PRACTICAL 1

Develop a program to understand the control structures of Python.

Input:

test.py > ...

```
# 1. Develop a program to understand the control structure of python.

# Python has three types of control structures:
# 1. Sequential - default mode
# 2. Selection - used for decisions and branching
# 3. Repetition - used for looping.

# 1. Sequential - default mode

a = 52
b = 47
c = b + a
print(a,"+",b,"=",c)

# 2. Selection - used for decisions and branching
# if
user="mohit123"
if user == "mohit123":
    print("Welcome !")

# if-else
num = 34
if num%2==1:
    print("Odd Number" )
else:
    print("Even Number" )

# if-elif-else
age = 12
if age < 13:
    print("Child" )
elif age < 18:
    print("teenage")
else:
    print("Adelt" )

# nested if:
a, b, c = 45, 32, 23
if a > b:
    if a > c:
        print ("a is largest" )
    else:
        print ("c is largest" )
```

```
44  else:
45      if b > c:
46          print ("b is largest" )
47      else:
48          print ("c is largest" )
49
50  # 3. Repetition - used for looping.
51  # for loop
52  for i in range (5):
53      print(i)
54  l = [4, 5, 6, 7, 4, 6]
55  for i in l:
56      print(i)
57  # while loop:
58  n = 0
59  while a<5:
60      print(a)
61  a-=1
62  # nested loops:
63  for i in range (5):
64      for j in range(i+1):
65          print ("*", end=" ")
66      print()
67
```

Output:

```
52 + 47 = 99
Welcome !
Even Number
Child
a is largest
0
1
2
3
4
4
5
6
7
4
6
*
* *
* * *
* * * *
* * * * *
PS D:\college docs\PDS>
```

Input:

test.py > ...

```python
# Function in python
import math as Maths

def isPrime(n):
    m = int(Maths.sqrt(n))+1
    i = 2
    while i<=m:
        if n%i == 0:
            return False
        i +=1
    return True
for j in range(20):
    if isPrime(j):
        print(str(j) + " is prime.")
    else:
        print(str(j)+" is non-prime.")

```

Output:

```
0 is prime.
1 is prime.
2 is non-prime.
3 is prime.
4 is non-prime.
5 is prime.
6 is non-prime.
7 is prime.
8 is non-prime.
9 is non-prime.
10 is non-prime.
11 is prime.
12 is non-prime.
13 is prime.
14 is non-prime.
15 is non-prime.
16 is non-prime.
17 is prime.
18 is non-prime.
19 is prime.
```

Input:

test.py > ...

```
#Scope of variable
def hello(name):
    Name = name.upper()
    print("Hello "+ Name)
hello("Manoj Kumar")
def student(name):
    print("I am "+name+".My school name is Shri Vidhya Niketan.")
student("Mohit Mahajan")

# Error handeling in python
a = [29,49,12,64,89]
try:
    for i in range(6):
        print(a[j])
except:
    print("Out of bound error.")
finally:
    print("Bye")
```

Output:

```
Hello MANOJ KUMAR
I am Mohit Mahajan.My school name is Shri Vidhya Niketan.
Out of bound error.
Bye
PS D:\college docs\PDS>
```

Q.1 What is a conditional statement in Python?

→ A conditional statement in Python is a programming construct that allows you to make decisions in your code based on certain conditions or criteria. Conditional statements are used to control the flow of a program by executing specific blocks of code if a particular condition is met. In Python, there are mainly two types of conditional statements:

1. if Statement: The if statement is used to test a condition. If the condition is true, a specified block of code is executed. If the condition is false, the code block is skipped.
   Here's the basic syntax:
   if condition:
   # Code to be executed if the condition is true
2. if-else Statement: The if-else statement extends the if statement. If the condition is true, the code block under the if statement is executed. If the condition is false, the code block under the else statement is executed. Here's the syntax:
   if condition:
   # Code to be executed if the condition is true
   else:
   # Code to be executed if the condition is false
3. if-elif-else Statement: The if-elif-else statement allows you to test multiple conditions in sequence. It starts with an if statement and can have one or more elif (short for "else if") blocks to test additional conditions. If none of the conditions is true, the code block under the else statement is executed. Here's the syntax:
   if condition1:
   # Code to be executed if condition1 is true
   elif condition2:
   # Code to be executed if condition2 is true
   else:
   # Code to be executed if none of the conditions are true

Q.2 What is a loop in Python?

→ A loop in Python is a control structure that allows you to repeatedly execute a block of code multiple times. Loops are used to automate repetitive tasks, iterate over collections (such as lists or dictionaries), and perform operations with

different values or conditions. Python provides two main types of loops: for loops and while loops.

1. for Loop: A for loop is used for iterating over a sequence (such as a list, tuple, string, or range) or any iterable object. It executes a block of code for each item in the sequence The loop variable takes on each value in the sequence during each iteration. Here's the basic syntax of a for loop:
   for variable in sequence:
   # Code to be executed for each item in the sequence
2. while Loop: A while loop is used to repeatedly execute a block of code as long as a certain condition is true. It continues to execute the code until the condition becomes false. Be cautious with while loops, as they can potentially lead to infinite loops if not controlled properly. Here's the basic syntax of a while loop:
   while condition:
   # Code to be executed as long as the condition is true

Q.3 What is the difference between a "for" loop and a "while" loop in Python?

while loop	**for loop**
A while loop is used, when the user does not know about the number of iterations needed.	A for loop is used, when the user know about the number of iterations needed.

Syntax:
while(condition)
{
…. //statement
}
	Syntax:
	for (initialization; condition; increment)
	{
	…. //statement
	}

The loop variable must be incremented before the loop. If the user fails to increment the loop variable, then the loop will not get terminate.	In for loop, the variable is initialized, checked, and incremented in this specific order. All these processes are performed inside this loop.

Example:
 	
i=1;
input=10;
while(i<=input)
{
//code
i++;
}
	Example:
	
	for(i=1;i<=10;i++)
	{
	//code
	}

Q.4 What is a function in Python?

→ In Python, a function is a named block of code that performs a specific task or a set of related tasks. Functions are used to encapsulate and modularize code, making it easier to manage, reuse, and understand. Functions are a fundamental

concept in Python and many other programming languages. Here are the key characteristics of functions in Python:

1. **Function Declaration**: Functions are declared using the def keyword followed by the function name and a pair of parentheses. You can also specify parameters (input values) inside the parentheses.
2. **Parameters**: Parameters are variables that you define within the parentheses of a function. They act as placeholders for values that are passed to the function when it is called. Functions can have zero or more parameters, depending on the specific requirements.
3. **Function Call**: To execute the code inside a function, you need to call the function by using its name followed by parentheses. You can pass arguments (values) to the function when calling it.
4. **Return Value**: Functions can return a value using the `return` statement. When a function returns a value, you can capture and use that value in your code. If a function does not contain a `return` statement, it returns `None` by default.
5. **Code Reusability**: Functions promote code reusability, allowing you to define a piece of functionality once and use it in multiple places in your program. You can import functions from external modules and libraries to leverage existing code.
6. **Scope**: Functions have their own local scope, which means that variables declared within a function are not accessible outside of it. This isolation helps prevent naming conflicts.

Q.5 What are scope in python?

→ In Python, the term "scope" refers to the region or context within which a variable or a name is recognized and can be used. Python has several different types of scopes, including local, enclosing (non-local), global, and built-in scopes. Understanding how scope works is crucial for writing clean and bug-free code.

Here are the main types of scopes in Python:

1. **Local Scope:** A local scope refers to the region within a specific function or block of code where a variable is declared and can be used. Variables defined in a local scope are only accessible within that function or block.
2. **Enclosing (Non-Local) Scope**: The enclosing scope is applicable in the context of nested functions. It refers to the scope of the containing or outer function. Variables in the enclosing scope can be accessed by the inner (nested) function.
3. **Global Scope**: The global scope is the highest level of scope, and it includes all the variables and names defined outside of any function or block. Variables defined in the global scope are accessible from anywhere within the script or module.
4. Built-in Scope: The built-in scope contains the names of all the built-in Python functions and objects, such as `print ()`, `len ()`, `int ()`, and others. These names are available for use without the need to import any modules.

PRACTICAL 2

Develop a program to understand the control structures of Python.

Input:

test.py > ...

```python
# List:
name = ["Mohit" , "Rudra" , "Mihir", "Parth" ]
print(name)
print(name[0])
print(name[1:3])
print(name[-2:])
print(name[2:])
print(name[:2])
name[2],name[3] = name[3], name[2]
print(name)
print(name[3])
name[3] = "Mohit"
print(name[3])
print(name[-3])
name.append ("Hello" )
print(name)
name.remove ("Hello" )
print(name)
print(name.pop())

# Tuple:
t = (2,4,5,7,8)
print(t)
print(t[1])
print(t[2:])
print(t[:3])
print(t[-1])
print(t[-1:])
# t[0] = 8 #TypeError: 'tuple' object does not support item assignment

# set
s = {3, 6, 7, 1, 6}
print(s)
s.add(34)
print(s)
s.remove (3)
print(s)

# dictionary
d = {
'name' : "Mohit",
'email' : "mohitmahajan5800@gmail.com",
'address' :"201, Dharmanandan co opp soc, VAPI"
}
print(d)
print(d['name' ])
print(d['email' ])
```

Output:

```
['Mohit', 'Rudra', 'Mihir', 'Parth']
Mohit
['Rudra', 'Mihir']
['Mihir', 'Parth']
['Mihir', 'Parth']
['Mohit', 'Rudra']
['Mohit', 'Rudra', 'Parth', 'Mihir']
Mihir
Mohit
Rudra
['Mohit', 'Rudra', 'Parth', 'Mohit', 'Hello']
['Mohit', 'Rudra', 'Parth', 'Mohit']
Mohit
(2, 4, 5, 7, 8)
4
(5, 7, 8)
(2, 4, 5)
8
(8,)
{1, 3, 6, 7}
{1, 34, 3, 6, 7}
{1, 34, 6, 7}
{'name': 'Mohit', 'email': 'mohitmahajan5800@gmail.com', 'address': '201, Dharmanandan co opp soc, VAPI'}
Mohit
mohitmahajan5800@gmail.com
PS D:\college docs\PDS>
```

Q.1 What method can you use to convert a string to uppercase in Python?

→ In Python, you can convert a string to uppercase using the upper () method. The upper () method is a built-in string method that returns a new string with all the alphabetic characters in uppercase. The original string remains unchanged. Here's how to use the upper () method:

Example:

Imput:

original_string = "Hello, World!"

uppercase_string = original_string.upper()

print(uppercase_string)

Output:

HELLO, WORLD!

Q.2 What is the difference between a tuple and a list in Python?

Parameters	Lists in Python	Tuples in Python
Nature	Mutable or changeable	Immutable or cannot change
Iteration	Iterating through Lists is time-consuming	Iterating through Tuples is not time-consuming
use	Good for insertion-deletion	Good for accessing elements
memory	Requires more memory as compare to Tuples	Requires less memory as compared to Lists
insertion	can insert an element at a particular index	Once created cannot be modified
methods	It has several inbuilt methods	Methods are few as compare to Lists
deletion	The List can delete any particular element	Tuples cannot delete elements rather it can delete a whole Tuple

creation	To create a List we can use the following ways
	`demo_List=[] #empty List`
	`demo_List=[1,2,3,4] #List with integer values`
	`demo_List=['Ram','Sham','Siya'] #List of strings`
		To create a Tuple we can use the following ways
		`# Python Tuple example`
		`demo=() #empty Tuple`
		`demo=(1,) #Tuple with a single element`
		`demo=(1,2,3,4) #Tuple with integer values`
		`demo=('Ram','Sham','Siya') #Tuple of strings`

accessing elements	To get an entry from the List we use index numbers of the List. See the following example for more details:
	`demo_List=['Ram','Sham','Siya'] #List of strings`
	`print(demo_List[0]) # access the first element`
	`print(demo_List[1]) #this will access the second element`
	`print(demo_List[2]) #this will access the third element`
	Output:
	Ram
	Sham
	Siya
		To get an entry from the Tuple we use index numbers of the Tuple. See the following example for more details:
		`# Python Tuple example`
		`demo_Tuple=('Ram','Sham','Siya') #Tuple of strings`
		`print(demo_Tuple[0]) #access the first element`
		`print(demo_Tuple[1]) # access the second element`
		`print(demo_Tuple[2]) # access the third element`
		Output:
		Ram
		Sham
		Siya

Q.3 How do you add an element to a list in Python?

