ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B. E. (CSE)

IV Semester

22CSCP410 - Python Programming Lab

Name : ....

Reg. No. : ....

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B. E. (CSE)

IV Semester

22CSCP410 - Python Programming Lab

Certified that this is a bona fide record of work done by

Mr./Ms. ....

Reg. No. .... of B.E. (CSE) in the

22CSCP410 – PYTHON PROGRAMMING LAB during the

even semester of the academic year 2023–24.

Staff-in-charge

Internal Examiner

Place: Annamalai Nagar
Date :

External Examiner

CONTENTS

Annamalai University
Department of Computer Science and Engineering

VISION

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISSION

- Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.

- Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment.

- Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs.

- Expose the students to the emerging technological advancements for meeting the demands of the industry.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO	PEO Statements
PEO1	To prepare the graduates with the potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursue higher education and carry out research in the field of Computer Science.
PEO3	To equip the graduates with the skills required to stay motivated and adapt to the dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates to communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

PROGRAM OUTCOMES (POs)

S. No.	Program Outcomes
PO1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem Analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
PO3	**Design/Development of Solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct Investigations of Complex Problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern Tool Usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The Engineer and Society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

S.no	Program Specific Outcomes
PSO1	Acquire the ability to understand basic sciences, humanity sciences, basic engineering sciences and fundamental core courses in Computer Science and Engineering to realize and appreciate real life problems in diverse fields for proficient design of computer based systems of varying complexity.
PSO2	Learn specialized courses in Computer Science and Engineering to build up the aptitude for applying typical practices and approaches to deliver quality products intended for business and industry requirements.
PSO3	Apply technical and programming skills in Computer Science and Engineering essential for employing current techniques in software development crucial in industries, to create pioneering career paths for pursuing higher studies, research and to be an entrepreneur.

***

Rubrics for Laboratory Examination (Internal/External)

(**Internal:** Two tests - 15 marks each, **External:** Two questions - 25 marks each)

Rubric	**Poor Up to (1/2)**	**Average Up to (2/4)**	**Good Up to (3/6)**	**Excellent Up to (5/8*)**
Syntax and Logic **Ability to understand, specify the data structures appropriate for the problem domain**	Program does not compile with typographical errors and incorrect logic leading to infinite loops.	Program compiles that signals major syntactic errors and logic shows severe errors.	Program compiles with minor syntactic errors and logic is mostly correct with occasional errors.	Program compiles with evidence of good syntactic understanding of the syntax and logic used.
Modularity **Ability to decompose a problem into coherent and reusable functions, files, classes, or objects (as appropriate for the programming language and platform).**	Program is one big Function or is decomposed in ways that make little/no sense.	Program is decomposed into units of appropriate size, but they lack coherence or reusability. Program contains unnecessary repetition.	Program is decomposed into coherent units, but may still contain some unnecessary repetition.	Program is decomposed into coherent and reusable units, and unnecessary repetition are eliminated.
Clarity and Completeness **Ability to code formulae and algorithms that produce appropriate results. Ability to apply rigorous test case analysis to the problem domain.**	Program does not produce appropriate results for most inputs. Program shows little/no ability to apply different test cases.	Program approaches appropriate results for most inputs, but contain some miscalculations. Program shows evidence of test case analysis, but missing significant test cases or mistaken some test cases.	Program produces appropriate results for most inputs. Program shows evidence of test case analysis that is mostly complete, but missed to handle all possible test cases.	Program produces appropriate results for all inputs tested. Program shows evidence of excellent test case analysis, and all possible cases are handled appropriately.

* 8 marks for syntax and logic, 8 marks for modularity, and 9 marks for Clarity and Completeness.

Rubric for CO3

Rubric for CO3 in Laboratory Courses				
Rubric	**Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks**			
	Up To 2.5 Marks	**Up To 5 Marks**	**Up To 7.5 Marks**	**Up To 10 marks**
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.

Ex No: 01 **TUPLES**
Date:

Theoretical Concepts:

Tuples are used to store multiple items in a single variable.

Tuples is one of 4 built-in data types in Python used to store collections of data, the other are List, Set, and Dictionary, all with different qualities and usage.

A tuples is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

Example:

Create a Tuples:

```
thistuple = ("apple", "banana", "cherry") print(thistuple)
```

Tuples items are ordered, unchangeable, and allow duplicate values.

Tuples items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Tuples items can be of any data type: String, int and boolean data types.

```
  tuple1 = ("apple", "banana", "cherry") tuple2 = (1, 5, 7, 9, 3) tuple3 =
(True, False, False)
```

Aim:

To create a python function that takes the list and returns a new dictionary where the keys are student names and the values are their average scores using tuples unpacking and list comprehension.

Algorithm:

1) In this code, we use list comprehension to iterate through each tuple in list of students.
2) Using tuples unpacking, we assign the first element of tuple to variable ‘name’ and the remaining elements to variable ‘score’.

3) Then, we calculate the average score by summing up the scores and dividing by number of scores.
4) We create a dictionary comprehension to build the dictionary with student names as keys and their scores as values.
5) You can call the `‘average_scores’` function with your list of tuples and it will return the desired dictionary.

Source code:

```
def average_scores(studentd):
    scr_dict = {}
    for name, *scr in studentd:
        avgscr = sum(scr) / len(scr)
        scr_dict[name] = avgscr
    return scr_dict

std1 = [('Abi', 85, 90, 92), ('Bala', 78, 89, 90), ('Dharshini', 92,
       88, 95)]
c = average_scores(std1)
print(c)
```

Sample Input and Output:

```
{‘Abi’:89.0,‘Bala’:85.6666667,‘Dharshini’:91.6666667}
```

Result:

Thus, a python function that converts a list of tuples into dictionary is successfully created.

Ex No: 02 **LIST**

Date:

Theoretical Concepts:

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:

Example:

Create a List:

```
thislist = ["apple", "banana", "cherry"] print(thislist)
```

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.

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

List items can be of any data type: String, int and boolean data types.

```
list1 = ["apple", "banana", "cherry"] list2 = [1, 5, 7, 9, 3] list3 = [True,
False, False]
```

Aim:

To create a python function that takes a list of integers as input and returns a list all unique combinations of two numbers that sum to a prime number.

Algorithm:

1) Define a function `get_prime_sum_combinations(numbers)` that takes a list of integers as input.
2) Define a helper function `is_prime(n)` that checks if a number n is prime.
    - If n is less than 2, return False.
    - Iterate from 2 to the square root of n and check if n is divisible by any number in that range. If it is, return False.
    - If no divisor is found, return True.

3) Initialize an empty list called combinations to store the unique combinations of two numbers.
4) Iterate over the range of the length of the numbers list, using i as the index of the first number.
    - Within this loop, iterate over the range from `i+1` to the length of the numbers list, using j as the index of the second number.
    - Create a tuple called pair with the two numbers at indices `i` and `j`.
    - Check if the sum of the pair is a prime number using the `is_prime()` function.
    - If it is prime, append the pair to the combinations list.
5) Return the combinations list.
6) Test the function by calling `get_prime_sum_combinations()` with a list of integers.

Source code:

```
def get_prime_sum_combinations(numbers):
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                return False
        return True  # Return True if the number is prime

    combinations = []
    for i in range(len(numbers)):
        for j in range(i+1, len(numbers)):
            pair = (numbers[i], numbers[j])
            if is_prime(sum(pair)):
                combinations.append(pair)
    return combinations

numbers = [76, 87, 98, 78, 66, 87, 45, 23]
print(get_prime_sum_combinations(numbers))
```

Sample Input and Output:

```
[(76,87), (76,87), (78,23), (66,23)]
```

Results:

Thus, a python function that converts a list of integers into a list of all unique combinations of two numbers that sum to a prime number is successfully created.

Ex No: 03 **SET**

Date:

Theoretical Concepts:

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 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 unchangeable, but you can remove items and add new items.

Sets are unordered, so you cannot be sure in which order the items will appear.

Sets are written with curly brackets.

Example:

Create a Set:

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

Set items can be of any data type: String, int and boolean data types.

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

Aim:

To create a python function that takes two sets as input and returns a new set containing elements that are common to both sets.

Algorithm:

1) Start by defining the `find_common_elements` function that takes two sets, `set1` and `set2`, as input.
2) Inside the function, use the `&` operator to find the intersection of `set1` and `set2`.
3) Return the result of the intersection operation.
4) Outside the function, create two sets, `set_a` and `set_b`, with the desired elements.
5) Call the `find_common_elements` function, passing `set_a` and `set_b` as arguments.

6) Store the result in a variable, such as result.

7) Print the value of result to display the common elements between the two sets.

Source code:

```
def find_common_elements(set1, set2):
    return set1 & set2
set_a = {1, 2, 3, 4, 5}
set_b = {3, 4, 5, 6, 7}
result = find_common_elements(set_a, set_b)
print(result)
```

Sample Input and Output:

```
{3,4,5}
```

Results:

Thus, a python function that takes two sets as input and returns a new set containing common elements is successfully created.

Ex No: 04 DICTIONARY

Date:

Theoretical Concepts:

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.

Dictionaries are written with curly brackets, and have keys and values:

Example:

Create and print a dictionary:

```
thisdict = { "brand": "Ford", "model": "Mustang", "year": 1964 }
print(thisdict)
```

Dictionary items are ordered, changeable, and do not allow duplicates.

Dictionary items are presented in `key:value` pairs, and can be referred to by using the key name.

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

The values in dictionary items can be of any data type: String, int, boolean, and list data types.

```
thisdict = { "brand": "Ford", "electric": False, "year": 1964, "colors":
              ["red", "white", "blue"] }
```

Aim:

To create a python function that takes two dictionaries as input and returns a new dictionary containing merged key-value pairs. If there are common keys, sum the corresponding values.

Algorithm:

1) Start by defining the `merge_dictionaries` function that takes two dictionaries, `dict1` and `dict2`, as input.
2) Inside the function, use a dictionary comprehension to create a new dictionary.
3) Iterate over the union of keys from `dict1` and `dict2` using the set function and the | operator.
4) For each key, use the get method to retrieve the corresponding values from `dict1` and `dict2`. If a key is missing in either dictionary, default to 0.
5) Add the values together and assign the result as the value for the key in the new dictionary.

6) Return the new dictionary.
7) Outside the function, create two dictionaries, `dict_a` and `dict_b`, with the desired key-value pairs.
8) Call the `merge_dictionaries` function, passing `dict_a` and `dict_b` as arguments.
9) Store the result in a variable, such as result.
10) Print the value of result to display the merged dictionary.

