for at least two members and concatenate the tuples and print the concatenated tuples

Program:

```
S1=("Name= Aditya", "Age = 20", "Address = CSE", "College = Adithya")
S2=("Name= Afrid", "Age = 21", "Address = CSE", "College = Adithya")
print(S1)
print(S2)
S3=S1+S2
print(S3)
```

Output:

```
('Name= Aditya', 'Age = 20', 'Address = CSE', 'College = Adithya')
('Name= Afrid', 'Age = 21', 'Address = CSE', 'College = Adithya')
('Name= Aditya', 'Age = 20', 'Address = CSE', 'College = Adithya', 'Name= Afrid', 'Age = 21',
'Address = CSE', 'College = Adithya')
```

Experiment 15

15. Write a program to count the number of vowels in a string (No control flow allowed)

Aim: To count the number of vowels in a string (No control flow allowed)

Program:

```
string = "Python Coders To Rule The World!"
vowels = "aeiouAEIOU"
count = sum(string.count(vowel) for vowel in vowels)
print(count)
```

Output:

```
8
```

Experiment 16

16. Write a program to check if a given key exists in a dictionary or not

Aim: To check if a given key exists in a dictionary or not

Program:

```
d={'a':1, 'b':2, 'c':3}
key='b'
print(key in d)
key='g'
print(key in d)
```

Output:

```
True
False
```

Experiment 17

17. Write a program to add a new key-value pair to an existing dictionary

Aim: To add a new key-value pair to an existing dictionary

Program:

```
dict = {'key1':' Python ', 'key2':' is a '}
dict['key3'] = ' versatile '
dict['key4'] = ' and powerful '
dict['key5'] = ' programming '
dict['key6'] = ' language
'
print(dict)
```

Output:

{'key1': ' Python ', 'key2': ' is a ', 'key3': ' versatile ', 'key4': ‘and powerful ', 'key5': ' programming ', 'key6': ' language'}

Experiment 18

18. Write a program to sum all the items in a given dictionary

Aim: To sum all the items in a given dictionary

Program:

```
def returnSum(myDict):

    list = []
    for i in myDict:
        list.append(myDict[i])
    final = sum(list)

    return final

# Driver Function
dict = {'a': 100, 'b': 200, 'c': 300}
print("Sum :", returnSum(dict))
```

Output:

```
Sum : 600
```

UNIT-4

Files:

Types of Files:

- **Text Files**: Files that contain readable characters. Commonly have extensions like .txt, .csv, etc.
- **Binary Files**: Files that contain data in binary format (non-readable). Examples include images, audio files, and compiled code. Extensions include .bin, .exe, etc.

Creating and Reading Text Data:

Creating a Text File:

- You can create a text file in Python using the open() function with the "w" (write) mode.
- If the file does not exist, it will be created.
- If it does exist, the contents will be overwritten.
- The with statement ensures proper acquisition and release of resources.
- It automatically closes the file when the block is exited.
- **file.write()**: writes a string to the file.
- Each call to **write()** appends the content after the existing content.

Syntax:

```
with open("filename.txt", "w") as file:
    file.write("Your text here.")
```

Reading from a Text File:

- To read data from a text file, you can use the open() function with the "r" (read) mode. You can read the entire file or line by line.

Syntax:

```
with open("filename.txt", "r") as file:
    content = file.read()  # Reads the entire file
```

File Methods to Read and Write Data

- **file.read(size)**: Reads the specified number of bytes from the file.
  Syntax:
  ```
  content = file.read(size)  # size is optional
  ```
- **file.readline()**: Reads a single line from the file.
  Syntax:
  ```
  line = file.readline()
  ```
- **file.readlines()**: Reads all lines and returns them as a list.
  Syntax:
  ```
  lines = file.readlines()
  ```

- **file.write(string)**: Writes a string to the file.
  Syntax:
  ```
  file.write(string)
  ```
- **file.writelines(list)**: Writes a list of strings to the file.
  Syntax:

```
file.writelines(list)
```

Reading and Writing Binary Files:

Writing a Binary File:

- To write binary data to a file, use the "wb" mode.

Syntax:

```
with open("filename.bin", "wb") as file:
    file.write(data)  # data should be in bytes
```

Reading a Binary File:

- To read binary data from a file, use the "rb" mode.

Syntax:

```
with open("filename.bin", "rb") as file:
    data = file.read(size)  # size is optional; reads the entire file if omitted
```

Pickle Module:

- There are two type of pickle modules:

1. **Pickling**: Is the process of converting an object into a byte stream.

Key Functions:

a. **pickle.dump(obj, file):** Serializes obj and writes it to the file object file.

Syntax:

```
import pickle

with open("filename.pkl", "wb") as file:
    pickle.dump(obj, file)
```

b. **pickle.load(file):**Reads a byte stream from the file object file and deserializes it back into a Python object.

Syntax:

```
with open("filename.pkl", "rb") as file:
    obj = pickle.load(file)
```

2. **unpickling:** while deserialization or unpicking is the reverse process of converting a byte stream back into an object.

Key Functions:

a. **pickle.load(file):**Reads a pickled object from a file and returns the deserialized Python object.

Syntax:

```
import pickle
with open("filename.pkl", "rb") as file:
    obj = pickle.load(file)
```

b. **pickle.loads(bytes):**Takes a byte string that contains a pickled object and returns the deserialized Python object.