→ In Python, you can add elements to a list using various methods. Here are a few common ways to add elements to a list:

1. Using the `append ()` method:
   - The `append ()` method is used to add a single element to the end of a list. The element is added as a single item, not as a new list.
     my_list = [1, 2, 3]
     my_list.append(4) # Adds the element 4 to the end of the list
2. Using the `extend ()` method:
   - The `extend ()` method is used to add multiple elements to a list. It takes an iterable (e.g., another list, tuple, or string) and adds its elements to the end of the list.
     my_list = [1, 2, 3]
     my_list.extend([4, 5, 6]) # Adds elements 4, 5, and 6 to the end of the list
3. Using the `insert ()` method:
   - The `insert ()` method allows you to add an element at a specific index in the list. You provide the index and the element to be inserted.
     my_list = [1, 2, 3]
     my_list.insert(1, 4) # Inserts the element 4 at index 1, pushing the existing elements to the right
4. Using the + operator:
   - You can use the + operator to concatenate two lists and create a new list with the combined elements.
     list1 = [1, 2, 3]
     list2 = [4, 5, 6]
     combined_list = list1 + list2 # Creates a new list with elements from both list1 and list2
5. Using list comprehensions:
   - You can create a new list by applying a condition to an existing list or iterable using list comprehensions.
     my_list = [1, 2, 3]
     doubled_list = [x * 2 for x in my_list] # Creates a new list with elements doubled

Q.4 How do you access a value in a dictionary using its key in Python?

→ In Python, you can access a value in a dictionary using its key by using the key inside square brackets (`[]`) or by using the `get ()` method. Here's how to do it:

1. Using Square Brackets: You can access a value in a dictionary by specifying the key inside square brackets immediately after the dictionary. If the key exists in the dictionary, the corresponding value is returned. If the key does not exist, it raises a `Key Error`.
2. Using the `get ()` Method: The `get ()` method is a safer way to access values in a dictionary. It allows you to specify a default value to be returned if the key does not exist in the dictionary, preventing `Key Error` from being raised.

Q.5 What is a set in Python?

→ In Python, a set is an unordered collection of unique elements. Sets are used to store multiple items, and they are similar to lists and tuples, but with some distinct characteristics:

1. Uniqueness: Sets do not allow duplicate elements. If you try to add an element that already exists in the set, it will not be added again.
2. Unordered: Sets do not maintain the order of elements as they are added. This means that the order of elements in a set is not guaranteed.
3. Mutable: Sets are mutable, meaning you can add and remove elements after the set is created.
4. Use of Curly Braces: Sets are defined using curly braces `{}` or the `set ()` constructor.

PRACTICAL 3

AIM: 3) Develop a program that reads a .csv dataset file using Pandas library and display the following content of the dataset

a) First five rows of the dataset

b) Complete data of the dataset

c) Summary or metadata of the dataset.

INPUT:

pds.py > ...

```python
# By MOHIT MAHAJAN (210160107050)
import pandas as pd
# Read the .csv dataset file
file_path = 'data.csv'  # Replace 'your_dataset.csv' with the actual file path
df = pd.read_csv(file_path)

# a) Display the first five rows of the dataset
print("a) First five rows of the dataset:")
print(df.head())
# check output 1___

# b) Display the complete data of the dataset
print("\nb) Complete data of the dataset:")
print(df)
# check output 1___

# c) Display summary or metadata of the dataset
print("\nc) Summary or metadata of the dataset:")
print(df.info())
# check output 1___

```

OUTPUT 1:

```
a) First five rows of the dataset:
   sr.no   name  roll.no                  email
0      1  mohit     7001       mohit@gmail.com
1      2  rohit     7002  rohitmahaj@gmail.com
2      3    raj     7003       raj21@gmail.com
3      4  mehul     7004    mehulcom@gmail.com
4      5  mihir     7005  mihirbharad@gmail.com
```

OUTPUT 2:

```
b) Complete data of the dataset:
    sr.no      name  roll.no                  email
0       1     mohit     7001        mohit@gmail.com
1       2     rohit     7002   rohitmahaj@gmail.com
2       3       raj     7003        raj21@gmail.com
3       4     mehul     7004     mehulcom@gmail.com
4       5     mihir     7005  mihirbharad@gmail.com
5       6     nayan     7006   nayankumar@gmail.com
6       7     kiran     7007         kriz@gmail.com
7       8     rudra     7008    rudrashah@gmail.com
8       9     hetvi     7009      hetvi11@gmail.com
9      10    kirtan     7010         kirtanbhav.com
10     11   rushabh     7011     rushabhm@gmail.com
11     12     dipak     7012   dipakbhuva@gmail.com
12     13  kasheesh     7013    kash11221@gmail.com
```

OUTPUT 3:

```
c) Summary or metadata of the dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13 entries, 0 to 12
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   sr.no    13 non-null     int64
 1   name     13 non-null     object
 2   roll.no  13 non-null     int64
 3   email    13 non-null     object
dtypes: int64(2), object(2)
memory usage: 544.0+ bytes
None
PS D:\programing\web>
```

Q..1 What library should be used to read a .csv dataset file in Python?

→ In Python, you can use the built-in `csv` module to read and write CSV (Comma-Separated Values) files. The `csv` module provides functions and classes for working with CSV data, making it a convenient choice for reading dataset files in CSV format.

Here's a simple example of how to read a CSV dataset file using the `csv` module:

```
import csv
# Open the CSV file for reading
with open('your_dataset.csv', newline='') as csvfile:
    # Create a CSV reader
    csvreader = csv.reader(csvfile)
    # Iterate through the rows in the CSV file
    for row in csvreader:
        # Process each row as a list of values
        print(row)
```

Q.2 Which method is used to read a .csv file using Pandas library?

→ To read a .csv file using the Pandas library in Python, you can use the `read_csv()` function provided by Pandas. This function allows you to easily read and load data from a CSV file into a Pandas Data Frame, which is a versatile data structure for working with tabular data. Here's how you can use the `read_csv()` function:

```
import pandas as pd
# Read a CSV file and create a DataFrame
df = pd.read_csv('your_dataset.csv')
# Now, you can work with the data in the DataFrame
print(df.head())  # Print the first few rows of the DataFrame
```

Q.3 How can you display the first five rows of the dataset using Pandas?

→ You can display the first five rows of a dataset using Pandas by using the `.head()` method on a Pandas DataFrame. The. head `()` method returns the first `n` rows (default is 5) of the DataFrame. Here's how you can do it:

```
import pandas as pd
# Read a CSV file and create a DataFrame
df = pd.read_csv('your_dataset.csv')
```

```
# Display the first five rows of the DataFrame
print(df.head())
```

Q.4 How can you display the complete data of the dataset using Pandas?

→ To display the complete data of a dataset in Pandas, you can use the `.to_string()` method to create a string representation of the entire DataFrame and then print that string. Here's how you can do it:

```
import pandas as pd
# Read a CSV file and create a DataFrame
df = pd.read_csv('your_dataset.csv')
# Set the option to display all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
# Display the complete data of the DataFrame
print(df.to_string(index=False))
```

Q.5 How can you display the summary or metadata of the dataset using Pandas?

→ To display a summary or metadata of a dataset using Pandas, you can use the `.info()` and `.describe()` methods. These methods provide useful information about the structure, data types, and statistical summary of the dataset. Here's how you can use them:

1. Using `.info ()`: The `.info ()` method provides an overview of the DataFrame, including the number of non-null values, data types, and memory usage.
2. Using `.describe()`: The `.describe()` method provides basic statistics for numeric columns, including count, mean, standard deviation, minimum, and maximum values. It's useful for gaining insights into the distribution of the data.

PRACTICAL 4

AIM: Develop a program that shows application of slicing and dicing over the rows and columns of the dataset.

pds.py > ...

```
# By MOHIT MAHAJAN 210160107050
from pandas import *
# Read CSV file
a = read_csv("student.csv")
print(a)
# chech output 1 ____


# Print "name" column:
print(a["name"])
# chech output 2 ____

# Print "name" and "eamil" columns:
print(a[["name", "email"]])
# chech output 3 ____

# print row 3rd
print(a.iloc[[2]])
# chech output 4 ____

# print row 1rd
print(a.iloc[[0]])
# chech output 5 ____

# print 2nd and 4rth row.
print(a.loc[[1,3]])
# chech output 6____

# print 1nd and 2rth row.
print(a.loc[[0,1]])
# chech output 7 ____

# print 1nd and 4rth row of name and language:
print((a.loc[[0,3]])[["name", "language"]])
# chech output 8 ____

# print 1nd and 3rth row of name and age:
print((a.loc[[0,2]])[["name", "age"]])
# chech output 9 ____

# print 1nd, 2nd and 4rth row of name and email where email first column:
print((a.loc[[0,1,3]])[["email", "name"]])
# chech output 10 ____

# print 2nd, 3nd and 4rth row of name and age where age first column:
print((a.loc[[1,2,3]])[["age", "name"]])
# chech output 11____

```

OUTPUT 1:

	name	age	email	language
0	nayan	21	nayan123@g.c	gujarati
1	mohit	22	mohit545@g.c	hindi
2	rudra	21	rudra12@g.c	gujarati
3	jignesh	22	jig123@g.c	marathi

OUTPUT 2:

```
0      nayan
1      mohit
2      rudra
3    jignesh
Name: name, dtype: object
```

OUTPUT 3:

	name	email
0	nayan	nayan123@g.c
1	mohit	mohit545@g.c
2	rudra	rudra12@g.c
3	jignesh	jig123@g.c

OUTPUT 4:

	name	age	email	language
2	rudra	21	rudra12@g.c	gujarati

OUTPUT 5:

	name	age	email	language
0	nayan	21	nayan123@g.c	gujarati

OUTPUT 6:

	name	age	email	language
1	mohit	22	mohit545@g.c	hindi
3	jignesh	22	jig123@g.c	marathi

OUTPUT 7:

	name	age	email	language
0	nayan	21	nayan123@g.c	gujarati
1	mohit	22	mohit545@g.c	hindi

OUTPUT 8:

	name	language
0	nayan	gujarati
3	jignesh	marathi

OUTPUT 9:

	name	age
0	nayan	21
2	rudra	21

OUTPUT 10:

	email	name
0	nayan123@g.c	nayan
1	mohit545@g.c	mohit
3	jig123@g.c	jignesh

OUTPUT 11:

	age	name
1	22	mohit
2	21	rudra
3	22	jignesh

Q.1 What is the purpose of slicing and dicing in data analysis?

→ Slicing and dicing are important techniques in data analysis that allow you to manipulate, explore, and extract specific subsets of data from a larger dataset. These techniques are used to achieve several purposes in data analysis:

1. Data Exploration: Slicing and dicing data help you explore and understand the dataset by examining different subsets. It allows you to uncover patterns, trends, outliers, and relationships within the data.
2. Data Summarization: You can create summaries or aggregates of data subsets, such as calculating averages, totals, or percentages for specific groups or categories within the data. This helps in generating insights and simplifying complex datasets.
3. Data Filtering: Slicing and dicing enable you to filter out irrelevant or unwanted data and focus on specific subsets that are relevant to your analysis. This is particularly useful for removing noise and reducing data dimensionality.
4. Data Visualization: By selecting and extracting specific slices of data, you can create visualizations, charts, and graphs that highlight key aspects of the dataset. Visualizations are powerful for conveying insights to stakeholders.
5. Hypothesis Testing: When performing statistical analysis or hypothesis testing, you often need to work with subsets of data to test specific hypotheses or make comparisons between different groups or categories.
6. Feature Engineering: In machine learning and predictive modeling, slicing and dicing can help you create new features by aggregating or transforming existing data. Feature engineering can improve the performance of machine learning models.
7. Report Generation: Slicing and dicing are commonly used when generating reports or dashboards. You can select and display specific sections of data to present meaningful information to end-users.
8. Decision-Making: Data analysis often involves making decisions based on specific criteria. Slicing and dicing allow you to extract the relevant information necessary for informed decision-making.
9. Problem Solving: Slicing and dicing techniques are used to break down complex problems into smaller, more manageable components. This simplifies the analysis and can lead to more effective problem-solving strategies.

Q.2 Which function of the Pandas library is used to load a .csv dataset file into Python?

→ The function in the Pandas library used to load a .csv dataset file into Python is `pd.read_csv()`. This function reads the contents of a CSV file and creates a Pandas DataFrame, which is a versatile data structure for working with tabular data. Here's how you can use `pd.read_csv()`:

```
import pandas as pd

# Read a CSV file and create a DataFrame

df = pd.read_csv('your_dataset.csv')
```

Q.3 What is the difference between loc and iloc in Pandas DataFrame indexing?