Source code:

```
def merge_dictionaries(dict1, dict2):
    return {key: dict1.get(key, 0) + dict2.get(key, 0) for key in
set(dict1) | set(dict2)}
dict_a = {'a': 1, 'b': 2, 'c': 3}
dict_b = {'b': 3, 'c': 4, 'd': 5}
result = merge_dictionaries(dict_a, dict_b)
print(result)
```

Sample Input and Output:

```
{‘a’:1,’b’:5,’c’:7,’d’:5}
```

Results:

Thus, a python function of merged new dictionary is successfully created.

Ex No: 05 **CONDITIONAL STATEMENT**

Date:

Theoretical Concepts:

In Python, the if statement is a conditional statement used for decision making. there are several types of if statements,

1.Simple if statement:

It checks a condition and executes a block of code if the condition is true.

Syntax:

```
if condition:
  # code block
```

Example:

```
x= 10
if x > 5:
  print(“x is greater than 5”) #output: x is greater than 5
```

2.If-else statement:

It checks a condition and executes one block of code if the condition is true, and another block if the condition is false.

Syntax:

```
if condition:
       # code block if condition is true
else:
       # code block if condition is false
```

Example:

```
x= 3
if x % 2 == 0:
  print(“x is even”)
else:
  print(“x is odd”)
```

3.If-elif-else statement:

It checks multiple conditions sequentially and executes the block of code associated with the first true condition. If none of the conditions are true, it executes the else block.

Syntax:

```
if condition1:
  # code block if condition1 is true
elif condition2:
  # code block if condition2 is true
else:
  # code block if no condition is false
```

Example:

```
x = 10
if x < 0:
 print(“x is negative”)
elif x == 0:
 print(“x is zero”)
else:
 print(“x is positive”)
```

4.Nested if statements:

If statements can be nested within each other to create more complex conditional logic.

Syntax:

```
if condition1:
 if condition2:
     # code block
```

Example:

```
x= 10
if x > 0:
 if x % 2 == 0:
     print(“x is a positive even number”)
 else:
     print(“x is a positive odd number”)
else:
  print(“x is non-positive”)
```

Aim:

To write a Python program that calculates a student’s grade using if statements.

Algorithm:

1. Define the weights for exams, assignments, and participation.
2. Input the scores for exams, assignments, and participation.
3. Calculate the overall score using the weighted averages.
4. Apply conditions to determine if any individual component score is below 40 or if participation score is 0.
5. Determine the final grade based on the overall score and the specified grade ranges.

Source code:

```
def calculate_grade(exam_score, assignment_score, participation_score):
    exam_weight = 0.4
    assignment_weight = 0.3
    participation_weight = 0.3
    overall_score = (exam_score * exam_weight) + (assignment_score *
    assignment_weight) + (participation_score * participation_weight)
    if exam_score < 40 or assignment_score < 40 or participation_score == 0:
        return "F"
    elif overall_score >= 90:
        return "A"
    elif overall_score >= 80:
        return "B"
    elif overall_score >= 70:
        return "C"
    elif overall_score >= 60:
        return "D"
    else:
        return "F"

exam_score = float(input("Enter exam score: "))
assignment_score = float(input("Enter assignment score: "))
participation_score = float(input("Enter participation score: "))
# Calculate grade
final_grade = calculate_grade(exam_score, assignment_score,
participation_score)
print("Final Grade:", final_grade)
```

Sample Input and Output:

```
Enter exam score: 90
Enter assignment score: 95
Enter participation score: 90
Final Grade: A
```

Results:

Thus, Final grade calculator using student ‘s exam score, assignment score and participation score have been implemented in python language and tested for various sample inputs.

Ex No: 06 STRING MANIPULATION

Date:

Theoretical Concepts:

String manipulation in Python involves performing various operations on strings, such as concatenation, splitting, slicing, formatting, and modifying the case. Below is a description along with syntax and examples for some common string manipulation operations:

1. **Concatenation:** Combining two or more strings into a single string.

```
string1 = “Hello”
string2 = “World”
New_string = string1 + “ “ + string2 # Output: “Hello World”
```

2. **String Slicing:** Extracting a substring from a string.

```
string = “Python”
Substring = string[1:4] # Output: “yth”
```

3. **String Formatting:** Inserting variable values into a string.

```
name = “ABC”
age = 18
formatted_string = f”My name is {name} and I am {age} years old.”
# Output: “My name is ABC and I am 18 years old.”
index = string.find(“World”) # Output: 6
```

4. **Changing case:** Converting the case of characters in a string.

```
string = “hello world”
upper_case = string.upper() # Output: “HELLO WORLD”
title_case = string.title() # Output: “Hello World”
```

5. **Splitting:** Breaking a string into a list of substrings based on a delimiter.

```
string = “apple,banana,orange”
string_list = string.split(“,”) # Output: [‘apple’, ‘banana’, ‘orange’]
```

6. **Joining :** Combining elements of a list into a single string with a specified separator.

```
string_list = [‘apple’, ‘banana’, ‘orange’]
new_string = “, “.join(string_list) # Output: “apple, banana, orange”
```

7. **Finding Substrings:** Locating the index of a substring within a string.

```
string = “Hello World”
index = string.find(“World”) # Output: 6
```

Aim:

To write a Python program that performs string manipulation functions on the given string.

Algorithm:

1. Define a Python function named `` `title_case` `` that takes a sentence as input.
2. Split the input sentence into a list of words using the `` `split()` `` method.
3. Iterate through each word in the list using list comprehension.
4. If the word is a common word `(e.g., “and,” “the,” “in”)`, lowercase it unless it appears at the beginning of the sentence.
5. Otherwise, capitalize the first letter of the word and lowercase the rest.
6. Join the modified list of words back into a string using the `` `join()` `` method.
7. Return the resulting title-cased sentence.

Source code:

```
def title_case(sentence):
    common_words = ["and", "the", "in"]
    words = sentence.split()
    title_cased_words = [word.capitalize() if i == 0 or word.lower() not
    in common_words else word.lower() for i, word in enumerate(words)]
    return ' '.join(title_cased_words)

input_sentence = input("Enter your sentence: ")
output_sentence = title_case(input_sentence)
print("Original Sentence:", input_sentence)
print("Title Cased Sentence:", output_sentence)
```

Sample Input and Output:

```
Enter your sentence: “the quick brown fox jumps over the lazy dog”
Original Sentence: the quick brown fox jumps over the lazy dog
Title Cased Sentence: The Quick Brown Fox Jumps Over the Lazy Dog
```

Results:

Thus, python program to implement different string manipulation techniques have been written successfully and tested with various samples.

Ex No: 07 EXTRACTING TITLES USING LAMDA FUNCTION

Date:

Theoretical Concepts:

Sorting:

The program uses the `sorted()` function to sort the list of dictionaries based on the 'year' key. Sorting is a fundamental operation in programming, and in this case, it allows us to organize the books by their publication year**.**

Syntax:

```
sorted(iterable, key=None, reverse=False)
```

Example:

```
# Sort a list in ascending order
sorted_list = sorted([3, 1, 4, 1, 5, 9])
print(sorted_list) # Output: [1, 1, 3, 4, 5, 9]
```

Lambda function:

A lambda function is an anonymous function that can be defined in a single line of code. In this program, a lambda function is used to extract the 'title' key from each dictionary in the sorted list. This demonstrates the flexibility and conciseness of lambda functions for simple operations.

Syntax:

```
lambda arguments: expression
```

Example:

```
# A lambda function that adds two numbers
addition = lambda x, y: x + y
print(addition(3, 5)) # Output: 8
# A lambda function that squares a number
square = lambda x: x ** 2
print(square(4)) # Output: 16
```

Mapping:

The `map()` function applies a function (in this case, the lambda function) to each element of an iterable (in this case, the sorted list of dictionaries). It creates a new list containing the results of applying the function to each element. Here, it creates a new list containing only the titles of the books extracted using the lambda function.

Syntax:

```
map(function, iterable)
```

Example:

```
# Use map() with a lambda function to square each number in the list
squared_numbers = map(lambda x: x ** 2, numbers)
# Convert the map object to a list
squared_numbers_list = list(squared_numbers)
print(squared_numbers_list) # Output: [1, 4, 9, 16, 25]
```

List comprehension:

The program also demonstrates list comprehension, a concise way to create lists in Python. The list comprehension is used to extract the 'title' key from each dictionary in the sorted list.

Example:

```
# Create a list of squares of numbers from 0 to 4 using list comprehension
squares = [x ** 2 for x in range(5)]
print(squares) # Output: [0, 1, 4, 9, 16]
```

Aim:

To write a Python program that sorts a list of books by year and extracts the books accordingly.

Algorithm:

1. Start
2. Define a list of dictionaries representing books, where each dictionary has '`title`', '`author`', and '`year`' keys.
3. Use the `sorted()` function to sort the list of dictionaries based on the '`year`' key in ascending order.
4. Use a lambda function with the `map()` function to create a new list containing only the titles of the books.
5. Return the sorted list of dictionaries and the list of book titles.
6. Stop.

Source code:

```
books = [
    {'title': 'Book1', 'author': 'Author1', 'year': 2005},
    {'title': 'Book2', 'author': 'Author2', 'year': 1998},
    {'title': 'Book3', 'author': 'Author3', 'year': 2010},
    {'title': 'Book4', 'author': 'Author4', 'year': 2000}
]
```

```
    sorted_books = sorted(books, key=lambda x: x['year'])
    titles = list(map(lambda x: x['title'], sorted_books))

    print("Sorted Books:")
    for book in sorted_books:
        print(book)

    print("\nTitles of the Books:")
    for title in titles:
        print(title)
```

Sample Input and Output:

```
Sorted Books:
{'title': 'Book2', 'author': 'Author2', 'year': 1998}
{'title': 'Book4', 'author': 'Author4', 'year': 2000}
{'title': 'Book1', 'author': 'Author1', 'year': 2005}
{'title': 'Book3', 'author': 'Author3', 'year': 2010}

Titles of the Books:
Book2
Book4
Book1
Book3
```

Result:

Thus, the python program for sorting the books by using publishing year and also print only the sorted list of books was executed and verified successfully.

Ex No: 08 STUDENT GRADE CALCULATOR WITH CLASS AND OBJECTS

Date:

Theoretical Concepts:

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. The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
  Eg.: My class.Myattribute

Syntax:

```
class ClassName:
    # Statement
```

Example:

```
class Dog:
    sound = "bark"
```

Object

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.