Syntax:

```
import pickle
```

```
obj = pickle.loads(byte_string)
```

Reading and Writing CSV Files:

- Reading and writing CSV (Comma-Separated Values) files in Python can be easily accomplished using the built-in csv module.
- This module provides functionality to both read from and write to CSV files in a straightforward manner.

Reading CSV Files:

- To read CSV files, you can use the csv.reader function, which reads each row of the CSV file as a list of strings.

Syntax:

```
import csv

with open('filename.csv', mode='r', newline='') as file:
    reader = csv.reader(file)
for row in reader:
    print(row)  # Each row is a list of strings
```

Reading CSV Files with Headers:

- If your CSV file has a header row, you can use csv.DictReader, which reads each row as a dictionary

Syntax:

```
import csv

with open('filename.csv', mode='r', newline='') as file:
    reader = csv.DictReader(file)  # Create a DictReader object
for row in reader:
     # Access values by column names (keys)
    print(row['Column1'], row['Column2'])  # Replace with actual column names\
```

Writing CSV Files:

- To write to a CSV file, you can use the csv.writer function, which writes data as rows.

```
import csv
```

Syntax:

```
with open('filename.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Column1', 'Column2', 'Column3'])  # Write header
    writer.writerow(['Value1', 'Value2', 'Value3'])  # Write a row
```

Writing CSV Files with Lists:

- You can also write multiple rows using the writerows() method, which takes an iterable of rows.

Syntax:

```
import csv

with open('filename.csv', mode='w', newline='') as file:
    writer = csv.writer(file)  # Create a CSV writer object
    writer.writerow(['Column1', 'Column2', 'Column3'])  # Write the header (optional)
```

```
writer.writerow(['Value1', 'Value2', 'Value3'])  # Write a single row
```

Python os and os.path Modules:

Using os Module:

- **os.mkdir(path)**: Creates a new directory.
  Syntax:

```
import os

os.mkdir(path)  # Creates a new directory at the specified path.
```

- **os.listdir(path)**: Returns a list of files and directories in the specified path.
  Syntax:

```
import os

files_and_dirs = os.listdir(path)  # Returns a list of files and directories in the specified path
```

- **os.remove(path)**: Deletes a file.
  Syntax:

```
import os

os.remove(path)  # Deletes the specified file
```

Object-Oriented Programming:

Classes and Objects:

- Classes are blueprints for creating objects (instances).

Syntax:

```
class ClassName:
  # Class body
  pass  # Placeholder for class body
```

- Objects are instances of classes.

Syntax:

```
object_name = ClassName()
```

Creating class in python:

- Creating classes in Python is a fundamental concept of Object-Oriented Programming (OOP). Classes allow you to define your own data types, encapsulating data and functionality together.

Basic structure of class in python:

- To define a class in Python, you use the class keyword followed by the class name and a colon. Inside the class, you can define attributes (data) and methods (functions) that operate on the data.

Syntax:

```
class ClassName:
    def __init__(self, parameters):
```

```
        # Constructor method to initialize attributes
        self.attribute = parameters

    def method_name(self, parameters):
        # Method that does something
        pass
```

Creating Objects in python:

- Creating objects in Python is a key aspect of object-oriented programming (OOP). An object is an instance of a class, and it can hold data (attributes) and perform operations (methods) defined in the class. Here's a guide on how to create and work with objects in Python.

Steps to Create Objects

1. **Define a Class**: First, you need to define a class that describes the properties and behaviors of the objects you want to create.

   Syntax:

```
class ClassName:
    def __init__(self, parameter1, parameter2, ...):
    # Constructor method to initialize attributes
    self.attribute1 = parameter1
    self.attribute2 = parameter2
    # Additional initialization as needed

    def method_name(self):
    # Method definition
    pass  # Replace with method logic
```

2. **Create an Object**: You can create an object (or instance) of the class by calling the class as if it were a function.

   Syntax:

```
# Creating an instance (object) of the class
object_name = ClassName(parameter1_value, parameter2_value, ...)
```

3. **Access Attributes and Methods**: You can use dot notation to access the object's attributes and methods.

   Syntax:

```
value = object_name.attribute_name #Accessing Attributes
result = object_name.method_name() #calling methods
```

Constructor Method:

- The constructor method __init__ is called when an object is created.

Syntax:

```
class ClassName:
    def __init__(self, parameters):
        # Initialize object attributes
        Pass
```

Classes with Multiple Objects:

- When you create multiple objects (instances) of the same class, each object can have its own unique data attributes while sharing the class attributes.

Syntax:

```
class ClassName:
    def __init__(self, parameter1, parameter2):
        # Initialize instance attributes
        self.attribute1 = parameter1
        self.attribute2 = parameter2

    def method_name(self):
        # Method for the class
        pass

# Creating multiple objects
object1 = ClassName(value1a, value1b)
object2 = ClassName(value2a, value2b)
```

Class Attributes vs. Data Attributes:

- **Class Attributes**: Shared by all instances of the class, defined directly in the class body.

Syntax:

```
class ClassName:
    class_attribute = value  # Class attribute

    def __init__(self, instance_value):
        self.instance_attribute = instance_value  # Data attribute
```

- **Data Attributes**: Unique to each instance, defined in the __init__ constructor.