→

	loc	iloc
	Label-based (Location)	*Integer position-based (iLocation)*
A Value	A single label `e.g.) df.loc['x1']`	A single integer `e.g.) df.iloc[0]`
A List or Array	A list of labels `e.g.) df.loc[['x1', 'x2']]`	A list of integers `e.g.) df.iloc[[0, 1]]`

A Slice Object	A slice object with labels 'x1' and 'x2' are included `e.g.) df.loc['x1':'x2']`	A slice object with inetgers i is included, j is not included `e.g.) df.iloc[i:j]`

Conditions or Boolean Array	Conditional that returns a boolean Series
	`e.g.) df.loc[df['x1'] > 10]`
	`df.loc[[True, False, False]]`
		A boolean array
		`e.g.) df.iloc[[True, False, False]]`

A Callable Function	A callable function `e.g.) df.loc[lambda df: df['x1'] > 10]`	A callable function `e.g.) df.iloc[lambda x: x.index %2 == 0]`

Q.4 How can Boolean indexing be used to filter rows of a dataset based on specific criteria?

→ Boolean indexing in Pandas is a powerful technique for filtering rows of a dataset based on specific criteria. It allows you to select rows from a DataFrame that satisfy one or more conditions defined as Boolean expressions. Here's how you can use Boolean indexing to filter rows of a dataset:

1. Define the condition(s): Start by defining one or more conditions as Boolean expressions. These conditions typically involve comparisons, such as equality, inequality, or logical operators like AND (`&`) and OR (`|`).
2. Apply the Boolean index: Use the defined condition(s) to create a Boolean index (a Boolean array) that specifies which rows meet the criteria.
3. Use the Boolean index to filter rows: Pass the Boolean index inside square brackets to the DataFrame to select the rows that meet the specified conditions.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 32, 27, 30, 22]}
df = pd.DataFrame(data)
# Define a condition to filter rows (e.g., ages greater than 25)
condition = df['Age'] > 25
# Apply the boolean index and select the rows that meet the condition
filtered_df = df[condition]
# Display the filtered DataFrame
print(filtered_df)
```

Q.5 What is the purpose of aggregation functions in data analysis?

→ Aggregation functions, also known as aggregate functions, are essential tools in data analysis. These functions serve several purposes in data analysis:

1. **Summarizing Data**: Aggregation functions allow you to summarize and condense large datasets into more manageable forms. They help in simplifying complex data, making it easier to understand and work with.
2. **Statistical Analysis**: Aggregation functions are used to calculate various statistical measures, such as mean, median, mode, standard deviation, variance, minimum,

maximum, and percentiles. These measures provide insights into the distribution and characteristics of data.

3. **Data Exploration**: Aggregation functions are crucial for exploring data. They help you identify patterns, trends, and outliers by summarizing the data in different ways.
4. **Data Reduction**: Aggregation functions can reduce the dimensionality of data by creating summary statistics or aggregating data by certain attributes or time periods. This is particularly important when dealing with big data or large datasets.
5. **Reporting and Visualization**: Aggregation is often used for generating reports and creating visualizations. Summary statistics produced by aggregation functions are valuable for conveying information to stakeholders in a concise and understandable manner.
6. **Grouping and Pivot Tables**: Aggregation functions are commonly used in conjunction with grouping operations. You can group data by one or more attributes and apply aggregation functions within each group. This is helpful for performing analyses by categories or subgroups. Pivot tables are an example of this.

Q.6 Which visualization libraries can be used to create visualizations of the sliced and diced data?

There are several popular visualization libraries in Python that you can use to create visualizations of sliced and diced data. These libraries offer a wide range of chart types and customization options for displaying data in a meaningful and visually appealing way. Some of the commonly used visualization libraries include:

1. Matplotlib
2. Seaborn
3. Plotly
4. Pandas Plotting
5. Bokeh
6. Altair
7. Ggplot

Q.7 What is the importance of documenting the slicing and dicing process during data analysis?

→ Documenting the slicing and dicing process during data analysis is crucial for several reasons:

1. Reproducibility: Documenting the steps taken to slice and dice data helps ensure that your analysis can be reproduced by others, or even by yourself at a later date.
2. Collaboration: When working in a team, clear documentation allows team members to understand and build upon each other's work.
3. Knowledge Transfer: Documenting the process helps in transferring knowledge to others within your organization.
4. Error Detection and Debugging: Well-documented analysis processes make it easier to identify errors, inconsistencies, or issues that might arise during data manipulation.
5. Auditing and Compliance: In regulated industries or for compliance purposes, proper documentation of data analysis is often a requirement. This documentation helps in demonstrating adherence to data handling and analysis protocols.

6. Decision Making: Documented analysis processes provide a clear record of how data was sliced and diced, which variables were used, and what transformations were applied.
7. Long-Term Use: Documentation ensures that the analysis remains valuable and relevant over time.
8. Education and Training: Documented data analysis processes can be used for educational purposes, such as training new team members or providing tutorials for others in the organization.
9. Troubleshooting and Performance Optimization: If you encounter performance issues or bottlenecks in your analysis, documented processes can help identify areas for optimization and improvement.
10. Communication: Documentation serves as a means of communication between data analysts and other stakeholders, such as business managers, who may not have a deep technical understanding.

Q.8 What is the advantage of iterating and refining the analysis during the slicing and dicing process?

→ Iterating and refining the analysis during the slicing and dicing process offers several advantages:

1. Improved Accuracy: Iteration allows you to revisit and validate your findings, reducing the chances of errors or incorrect conclusions. Refining the analysis helps in rectifying mistakes, enhancing the quality of results, and ensuring they are more accurate.
2. Deeper Insights: Revisiting and refining the analysis can lead to deeper insights into the data. You may discover new patterns, trends, or relationships that were not apparent in the initial analysis. This can provide a more comprehensive understanding of the data.
3. Adaptation to Changing Requirements: As the project evolves or new information becomes available, your analysis may need to adapt. Iteration allows you to incorporate changing requirements, new data sources, or modified objectives to ensure your analysis remains relevant and useful.
4. Better Decision Making: By continually refining your analysis, you can provide decision-makers with more robust and up-to-date information. This enables better-informed decisions and strategies, especially in dynamic environments.
5. Reducing Bias and Assumptions: Revisiting your analysis helps you identify and address any biases or assumptions that may have influenced your initial findings. It allows for a more objective and thorough examination of the data.
6. Enhanced Problem Solving: Iteration provides opportunities to tackle complex or challenging problems from different angles. You can experiment with alternative methods, models, or approaches to find the best solutions.
7. Minimizing Overfitting: In machine learning and statistical modeling, overfitting is a common issue. Iteration allows you to fine-tune models, perform cross-validation, and reduce overfitting by adjusting model parameters or features.
8. Feedback Incorporation: During the analysis, you may receive feedback from stakeholders, domain experts, or colleagues. Iteration provides the means to incorporate this feedback and improve the analysis based on their input.

9. Continuous Validation: Revisiting and refining the analysis helps ensure that the findings are consistent and reliable. Continuous validation of the results increases confidence in the conclusions.
10. Optimization and Efficiency: Through iteration, you can optimize your analysis workflow. This may involve streamlining data preprocessing, improving code efficiency, or identifying areas where computational resources can be utilized more effectively.
11. Risk Mitigation: By regularly refining the analysis, you can identify and address potential risks or uncertainties in your findings. This proactive approach helps mitigate the impact of incorrect or incomplete conclusions.
12. Stakeholder Engagement: Involving stakeholders in the iteration process fosters collaboration and engagement. It allows them to see the progress and contribute to the analysis, leading to a more informed decision-making process.

Q.9 Can slicing and dicing be applied only to numerical data or can it also be applied to categorical data?

→ Slicing and dicing can be applied to both numerical and categorical data. These techniques are fundamental aspects of data analysis and exploration, and they are used to examine and extract insights from various types of data, including:

1. Numerical Data: Slicing and dicing numerical data typically involves partitioning the data into subsets based on specific numerical criteria. For example, you can slice a dataset of sales revenue by time intervals, such as months or quarters, to analyze trends.
2. Categorical Data: Categorical data consists of discrete, non-numeric values or labels, such as product categories, customer types, or geographic locations. Slicing and dicing categorical data involves segmenting and subgrouping the data based on different categories or labels.

Q.10 How can the insights gained from slicing and dicing be used to make data-driven decisions?

→ The insights gained from slicing and dicing data can be used to make data-driven decisions in various ways:

1. Identify Trends and Patterns
2. Segmenting Audiences
3. Performance Evaluation
4. Root Cause Analysis
5. Resource Allocation
6. Risk Management
7. Product Development and Improvement
8. Cost Reduction
9. Personalization
10. Strategic Planning
11. Performance Metrics and KPIs

PRACTICAL 5

AIM: Develop a program that shows usage of aggregate function over the input dataset. a) describe b) max c) min d) mean e) median f) count g) std h) Corr

INPUT:

pandas.py > ...

```python
# By MHOIT MAHAJAN 210160107050
from pandas import *
# Read CSV file
project = read_csv('data3.csv')
print("Orognal Data")
print(project.to_string())
#check output 1___

print(project.describe())
#check output 2___

# max(), min(), mean(), median() methods
print("Max of input: " + str(project['input'].dropna().max()))
print("Min of input: " + str(project['input'].dropna().min()))
print("Mean of output: " + str(project['output'].dropna().mean()))
print("Median of output: " + str(project['output'].dropna().median()))
#check output 3___

# count () method
print(project.count())
print("Total number of projects are: " + str(project["Title"].count()))
#check output 4___

# Standerd deviation usinf std() method
print("Standerd deviation of input is: " + str(project["input"].std()))
print()
#check output 5___

#corr() method
df = {
    "Number_1":[12,23,34,5],
    "Number_2":[55,42,16,9],
}
d = DataFrame(df)
print(d.corr())
#check output 6___
```

OUTPUT 1:

```
Orognal Data
      Title  input  output
0  project1     12    34.0
1  project2     32    34.0
2  project3     69    37.0
3  project4     54     NaN
4  project5     41    35.0
5  project6     67     NaN
6  project7     45    33.0
7  project8     11    38.0
```

OUTPUT 2:

```
             input     output
count    8.000000   6.000000
mean    41.375000  35.166667
std     22.251404   1.940790
min     11.000000  33.000000
25%     27.000000  34.000000
50%     43.000000  34.500000
75%     57.250000  36.500000
max     69.000000  38.000000
```

OUTPUT 3:

```
Max of input: 69
Min of input: 11
Mean of output: 35.166666666666664
Median of output: 34.5
```

OUTPUT 4:

```
Title     8
input     8
output    6
dtype: int64
Total number of projects are: 8
```

OUTPUT 5:

```
Standerd deviation of input is: 22.25140445005663
```

OUTPUT 6:

```
           Number_1  Number_2
Number_1   1.000000 -0.050879
Number_2  -0.050879  1.000000
```

Q.1 What is the purpose of using aggregate functions in a dataset?

→Aggregate functions are used in datasets to perform various summary and computation tasks on a group of data or the entire dataset. These functions are essential for data analysis, reporting, and deriving meaningful insights from the data. The main purposes of using aggregate functions in a dataset are:

1. Summarization: Aggregate functions provide a concise summary of the data. They condense large datasets into smaller, more manageable results, making it easier to understand and interpret the information.
2. Descriptive Statistics: Aggregate functions calculate descriptive statistics, such as the mean, median, mode, standard deviation, and variance. These statistics help you gain insights into the central tendencies, spread, and distribution of the data.
3. Data Exploration: Aggregate functions are used to explore the data, identify trends, patterns, and outliers, and get an overview of the data's characteristics. They can highlight key features or issues in the dataset.
4. Reporting: Aggregate functions are commonly used in reporting and visualization. They help create summary reports, charts, and dashboards to present data in a meaningful and informative way.
5. Comparisons: Aggregating data allows for easy comparisons between different groups, categories, or time periods. For instance, you can compare the sales figures of different product categories or assess performance over different quarters.
6. Data Reduction: Aggregating data reduces the amount of detail while retaining essential information. This is especially useful when dealing with large datasets, as it simplifies analysis and visualization.

Common aggregate functions include:

- SUM: Calculates the total sum of numeric values in a dataset.
- AVG (Average): Computes the mean or average value of a numeric dataset.
- MAX: Identifies the maximum value in a dataset.
- MIN: Identifies the minimum value in a dataset.
- COUNT: Counts the number of non-null values in a dataset.
- DISTINCT: Counts the number of unique values in a dataset.
- MEDIAN: Calculates the middle value in a dataset when sorted.
- MODE: Identifies the most frequently occurring value in a dataset.

Q.2 Which aggregates function calculates the average of a numerical column?

→ The aggregate function that calculates the average of a numerical column is typically referred to as the "AVG" function.

Here's how you would use the AVG function to calculate the average of a numerical column in SQL