An object consists of:

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

Syntax:

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

Example

```
# A class
class Dog:

    # A simple class
    # attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)


# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr1)
Rodger.fun()
```

Self-Parameter

`self represents` the instance of the class that is currently being used. It is customary to use self as the first parameter in instance methods of a class. Whenever you call a method of an object created from a class, the object is automatically passed as the first argument using the `self parameter`. This enables you to modify the object's properties and execute tasks unique to that particular instance.

Example

```
Refer the previous example code.
```

Constructor method

The `__init__` method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements (i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Syntax

```
class ClassName:
    def __init__(self, arguments):
    #statements
```

Aim:

To create a Python program for managing student data, calculating their average marks, determining grades using classes and objects.

Algorithm:

1. Define a class named `Student`.
2. Initialize the class with attributes `name`, `roll_number`, and `marks`.
3. Define a method `calculate_average()` to calculate the average marks of the student.
4. Define a method `get_grade()` to determine the grade based on the average marks calculated.
5. Define a method `display_info()` to display the student's name, roll number, and average marks.
6. Create a dictionary `student_marks` containing subject names as keys and marks as values.
7. Create an instance `student1` of the `Student` class with name "Ragu", roll number "S001", and the dictionary of marks `student_marks`.
8. Display the student's information using the `display_info()` method.
9. Print the grade obtained by the student using the `get_grade()` method.

Source code:

```
class Student:
    def __init__(self, name, roll_number, marks):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks

    def calculate_average(self):
        total_marks = sum(self.marks.values())
        return total_marks / len(self.marks)

    def get_grade(self):
        average = self.calculate_average()
        if average >= 90:
            return 'A'
        elif 80 <= average < 90:
            return 'B'
        elif 70 <= average < 80:
            return 'C'
        elif 60 <= average < 70:
            return 'D'
        else:
            return 'F'

    def display_info(self):
        print("Student Name:", self.name)
        print("Roll Number:", self.roll_number)
        print("Average Marks:", self.calculate_average())


# Example usage:
student_marks = {'Math': 85, 'Science': 90, 'History': 75}
student1 = Student("Ragu", "S001", student_marks)
student1.display_info()
print("Grade:", student1.get_grade())
```

Sample Input and Output:

```
Student Name: Ragu
Roll Number: S001
Average Marks: 83.33333333333333
Grade: B
```

Result:

Thus, the Python program for managing student data, calculating their average marks, determining grades using classes and objects has been executed successfully.

Ex No: 09 LIBRARY MANAGEMENT WITH CLASS AND OBJECTS

Date:

Theoretical Concepts:

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. The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
  Eg.: My class.Myattribute

Syntax:

```
class ClassName:
    # Statement
```

Example:

```
class Dog:
    sound = "bark"
```

Object

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.

An object consists of:

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

Syntax:

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

Example

```
# A class
class Dog:

    # A simple class
    # attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)


# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr1)
Rodger.fun()
```

Self Parameter

`self represents` the instance of the class that is currently being used. It is customary to use self as the first parameter in instance methods of a class. Whenever you call a method of an object created from a class, the object is automatically passed as the first argument using the `self parameter`.This enables you to modify the object's properties and execute tasks unique to that particular instance.

Example

```
Refer the previous example code.
```

Constructor method

The `__init__` method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements (i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Syntax

```
class ClassName:
    def __init__(self, arguments):
    #statements
```

Aim:

To create a Python Program using class and objects where books can be added to a library, members can borrow and return books, and the status of books and members can be displayed.

Algorithm:

1. Define a class `Book` with attributes `title`, `author`, `isbn`, and `availability`.
2. Define a class `Library` with a list to store books and methods to add books and display book information.
3. Define a class `Member` with attributes `member_id`, `name`, and a `list` to store borrowed books. It also includes methods to borrow and return books.
4. Define a class `LibrarySystem` which contains a library and a list of members. It includes methods to register members and display member information.
5. Create an instance of `LibrarySystem`.
6. Create instances of Book and add them to the library system's library.
7. Display the books in the library.
8. Create an instance of Member and register them with the library system.
9. Borrow a book for the member and display member information.

Source code:

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.availability = True

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def display_books(self):
        for book in self.books:
            print("Title:", book.title)
            print("Author:", book.author)
            print("ISBN:", book.isbn)
            print("Availability:", "Available" if book.availability else
            "Not Available")
            print()

class Member:
    def __init__(self, member_id, name):
        self.member_id = member_id
        self.name = name
        self.borrowed_books = []

    def borrow_book(self, book):
        if book.availability:
            self.borrowed_books.append(book)
            book.availability = False
            print("Book", book.title, "borrowed successfully.")
        else:
            print("Book", book.title, "is not available for borrowing.")

    def return_book(self, book):
        if book in self.borrowed_books:
            self.borrowed_books.remove(book)
            book.availability = True
            print("Book", book.title, "returned successfully.")
        else:
            print("Book", book.title, "was not borrowed by this member.")

class LibrarySystem:
    def __init__(self):
        self.library = Library()
        self.members = []

    def register_member(self, member):
        self.members.append(member)

    def display_members(self):
        for member in self.members:
            print("Member ID:", member.member_id)
            print("Name:", member.name)
            print("Borrowed Books:", [book.title for book in
            member.borrowed_books])
            print()


library_system = LibrarySystem()
book1 = Book("Python Programming"," Guido van Rossum","978-0134444321")
book2 = Book("Internet of Things","Kalaiselvi Geetha"," 978-3-319-53470-1")
library_system.library.add_book(book1)
library_system.library.add_book(book2)
library_system.library.display_books()
member1 = Member("CS01", "Madhan")
library_system.register_member(member1)
member1.borrow_book(book1)
library_system.display_members()
```

Sample Input and Output:

```
Title: Python Programming
Author:  Guido van Rossum
ISBN: 978-0134444321
Availability: Available

Title: Internet of Things
Author: Kalaiselvi Geetha
ISBN:  978-3-319-53470-1
Availability: Available

Book Python Programming borrowed successfully.
Member ID: CS01
Name: Madhan
Borrowed Books: ['Python Programming']
```

Result:

Thus, the Python Program where books can be added to a library, members can borrow and return books, and the status of books and members can be displayed using class and objects has been executed successfully.

Ex No: 10 SIMPLE BANK ACCOUNT USING OPERATOR OVERLOADING

Date:

Theoretical Concepts:

Operator Overloading:

Operator overloading in Python allows the same operator to have different meanings depending on the context of its usage. It is achieved by defining special methods in a class that correspond to the operator being overloaded.

Unary Operator Overloading:

Unary operators work with a single operand. They are defined using special methods in a class with names like neg for unary minus, pos for unary plus, abs for absolute value, invert for bitwise negation, etc.

Unary operators include:

- Unary plus `(+)`
- Unary minus `(-)`
- Logical NOT `(!)`
- Bitwise NOT `(~)`

For example:

```
class MyClass:
    def neg(self):
        #Define behavior for unary minus pass
```

Binary Operator Overloading:

Binary operators work with two operands. They are defined using special methods in a class with names like add for addition, sub for subtraction, mul for multiplication, truediv for division, mod for modulus, etc.

Binay operators include:

Arithmetic operators:

`+, -, *, /, **, %, //`

Comparison operators:

`==, !=, <, >, <=, >=`

Bitwise operators:

`&, |, ^, <<, >>`

For example:

```
class MyClass:
    def add(self, other):
        # Define behavior for addition pass
```

Aim:

To Create a Python program to simulate a basic bank account system with deposit, withdrawal, and balance inquiry functionalities, along with operator overloading for account operations.

Algorithm:

1. Define a class `BankAccount` with attributes `account_number`, `account_holder`, and `balance`.
2. Include methods to `deposit`, `withdraw`, `get_balance`, `display_account_info`, and `overload operators` for `addition`, `subtraction`, and `equality`.
3. Define the `__init__` method to initialize the account with the account number, account holder's name, and initial balance.
4. Implement the deposit method to increase the balance by the deposited amount.
5. Implement the withdraw method to decrease the balance if sufficient funds are available.
6. Implement the `get_balance` method to return the current balance.
7. Implement the `display_account_info` method to print the account information.
8. Overload the addition operator `__add__` to combine balances of two accounts into a new account.
9. Overload the subtraction operator `__sub__` to find the difference between balances of two accounts.
10. Overload the equality operator `__eq__` to compare account numbers.
11. Create instances of `BankAccount` with different account details.
12. Test equality between two accounts.
13. Perform addition and subtraction operations between accounts and display the resulting account information.

Source code:

```
class BankAccount:
    def __init__(self, account_number, account_holder, balance):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print("Withdrawal successful. Current balance:", self.balance)
        else:
            print("Insufficient funds.")

    def get_balance(self):
        return self.balance
    def display_account_info(self):
        print("Account Number:", self.account_number)
        print("Account Holder:", self.account_holder)
        print("Balance:", self.balance)

    def __add__(self, other):
        new_balance = self.balance + other.balance
        return BankAccount("Combined Account", "Joint Account", new_balance)

    def __sub__(self, other):
        new_balance = self.balance - other.balance
        return BankAccount("Difference Account", "Difference Holder",
        new_balance)

    def __eq__(self, other):
        return self.account_number == other.account_number

account1 = BankAccount("A001", "Kanthi", 1000)
account2 = BankAccount("A002", "Madhan", 500)
account3 = BankAccount("A001", "Pathy", 1500)
print(account1 == account2)
print(account1 == account3)
combined_account = account1 + account2
combined_account.display_account_info()
difference_account = account1 - account2
difference_account.display_account_info()
```

Sample Input and Output:

```
False
True
Account Number: Combined Account
Account Holder: Joint Account
Balance: 1500
Account Number: Difference Account
Account Holder: Difference Holder
Balance: 500
```

Result:

Thus, the Python program to simulate a basic bank account system with deposit, withdrawal, and balance inquiry functionalities, along with operator overloading for account operations has been executed successfully.

Ex No: 11 INHERITANCE VEHICLE MANAGEMENT SYSTEM

Date:

Theoretical Concepts:

Inheritance:

Inheritance could be a feature of object-oriented programming that permits one class to acquire characteristics from another class. In other words, inheritance permits a class to be characterized in terms of another class, which makes it simpler to make and keep up an application.

Types of Inheritance in Python Single inheritance: A derived class inherits from one base class. Multiple inheritance: A derived class inherits from multiple base classes. Multilevel inheritance: A derived class inherits from a base class that inherits from another base class. Hierarchical inheritance: Multiple derived classes inherit from the same base class. Hybrid inheritance: A combination of two or more of the above inheritance types.

Inheritance Syntax:

Inheritance allows a class (subclass/derived class) to inherit attributes and methods from another class (superclass/base class).

Syntax for inheritance:

```
class DerivedClassName(BaseClassName):
    # class body
```

Example:

```
class Animal:
    def speak(self):
            print("Animal speaks")

class Dog(Animal):
    def bark(self):
            print("Dog barks")
```

Example usage

```
dog = Dog()
dog.speak()
# Output: Animal speaks
dog.bark()
# Output: Dog barks
```

Polymorphism:

Polymorphism in Python refers to the ability of different objects to respond to the same method or attribute call in different ways. It allows objects of different classes to be treated as objects of a common superclass. This concept is fundamental to object-oriented programming and enables flexibility and extensibility in code.

Compile-Time Polymorphism:

Achieved through method overloading and operator overloading. Resolves at compile time based on the number and types of arguments.

Run-Time Polymorphism:

Achieved through method overriding. Resolves at runtime based on the actual type of the object.

Polymorphism Syntax:

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It allows methods to be overridden in a subclass with the same name but different implementation.

Syntax for polymorphism:

```
class BaseClassName:
    def method_name(self):
        # base class method implementation

class DerivedClassName(BaseClassName):
    def method_name(self):
        # derived class method implementation
```

Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")
```

Example usage

```
animal = Animal()
dog = Dog()
```

```
animal.speak()
# Output: Animal speaks
dog.speak()
# Output: Dog barks
```

Aim:

To create a Python program that models a transportation company's vehicle management system with a hierarchy of classes representing various vehicle types, allowing flexible management and polymorphic display of vehicle information.