Syntax:

```
class ClassName:
    def __init__(self, parameter1, parameter2):
        self.attribute1 = parameter1  # Data attribute 1
        self.attribute2 = parameter2  # Data attribute 2
```

Encapsulation:

- Encapsulation restricts access to certain attributes or methods.
- Private attributes are defined using double underscores (__) before the attribute name.

Syntax:

```
class ClassName:
    def __init__(self):
        self.__private_attribute = value  # Private attribute

    def get_private_attribute(self):  # Public method to access the private attribute
        return self.__private_attribute

    def set_private_attribute(self, value):  # Public method to modify the private attribute
        self.__private_attribute = value
```

Inheritance:

- Inheritance allows a class (child class) to inherit attributes and methods from another class (parent class).

Syntax:

```
class ChildClass(ParentClass):
        pass
```

Polymorphism:

- Polymorphism allows methods to do different things based on the object it is acting upon.

Syntax:

```
class Base:
   def method_name(self):  # Method to be overridden
      pass

class DerivedA(Base):
   def method_name(self):  # Overriding the method
      # Implementation for DerivedA
      pass

class DerivedB(Base):
   def method_name(self):  # Overriding the method
      # Implementation for DerivedB
      pass

def polymorphic_function(obj):  # Function that uses polymorphism
   obj.method_name()  # Calls the method defined in the respective class
# Creating instances
obj_a = DerivedA()
obj_b = DerivedB()
# Calling the polymorphic function
polymorphic_function(obj_a)  # Calls DerivedA's method
polymorphic_function(obj_b)  # Calls DerivedB's method
```

Experiment 19

19. Write a python program to sort words in a file and put them in another file. The output file should have only lower-case words, so any upper-case words from source must be lowered.

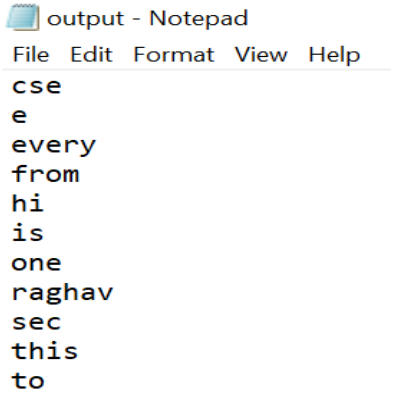
Aim: To sort words in a file and put them in another file. The output file should have only lower-case words, so any upper-case words from source must be lowered.

Program:

```
def sort_words_in_file(input_file, output_file):
   try:

# Open the input file and read the words
      with open(input_file, 'r') as f:
         words = f.read().split()

# Convert words to lowercase and sort them alphabetically
      words = [word.lower() for word in words]
      sorted_words = sorted(words)

# Write the sorted words to the output file
      with open(output_file, 'w') as f:
         for word in sorted_words:
            f.write(word + '\n')

      print(f"Words successfully sorted and written to {output_file}")

   except FileNotFoundError:
      print(f"The file {input_file} does not exist.")
   except Exception as e:
      print(f"An error occurred: {e}")

# Get input file and output file names from the user
input_file = input("Enter the input file name (with extension): ")
output_file = input("Enter the output file name (with extension): ")

sort_words_in_file(input_file, output_file)
ds_in_file(input_file, output_file)
```

Output:

Enter the input file name (with extension): c:\Notepad\input.txt
Enter the output file name (with extension): c:\Notepad\output.txt
Words successfully sorted and written to c:\Notepad\output.txt



Experiment 20

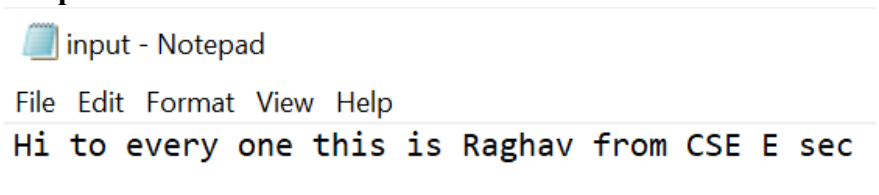
20. Python program to print each line of a file in reverse order.

Aim: To print each line of a file in reverse order.

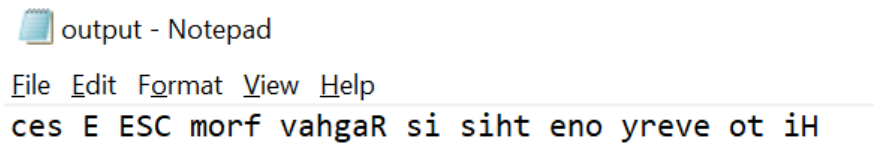
Program:

```
def print_lines_in_reverse(input_file_path, output_file_path):
    try:
        # Open the input file for reading
        with open(input_file_path, 'r') as input_file:
            lines = input_file.readlines()  # Read all lines in the file

 # Open the output file for writing
        with open(output_file_path, 'w') as output_file:

 # Reverse each line and write to the output file
            for line in lines:
                reversed_line = line.strip()[::-1]  # Reverse and strip whitespace
                output_file.write(reversed_line + '\n')  # Write reversed line to output

        print(f"Reversed lines successfully written to {output_file_path}")
    except FileNotFoundError:
        print(f"The file {input_file_path} does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")
# Get input and output file paths from the user
input_file_path = input("Enter the full path to the input file (with extension): ")
output_file_path = input("Enter the full path to the output file (with extension): ")

print_lines_in_reverse(input_file_path, output_file_path)
```