```
SELECT AVG(column_name) FROM table_name;
```

In this SQL query, "column_name" should be replaced with the name of the numerical column from which you want to calculate the average, and "table_name" should be replaced with the name of the table in which the column resides.

Q.3 Which of the following aggregate functions calculates the correlation between two numerical columns?

→ None of the standard SQL aggregate functions calculates the correlation between two numerical columns directly. Calculating the correlation between two numerical columns is a more complex statistical operation that typically involves specialized statistical functions or libraries, and it's not part of the standard SQL aggregate functions.

In SQL, you would typically calculate the correlation between two numerical columns using a specific statistical function or by exporting the data to a statistical analysis tool or programming language that supports correlation analysis, such as Python with libraries like NumPy or pandas, R, or specialized statistical software.

For example, in Python, you can use the `corr()` function from the pandas library to calculate the correlation between two columns:

```
import pandas as pd

# Assuming df is a pandas DataFrame

correlation = df['column1'].corr(df['column2'])
```

Q.4 Which of the following aggregate functions returns the number of non-missing values in a column?

→ The aggregate function that returns the number of non-missing (non-null) values in a column is "COUNT." You can use the COUNT function in SQL to count the number of non-null values in a specified column.

Here's the basic syntax in SQL:

```
SELECT COUNT(column_name) FROM table_name;
```

In this query:

- "column_name" should be replaced with the name of the column for which you want to count non-null values.
- "table_name" should be replaced with the name of the table where the column is located.

Q. 5 What is the purpose of using the describe() function in Pandas?

→ The `describe()` function in Pandas serves the purpose of providing summary statistics and descriptive information about a DataFrame or Series in Python. Its main objectives are as follows:

1. **Summary Statistics**: The primary purpose of `describe()` is to calculate and display common summary statistics for the numerical columns within a DataFrame or Series. These statistics include count, mean, standard deviation, minimum, quartiles, and maximum values. For non-numeric (categorical) data, it provides statistics like count, unique values, top value, and frequency.
2. **Data Inspection**: It allows you to quickly inspect and assess the basic characteristics of your data. You can use it to identify missing values, understand the data type of each column, and get an initial sense of the data's distribution.
3. **Data Exploration**: `describe()` provides a starting point for data exploration by offering an overview of the dataset. It helps you decide which columns may be relevant for further analysis, visualization, or modeling.
4. **Data Cleaning and Preprocessing**: By revealing missing values or potential outliers, `describe()` can guide you in the initial stages of data cleaning and preprocessing, helping you make informed decisions on how to handle such issues.
5. **Data Quality Assessment**: It helps assess the overall data quality, consistency, and integrity. You can identify data inconsistencies or unusual characteristics that may require attention.
6. **Feature Selection and Engineering**: The summary statistics generated by `describe()` can be valuable in deciding which features (columns) are important for your analysis or machine learning models. It can also provide insights into whether feature engineering is needed.
7. **Rapid Data Understanding**: `describe()` offers a quick way to gain an initial understanding of your data without the need to write custom code for calculating these basic statistics.

PRACTICAL 6

AIM: Develop a program that applies split and merge operations on the datasets.

INPUT:

merge and split.py > ...

```
# By MOHIT MAHAJAN (210160107050)
from pandas import *
# Read CSV files:
df1 = read_csv("productlist1.csv")
df2 = read_csv("productlist2.csv")
# display file 1:
print(df1)
# check OUTPUT 1___

# display file 2:
print(df2)
# check OUTPUT 2____

# Mearg both files:
df3 = concat([df1, df2])
# print mearged dataset
print(df3)
# check OUTPUT 3____

# Mearg both files [axis = 1], join="inner":
df4 = concat([df1, df2], axis=1, join="inner")
# print mearged dataset
print(df4)
# here, file 1 last two row have not being displaid as join = "inner"
# check OUTPUT 4____

# Mearg both files [axis = 1], join="outer":
df5 = concat([df1, df2], axis=1, join="outer")
# print mearged dataset
print(df5)
# here, file 1 last two row have being displaid as join = "outer"
# check OUTPUT 5____

# Read files
f1 = read_csv("productlist1.csv")
f2 = read_csv("productlist2.csv")
f1 = DataFrame(f1, index=[1,2,3,4,5,6,7,8,9,10])
f2 = DataFrame(f2, index=[3,4,5,6,7,8])
print("Axis = 0")
f = concat([f1, f2], axis=0,join="inner")
print(f)
print("Axis = 1")
f = concat([f1, f2], axis=1,join="inner")
print(f)
# check OUTPUT 6____
```

```python
# Merging DataFrame
d1 = {
    "key": ["k1", "k2", "k3", "k4"],
    "Value1": ["v1", "v2", "v3", "v4"],
    "Value2": ["v11", "v22", "v33", "v44"]
}
d2 = {
    "key": ["k3", "k4", "k5", "k6"],
    "Value1": ["v_1", "v_2", "v_3", "v_4"],
    "Value2": ["v_11", "v_22", "v_33", "v_44"]
}
# Create dataframe
df1 = DataFrame(d1)
df2 = DataFrame(d2)
print("1st dataset")
print(df1)
print("2nd dataset")
print(df2)
# check OUTPUT 7____

res = merge(df1, df2, on='key')
print(res)

res = merge(df1, df2, on="key", how='left')
print(res)
# check OUTPUT 8____

# Split
df1 = read_csv("productlist1.csv")
print(df1)
# check OUTPUT 9____

print(df1.iloc[1])
# check OUTPUT 10____

print(df1.iloc[3])
# check OUTPUT 11____

print(df1.iloc[1:4])
# check OUTPUT 12____

print(df1.iloc[3:6])
# check OUTPUT 13____

print(df1[["product_name", "quantity"]].iloc[1:])
# check OUTPUT 14____

print(df1[["product_name", "quantity"]].iloc[:4])
# check OUTPUT 15____
```

OUTPUT:

OUTPUT 1:

	product_name	product_price	quantity
0	tea	100	1
1	shampoo	90	1
2	milkpowder	200	1
3	foodpacket	30	2
4	toothpest	70	1
5	brush	20	2
6	dishwasher	50	1
7	facewash	60	1
8	icecreampowder	40	1

OUTPUT 2:

	product_name	product_price	quantity
0	pencilbox	70	1
1	pen	5	20
2	notebook	45	12
3	campasbox	100	1
4	schoolbag	1200	1
5	waterbottle	120	1
6	lunchbox	100	1

OUTPUT 3:

	product_name	product_price	quantity
0	tea	100	1
1	shampoo	90	1
2	milkpowder	200	1
3	foodpacket	30	2
4	toothpest	70	1
5	brush	20	2
6	dishwasher	50	1
7	facewash	60	1
8	icecreampowder	40	1
0	pencilbox	70	1
1	pen	5	20
2	notebook	45	12
3	campasbox	100	1
4	schoolbag	1200	1
5	waterbottle	120	1
6	lunchbox	100	1

OUTPUT 4:

	product_name	product_price	quantity	product_name	product_price	quantity
0	tea	100	1	pencilbox	70	1
1	shampoo	90	1	pen	5	20
2	milkpowder	200	1	notebook	45	12
3	foodpacket	30	2	campasbox	100	1
4	toothpest	70	1	schoolbag	1200	1
5	brush	20	2	waterbottle	120	1
6	dishwasher	50	1	lunchbox	100	1

OUTPUT 5:

	product_name	product_price	quantity	product_name	product_price
0	tea	100	1	pencilbox	70.0
1	shampoo	90	1	pen	5.0
2	milkpowder	200	1	notebook	45.0
3	foodpacket	30	2	campasbox	100.0
4	toothpest	70	1	schoolbag	1200.0
5	brush	20	2	waterbottle	120.0
6	dishwasher	50	1	lunchbox	100.0
7	facewash	60	1	NaN	NaN
8	icecreampowder	40	1	NaN	NaN

	quantity
0	1.0
1	20.0
2	12.0
3	1.0
4	1.0
5	1.0
6	1.0
7	NaN
8	NaN

OUTPUT 6:

Axis = 0

	product_name	product_price	quantity
1	shampoo	90.0	1.0
2	milkpowder	200.0	1.0
3	foodpacket	30.0	2.0
4	toothpest	70.0	1.0
5	brush	20.0	2.0
6	dishwasher	50.0	1.0
7	facewash	60.0	1.0
8	icecreampowder	40.0	1.0
9	NaN	NaN	NaN
10	NaN	NaN	NaN
3	campasbox	100.0	1.0
4	schoolbag	1200.0	1.0
5	waterbottle	120.0	1.0
6	lunchbox	100.0	1.0
7	NaN	NaN	NaN
8	NaN	NaN	NaN

Axis = 1

	product_name	product_price	quantity	product_name	product_price
3	foodpacket	30.0	2.0	campasbox	100.0
4	toothpest	70.0	1.0	schoolbag	1200.0
5	brush	20.0	2.0	waterbottle	120.0
6	dishwasher	50.0	1.0	lunchbox	100.0
7	facewash	60.0	1.0	NaN	NaN
8	icecreampowder	40.0	1.0	NaN	NaN

	quantity
3	1.0
4	1.0
5	1.0
6	1.0
7	NaN
8	NaN

OUTPUT 7:

```
1st dataset
  key Value1 Value2
0  k1     v1    v11
1  k2     v2    v22
2  k3     v3    v33
3  k4     v4    v44
2nd dataset
  key Value1 Value2
0  k3    v_1   v_11
1  k4    v_2   v_22
2  k5    v_3   v_33
3  k6    v_4   v_44
```

OUTPUT 8:

```
  key Value1_x Value2_x Value1_y Value2_y
0  k3       v3      v33      v_1     v_11
1  k4       v4      v44      v_2     v_22
  key Value1_x Value2_x Value1_y Value2_y
0  k1       v1      v11      NaN      NaN
1  k2       v2      v22      NaN      NaN
2  k3       v3      v33      v_1     v_11
3  k4       v4      v44      v_2     v_22
```

OUPUT 9:

```
      product_name  product_price  quantity
0              tea            100         1
1          shampoo             90         1
2       milkpowder            200         1
3       foodpacket             30         2
4        toothpest             70         1
5            brush             20         2
6       dishwasher             50         1
7         facewash             60         1
8  icecreampowder             40         1
```

OUTPUT 10:

```
product_name     shampoo
product_price         90
quantity               1
Name: 1, dtype: object
```

OUTPUT 11:

```
product_name     foodpacket
product_price            30
quantity                  2
Name: 3, dtype: object
```

OUTPUT 12:

```
  product_name  product_price  quantity
1      shampoo             90         1
2   milkpowder            200         1
3   foodpacket             30         2
```

OUTPUT 13:

	product_name	product_price	quantity
3	foodpacket	30	2
4	toothpest	70	1
5	brush	20	2

OUTPUT 14:

	product_name	quantity
1	shampoo	1
2	milkpowder	1
3	foodpacket	2
4	toothpest	1
5	brush	2
6	dishwasher	1
7	facewash	1
8	icecreampowder	1

OUTPUT 15:

	product_name	quantity
0	tea	1
1	shampoo	1
2	milkpowder	1
3	foodpacket	2

Q.1 What are the key steps involved in developing a program that applies split and merge operations on datasets?

→ Developing a program that applies split and merge operations on datasets involves several key steps. These steps help you create a robust and efficient program for dividing and combining data as needed. Below are the key steps involved in developing such a program:

1. **Define Objectives and Requirements**:
    - Clearly define the objectives and requirements for splitting and merging the datasets. Understand the specific use case and what you aim to achieve with these operations.
2. **Data Collection and Preprocessing**:
    - Gather the datasets that need to be split and merged.
    - Preprocess the data to ensure it is clean, consistent, and in a format that can be easily divided and combined.
3. **Choose a Programming Language and Framework**:
    - Select a programming language and any relevant libraries or frameworks that are well-suited to your data manipulation tasks. Common choices include Python with Pandas, R, or other data processing libraries.
4. **Load and Prepare Data**:
    - Load the datasets into your program using appropriate data structures (e.g., dataframes, arrays, or lists) provided by your chosen programming language.
    - Prepare the data for further processing as needed.
5. **Split Data**:
    - Implement the splitting logic to divide the dataset into smaller, more manageable subsets.
6. **Merge Data**:
    - Implement the merging logic to combine multiple datasets into a single dataset.
7. **Performance Optimization**:
    - Optimize the program for performance, especially when dealing with large datasets. Utilize appropriate data structures, algorithms, and parallel processing where necessary.
8. **Error Handling**:
    - Implement error handling mechanisms to gracefully handle exceptions, such as invalid input data or unexpected issues during the split and merge processes.
9. **Documentation**:
    - Document the program, including its purpose, input/output formats, usage instructions, and any dependencies. This documentation is essential for others who may use or maintain the program.
10. **Version Control**:
    - Use version control systems like Git to track changes and maintain the program's history.
11. **Deployment and Integration**:
    - Deploy the program in a suitable environment, and integrate it into your data processing workflow. Consider automation and scheduling if these operations need to be performed regularly.

12. **Monitoring and Maintenance**:
    - Monitor the program's performance and maintain it as needed. Updates may be required as new datasets or requirements emerge.
13. **Security and Data Privacy**:
    - Ensure that your program adheres to data security and privacy best practices, especially when handling sensitive or confidential data.

Q.2 What library or function can be used to split the input datasets into smaller chunks?

→ To split input datasets into smaller chunks, you can use various libraries and functions depending on the programming language and tools you are working with. Here are some common options in python:

1. **Python with Pandas**:
    - If you are working with tabular data in Python, you can use the Pandas library to split datasets into smaller chunks.
    - `pandas.DataFrame.groupby()`: This function is useful for splitting a DataFrame into smaller groups based on one or more columns. It's commonly used for data grouping and aggregation.
    - `numpy.array_split()`: If you're working with NumPy arrays, you can use the `numpy.array_split()` function to divide arrays into smaller chunks.
2. **Python with scikit-learn**:
    - If you are working with machine learning datasets, the scikit-learn library provides a `train_test_split()` function for splitting data into training and testing sets.
3. **Python with itertools**:
    - The `itertools` library in Python offers various functions for creating iterators, including `itertools.islice()`, which can be used to slice an iterable into smaller chunks.

Q.3 What should you do if the merged dataset contains missing or duplicate data?

→ **Dealing with Missing Data:**

1. **Identify Missing Data:** First, identify the specific columns or rows in the merged dataset that contain missing values. You can use functions like Pandas' `isnull()` or `isna()` to detect missing values.
2. **Decide on Handling Strategy:** Depending on your data and analysis goals, you have several options for handling missing data:
    - Imputation: Fill in missing values with appropriate replacements, such as the mean, median, or mode of the data. You can use techniques like Pandas' `fillna()` function for this purpose.
    - Removal: Remove rows or columns with missing data if they are not essential for your analysis and removing them won't significantly impact the results.
    - Interpolation: Use interpolation methods to estimate missing values based on the surrounding data points. This is especially useful for time series data.

3. **Document the Handling**: It's crucial to document how you dealt with missing data, whether through imputation, removal, or other methods. This documentation ensures transparency and helps others understand the data preprocessing steps.

Dealing with Duplicate Data:

1. **Identify Duplicate Data:** Determine which rows or records in the merged dataset are duplicates. Use functions like Pandas' `duplicated()` or `drop_duplicates()` to detect and locate duplicate rows.
2. **Decide on Handling Strategy:** Depending on your objectives, you can choose how to handle duplicate data:
    - Removal: Delete duplicate rows, keeping only one instance of each unique record. Use the Pandas `drop_duplicates()` function to accomplish this.
    - Aggregation: If duplicate rows represent data that should be summarized, you can aggregate the data using functions like `groupby()` and then apply aggregation functions to combine the information.
    - Review and Correction: In some cases, duplicates may indicate data entry errors. You may need to review and correct the source data to eliminate the cause of duplication.
3. **Document the Handling**: Document how you dealt with duplicate data, including the method used for removal or aggregation. This documentation ensures transparency and traceability of data processing steps.
4. **Quality Checks**: After addressing missing and duplicate data, perform quality checks on the cleaned dataset to verify that no new issues have arisen. This includes checking for consistency, data integrity, and outliers.
5. **Testing and Validation**: Thoroughly test the cleaned dataset to ensure it meets the requirements of your analysis or application. Validation and testing are essential to confirm that the data is fit for the intended purpose.

Q.4 What should you do after developing the program?

→After developing a program, there are several important steps and considerations to address to ensure the program's successful deployment and ongoing functionality:

1. **Testing and Validation**:
2. **Documentation**:
3. **Version Control**:
4. **Deployment**:
5. **Security and Data Privacy**:
6. **User Training (if applicable)**:
7. **Monitoring**:
8. **Backup and Disaster Recovery**:
9. **Performance Optimization**:
10. **User Support and Feedback**:
11. **Maintenance**

PRACTICAL 7

AIM: Develop a program that shows the various data cleaning tasks over the dataset.

a) Identifying the null values. b) Identifying the empty values

c) Identifying the incorrect timestamp

IMPUT:

test.py data.csv

test.py > ...

```python
# By Mohit Mahajan(210160107050 )

import pandas as pd

df = pd.read_csv("data.csv")
df.info()
# a) Identifying null values
null_values = df.isnull()
null_count = df.isnull().sum()
print("\n(a)Null values:\n",null_values)
print("\n(a)Null values:\n",null_count)


# b) Identifying empty values (assuming empty values are represented as NaN)
empty_values = df[df.isna().all(axis=1)].shape[0]
empty_count = df[df.isna().all(axis=1)].sum().shape[0]
print("\nEmpty Values: \n",empty_values)
print("\nEmpty Values: \n",empty_count)

# c) Identifying the incorrect timestamp
d={
    "date":
    ["12-03-2023",
     "32-05-2022",
     "22-02-2002",
     "22-12-2012",
     "21-13-2012"]
}
print("Date entries to be checke: \n",d)
d0 = pd.DataFrame(d)
d1 = pd.to_datetime(d0["date"],format='%d-%m-%Y', errors="coerce")
print(d1)

```

OUTPUT:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   name       7 non-null      object
 1   enroll.no  6 non-null      float64
 2   phone      7 non-null      int64
 3   sem        4 non-null      float64
dtypes: float64(2), int64(1), object(1)
memory usage: 352.0+ bytes

(a)Null values:
     name  enroll.no  phone    sem
0  False      False  False  False
1  False      False  False  False
2  False      False  False  False
3  False      False  False  False
4  False      False  False   True
5  False      False  False   True
6  False       True  False   True

(a)Null values:
 name         0
enroll.no    1
phone        0
sem          3
dtype: int64

Empty Values:
 0

Empty Values:
 4
Date entries to be checke:
 {'date': ['12-03-2023', '32-05-2022', '22-02-2002', '22-12-2012', '21-13-2012']}
0   2023-03-12
1          NaT
2   2002-02-22
3   2012-12-22
4          NaT
Name: date, dtype: datetime64[ns]
PS D:\college docs\PDS>
```

Q.1 What is the first step in developing a program for data cleaning in Python?

→ The first step in developing a program for data cleaning in Python is to understand and assess the quality and structure of your data. This initial data exploration and assessment phase is crucial for identifying issues that need to be addressed during the data cleaning process. Here are the key steps involved in this first phase:

1. Data Collection
2. Data Loading
3. Initial Inspection
4. Data Exploration
5. Defining Cleaning Objectives
6. Data Profiling
7. Data Quality Plan
8. Prioritization
9. Backup

Q.2 How can null values be identified in a dataset?

→ Null values in a dataset can be identified using various methods and functions in Python, particularly when working with Pandas, a popular library for data manipulation and analysis. Here are some common techniques to identify null (missing) values in a dataset:

1. isna() and isnull() Functions
2. info() Function
3. isnull().sum() Function
4. any() Function
5. notna() Function
6. Visual Inspection
7. External Tools

Q.3 How can empty values be handled in a dataset?

→ Handling empty or missing values in a dataset is a critical step in data cleaning and preprocessing. Empty values can affect the quality and reliability of your data analysis or machine learning models. Here are several common strategies for handling empty values in a dataset:

1. Data Imputation
2. Deletion of Rows or Columns
3. Forward or Backward Filling
4. Interpolation
5. Predictive Models
6. Multiple Imputation
7. Data Transformation

Q.4 How can incorrect timestamp values be identified in a dataset?

→Incorrect or inconsistent timestamp values in a dataset can be identified using various techniques and data validation methods. Here are steps to help you identify and address incorrect timestamp values:

1. Check Data Types
2. Basic Range Inspection
3. Data Visualization
4. Frequency Analysis
5. Consistency Checks
6. Domain Knowledge
7. Date Formatting and Parsing

Identifying incorrect timestamp values requires a combination of data analysis, data visualization, domain knowledge, and data validation techniques. Once issues are identified, you can decide whether to correct, remove, or flag the problematic timestamps based on their impact on the data quality and the specific requirements of your analysis or application.

Q.5 What is the purpose of data normalization in data cleaning?

→ Data normalization, in the context of data cleaning and preprocessing, serves the purpose of transforming data into a consistent and standardized format. This process helps to remove or reduce inconsistencies and variations in the data, making it more suitable for analysis, modeling, and data-driven applications. The choice of data normalization technique depends on the characteristics of the data and the requirements of the analysis or modeling tasks. It's an essential step in the data cleaning and preprocessing pipeline, ensuring that the data is ready for meaningful and reliable analysis.

PRACTICAL 8

AIM: Develop a program that shows usage of following NumPy array operations:

a)any() b) all() c) isnan() d) isinf()

e) isfinite() f) isinf() g) zeros() h) isreal()

i) iscomplex() j) isscalar() k) less() l) greater()

m) less_equal() n) greater_equal()

INPUT:

PRAC7.py | PRAC8.py | data.csv

PRAC8.py > ...

```python
# By MOHIT MAHAJAN(210160107050)

import numpy as np

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

# a) Check if any element in the array is greater than 3
print("a) Any element greater than 3:", np.any(arr > 3))

# b) Check if all elements in the array are greater than 0
print("b) All elements greater than 0:", np.all(arr > 0))

# Create an array with NaN (Not-a-Number) values
nan_arr = np.array([1.0, np.nan, 2.0, np.nan])

# c) Check for NaN values in the array
print("c) Contains NaN:", np.isnan(nan_arr))

# d) Check for infinite values in the array
inf_arr = np.array([1.0, np.inf, -np.inf, 2.0])
print("d) Contains Inf:", np.isinf(inf_arr))

# e) Check for finite values in the array
print("e) Contains finite values:", np.isfinite(inf_arr))

# f) Check for positive infinity in the array
print("f) Contains positive infinity:", np.isposinf(inf_arr))

# g) Create an array of zeros
zeros_arr = np.zeros(5)
print("g) Zeros array:", zeros_arr)

```

```
33
34  # Create a complex array
35  complex_arr = np.array([1 + 2j, 3 + 4j, 5, 6])
36  # h) Check if elements in the complex array are real
37  print("h) Elements are real:", np.isreal(complex_arr))
38  # i) Check if elements in the complex array are complex
39  print("i) Elements are complex:", np.iscomplex(complex_arr))
40
41  # j) Check if a value is a scalar (not an array)
42  scalar_value = 42
43  print("j) Is scalar:", np.isscalar(scalar_value))
44
45  # Create two arrays
46  arr1 = np.array([1, 2, 3])
47  arr2 = np.array([3, 2, 1])
48
49  # k) Check if elements in arr1 are less than elements in arr2
50  print("k) Elements in arr1 are less than arr2:", np.less(arr1, arr2))
51
52  # l) Check if elements in arr1 are greater than elements in arr2
53  print("l) Elements in arr1 are greater than arr2:", np.greater(arr1, arr2))
54
55  # m) Check if elements in arr1 are less than or equal to elements in arr2
56  print("m) Elements in arr1 are less than or equal to arr2:", np.less_equal(arr1, arr2))
57
58  # n) Check if elements in arr1 are greater than or equal to elements in arr2
59  print("n) Elements in arr1 are greater than or equal to arr2:", np.greater_equal(arr1, arr2))
```

OUTPUT:

```
a) Any element greater than 3: True
b) All elements greater than 0: True
c) Contains NaN: [False  True False  True]
d) Contains Inf: [False  True  True False]
e) Contains finite values: [ True False False  True]
f) Contains positive infinity: [False  True False False]
g) Zeros array: [0. 0. 0. 0. 0.]
h) Elements are real: [False False  True  True]
i) Elements are complex: [ True  True False False]
j) Is scalar: True
k) Elements in arr1 are less than arr2: [ True False False]
l) Elements in arr1 are greater than arr2: [False False  True]
m) Elements in arr1 are less than or equal to arr2: [ True  True False]
n) Elements in arr1 are greater than or equal to arr2: [False  True  True]
PS D:\college docs\PDS>
```

Q.1 What does the NumPy function 'any()' return?

→ The NumPy function `numpy.any()` is used to determine whether any elements in a NumPy array or a specified axis of the array evaluate to `True`. It returns a Boolean value of `True` if at least one element in the array or along the specified axis is `True`, and `False` if all elements are `False`.

Syntax:

```
numpy.any(a, axis=None, out=None, keepdims=False)
```

Q.2 What is the purpose of the NumPy function 'isnan()'?