Algorithm:

1. Define a class `Vehicle` with attributes `make`, `model`, `year`, and `fuel_type`, and a method `display_info`.
2. Define a class `Car` inheriting from `Vehicle`, with additional attributes `num_doors`, `num_passengers`, and `car_type`, and override the `display_info` method to include car-specific information.
3. Define a class `Truck` inheriting from `Vehicle`, with additional attributes `payload_capacity` and `four_wheel_drive`, and override the `display_info` method to include truck-specific information.
4. Define a class `ElectricCar` inheriting from `Car`, with additional attributes `battery_capacity` and `charging_time`, and override the `display_info` method to include electric car-specific information.
5. Define a class `Motorcycle` inheriting from `Vehicle`, with additional attributes `num_wheels`, `has_sidecar`, and `motorcycle_type`, and override the `display_info` method to include motorcycle-specific information.
6. Define a function `display_vehicle_info(vehicles)` to display information for a list of vehicles, utilizing their `display_info` methods.
7. Create instances of various vehicle types (Car, Truck, ElectricCar, Motorcycle).
8. Store these instances in a list.
9. Call `display_vehicle_info` function with the list of vehicles to print their information.

Source code:

```
class Vehicle:
    def __init__(self, make, model, year, fuel_type):
        self.make = make
        self.model = model
        self.year = year
        self.fuel_type = fuel_type

    def display_info(self):
        pass
```

```
class Car(Vehicle):
     def __init__(self, make, model, year, fuel_type, num_doors,
          num_passengers, car_type):
        super().__init__(make, model, year, fuel_type)
        self.num_doors = num_doors
        self.num_passengers = num_passengers
        self.car_type = car_type

    def display_info(self):
         return f"Car: {self.make} {self.model} ({self.year}), Fuel:
          {self.fuel_type}, Doors: {self.num_doors}, Passengers:
          {self.num_passengers}, Type: {self.car_type}"


class Truck(Vehicle):
    def __init__(self, make, model, year, fuel_type, payload_capacity,
          four_wheel_drive):
        super().__init__(make, model, year, fuel_type)
        self.payload_capacity = payload_capacity
        self.four_wheel_drive = four_wheel_drive

    def display_info(self):
        return f"Truck: {self.make} {self.model} ({self.year}), Fuel:
          {self.fuel_type}, Payload Capacity: {self.payload_capacity}, 4WD:
          {self.four_wheel_drive}"


class ElectricCar(Car):
     def __init__(self, make, model, year, num_doors, num_passengers,
                   car_type, battery_capacity, charging_time):
        super().__init__(make, model, year, "Electric", num_doors,
num_passengers, car_type)
        self.battery_capacity = battery_capacity
        self.charging_time = charging_time

    def display_info(self):
        return f"Electric Car: {self.make} {self.model} ({self.year}),
          Battery Capacity: {self.battery_capacity}, Charging Time:
          {self.charging_time}, {super().display_info()}"


class Motorcycle(Vehicle):
    def __init__(self, make, model, year, fuel_type, num_wheels,
        has_sidecar, motorcycle_type):
        super().__init__(make, model, year, fuel_type)
        self.num_wheels = num_wheels
        self.has_sidecar = has_sidecar
        self.motorcycle_type = motorcycle_type
    def display_info(self):
         return f"Motorcycle: {self.make} {self.model} ({self.year}), Fuel:
          {self.fuel_type}, Wheels: {self.num_wheels}, Sidecar:
          {self.has_sidecar}, Type: {self.motorcycle_type}"


def display_vehicle_info(vehicles):
    for vehicle in vehicles:
        print(vehicle.display_info())


car1 = Car("Toyota", "Camry", 2022, "Gasoline", 4, 5, "Sedan")
truck1 = Truck("Ford", "F-150", 2022, "Gasoline", 1500, True)
electric_car1 = ElectricCar("Tesla", "Model S", 2022, 4, 5, "Sedan", 100, 8)
motorcycle1 = Motorcycle("Harley-Davidson", "Sportster", 2022, "Gasoline",
              2, False, "Cruiser")

vehicles_list = [car1, truck1, electric_car1, motorcycle1]

display_vehicle_info(vehicles_list)
```

Sample Input and Output:

```
Car: Toyota Camry (2022), Fuel: Gasoline, Doors: 4, Passengers: 5, Type: Sedan
Truck: Ford F-150 (2022), Fuel: Gasoline, Payload Capacity: 1500, 4WD: True
Electric Car: Tesla Model S (2022), Battery Capacity: 100, Charging Time: 8, Car: Tesla Model S
        (2022), Fuel: Electric, Doors: 4, Passengers: 5, Type: Sedan
Motorcycle: Harley-Davidson Sportster (2022), Fuel: Gasoline, Wheels: 2, Sidecar: False, Type: Cruiser
```

Result:

Thus, the Python program that models a transportation company's vehicle management system with a hierarchy of classes representing various vehicle types, allowing flexible management and polymorphic display of vehicle information has been executed successfully.

Ex No: 12 **FILE HANDLING ON LOG FILES**

Date:

Theoretical Concepts:

1. Log File Structure:

   - Log files typically contain entries with timestamps, severity levels, and messages.
   - Entries might follow a specific format, such as "`YYYY-MM-DD HH:MM:SS - SEVERITY: MESSAGE`".

2. File Handling:

   - File handling in Python involves opening, reading, and closing files.
   - The `` `open()` `` function is used to open a file, and the `` `readlines()` `` method is used to read lines from the file.

3. Regular Expressions:

   - Regular expressions `(regex)` are patterns used for matching character combinations in strings.
   - The `` `re.match()` `` function is employed to extract information from log entries using a regex pattern.

4. Datetime Module:

   - The `` `datetime` `` module in Python is used to work with dates and times.
   - `` `datetime.strptime()` `` is used to convert a string representation of a timestamp to a datetime object.
   - Time intervals between entries are calculated using `` `datetime.timedelta` ``.

Aim:

To read and analyze a log file, extracting timestamped entries to determine the total number of entries, count occurrences of each severity level, and calculate the average time gap between consecutive log entries, facilitating effective log data assessment and system monitoring using python.

Algorithm:

1. Reading the log file:

   - Opens a log file and reads its contents line by line into a list.

2. Extracting information from log entries:
   - Defines a pattern to extract timestamp, severity, and message from each log entry using regular expressions.
   - Parses each log entry to extract this information and converts the timestamp into a datetime object.

3. Analyzing the log entries:
   - Counts the total number of log entries.
   - Counts the occurrences of different severity levels.
   - Calculates the time gap between consecutive log entries and computes the average time gap.

4. Main Functionality:
   - Runs the main code if the script is executed directly.
   - Calls functions to read the log file, analyze its contents, and print the results.

Source code:

```
import re
from datetime import datetime, timedelta

def read_log_file(file_path):
    with open(file_path, 'r') as file:
        log_entries = file.readlines()
    return log_entries

def extract_information(log_entry):
    # Define a regular expression pattern to extract timestamp, severity,
    and message
    pattern = r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) - (\w+): (.*)'
    match = re.match(pattern, log_entry)

    if match:
        timestamp_str, severity, message = match.groups()
        timestamp = datetime.strptime(timestamp_str, '%Y-%m-%d %H:%M:%S')
        return timestamp, severity, message
    else:
        return None

def analyze_log(log_entries):
    total_entries = len(log_entries)
    severity_counts = {}
    time_gaps = []
    for i in range(1, total_entries):
        current_entry = extract_information(log_entries[i])
        previous_entry = extract_information(log_entries[i - 1])

        if current_entry and previous_entry:
            time_gap = current_entry[0] - previous_entry[0]
            time_gaps.append(time_gap.total_seconds())

            # Count severity levels
            severity_counts[current_entry[1]] =
            severity_counts.get(current_entry[1], 0) + 1

    average_time_gap = sum(time_gaps) / len(time_gaps) if time_gaps else 0

    return total_entries, severity_counts, average_time_gap

if __name__ == "__main__":
    log_file_path = "log_file.log"

    log_entries = read_log_file(log_file_path)

    total_entries, severity_counts, average_time_gap =
    analyze_log(log_entries)

    print(f"Total Entries: {total_entries}")
    print("Severity Counts:")
    for severity, count in severity_counts.items():
        print(f"  {severity}: {count}")
    print(f"Average Time Gap between Entries: {average_time_gap} seconds")
```

Source Code (Log File):

Note: Save a Log File as log_file.log

```
2024-02-27 10:00:00 - INFO: Application started
2024-02-27 10:05:30 - ERROR: Critical error occurred - Server crashed
2024-02-27 10:10:45 - WARNING: Resource usage high
2024-02-27 10:15:20 - INFO: User logged in
2024-02-27 10:20:05 - DEBUG: Debugging message - Step 1
```

Note: Create and save LOG file and change the directory name to the LOG file directory which you created in the python source code and execute.

Sample Input and Output:

```
Total Entries: 5
Severity Counts:
  ERROR: 1
  WARNING: 1
  INFO: 1
  DEBUG: 1
Average Time Gap between Entries: 301.25 seconds
```

Result:

Thus, the log analysis program executed successfully, determining the total entries, average time gap, and severity level counts. It provided concise insights into the log data.