Output:




```
Enter the full path to the input file (with extension):  C:\Notepad\input.txt
Enter the full path to the output file (with extension):  C:\Notepad\output.txt
Reversed lines successfully written to C:\Notepad\output.txt
```



Experiment 21

21. Python program to compute the number of characters, words and lines in a file

Aim: To compute the number of characters, words and lines in a file

Program:

```
def count_file_contents():
    # Get the file path from the user
    file_path = input("Enter the full path to the input file (with extension): ")

    try:
        with open(file_path, 'r') as file:
            lines = file.readlines()  # Read all lines from the file

        num_lines = len(lines)  # Count lines
        num_words = 0  # Initialize word count
        num_characters = 0  # Initialize character count

        for line in lines:
            num_characters += len(line)  # Count characters in the line
            num_words += len(line.split())  # Count words in the line

        # Display results
        print(f"Number of lines: {num_lines}")
        print(f"Number of words: {num_words}")
        print(f"Number of characters: {num_characters}")

    except FileNotFoundError:
        print(f"The file {file_path} does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Call the function to execute the program
count_file_contents()
```

Output:

```
Enter the full path to the input file (with extension):  C:\Notepad\input.txt
Number of lines: 1
Number of words: 11
Number of characters: 47
```

Experiment 22

22. Python program to create, display, append, insert and reverse the order of the items in the array

Aim: To create, display, append, insert and reverse the order of the items in the array

Program:

```
def create_array():
    """Create an array with initial items."""
    return [1, 2, 3, 4, 5]  # Example initial items

def display_array(arr):
    """Display the items in the array."""
    print("Current array:", arr)

def append_to_array(arr, item):
    """Append an item to the end of the array."""
    arr.append(item)
    print(f"Item {item} appended to the array.")

def insert_into_array(arr, index, item):
    """Insert an item at a specific index in the array."""
    if 0 <= index <= len(arr):
        arr.insert(index, item)
        print(f"Item {item} inserted at index {index}.")
    else:
        print("Index out of bounds.")

def reverse_array(arr):
    """Reverse the order of the items in the array."""
    arr.reverse()
    print("Array reversed.")

def main():
    # Create an array
    array = create_array()

    while True:
        print("\nMenu:")
        print("1. Display array\t 2. Append item")
        print("3. Insert item\t 4. Reverse array\t 5. Exit")
        choice = input("Enter your choice (1-5): ")
        if choice == '1':
            display_array(array)
        elif choice == '2':
            item = input("Enter item to append: ")
```

```
            append_to_array(array, item)
        elif choice == '3':
            index = int(input("Enter index to insert at: "))
            item = input("Enter item to insert: ")
            insert_into_array(array, index, item)
        elif choice == '4':
            reverse_array(array)
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please select again.")

if __name__ == "__main__":
    main()
```

Output:

```
Menu:
1. Display array          2. Append item
3. Insert item    4. Reverse array          5. Exit
Enter your choice (1-5):  1
Current array: [1, 2, 3, 4, 5]

Menu:
1. Display array          2. Append item
3. Insert item    4. Reverse array          5. Exit
Enter your choice (1-5):  2
Enter item to append:  6, 9, 10
Item 6, 9, 10 appended to the array.

Menu:
1. Display array          2. Append item
3. Insert item    4. Reverse array          5. Exit
Enter your choice (1-5):  3
Enter index to insert at:  3
Enter item to insert:  20
Item 20 inserted at index 3.

Menu:
1. Display array          2. Append item
3. Insert item    4. Reverse array          5. Exit
Enter your choice (1-5):  4
Array reversed.

Menu:
1. Display array          2. Append item
3. Insert item    4. Reverse array          5. Exit
```

```
Enter your choice (1-5):  1
Current array: ['6, 9, 10', 5, 4, '20', 3, 2, 1]

Menu:
1. Display array          2. Append item
3. Insert item    4. Reverse array          5. Exit
Enter your choice (1-5):  5
Exiting the program.
```

Experiment 23

23. Python program to add, transpose and multiply two matrices.

Aim: To add, transpose and multiply two matrices.

Program:

```
def input_matrix(rows, cols):
    """Input a matrix from the user."""
    matrix = []
    for _ in range(rows):
        row = list(map(int, input("Enter row (space-separated): ").split()))
        matrix.append(row)
    return matrix

def add_matrices(matrix1, matrix2):
    """Add two matrices."""
    return [[matrix1[i][j] + matrix2[i][j] for j in range(len(matrix1[0]))] for i in range(len(matrix1))]

def transpose_matrix(matrix):
    """Transpose a matrix."""
    return [[matrix[j][i] for j in range(len(matrix))] for i in range(len(matrix[0]))]

def multiply_matrices(matrix1, matrix2):
    """Multiply two matrices."""
    result = []
    for i in range(len(matrix1)):
        row = []
        for j in range(len(matrix2[0])):
            sum = 0
            for k in range(len(matrix2)):
                sum += matrix1[i][k] * matrix2[k][j]
            row.append(sum)
        result.append(row)
    return result

def main():
    # Input first matrix
    rows1 = int(input("Enter the number of rows for the first matrix: "))
    cols1 = int(input("Enter the number of columns for the first matrix: "))
    print("Enter the first matrix:")
    matrix1 = input_matrix(rows1, cols1)

    # Input second matrix
    rows2 = int(input("Enter the number of rows for the second matrix: "))
    cols2 = int(input("Enter the number of columns for the second matrix: "))
    print("Enter the second matrix:")
```

```
    matrix2 = input_matrix(rows2, cols2)

    # Matrix Addition
    if rows1 == rows2 and cols1 == cols2:
        print("\nAddition of matrices:")
        result_add = add_matrices(matrix1, matrix2)
        for row in result_add:
            print(row)
    else:
        print("Matrices cannot be added (dimensions do not match).")

    # Matrix Transpose
    print("\nTranspose of the first matrix:")
    transposed1 = transpose_matrix(matrix1)
    for row in transposed1:
        print(row)

    print("\nTranspose of the second matrix:")
    transposed2 = transpose_matrix(matrix2)
    for row in transposed2:
        print(row)

    # Matrix Multiplication
    if cols1 == rows2:
        print("\nMultiplication of matrices:")
        result_multiply = multiply_matrices(matrix1, matrix2)
        for row in result_multiply:
            print(row)
    else:
        print("Matrices cannot be multiplied (inner dimensions do not match).")

if __name__ == "__main__":
    main()
```