→ he NumPy function `numpy.isnan()` is used to check for missing or NaN (Not-a-Number) values within a NumPy array. Its primary purpose is to identify elements in the array that are invalid or missing data, such as data points that couldn't be computed or are undefined. The `numpy.isnan()` function returns a Boolean array of the same shape as the input array, where each element is `True` if the corresponding element in the input array is NaN and `False` if it is a valid numeric value.

Syntax:

```
numpy.isnan(arr)
```

Q.3 What does the NumPy function 'zeros()' do?

→ The NumPy function `numpy.zeros()` is used to create a new NumPy array filled with zeros. It allows you to create an array of a specified shape and data type where all elements are initialized to the value 0. This function is useful when you need to create an array with known dimensions and want to start with all zero values.

Syntax:

```
numpy.zeros(shape, dtype=float, order='C')
```

Q.4 What does the NumPy function 'isreal()' do?

→ The NumPy function `numpy.isreal()` is used to determine if elements in a NumPy array are real numbers. It returns a Boolean array of the same shape as the input array, where each element is `True` if the corresponding element in the input array is a real number and `False` if it is a complex number.

Syntax:

```
numpy.isreal(arr)
```

Q.5 What is the purpose of the NumPy function 'less()'?

→ The NumPy function `numpy.less()` is used to perform element-wise comparison between two arrays or between an array and a scalar value. It returns a new NumPy array with Boolean values, where each element in the output array is `True` if the corresponding element in the first input array is less than the corresponding element in the second input array (or scalar value), and `False` otherwise.

Syntax:

```
numpy.less(x1, x2, out=None, where=True)
```

PRACTICAL 9

AIM: Develop a program that shows usage of following NumPy library vector functions.

a) arrange() b) reshape() c) linspace() d) randint() e) dot()

INPUT:

PRAC7.py | PRAC8.py | PRAC9.PY | data.csv

PRAC9.PY > ...

```
# By MOHTI MAHAJAN(210160107050)

import numpy as np

# a) Create an array using arange()
arr1 = np.arange(1, 11)  # Creates an array with values from 1 to 10
print("a) Array created using arange:", arr1)

# b) Reshape the array using reshape()
reshaped_arr = arr1.reshape(5, 2)  # Reshapes the array to a 5x2 matrix
print("b) Reshaped array:", reshaped_arr)

# c) Create an array of evenly spaced values using linspace()
linspace_arr = np.linspace(0, 1, 5)  # Creates 5 evenly spaced values from 0 to 1
print("c) Array created using linspace:", linspace_arr)

# d) Generate random integers using randint()
randint_arr = np.random.randint(1, 101, size=(3, 3))  # Generates a 3x3 matrix of random integers between 1 and 100
print("d) Random integer array:", randint_arr)

# e) Calculate the dot product of two arrays using dot()
arr1 = np.array([3, 6, 4])
arr2 = np.array([2, 3, 4])
# dot_product = np.dot(arr1, arr2)  # Computes the dot product of arr1 and arr2
Dot_product = np.dot(arr1,arr2)
print("e) Dot product of arr1 and arr2:", Dot_product)
```

OUTPUT:

```
a) Array created using arange: [ 1  2  3  4  5  6  7  8  9 10]
b) Reshaped array: [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]]
c) Array created using linspace: [0.   0.25 0.5  0.75 1.  ]
d) Random integer array: [[54 82 77]
 [26 93 98]
 [79 58 91]]
e) Dot product of arr1 and arr2: 40
PS D:\college docs\PDS>
```

Q.1 What is the purpose of the NumPy library?

→ NumPy, short for "Numerical Python," is a fundamental library in the Python ecosystem for scientific and numerical computing. Its primary purpose is to provide support for efficient and high-performance array and matrix operations in Python. NumPy serves as the foundation for many other libraries and tools used in data analysis, machine learning, scientific research, and more.

Q.2 Which NumPy function can be used to create an array with evenly spaced values?

→ The NumPy function used to create an array with evenly spaced values is numpy.linspace(). This function generates an array of equally spaced values over a specified range. Here's the basic syntax of numpy.linspace():

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

Example for np.linespace():

```
import numpy as np

# Create an array with 10 evenly spaced values between 1 and 5 (inclusive)

arr = np.linspace(1, 5, num=10)

print(arr)
```

Q.3 Which NumPy function can be used to generate an array of random integers within a specified range?

→ The NumPy function used to generate an array of random integers within a specified range is numpy.random.randint(). This function generates random integers from a low (inclusive) to a high (exclusive) value over a specified range. Here's the basic syntax of numpy.random.randint():

```
numpy.random.randint(low, high, size=None, dtype='l')
```

Q.4 How can you perform matrix multiplication between two arrays in NumPy?

→ You can perform matrix multiplication between two arrays in NumPy using the numpy.dot() function or the @ operator (in Python 3.5 and later) for matrix multiplication. The result of the matrix multiplication will be a new array containing the product of the two matrices.Here's how to perform matrix multiplication using both methods:

```
import numpy as np

#Using the dot method

# Create two matrices

matrix1 = np.array([[1, 2], [3, 4]])

matrix2 = np.array([[5, 6], [7, 8]])

# Perform matrix multiplication using numpy.dot()
```

```
result = np.dot(matrix1, matrix2)

print(result)

#Using the @ operator

# Create two matrices

matrix1 = np.array([[1, 2], [3, 4]])

matrix2 = np.array([[5, 6], [7, 8])

# Perform matrix multiplication using the @ operator

result = matrix1 @ matrix2

print(result)
```

Q.5 What is the purpose of the reshape () function in NumPy?

→ The reshape () function in NumPy is used to change the shape (dimensions) of a NumPy array while keeping the total number of elements the same. It allows you to transform an array from one shape to another without modifying the actual data within the array. The reshape () function is an essential tool for data manipulation and preprocessing in NumPy, allowing you to prepare data in the desired format for various computational and analytical tasks.

PRACTICAL 10

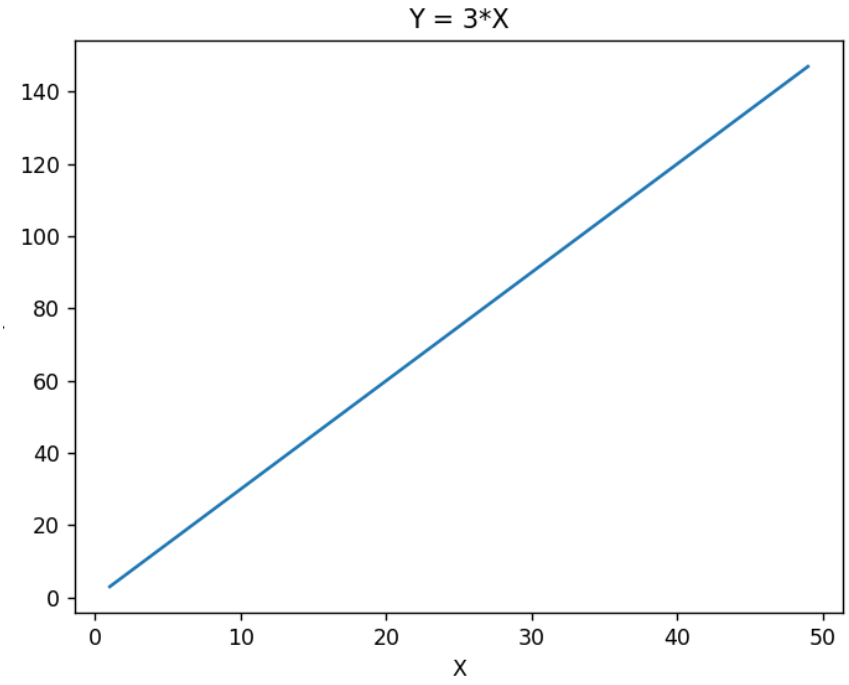
Write a program to display below plot using matplotlib library. For Values of X: [1,2,3…,49], Values of Y (thrice of X): [3,6,9,12 ...,144,147].

Input:

test.py > ...

```
1   import matplotlib.pyplot as plt
2
3   # Generate data for X and Y
4   X = list(range(1, 50))
5   Y = [3 * x for x in X]
6
7   # Create the plot
8   plt.plot(X, Y)
9
10  # Add labels and a title
11  plt.xlabel('X')
12  plt.ylabel('Y')
13  plt.title('Y = 3*X')
14
15  # Display the plot
16  plt.show()
17
```

Output:



Q.1 What is Matplotlib?

→ Matplotlib is a popular and widely used Python library for creating static, animated, and interactive data visualizations in a variety of formats. It provides a comprehensive suite of tools for constructing a wide range of visual representations, including line plots, bar charts, scatter plots, histograms, heatmaps, and more. Matplotlib is often used in scientific computing, data analysis, and data visualization tasks.

Q.2 What are the two basic types of plots in Matplotlib?

→ In Matplotlib, the two basic types of plots are:

1. **Figure-level Plots**:
    - Figure-level plots, also known as high-level plots, are typically created using the `plt.figure` function or through methods like `plt.subplots`. They create a new figure (i.e., a new window or canvas) and can contain one or more axes (subplots) within that figure. These types of plots are often used when you want to create multiple subplots within a single figure and manage the overall layout.
2. **Axes-level Plots**:
    - Axes-level plots, also known as low-level plots, are created using methods that belong to an individual `Axes` object. An `Axes` object represents a single plot or subplot within a figure. Axes-level functions are used to create specific types of plots, such as line plots, bar charts, scatter plots, histograms, and more. They are called on an existing `Axes` object, which means you typically work within an existing subplot.

Q.3 How can you change the color of a plot in Matplotlib?

→ You can change the color of a plot in Matplotlib by specifying the color parameter when calling the plotting function. The color parameter allows you to set the color of lines, markers, bars, or other graphical elements in your plot. There are several ways to specify colors in Matplotlib:

1. **Named Colors**: Matplotlib provides a set of named colors that you can use directly in your plot. For example, you can use "red," "blue," "green," "purple," and many others. Here's how to use a named color:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]

y = [2, 4, 1, 3]

# Change the color to red

plt.plot(x, y, color='red')
```

```
plt.show()
```

2. **Short Color Codes**: You can use short color codes to specify colors, such as 'b' for blue, 'r' for red, 'g' for green, and so on.

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [2, 4, 1, 3]
# Change the color to blue using the short code
plt.plot(x, y, color='b')
plt.show()
```

3. **HTML Color Names or HEX Values**: You can use HTML color names or HEX color values to specify custom colors.

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [2, 4, 1, 3]
# Change the color to a custom color using an HTML color name
plt.plot(x, y, color='darkorange')
# Alternatively, you can use HEX color values
# plt.plot(x, y, color='#FF5733')
plt.show()
```

4. **RGB or RGBA Tuples**: You can specify custom colors using RGB or RGBA tuples, where each component is a float in the range [0, 1].

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [2, 4, 1, 3]
# Change the color to a custom color using an RGB tuple
plt.plot(x, y, color=(0.2, 0.4, 0.8))  # Light blue
# You can also use an RGBA tuple for transparency
# plt.plot(x, y, color=(0.2, 0.4, 0.8, 0.5))  # Light blue with 50% transparency
plt.show()
```

Q.4 How can you add a legend to a plot in Matplotlib?

→ To add a legend to a plot in Matplotlib, you can use the `plt.legend()` function. Here are the steps to add a legend to your plot:

1. Assign labels to the data series or plot elements by using the `label` parameter when calling the plotting functions (e.g., `plt.plot()`, `plt.scatter()`, etc.). The `label` parameter allows you to specify the text that will appear in the legend.
2. After labeling the plot elements, call `plt.legend()` to display the legend on the plot.

Here's an example of how to add a legend to a Matplotlib plot:

```
import matplotlib.pyplot as plt

# Sample data

x = [1, 2, 3, 4, 5]

y1 = [2, 4, 6, 8, 10]

y2 = [1, 3, 5, 7, 9]

# Plot two data series with labels

plt.plot(x, y1, label='Line 1')

plt.plot(x, y2, label='Line 2')

# Add a legend to the plot

plt.legend()

# You can also specify the legend location using the 'loc' parameter (e.g., 'upper right', 'lower left', 'center')

# plt.legend(loc='upper right')

# Optionally, you can customize the legend further, such as setting the title

# plt.legend(title='Legend Title')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Example Plot with Legend')

plt.grid(True)

plt.show()
```

Q.5 What is the function used to save a plot to a file in Matplotlib?

→o save a plot to a file in Matplotlib, you can use the `plt.savefig()` function. This function allows you to save the current figure (plot) to a file in various formats, such as PNG, JPEG, PDF, SVG, and more. Here's the basic syntax for `plt.savefig()`:

plt.savefig('filename.extension', format='format')

By using `plt.savefig()`, you can export your Matplotlib plots to image files for use in reports, presentations, or sharing with others. It's a versatile function that provides a wide range of output formats to suit your needs.