Ex No: 13 **FILE HANDLING ON CSV FILES**

Date:

Theoretical Concepts:

1. CSV file:

A CSV (Comma-Separated Values) file is a plain text file that stores tabular data. In Python, the csv module simplifies the process of working with such files. It allows you to read data from and write data to CSV files.

2. File Handling:

- ➢ File handling in Python involves opening, reading, and closing files.
- ➢ The `` `open()` `` function is used to open a file, and the `` `readlines()` `` method is used to read lines from the file.

3. Functions :

Functions help modularize the code for better organization and readability. For example, the `read_csv` function handles reading data from the CSV file, making the code more maintainable.

Aim:

To Design a Python program to efficiently handle and analyze employee data stored in a CSV file, reading the file, finding the highest-paid employee, sorting employees by department, and calculating the average salary for each department.

Algorithm:

1. Import Libraries:
   - Import `` `csv` `` and `` `operator` `` for CSV handling and sorting.
2. Read CSV:
   - Use `` `csv.reader` `` to read the employee data from the CSV file.
3. Highest-Paid Employee:
   - Track the highest-paid employee while iterating through the data.
4. Sort Employees by Department:
   - Utilize the `` `sorted` `` function to sort employees based on department.
5. Average Salary per Department:
   - Calculate the average salary for each department using a dictionary.
6. Display Results:
   - Print the details of the highest-paid employee, the sorted employee list, and the average salary for each department.
7. Exception Handling:
   - Implement basic error handling for file reading or data processing issues.

8. Close File:
    - Ensure proper closure of the CSV file.

Source code:

```
import csv
from collections import defaultdict

def read_csv_file(file_path):
    employees = []
    with open(file_path, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            employees.append(row)
    return employees

def find_highest_paid_employee(employees):
    highest_paid_employee = max(employees, key=lambda x: float(x['salary']))
    return highest_paid_employee

def sort_employees_by_department(employees):
    sorted_employees = sorted(employees, key=lambda x: x['department'])
    return sorted_employees

def calculate_average_salary_by_department(employees):
    department_salaries = defaultdict(list)

    for employee in employees:
        department_salaries[employee['department']].append(float(employee['sa
        lary']))
    average_salaries = {department: sum(salaries) / len(salaries) for
    department, salaries in department_salaries.items()}
    return average_salaries

def main():
    file_path = 'emp.csv'
    employees = read_csv_file(file_path)

    highest_paid_employee = find_highest_paid_employee(employees)
    print(f"Highest Paid Employee: {highest_paid_employee['name']} (ID:
    {highest_paid_employee['employee_id']}, Salary:
    {highest_paid_employee['salary']})")
    sorted_employees = sort_employees_by_department(employees)
    print("\nEmployees Sorted by Department:")
    for employee in sorted_employees:
        print(f"{employee['name']} (ID: {employee['employee_id']},
        Department: {employee['department']}, Salary: {employee['salary']})")

    average_salaries = calculate_average_salary_by_department(employees)
    print("\nAverage Salary by Department:")
    for department, avg_salary in average_salaries.items():
        print(f"{department}: {avg_salary:.2f}")

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

Source Code (CSV File):

Note: Save a CSV File as ***`employee_data.csv`***

```
employee_id,name,department,salary
1,Sriram ,HR,50000
2,Vasanth,IT,60000
3,Praneeth,HR,55000
4,Suresh,IT,65000
5,Ramesh,Finance,70000
```

Note: Create and save CSV file and change the directory name to the CSV file directory which you created in the python source code and execute.

Sample Input and Output:

```
Highest Paid Employee: Ramesh (ID: 5, Salary: 70000)

Employees Sorted by Department:
Ramesh (ID: 5, Department: Finance, Salary: 70000)
Sriram (ID: 1, Department: HR, Salary: 50000)
Praneeth (ID: 3, Department: HR, Salary: 55000)
Vasanth (ID: 2, Department: IT, Salary: 60000)
Suresh (ID: 4, Department: IT, Salary: 65000)

Average Salary by Department:
HR: 52500.00
IT: 62500.00
Finance: 70000.00
```

Result:

Thus, the File handling on csv file is executed successfully, calculating the highest-paid employee, sorting employees by department, and calculating the average salary for each department.

Ex No: 14 CALCULATOR USING EXCEPTION HANDLING

Date:

Theoretical Concepts:

Exception Handling:

1. Definition:

- Exception handling is a programming paradigm that addresses errors during program execution.

2. Try-Catch Block:

Try:

- Contains code that might raise an exception.

Catch:

- Handles exceptions and specifies actions for error scenarios.

3. Types of Exceptions:

Checked Exceptions:

- Compiler-enforced handling (e.g., file IO errors).

Unchecked Exceptions:

- Runtime errors (e.g., division by zero).

4. Throw and Throws:

Throw :

Explicitly throws an exception.

Throws :

Declares potential exceptions in a method signature.

5. Finally Block :

Executes code regardless of exceptions.

Useful for cleanup tasks (e.g., closing files).

Aim:

To Develop a user-friendly calculator program with exception handling to ensure error-free input for basic arithmetic operations, including informative error messages for potential issues.

Algorithm:

1. The calculate function:
   - It takes three parameters: two numbers (num1 and num2) and an operation (operation).
   - It performs the specified operation (+, -, *, /) on the numbers and returns the result.
   - It includes error handling for division by zero, invalid operations, and invalid input types.

2. The get_user_input function:
   - It prompts the user to enter two numbers and an operation.
   - It converts the user input into floating-point numbers.
   - It includes error handling for invalid input types.

3. The main part of the script:
   - It runs a loop to continuously prompt the user for input and perform calculations.
   - It calls the get_user_input function to get user input.
   - It calls the calculate function to perform the calculation and prints the result.
   - It asks the user if they want to continue, and if not, it breaks the loop.

Source code:

```
def calculate(num1, num2, operation):
    try:
        if operation == '+':
            result = num1 + num2
        elif operation == '-':
            result = num1 - num2
        elif operation == '*':
            result = num1 * num2
        elif operation == '/':
            if num2 == 0:
                raise ZeroDivisionError("Cannot divide by zero")
            result = num1 / num2
        else:
            raise ValueError("Invalid operation. Please use '+', '-', '*',
            or '/'.")

        return result
    except (ValueError, TypeError) as e:
        print(f"Error: {e}")
    except ZeroDivisionError as e:
        print(f"Error: {e}")

def get_user_input():
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        operation = input("Enter the operation (+, -, *, /): ")

        return num1, num2, operation
    except ValueError:
        print("Error: Invalid input for numbers.")
        return None, None, None

if __name__ == "__main__":
    while True:
        num1, num2, operation = get_user_input()

        if num1 is not None and num2 is not None and operation is not None:
            result = calculate(num1, num2, operation)
            if result is not None:
                print(f"Result: {result}")

        user_input = input("Do you want to continue? (y/n): ").lower()
        if user_input != 'y':
            break
```

Sample Input and Output:

```
Enter the first number: 6
Enter the second number: 9
Enter the operation (+, -, *, /): +
Result: 15.0
Do you want to continue? (y/n): n
```

Result:

Thus, the arithmetic calculator program executed successfully. Users can perform basic arithmetic operations on two numbers. The program handles invalid input types gracefully, ensuring a smooth user experience.

Ex No: 15 NUMERICAL DATA PROCESSING USING PANDAS

Date:

Theoretical Concepts:

Exception Handling:

1. Definition:

- Exception handling is a programming paradigm that addresses errors during program execution.

2. Try-Catch Block:

Try:

- Contains code that might raise an exception.

Catch:

- Handles exceptions and specifies actions for error scenarios.

3. Types of Exceptions:

Checked Exceptions:

- Compiler-enforced handling (e.g., file IO errors).

Unchecked Exceptions:

- Runtime errors (e.g., division by zero).

4. Throw and Throws:

Throw :

Explicitly throws an exception.

Throws :

Declares potential exceptions in a method signature.

5. Finally Block :

Executes code regardless of exceptions.
Useful for cleanup tasks (e.g., closing files).

Aim:

To create a Python program that reads numerical data from a file, performs calculations, and handles potential errors gracefully.

Algorithm:

1. Define a function to read numerical data from a file.
2. Open the specified file, iterating through each line.
3. Attempt to convert each line to a float, appending valid values to a list.
4. Handle potential errors, such as a missing file or invalid data formats.
5. Perform numerical calculations on the collected data.

6. Handle potential errors during calculations, like division by zero.
7. Print the total and average if calculations are successful.
8. Provide clear feedback to the user throughout the process.

Source code:

```
def read_data_from_file(file_path):
    data = []
    try:
        with open(file_path, 'r') as file:
            for line in file:
                try:
                    data.append(float(line.strip()))
                except ValueError:
                    print(f"Ignoring non-numeric data: {line.strip()}")
    except FileNotFoundError:
        print(f"File '{file_path}' not found.")
    except Exception as e:
        print(f"An error occurred while reading the file: {e}")
    return data

def perform_numerical_calculations(data):
    try:
        if not data:
            raise ValueError("No numerical data found.")
        total = sum(data)
        average = total / len(data)
        return total, average
    except ZeroDivisionError:
        print("Cannot calculate average: Division by zero.")
    except Exception as e:
        print(f" An error occurred during numerical calculations: {e}")

if __name__ == "__main__":
    file_path = 'data.txt'
    data = read_data_from_file(file_path)
    total, average = perform_numerical_calculations(data)
    if total is not None and average is not None:
        print(f"Total: {total}")
        print(f"Average: {average}")
```

Source Code (TXT File):

Note: Save a TXT File as `data.txt`

```
10
20
30
40
50
abc
60
70
80
```

Note: Create and save TXT file and change the directory name to the TXT file directory which you created in the python source code and execute.

Sample Input and Output:

```
Ignoring non-numeric data: abc
Total: 360.0
Average: 45.0
```

Result:

Thus, the Python program for numerical data analysis and error handling has been executed successfully.

Ex No: 16 E-COMMERCE SALES ANALYSIS WITH MATPLOTLIB

Date:

Theoretical Concepts:

NumPy:

Concept:

NumPy, short for Numerical Python, is a fundamental package for scientific computing in Python. It provides support for arrays, matrices, and a collection of mathematical functions to operate on these arrays efficiently.

Key Features:

- Arrays for efficient numerical computing.
- Mathematical operations and functions.
- Broadcasting for implicit looping.
- Linear algebra support.

Pandas:

Concept:

Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like Series and DataFrame, which are built on top of NumPy arrays, along with methods for data cleaning, manipulation, and analysis.

Key Features:

- DataFrame for structured data manipulation.
- Handling missing data and alignment.
- Versatile data manipulation methods.
- Time series support.

Matplotlib:

Concept:

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It provides a MATLAB-like interface for creating plots and charts to visualize data in various formats.

Key Features:

- Flexible plotting functions.
- Customization options for plots.
- Exporting plots to various formats.

- Easy integration with other libraries.

Correlation Coefficient:

It's a measure that describes the strength and direction of a relationship between two variables. It ranges from -1 to 1, where 1 indicates a perfect positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation.

Groupby in Pandas:

It's a powerful function that allows you to split the data into groups based on some criteria, apply a function to each group independently, and then combine the results back into a `DataFrame`. In this exercise, we used `groupby` to group the data by `'Customer_ID'` and `'Product_Name'`, and then applied aggregation functions like `'sum'` and `'mean'`.

Visualization:

Matplotlib is a popular plotting library in Python. We used it to create various types of plots like bar charts, line charts, and scatter plots to visualize the data and gain insights.

The concept and theory for each part of the Program:

a) Data Loading and Exploration:

Concept: Descriptive statistics provide insights into the basic characteristics of a dataset, such as measures of central tendency, variability, and distribution.

Syntax:

```
#Loading dataset into Pandas DataFrame
data = pd.read_csv('sales_data.csv')

#Descriptive statistics with Pandas
data.describe()

#Descriptive statistics with NumPy
np.describe(data)

#Displaying first few rows
data.head()
```

b) Data Cleaning and Manipulation:

Concept: Handling missing values and converting data types are essential steps in preparing the dataset for analysis.

Syntax:
```
#Checking for missing values
data.isnull().sum()

#Converting 'Date' column to datetime object
data['Date'] = pd.to_datetime(data['Date'])

#Creating new column 'Total_Price'
data['Total_Price'] = data['Quantity'] * data['Price_per_Unit']
```

c) Data Visualization:

Concept: Visualization helps in understanding patterns and relationships within the data.

Syntax:
```
#Bar chart showing total sales for each product
product_sales.plot(kind='bar')

#Line chart for sales trend over time
sales_over_time.plot(kind='line')

#Scatter plot to explore relationship between variables
plt.scatter(data['Quantity'], data['Total_Price'])
```

d) Advanced Analysis:

Concept: Advanced analysis involves calculating statistical measures and performing group-wise operations.

Syntax:
```
#Calculating correlation coefficient
np.corrcoef(data['Quantity'], data['Total_Price'])

#Grouping data and finding average spending per customer
data.groupby('Customer_ID')['Total_Price'].mean()

#Identifying top products by total sales
data.groupby('Product_Name')['Total_Price'].sum().nlargest(5)
```

Aim:

To create a Python program for analyzing sales transactions dataset, including data loading, exploration, cleaning, manipulation, visualization, and advanced analysis using NumPy, Pandas, and Matplotlib.

Algorithm:

1) Import Libraries:

`import pandas (pd), numpy (np), and matplotlib.pyplot (plt).`

2) Load and Explore Data:

- Load the dataset into a `DataFrame (df)` using `pd.read_csv()`.
- Print descriptive statistics with `df.describe()` and display the first few rows with `df.head()`.

3) Data Cleaning and Manipulation:

- Check for missing values with `df.isnull().sum().`
- Convert `'Date`' column to datetime format using `pd.to_datetime().`
- Calculate `'Total_Price'` by multiplying `'Quantity'` and `'Price_per_Unit'`.

4) Data Visualization:

- Group data by `'Product_Name'` and plot total sales for each product as a bar chart.
- Group data by `'Date'` and plot sales trend over time as a line chart.
- Create a scatter plot to visualize the relationship between `'Quantity'` and `'Total_Price'.`

5) Advanced Analysis:

- Calculate correlation coefficient between `'Quantity'` and `'Total_Price'` using `np.corrcoef()`.
- Find average spending per customer by grouping data by `'Customer_ID'` and calculating mean `'Total_Price'`.
- Identify top 5 products based on total sales using `product_sales.nlargest(5)`.

6) Display Visualization:

Use `plt.show()` to display each plot.

Source code:

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

# Data Loading and Exploration
df = pd.read_csv('sales_data.csv')
print("Descriptive Statistics using NumPy:")
print(np.array(df.describe()))
print("\nDescriptive Statistics using Pandas:")
print(df.describe())
print("\nFirst few rows of the dataset:")
print(df.head())

# Data Cleaning and Manipulation
print("\nMissing values:")
print(df.isnull().sum())
df['Date'] = pd.to_datetime(df['Date'])
df['Total_Price'] = df['Quantity'] * df['Price_per_Unit']

# Data Visualization
product_sales = df.groupby('Product_Name')['Total_Price'].sum()
plt.figure(figsize=(10, 6))
product_sales.plot(kind='bar', color='red')
plt.title('Total Sales for Each Product')
plt.xlabel('Product')
plt.ylabel('Total Sales ($)')
plt.xticks(rotation=45)
plt.show()

plt.figure(figsize=(10, 6))
sales_trend = df.groupby('Date')['Total_Price'].sum()
sales_trend.plot(kind='line', marker='o', color='orange')
plt.title('Sales Trend Over Time')
plt.xlabel('Date')
plt.ylabel('Total Sales ($)')
plt.show()

plt.figure(figsize=(10, 6))
plt.scatter(df['Quantity'], df['Total_Price'], color='green')
plt.title('Relationship between Quantity and Total Price')
plt.xlabel('Quantity')
plt.ylabel('Total Price ($)')
plt.show()

# Advanced Analysis
correlation_coefficient = np.corrcoef(df['Quantity'], df['Total_Price'])[0,1]
print("\nCorrelation Coefficient between Quantity and Total Price:",
correlation_coefficient)

average_spending_per_customer =
df.groupby('Customer_ID')['Total_Price'].mean()
print("\nAverage Total Spending per Customer:")
print(average_spending_per_customer)

top_5_products = product_sales.nlargest(5)
plt.figure(figsize=(10, 6))
top_5_products.plot(kind='bar', color='purple')
plt.title('Top 5 Products by Total Sales')
plt.xlabel('Product')
plt.ylabel('Total Sales ($)')
plt.xticks(rotation=45)
plt.show()
```

Source Code (TXT File):

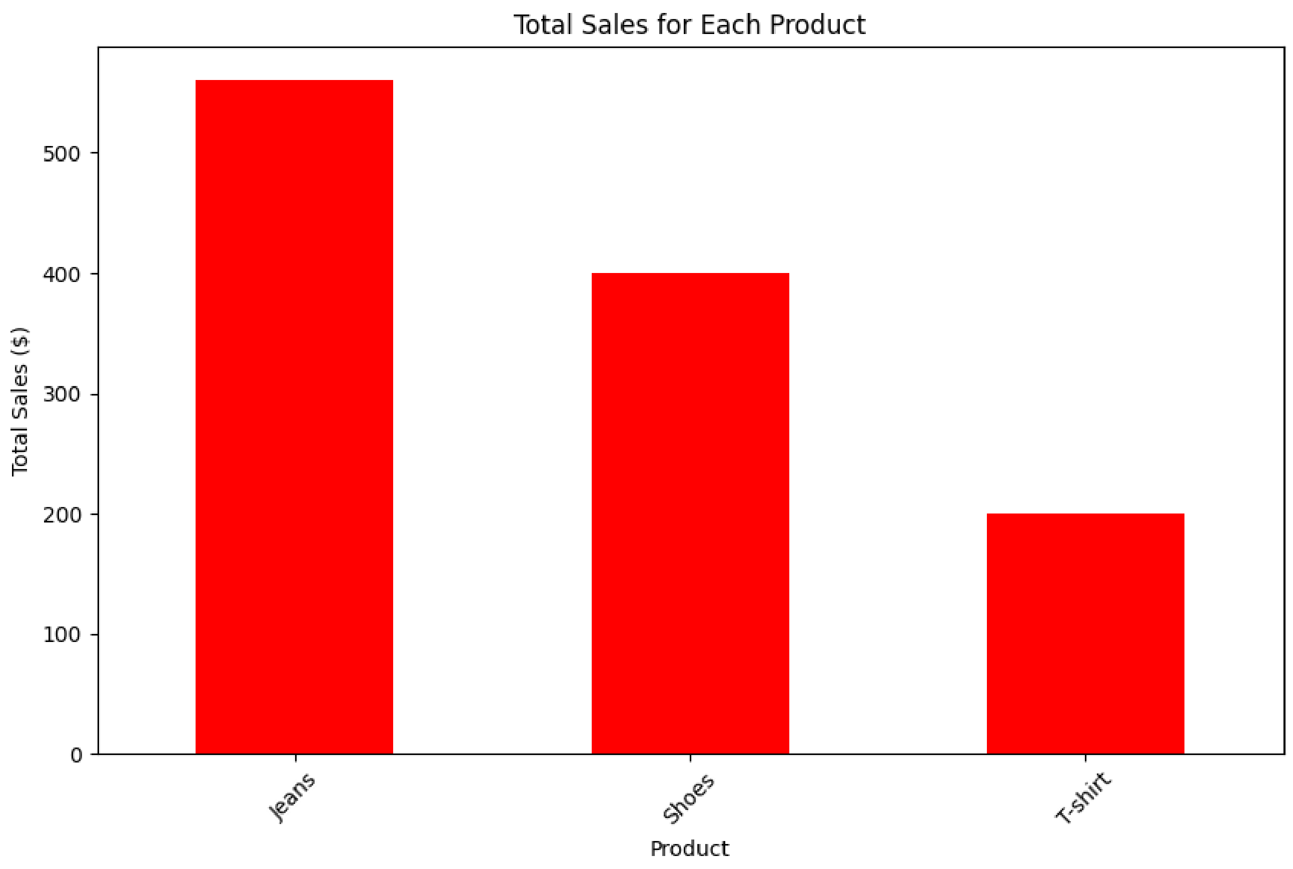
Note: Save a CSV File as `sales_data.csv`