Output:

```
Enter the number of rows for the first matrix:  2
Enter the number of columns for the first matrix:  2
Enter row (space-separated):  1 2
Enter row (space-separated):  3 4
Enter the number of rows for the second matrix:  2
Enter the number of columns for the second matrix:  2
Enter row (space-separated):  5 6
Enter row (space-separated):  7 8

Addition of matrices:
[6, 8]
[10, 12]
```

Transpose of the first matrix:
[1, 3]
[2, 4]

Transpose of the second matrix:
[5, 7]
[6, 8]

Multiplication of matrices:
[19, 22]
[43, 50]

Experiment 24

24. Python program to create a class that represents a shape. Include methods to calculate its area and perimeter. Implement subclasses for different shapes like circle, triangle, and square.

Aim: To create a class that represents a shape. Include methods to calculate its area and perimeter. Implement subclasses for different shapes like circle, triangle, and square.

Program:

```
import math

class Shape:
    def area(self):
        raise NotImplementedError

    def perimeter(self):
        raise NotImplementedError

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius

class Triangle(Shape):
    def __init__(self, base, height, side1, side2):
        self.base = base
        self.height = height
        self.side1 = side1
        self.side2 = side2

    def area(self):
        return 0.5 * self.base * self.height

    def perimeter(self):
        return self.base + self.side1 + self.side2

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2
```

```
    def perimeter(self):
        return 4 * self.side

def main():
    shapes = []
    # Get Circle input
    radius = float(input("Enter the radius of the circle: "))
    shapes.append(Circle(radius))
    # Get Triangle input
    base = float(input("Enter the base of the triangle: "))
    height = float(input("Enter the height of the triangle: "))
    side1 = float(input("Enter the length of side 1 of the triangle: "))
    side2 = float(input("Enter the length of side 2 of the triangle: "))
    shapes.append(Triangle(base, height, side1, side2))
    # Get Square input
    side = float(input("Enter the side length of the square: "))
    shapes.append(Square(side))
    # Display area and perimeter for each shape
    for shape in shapes:
        print(f"{shape.__class__.__name__}: Area = {shape.area():.2f}, Perimeter =
{shape.perimeter():.2f}")

if __name__ == "__main__":
    main()
```

Output:

```
Enter the radius of the circle:  4
Enter the base of the triangle:  4
Enter the height of the triangle:  4
Enter the length of side 1 of the triangle:  4
Enter the length of side 2 of the triangle:  4
Enter the side length of the square:  4
Circle: Area = 50.27, Perimeter = 25.13
Triangle: Area = 8.00, Perimeter = 12.00
Square: Area = 16.00, Perimeter = 16.00
```

UNIT-5

Introduction to Data Science:

Functional Programming in Python:

- Functional programming is a programming paradigm where functions are treated as first-class citizens. This means functions can be passed as arguments, returned from other functions, and assigned to variables.

Key concepts include:

- **Higher-Order Functions:** Functions that take other functions as arguments or return them.
- **Lambda Functions:** Anonymous functions defined with the lambda keyword.
- **Map, Filter, and Reduce:** Functions used to apply operations on collections.

Syntax:

```
# Higher-Order Function
def apply_function(func, value):
   return func(value)
# Lambda Function
square = lambda x: x * x
# Using map
numbers = [1, 2, 3, 4]
squared_numbers = list(map(square, numbers))
# Using filter
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
# Using reduce
from functools import reduce
total = reduce(lambda x, y: x + y, numbers)
```

JSON and XML in Python:

- Both JSON (JavaScript Object Notation) and XML (eXtensible Markup Language) are widely used formats for data interchange.
- JSON represents data in key-value pairs. It uses objects (curly braces) and arrays (square brackets) to structure data.
- XML represents data in a tree structure using nested tags.
- Python provides libraries to work with both formats.
-

Syntax for JSON:

```
import json
# Converting Python object to JSON
data = {"name": "Alice", "age": 30}
json_data = json.dumps(data)  # Convert to JSON string
# Converting JSON string to Python object
python_data = json.loads(json_data)  # Convert back to Python object
```

Syntax for XML:

```
import xml.etree.ElementTree as ET
# Creating XML
root = ET.Element("data")
child = ET.SubElement(root, "item")
child.text = "Sample Item"
```

```
xml_data = ET.tostring(root)
# Parsing XML
tree = ET.ElementTree(ET.fromstring(xml_data))
root = tree.getroot()
for item in root:
   print(item.text)  # Output: Sample Item
```

NumPy with Python:

- NumPy (Numerical Python) is a library used for numerical computing in Python.
- It provides support for arrays, matrices, and many mathematical functions.