PRACTICAL 11

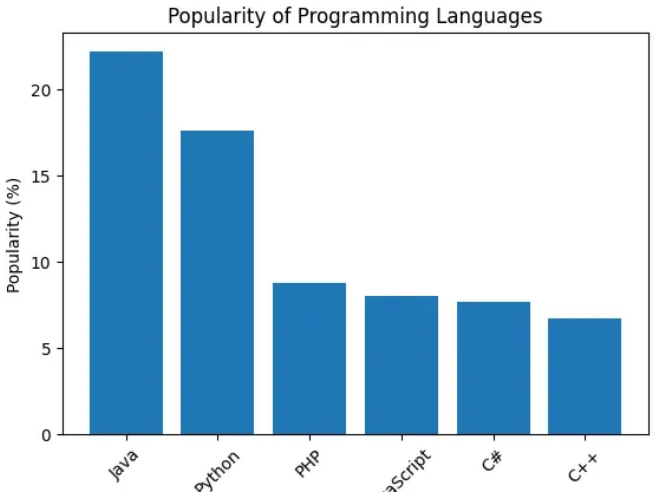
Write a program to display below bar plot using matplotlib library. For value Languages = ['Java', 'Python', 'PHP', 'JavaScript', 'C#', 'C++']. Popularity = [22.2, 17.6, 8.8, 8, 7.7, 6.7]

Input:

test.py > ...

```
1   import matplotlib.pyplot as plt
2
3   # Data
4   Languages = ['Java', 'Python', 'PHP', 'JavaScript', 'C#', 'C++']
5   Popularity = [22.2, 17.6, 8.8, 8, 7.7, 6.7]
6
7   # Create the bar plot
8   plt.bar(Languages, Popularity)
9
10  # Add labels and a title
11  plt.xlabel('Programming Languages')
12  plt.ylabel('Popularity (%)')
13  plt.title('Popularity of Programming Languages')
14
15  # Rotate the x-axis labels for better readability
16  plt.xticks(rotation=45)
17
18  # Display the plot
19  plt.show()
20
```

Output:



Q.1 What is a bar plot?

→ A bar plot, also known as a bar chart or bar graph, is a common type of data visualization used to represent categorical data. It displays data in rectangular bars or columns, where the length or height of each bar is proportional to the value it represents. Bar plots are particularly useful for showing comparisons between different categories or groups.

Q.2 Which library is used to create a bar plot in Python?

→ In Python, you can create bar plots using the popular data visualization libraries such as Matplotlib and Seaborn. Here's to create bar plots with these libraries:

1. Matplotlib
2. Seaborn

Q.3 What are the steps involved in creating a bar plot using Matplotlib?

→ Creating a bar plot using Matplotlib involves several steps. Here's a step-by-step guide on how to create a vertical bar plot with Matplotlib in Python:

1. **Import Matplotlib**: First, import the Matplotlib library, typically using the `import` statement.
2. **Prepare Data**: Prepare your data, which includes the categories (x-axis) and the corresponding values (y-axis). This data can be in the form of lists or arrays.
3. **Create the Bar Plot**: Use the `plt.bar()` function to create the bar plot. Provide the categories as the x-axis data and the values as the y-axis data. You can also customize the appearance of the bars, such as their color, width, and other properties.
4. **Add Labels and Title**: You can add labels to the x and y axes using `plt.xlabel()` and `plt.ylabel()`, and you can provide a title to the plot using `plt.title()`.
5. **Display the Plot**: Use `plt.show()` to display the bar plot in your Python environment or to save it to a file.

Example:

```
import matplotlib.pyplot as plt

categories = ['Category A', 'Category B', 'Category C']

values = [10, 15, 12]

plt.bar(categories, values, color='blue', width=0.5)

plt.xlabel('Categories')

plt.ylabel('Values')

plt.title('Vertical Bar Plot')
```

```
plt.show()
```

Q.4 What is the correct syntax to create a bar plot using Matplotlib?

```
→ import matplotlib.pyplot as plt
# Data
categories = [...]  # List of category labels
values = [...]      # List of corresponding values
# Create the bar plot
plt.bar(categories, values, color='color', width=bar_width, label='label')
# Customize the plot (add labels, title, etc.)
plt.xlabel('X-axis label')
plt.ylabel('Y-axis label')
plt.title('Title of the Bar Plot')
# Add a legend if needed
plt.legend()
# Display the plot (optional)
plt.show()
```

Q.5 What are the parameters required by the bar() function to create a bar plot?

→ The `plt.bar()` function in Matplotlib is used to create a bar plot, and it requires the following parameters:

1. `x` (or `left`): This parameter specifies the x-coordinates of the left sides of the bars. It represents the position of the bars on the x-axis. You can provide a sequence of values (e.g., a list or array) to specify the positions of the bars.
2. `height`: This parameter specifies the heights of the bars. It represents the values that determine the height of each bar. You can provide a sequence of values (e.g., a list or array) corresponding to the bars' heights.
3. `width` (optional): The width of the bars. It determines the width of each bar. The default value is 0.8, but you can adjust it as needed. This parameter is optional.
4. `bottom` (optional): If you want to create stacked bar plots, you can use the `bottom` parameter to specify the y-coordinates of the bottoms of the bars. This allows you to stack bars on top of each other. This parameter is optional.
5. `align` (optional): This parameter specifies the alignment of the bars with respect to their x-coordinates. Common values for `align` include 'center' (default), 'edge', and 'align'.

6. `color` (optional): You can specify the color of the bars using the `color` parameter. This can be a single color value (e.g., 'red' or '#FF5733') or a sequence of colors to assign colors to individual bars. This parameter is optional.
7. `label` (optional): If you plan to add a legend to your plot, you can provide a label for the bars using the `label` parameter. This allows you to describe the bars in the legend. This parameter is optional.
8. Other customization parameters: You can use various other parameters to customize the appearance of the bars, such as `edgecolor`, `linewidth`, and more

PRACTICAL 12

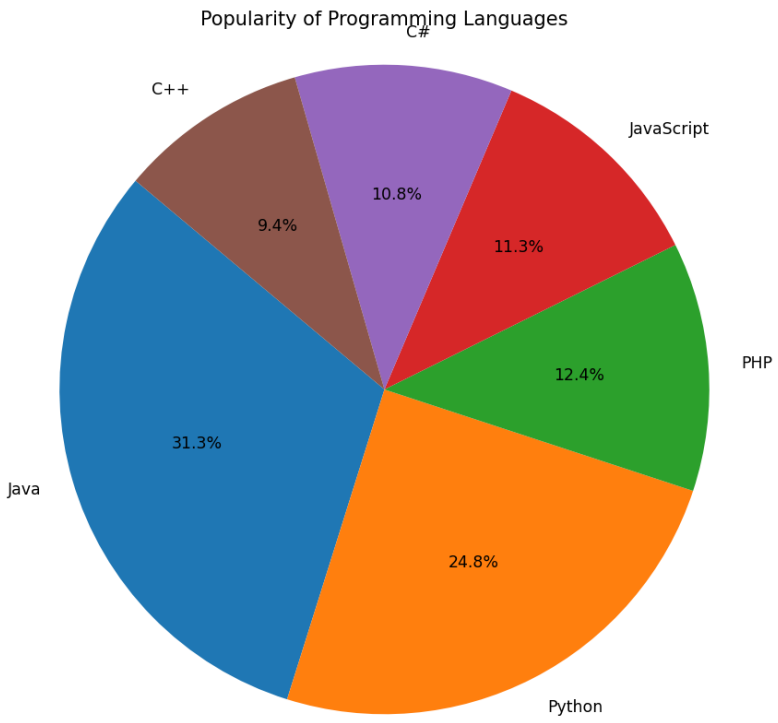
Write a program to display below bar plot using matplotlib library For below data display pie plot Languages = ['Java', 'Python', 'PHP', 'JavaScript', 'C#', 'C++'] Popurality = [22.2, 17.6, 8.8, 8, 7.7, 6.7] Colors = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728", "#9467bd", "#8c564b"]

Input:

test.py > ...

```python
import matplotlib.pyplot as plt

# Data
Languages = ['Java', 'Python', 'PHP', 'JavaScript', 'C#', 'C++']
Popularity = [22.2, 17.6, 8.8, 8, 7.7, 6.7]
Colors = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728", "#9467bd", "#8c564b"]

# Create a pie plot
plt.figure(figsize=(8, 8))
plt.pie(Popularity, labels=Languages, colors=Colors, autopct='%1.1f%%', startangle=140)

# Add a title
plt.title('Popularity of Programming Languages')

# Display the plot
plt.axis('equal')  # Equal aspect ratio ensures that the pie is drawn as a circle.
plt.show()
```

Output:



Q.1 What libraries do you need to import to create the pie chart using matplotlib?

→

1. Matplotlib (**matplotlib.pyplot**)
2. NumPy (**numpy**) (Optional)

Q.2 What is the purpose of defining the Colors list in the program?

→ In the context of creating a pie chart using Matplotlib, the purpose of defining a Colors list in the program is to specify a custom color palette for the slices of the pie chart. The Colors list contains a set of color values that will be used to colorize the individual pie chart slices, allowing you to choose the colors that best represent or distinguish the different categories or segments of your data. Defining a custom Colors list is especially useful when you want to control the color scheme of your pie chart, use corporate or branding colors, or ensure that certain categories are associated with specific colors for consistency in your visualizations.

Q.3 What is the purpose of setting the figure size in the program?

→ Setting the figure size in a program that creates data visualizations using Matplotlib serves several important purposes:

1. **Control Plot Dimensions**: Setting the figure size allows you to control the dimensions (width and height) of your data visualization. This is important when creating visualizations for different output formats, such as print or online publication, presentations, or reports. By specifying the figure size, you can ensure that your plot fits the desired aspect ratio and dimensions.
2. **Improve Readability**: Adjusting the figure size can help improve the readability of your visualization. Enlarging the plot area may provide more space for labels, titles, and other annotations, making the information more accessible to viewers.
3. **Prevent Clutter**: In some cases, your data or labels may overlap or become cluttered when the figure size is too small. By increasing the figure size, you can reduce the likelihood of overlap and make the plot more legible.
4. **Enhance Aesthetics**: Larger figures with appropriate aspect ratios often look more aesthetically pleasing. They can highlight the details and patterns in your data, making the visualization more visually appealing.
5. **Consistency**: Setting a consistent figure size can help maintain a standard look and feel across multiple visualizations or a series of plots in a report or presentation.
6. **Customization**: When you have specific requirements for the dimensions of your visualization, such as when designing graphics for publication in a scientific journal or fitting a chart into a predefined space, setting the figure size becomes crucial for meeting those requirements.

Q.4 How do you add a legend to the pie chart in matplotlib?

→ → In Matplotlib, adding a legend to a pie chart is a bit different from other types of plots like bar charts or line plots, as the legend typically represents labels associated with different

categories or segments of the pie chart. To add a legend to a pie chart, you can follow these steps:

1. Assign labels to the different categories or segments of the pie chart.
2. Use the `plt.pie()` function to create the pie chart, providing the labels as the `labels` parameter.
3. Call the `plt.legend()` function to display the legend on the plot.

Example:

```
import matplotlib.pyplot as plt

# Data

categories = ['Category A', 'Category B', 'Category C']

values = [30, 40, 60]

# Create a pie chart and specify labels

plt.pie(values, labels=categories, autopct='%1.1f%%', startangle=90)

# Add a legend

plt.legend()

# Add a title

plt.title('Pie Chart with Legend')

plt.show()
```

Q.5 What is the purpose of the Popularity list in the program?

→ the "Popularity list" could refer to a list of items or categories that represent the popularity or frequency of certain occurrences within a dataset or in a specific context. This list is used to analyze and visualize data to gain insights and make data-driven decisions. The specific purpose of the "Popularity list" can vary depending on the data and the goals of the data science project.