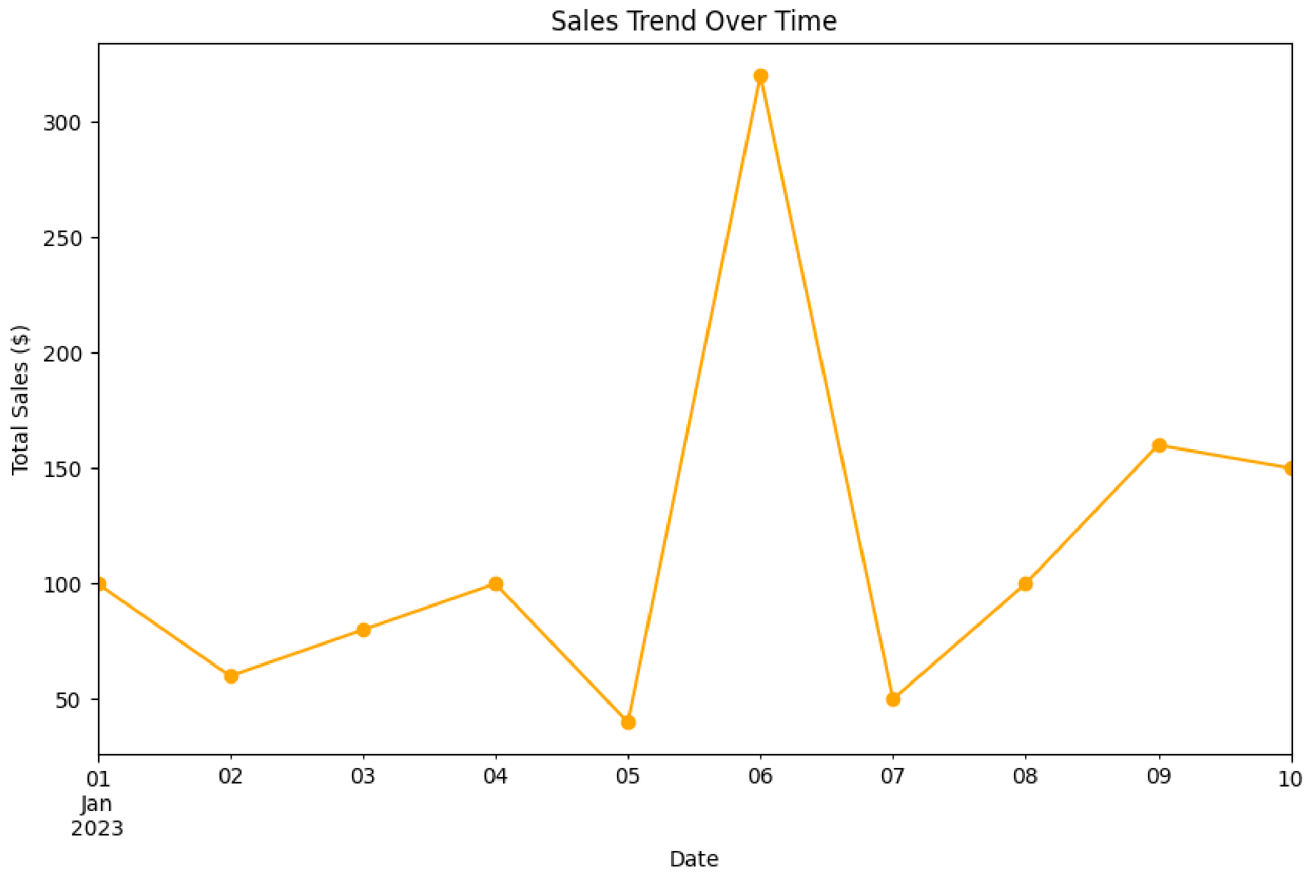
```
Transaction_ID,Product_Name,Quantity,Price_per_Unit,Customer_ID,Date
1,Shoes,2,50,101,2023-01-01
2,T-shirt,3,20,102,2023-01-02
3,Jeans,1,80,103,2023-01-03
4,Shoes,2,50,104,2023-01-04
5,T-shirt,2,20,101,2023-01-05
6,Jeans,4,80,102,2023-01-06
7,Shoes,1,50,103,2023-01-07
8,T-shirt,5,20,104,2023-01-08
9,Jeans,2,80,101,2023-01-09
10,Shoes,3,50,102,2023-01-10
```

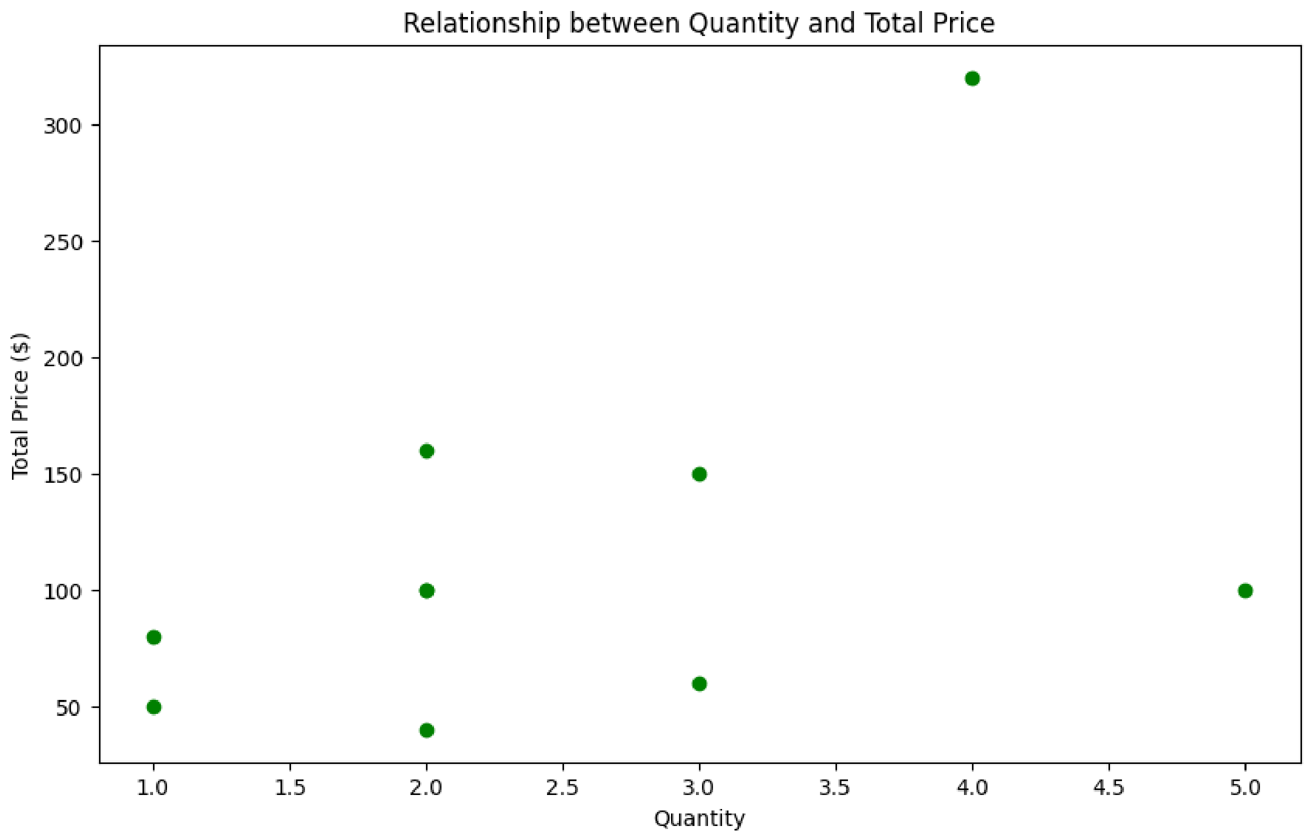
Note: Create and save CSV file and change the directory name to the CSV file directory which you created in the python source code and execute.

Sample Input and Output:

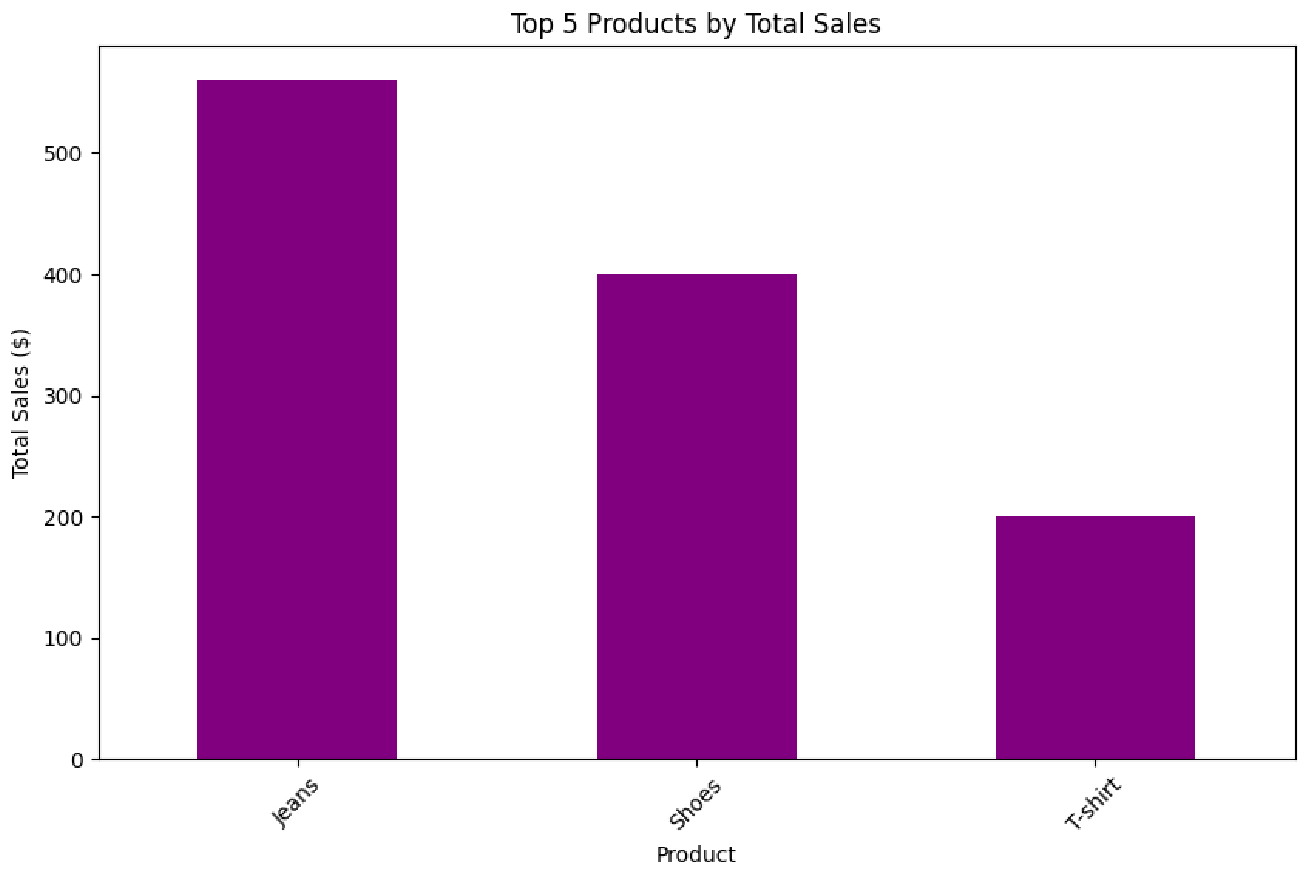
```
Descriptive Statistics using NumPy:

[[ 10.          10.          10.          10.        ]
 [  5.5          2.5         50.         102.3       ]
 [  3.02765035   1.26929552  24.49489743   1.15950181]
 [  1.           1.          20.         101.        ]
 [  3.25         2.          27.5        101.25      ]
 [  5.5          2.          50.         102.        ]
 [  7.75         3.          72.5        103.        ]
 [ 10.           5.          80.         104.        ]]

Descriptive Statistics using Pandas:

       Transaction_ID   Quantity  Price_per_Unit  Customer_ID
count        10.00000  10.000000       10.000000    10.000000
mean          5.50000   2.500000       50.000000   102.300000
std           3.02765   1.269296       24.494897     1.159502
min           1.00000   1.000000       20.000000   101.000000
25%           3.25000   2.000000       27.500000   101.250000
50%           5.50000   2.000000       50.000000   102.000000
75%           7.75000   3.000000       72.500000   103.000000
max          10.00000   5.000000       80.000000   104.000000

First few rows of the dataset:

   Transaction_ID Product_Name  Quantity  Price_per_Unit  Customer_ID        Date
0               1        Shoes         2              50          101  2023-01-01
1               2      T-shirt         3              20          102  2023-01-02
2               3        Jeans         1              80          103  2023-01-03
3               4        Shoes         2              50          104  2023-01-04
4               5      T-shirt         2              20          101  2023-01-05

Missing values:

Transaction_ID    0
Product_Name      0
Quantity          0
Price_per_Unit    0
Customer_ID       0
Date              0
dtype: int64
```