Syntax:

```
import numpy as np
# Creating a NumPy array
array = np.array([1, 2, 3, 4])
# Basic operations
sum_array = np.sum(array)
mean_array = np.mean(array)
reshaped_array = array.reshape(2, 2)  # Reshape array to 2x2 matrix
```

Pandas:

- Pandas is a powerful data manipulation and analysis library for Python.
- It provides data structures like Series and DataFrames for handling structured data.

Syntax:

```
import pandas as pd
# Creating a DataFrame
data = {
   "Name": ["Alice", "Bob", "Charlie"],
   "Age": [30, 25, 35]
}
df = pd.DataFrame(data)
# Basic operations
mean_age = df["Age"].mean()  # Calculate mean age
filtered_df = df[df["Age"] > 30]  # Filter rows where age > 30
```

Experiment 25

25. Python program to check whether a JSON string contains complex object or not.

Aim: To check whether a JSON string contains complex object or not.
Program:

```
import json

def is_complex_object(obj):
    """Check if the object is a complex object (dict or list)."""
    return isinstance(obj, (dict, list))

def check_json_complexity(json_string):
    try:
        # Parse the JSON string
        parsed_json = json.loads(json_string)

        # Check if the parsed JSON is a complex object
        if is_complex_object(parsed_json):
            return True
        else:
            return False
    except json.JSONDecodeError:
        print("Invalid JSON string.")
        return False
# Example JSON strings
json_string_1 = '{"name": "Alice", "age": 30, "hobbies": ["reading", "hiking"]}'  # complex object
json_string_2 = '"Hello, world!"'  # simple string
json_string_3 = '12345'  # simple number
# Check if they are complex objects
print(f'JSON String 1 is complex: {check_json_complexity(json_string_1)}')
print(f'JSON String 2 is complex: {check_json_complexity(json_string_2)}')
print(f'JSON String 3 is complex: {check_json_complexity(json_string_3)}')
```

Output:

```
JSON String 1 is complex: True
JSON String 2 is complex: False
JSON String 3 is complex: False
```

Experiment 26

26. Python Program to demonstrate NumPy arrays creation using array () function.

Aim: To demonstrate NumPy arrays creation using array () function.

Program:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Experiment 27

27. Python program to demonstrate use of ndim, shape, size, dtype.

Aim: To demonstrate use of ndim, shape, size, dtype

Program:

```
import numpy as np

# Create a 2D NumPy array
array_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Display the array
print("Array:")
print(array_2d)

# Demonstrate the use of ndim
print("\nNumber of dimensions (ndim):", array_2d.ndim)

# Demonstrate the use of shape
print("Shape of the array (shape):", array_2d.shape)

# Demonstrate the use of size
print("Total number of elements (size):", array_2d.size)

# Demonstrate the use of dtype
print("Data type of the elements (dtype):", array_2d.dtype)
```

Output:

```
Array:
[[1 2 3]
 [4 5 6]]

Number of dimensions (ndim): 2
Shape of the array (shape): (2, 3)
Total number of elements (size): 6
Data type of the elements (dtype): int32
```

Experiment 28

28. Python program to demonstrate basic slicing, integer and Boolean indexing

Aim: To demonstrate basic slicing, integer and Boolean indexing

Program:

```
import numpy as np
# Create a sample NumPy array
array = np.array([[10, 20, 30, 40],
                  [50, 60, 70, 80],
                  [90, 100, 110, 120]])

print("Original Array:")
print(array)
# Basic slicing
print("\nBasic Slicing (first two rows and first three columns):")
sliced_array = array[:2, :3]
print(sliced_array)
# Integer indexing
print("\nInteger Indexing (selecting specific elements):")
# Selecting elements at (0, 1), (1, 2), and (2, 3)
integer_indexed_array = array[[0, 1, 2], [1, 2, 3]]
print(integer_indexed_array)
# Boolean indexing
print("\nBoolean Indexing (selecting elements greater than 60):")
boolean_indexed_array = array[array > 60]
print(boolean_indexed_array)
```

Output:

```
Original Array:
[[ 10  20  30  40]
 [ 50  60  70  80]
 [ 90 100 110 120]]

Basic Slicing (first two rows and first three columns):
[[10 20 30]
 [50 60 70]]
Integer Indexing (selecting specific elements):
[ 20  70 120]
Boolean Indexing (selecting elements greater than 60):
[ 70  80  90 100 110 120]
```

Experiment 29

29. Python program to find min, max, sum, cumulative sum of array

Aim: To find min, max, sum, cumulative sum of array

Program:

```
import numpy as np

# Create a sample NumPy array
array = np.array([5, 10, 15, 20, 25])

# Find the minimum value
min_value = np.min(array)
print("Minimum value in the array:", min_value)

# Find the maximum value
max_value = np.max(array)
print("Maximum value in the array:", max_value)

# Find the sum of all elements
total_sum = np.sum(array)
print("Sum of all elements in the array:", total_sum)

# Find the cumulative sum of the array
cumulative_sum = np.cumsum(array)
print("Cumulative sum of the array:", cumulative_sum)
```

Output:

```
Minimum value in the array: 5
Maximum value in the array: 25
Sum of all elements in the array: 75
Cumulative sum of the array: [ 5 15 30 50 75]
```

Experiment 30

30. Create a dictionary with at least five keys and each key represent value as a list where this list contains at least ten values and convert this dictionary as a pandas data frame and explore the data through the data frame as follows:
a) Apply head () function to the pandas data frame.
b) Perform various data selection operations on Data Frame

Aim: To demonstrate dictionary with at least five keys and each key represent value as a list where this list contains at least ten values and convert this dictionary as a pandas data frame and explore the data through the data frame as follows:
a) Apply head () function to the pandas data frame.
b) Perform various data selection operations on Data Frame

Program:

```
import pandas as pd
# Step 1: Create a dictionary with at least five keys, each representing a list of ten values
data = {
   'Product': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
   'Price': [10.99, 12.50, 8.75, 5.00, 15.99, 7.50, 9.99, 20.00, 13.50, 11.25],
   'Stock': [100, 200, 150, 300, 120, 80, 60, 40, 250, 10],
   'Category': ['Electronics', 'Electronics', 'Home', 'Home', 'Fashion',
            'Fashion', 'Toys', 'Toys', 'Books', 'Books'],
   'Rating': [4.5, 4.0, 4.2, 3.8, 4.9, 4.0, 3.5, 4.3, 4.7, 4.1]
}
# Step 2: Convert the dictionary into a pandas DataFrame
df = pd.DataFrame(data)
# Step 3: Apply head() function to the pandas DataFrame
print("First five rows of the DataFrame:")
print(df.head())
# Step 4: Perform various data selection operations on the DataFrame
# 4a. Selecting a single column (Product)
print("\nSelecting the 'Product' column:")
product_column = df['Product']
print(product_column)
# 4b. Selecting multiple columns (Price and Stock)
print("\nSelecting 'Price' and 'Stock' columns:")
price_and_stock = df[['Price', 'Stock']]
print(price_and_stock)
# 4c. Selecting rows where Price is greater than 10
print("\nSelecting rows where Price > 10:")
high_price = df[df['Price'] > 10]
print(high_price)

# 4d. Selecting the first five rows using index position
print("\nSelecting the first five rows using iloc:")
first_five_rows = df.iloc[:5]
print(first_five_rows)
```

```
# 4e. Selecting a specific row by index (for example, index 2)
print("\nSelecting the row at index 2:")
specific_row = df.iloc[2]
print(specific_row)
```

Output:

```
First five rows of the DataFrame:
  Product  Price  Stock     Category  Rating
0       A  10.99    100  Electronics     4.5
1       B  12.50    200  Electronics     4.0
2       C   8.75    150         Home     4.2
3       D   5.00    300         Home     3.8
4       E  15.99    120      Fashion     4.9

Selecting the 'Product' column:
0    A
1    B
2    C
3    D
4    E
5    F
6    G
7    H
8    I
9    J
Name: Product, dtype: object

Selecting 'Price' and 'Stock' columns:
   Price  Stock
0  10.99    100
1  12.50    200
2   8.75    150
3   5.00    300
4  15.99    120
5   7.50     80
6   9.99     60
7  20.00     40
8  13.50    250
9  11.25     10

Selecting rows where Price > 10:
  Product  Price  Stock     Category  Rating
0       A  10.99    100  Electronics     4.5
1       B  12.50    200  Electronics     4.0
4       E  15.99    120      Fashion     4.9
7       H  20.00     40         Toys     4.3
8       I  13.50    250        Books     4.7
9       J  11.25     10        Books     4.1
```

Selecting the first five rows using iloc:

	Product	Price	Stock	Category	Rating
0	A	10.99	100	Electronics	4.5
1	B	12.50	200	Electronics	4.0
2	C	8.75	150	Home	4.2
3	D	5.00	300	Home	3.8
4	E	15.99	120	Fashion	4.9

Selecting the row at index 2:

Product C
Price 8.75
Stock 150
Category Home
Rating 4.2
Name: 2, dtype: object

Experiment 31

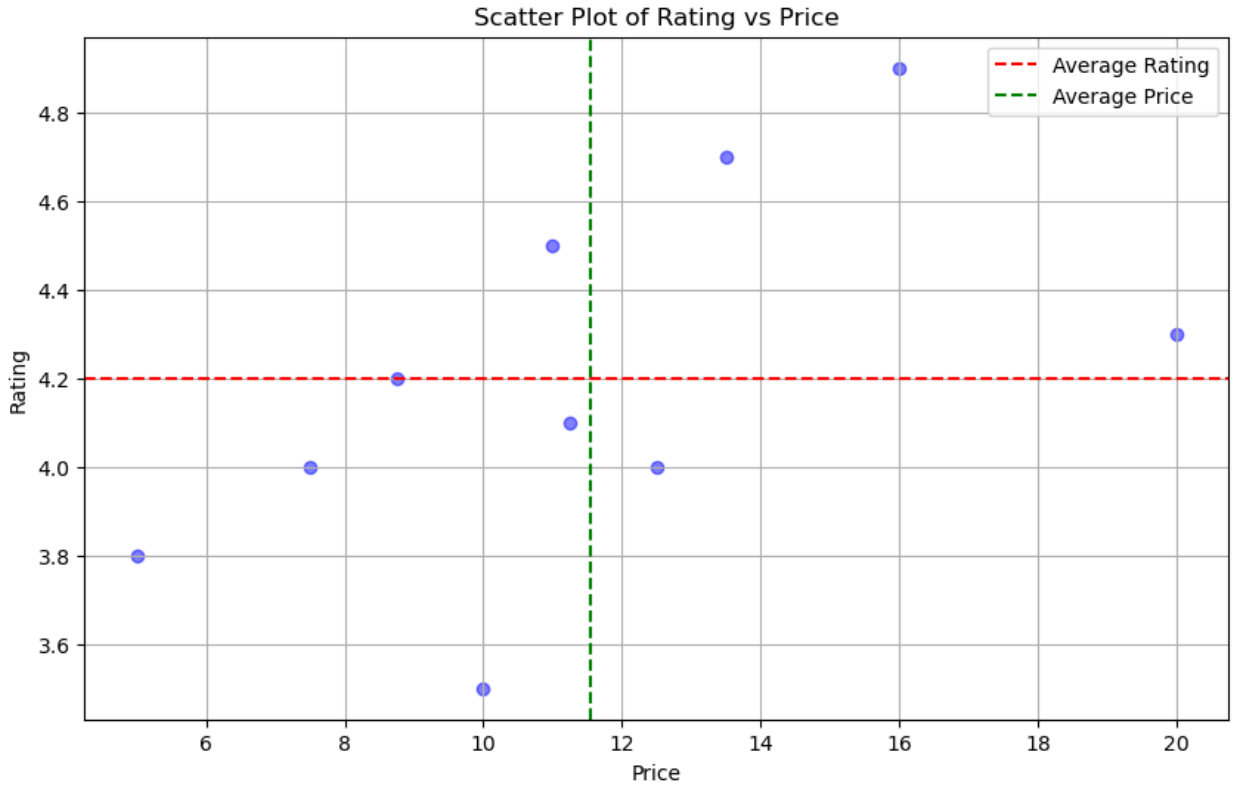
31. Select any two columns from the above data frame, and observe the change in one attribute with respect to other attribute with scatter and plot operations in matplotlib

Aim: Select any two columns from the above data frame, and observe the change in one attribute with respect to other attribute with scatter and plot operations in matplotlib

Program:

```
import pandas as pd
import matplotlib.pyplot as plt

# Step 1: Create a dictionary with at least five keys, each representing a list of ten values
data = {
   'Product': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
   'Price': [10.99, 12.50, 8.75, 5.00, 15.99, 7.50, 9.99, 20.00, 13.50, 11.25],
   'Stock': [100, 200, 150, 300, 120, 80, 60, 40, 250, 10],
   'Category': ['Electronics', 'Electronics', 'Home', 'Home', 'Fashion',
            'Fashion', 'Toys', 'Toys', 'Books', 'Books'],
   'Rating': [4.5, 4.0, 4.2, 3.8, 4.9, 4.0, 3.5, 4.3, 4.7, 4.1]
}

# Step 2: Convert the dictionary into a pandas DataFrame
df = pd.DataFrame(data)

# Step 3: Select two columns to explore their relationship
x_column = 'Price'  # Independent variable
y_column = 'Rating'  # Dependent variable

# Step 4: Create a scatter plot to observe the relationship
plt.figure(figsize=(10, 6))
plt.scatter(df[x_column], df[y_column], color='blue', alpha=0.5)
plt.title('Scatter Plot of Rating vs Price')
plt.xlabel('Price')
plt.ylabel('Rating')
plt.grid()
plt.axhline(y=df[y_column].mean(), color='r', linestyle='--', label='Average Rating')
plt.axvline(x=df[x_column].mean(), color='g', linestyle='--', label='Average Price')
plt.legend()
plt.show()

# Step 5: Display the DataFrame for reference
print("DataFrame:")
print(df)
```

Ouptut:



DataFrame:

	Product	Price	Stock	Category	Rating
0	A	10.99	100	Electronics	4.5
1	B	12.50	200	Electronics	4.0
2	C	8.75	150	Home	4.2
3	D	5.00	300	Home	3.8
4	E	15.99	120	Fashion	4.9
5	F	7.50	80	Fashion	4.0
6	G	9.99	60	Toys	3.5
7	H	20.00	40	Toys	4.3
8	I	13.50	250	Books	4.7
9	J	11.25	10	Books	4.

Additional Experiment
Experiment 32

32. Python program to check whether a XML string contains complex object or not.

Program:

```
import xml.etree.ElementTree as ET
def is_complex_object(element):
    # A complex object can be defined as:
    # - Element has child elements, or
    # - Element has attributes

    if len(element) > 0 or len(element.attrib) > 0:
        return True
    return False
def contains_complex_object(xml_string):
    try:
        # Parse the XML string into an ElementTree object
        root = ET.fromstring(xml_string)
        # Check if root or any child element is a complex object
        if is_complex_object(root):
            return True
        # Recursively check child elements for complex objects
        for child in root:
            if is_complex_object(child):
                return True
        return False
    except ET.ParseError as e:
        print(f"Invalid XML: {e}")
        return False
# Test the function
xml_string = """
<person>
    <name>John</name>
    <address>
        <street>Main St</street>
        <city>New York</city>
    </address>
</person>
"""
if contains_complex_object(xml_string):
    print("The XML contains a complex object.")
else:
    print("The XML does not contain a complex object.")
```

Output:

```
The XML contains a complex object.
```