Total Sales for Each Product
Total Sales ($)
0
100
200
300
400
500
Jeans
Shoes
T-shirt
Product


Sales Trend Over Time
Total Sales ($)
50
100
150
200
250
300
01
Jan
2023
02
03
04
05
06
07
08
09
10
Date



```
Correlation Coefficient between Quantity and Total Price: 0.4715723507347863

Average Total Spending per Customer:

Customer_ID
101    100.000000
102    176.666667
103     65.000000
104    100.000000
Name: Total_Price, dtype: float64
```



Result:

Thus, the Python program for analyzing sales transactions dataset, including data loading, exploration, cleaning, manipulation, visualization, and advanced analysis using NumPy, Pandas, and Matplotlib has been executed successfully.

Ex No: 17 TIC-TAC-TOE GAME WITH TKINTER

Date:

Theoretical Concepts:

1. Importing Tkinter and Creating a Main Application Window:

Concept: `Tkinter` is the standard GUI toolkit for Python. It provides a set of tools for building desktop applications with graphical interfaces.

Syntax:
```
import tkinter as tk
root = tk.Tk()
```

In this Code:

- `'import tkinter as tk'` imports the `Tkinter` module and aliases it as `'tk'`.
- `'tk.Tk()'` creates the main application window.

2. Creating a Grid Layout:

Concept: To display the Tic-Tac-Toe board, we can use a grid layout in `Tkinter`. Each cell in the grid can be a button where players can make their moves.

Syntax:
```
button1 = tk.Button(root, text='', command=lambda: make_move(0, 0))
button1.grid(row=0, column=0)
```

In this Code:

- `'tk.Button(root, text='', command=lambda: make_move(0, 0))'` creates a button widget with empty text and binds it to a command (function) that makes a move at position (0, 0).
- `'button1.grid(row=0, column=0)'` places the button in the grid layout at row 0, column 0.

3. Game Logic:

Concept: Implementing the game logic involves tracking the state of the board, validating moves, and checking for winning conditions or a draw.

Syntax:
```
def check_winner():
    # Check rows, columns, and diagonals for winning combinations

def make_move(row, col):
    # Update the board state and check for winner or draw

def reset_game():
    # Reset the board and game state
```

In this code:

- `'check_winner()'` function checks for winning combinations in rows, columns, and diagonals.
- `'make_move(row, col)'` function updates the board state with the player's move at a specified position and checks for a winner or draw.
- `'reset_game()'` function resets the board and game state.

4. Displaying Game Results:

Concept: After each move, the program should check for a winning condition or a draw and display the result on the GUI.

Syntax:
```
result_label = tk.Label(root, text='')
result_label.grid(row=4, columnspan=3)

def display_result(result):
    result_label.config(text=result)
```

In this code:

- `'tk.Label(root, text='')'` creates a label widget with empty text.
- `'result_label.grid(row=4, columnspan=3)'` places the label in the grid layout at row 4, spanning across 3 columns.
- `'display_result(result)'` function updates the label text to display the game result.

5. Main Loop:

Concept: The main loop of the Tkinter application continuously listens for events and updates the GUI accordingly.

Syntax:

```
root.mainloop()
```

In this code:

- 'root.mainloop()' starts the Tkinter main loop, which listens for events (such as button clicks) and updates the GUI accordingly.

Aim:

To create a Python program using Tkinter for a two-player Tic-Tac-Toe game.

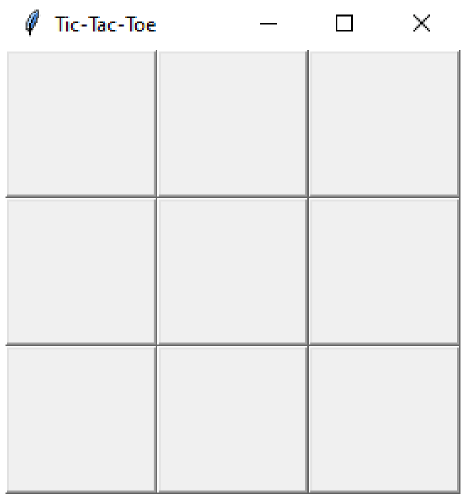
Algorithm:

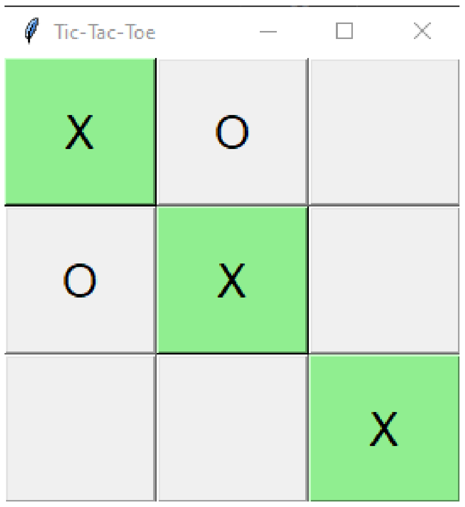
1. Import the necessary modules: `tkinter` and `messagebox`.
2. Define a class `TicTacToe` to represent the game.
3. Initialize the game attributes such as the `Tkinter` root window, current player, game board, and buttons grid in the constructor (`__init__` method).
4. Create a method `create_board()` to generate the game board with buttons.
5. Implement the `make_move()` method to handle player moves, update the board, and check for a winner or draw.
6. Define the `check_winner()` method to verify winning conditions by checking rows, columns, and diagonals.
7. Implement `highlight_winner()` method to visually highlight the winning combination on the GUI.
8. Implement `check_draw()` method to check for a draw condition.
9. Define `end_game()` method to display the result (winner or draw) using a `messagebox` and quit the game.
10. Add a `play()` method to start the main event loop using `root.mainloop()`.
11. In the main block, create an instance of `TicTacToe`, and call its `play()` method to start the game loop.

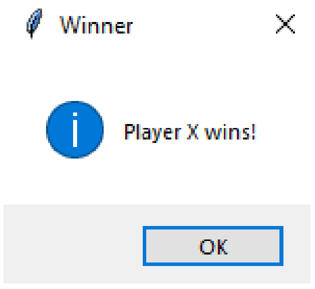
Source code:

```
import tkinter as tk
from tkinter import messagebox

class TicTacToe:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Tic-Tac-Toe")
        self.current_player = "X"
        self.board = [[' ' for _ in range(3)] for _ in range(3)]
        self.buttons = [[None for _ in range(3)] for _ in range(3)]
        self.create_board()

    def create_board(self):
        for i in range(3):
            for j in range(3):
                 self.buttons[i][j] = tk.Button(self.root, text="",
                font=("Helvetica", 20), width=5, height=2,command=lambda
                row=i, col=j: self.make_move(row, col))
                self.buttons[i][j].grid(row=i, column=j)

    def make_move(self, row, col):
        if self.board[row][col] == ' ':
            self.board[row][col] = self.current_player
            self.buttons[row][col].config(text=self.current_player)
            if self.check_winner() or self.check_draw():
                self.end_game()
            else:
                self.current_player = 'O' if self.current_player == 'X' else 'X'

    def check_winner(self):
        for i in range(3):
            if self.board[i][0] == self.board[i][1] == self.board[i][2] != ' ':
                self.highlight_winner(i, 0, i, 1, i, 2)
                return True
            if self.board[0][i] == self.board[1][i] == self.board[2][i] != ' ':
                self.highlight_winner(0, i, 1, i, 2, i)
                return True
        if self.board[0][0] == self.board[1][1] == self.board[2][2] != ' ':
            self.highlight_winner(0, 0, 1, 1, 2, 2)
            return True
        if self.board[0][2] == self.board[1][1] == self.board[2][0] != ' ':
            self.highlight_winner(0, 2, 1, 1, 2, 0)
            return True
        return False

    def highlight_winner(self, *coords):
        for i in range(0, len(coords), 2):
            self.buttons[coords[i]][coords[i+1]].config(bg='light green')
    def check_draw(self):
        for row in self.board:
            for cell in row:
                if cell == ' ':
                    return False
        return True

    def end_game(self):
        if self.check_winner():
            messagebox.showinfo("Winner", f"Player {self.current_player}
            wins!")
        else:
            messagebox.showinfo("Draw", "It's a draw!")
        self.root.quit()

    def play(self):
        self.root.mainloop()

if __name__ == "__main__":
    game = TicTacToe()
    game.play()
```

Sample Input and Output:







Player X wins!

Result:

Thus, the Python program using Tkinter for a two-player Tic-Tac-Toe game has been executed successfully.