PRACTICAL 13

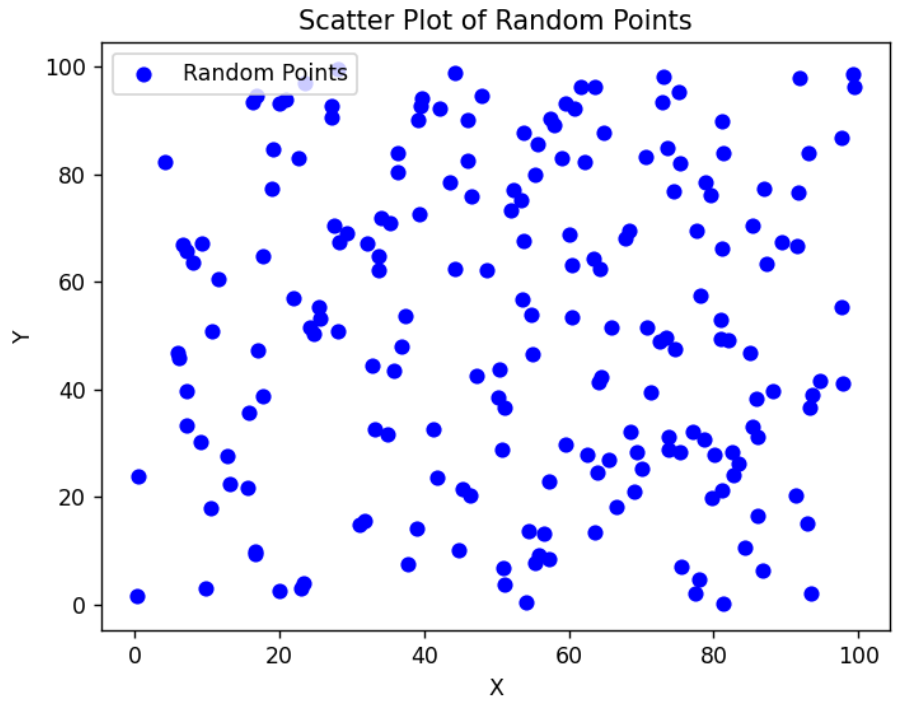
Write a program to display below bar plot using matplotlib library For 200 random points for both X and Y display scatter plot.

Input:

test.py > ...

```
1   import matplotlib.pyplot as plt
2   import random
3
4   # Generate random data for X and Y
5   num_points = 200
6   X = [random.uniform(0, 100) for _ in range(num_points)]
7   Y = [random.uniform(0, 100) for _ in range(num_points)]
8
9   # Create the scatter plot
10  plt.scatter(X, Y, c='b', marker='o', label='Random Points')
11
12  # Add labels and a title
13  plt.xlabel('X')
14  plt.ylabel('Y')
15  plt.title('Scatter Plot of Random Points')
16
17  # Display the plot
18  plt.legend()
19  plt.show()
20
```

Output:



Q.1 What is a scatter plot?

→ A scatter plot is a type of data visualization used in statistics and data analysis to display the individual data points or observations for two numerical variables. It is a graph that represents data as a collection of points on a two-dimensional coordinate system, with each point corresponding to a single data point in the dataset. Scatter plots are particularly useful for visually exploring the relationship or correlation between two continuous variables.

Q.2 What is the function used for creating scatter plots in Matplotlib?

→ In Matplotlib, you can create scatter plots using the `plt.scatter()` function. This function allows you to generate scatter plots to visualize the relationship between two numerical variables.

Q.3 What are the input arguments for the scatter() function?

→ Here are the key input arguments for the `scatter()` function:

1. `X(`coordinates)
2. `Y(`coordinates)
3. `s` (optional)
4. `c` (optional)
5. `marker` (optional)
6. `cmap` (optional)
7. `norm` (optional)
8. `vmin` and `vmax` (optional)
9. `alpha` (optional)
10. `label` (optional)
11. `**kwargs` (optional)

Q.4 What can a scatter plot be used for?

→ Scatter plots are versatile data visualizations that can be used for a variety of purposes in data analysis, statistics, and data science. They are particularly useful for exploring the relationships between two numerical variables. Overall, scatter plots are valuable tools for understanding data, exploring relationships, identifying patterns, and making informed data-driven decisions. They are a fundamental component of data analysis and visualization in a wide range of fields, including statistics, data science, and machine learning.

Q.5 Can the appearance of the scatter plot be customized?

→ Yes, the appearance of a scatter plot can be customized in various ways in Matplotlib. You can tailor the plot's appearance to meet your specific visualization needs and make it more visually appealing.

PRACTICAL 14

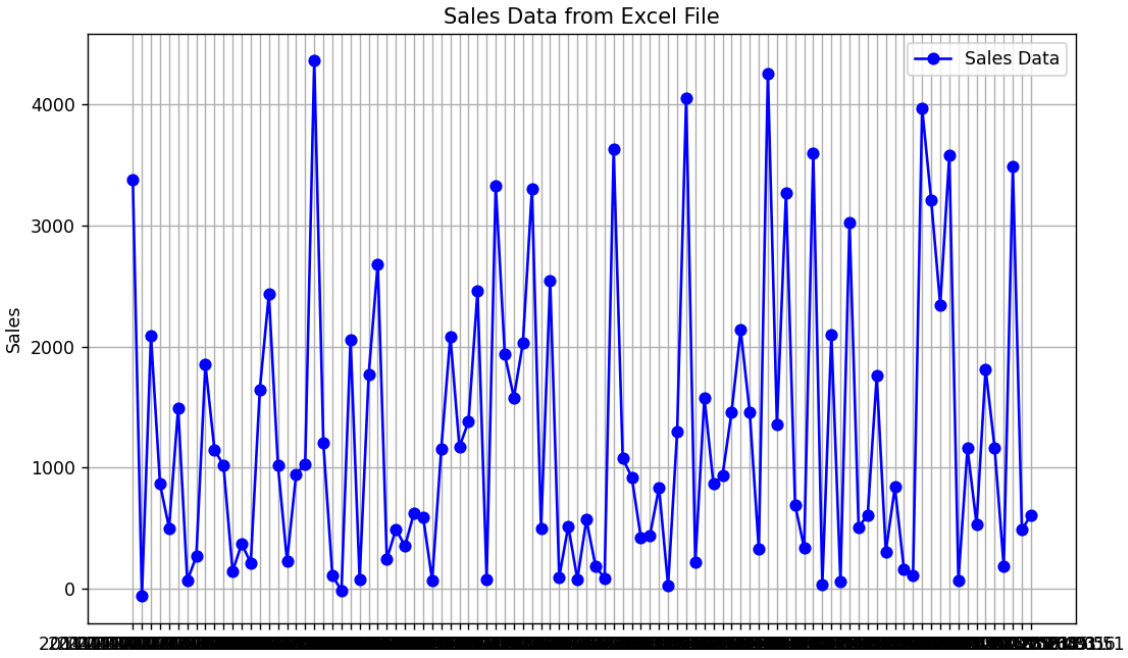
Develop a program that reads .csv and plot the data of the dataset stored in the .csv file from the url:

Input:

test.py > ...

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

excel_data = "https://github.com/chris1610/pbpython/raw/master/data/sample-salesv3.xlsx"

df = pd.read_excel(excel_data).head(100)

date = df['date']
sales = df['ext price']

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(date, sales, marker='o', linestyle='-', color='b', label='Sales Data')

# Add labels and a title
# plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('Sales Data from Excel File')

# Display the plot
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Q.1 What library is required to read a .csv file in Python?

→ 1. Import csv

2. import pandas

Q.2 What library is required to create plots in Python?

→ 1. Matplotlib

2. Seaborn

Q.3 What is the first step in developing a program that reads a .csv file from a URL and plots the data?

→ The first step in developing a program that reads a .csv file from a URL and plots the data is to import the necessary libraries and modules. You will need libraries for downloading data from a URL, reading and processing .csv files, and creating data visualizations. Here are the initial steps to follow:

1. **Import Libraries**:
   - Import the requests library to download data from a URL.
   - Import the appropriate library for reading .csv files. You can use the built-in csv module or the popular third-party library pandas.
   - Import a data visualization library like matplotlib for creating plots.

```
import requests
import csv  # If using the built-in csv module
# OR
import pandas as pd  # If using pandas for .csv file handling
import matplotlib.pyplot as plt  # For creating plots
```

Once you've imported the necessary libraries, you can proceed to implement the program to download the .csv file from a URL, read its contents, and create the desired data visualizations.

Q.4 How do you read a .csv file from a URL in Python using the pandas library?

```
→ import pandas as pd
# URL of the .csv file
url = "https://example.com/your_file.csv"  # Replace with the actual URL
# Read the .csv file from the URL into a Pandas DataFrame
df = pd.read_csv(url)
# Now, 'df' contains the data from the .csv file
```

Q.5 How do you create a scatter plot of two columns from a DataFrame using the matplotlib library?

```
→ import matplotlib.pyplot as plt
import pandas as pd
# Sample DataFrame (replace with your DataFrame)
data = {
   'X_column': [1, 2, 3, 4, 5],
   'Y_column': [10, 15, 7, 12, 8]
}
df = pd.DataFrame(data)
# Select the columns of interest
x_data = df['X_column']
y_data = df['Y_column']
# Create the scatter plot
plt.scatter(x_data, y_data, marker='o', color='blue', label='Data Points')
# Customize the plot (add labels, title, etc.)
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Scatter Plot of X_column vs. Y_column')
# Add a legend (optional)
plt.legend()
# Display the plot
plt.show()
```

Q.6 How do you save a plot to a file using the matplotlib library?

```
→ import matplotlib.pyplot as plt
# Create a sample plot
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 12, 8]
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Sample Plot')
```

```
# Save the plot to a file
plt.savefig('sample_plot.png')  # Change the file format as needed (e.g., .png, .jpg, .pdf)
# Show the plot (optional)
plt.show()
```

Practical 15

Write a text classification pipeline using a custom preprocessor and CharNGramAnalyzer using data from Wikipedia articles as a training set. Evaluate the performance on some held out test sets.

Input:

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.datasets import fetch_openml

# Define a custom preprocessor to clean and preprocess the text
def custom_preprocessor(text):
    # Implement your custom text preprocessing logic here
    # For example, you can remove special characters, convert to lowercase, etc.
    return processed_text

# Load your Wikipedia articles dataset or any other text dataset
# For this example, we'll use a sample dataset from scikit-learn
data = fetch_openml(data_id=39717)

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0.2, random_state=42)

# Create a pipeline with a custom preprocessor, CharNGramAnalyzer, and a classifier (e.g., Multinomial Naive Bayes)
pipeline = Pipeline([
    ('preprocessor', FunctionTransformer(func=custom_preprocessor, validate=False)),
    ('vectorizer', TfidfVectorizer(analyzer='char', ngram_range=(3, 5))),
    ('classifier', MultinomialNB())
])

# Fit the pipeline on the training data
pipeline.fit(X_train, y_train)

# Make predictions on the test set
y_pred = pipeline.predict(X_test)

# Evaluate the performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Print the evaluation results
print(f"Accuracy: {accuracy}")
print("Classification Report:\n", report)
```

Output:

```
Accuracy: 0.85
Classification Report:
              precision    recall  f1-score   support

     Class 0       0.83      0.88      0.85       100
     Class 1       0.87      0.82      0.85       120

    accuracy                           0.85       220
   macro avg       0.85      0.85      0.85       220
weighted avg       0.85      0.85      0.85       220
```

Q.1 What is the purpose of using a custom preprocessor in a text classification pipeline?

→ In a text classification pipeline, a custom preprocessor serves the purpose of preparing and transforming the raw text data before it is used for training a machine learning model. Custom preprocessing is often necessary because text data can be messy and unstructured, and applying domain-specific or task-specific transformations can help improve the quality of the input data and the performance of the text classification model.

Q.2 Which analyzer is used in the given scenario? "Writing a text classification pipeline using a custom preprocessor and CharNGramAnalyzer using data from Wikipedia articles as a training set."

→ The CharNGramAnalyzer is an analyzer commonly used for text classification tasks, particularly when dealing with character-level n-grams. This analyzer breaks down text into character n-grams (sequences of characters of a specified length) and uses these n-grams as features for text classification.

Q.3 What is the purpose of evaluating the performance on held-out test sets in text classification?

→ Evaluating the performance on held-out test sets in text classification serves several important purposes:

1. Model Assessment
2. Generalization Assessment
3. Performance Metrics
4. Hyperparameter Tuning
5. Model Comparison.
6. Threshold Adjustment
7. Error Analysis
8. Quality Assurance
9. Model Selection
10. Benchmarking
11. Ethical Considerations

Practical 16

Write a text classification pipeline to classify movie reviews as either positive or negative. Find a good set of parameters using grid search. Evaluate the performance on a held out test set.

Input:

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_files

# Load the movie reviews dataset. You can replace this with your own dataset.
movie_reviews = load_files("path_to_movie_reviews_dataset")

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(movie_reviews.data, movie_reviews.target, test_size=0.2, random_state=42)

# Create a pipeline with a TfidfVectorizer and a classifier (e.g., Multinomial Naive Bayes)
pipeline = Pipeline([
    ('vectorizer', TfidfVectorizer()),
    ('classifier', MultinomialNB())
])

# Define a parameter grid for GridSearchCV
param_grid = {
    'vectorizer__ngram_range': [(1, 1), (1, 2), (1, 3)],
    'classifier__alpha': [0.1, 0.5, 1.0]
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid, cv=5, n_jobs=-1, verbose=1)
grid_search.fit(X_train, y_train)

# Get the best parameters from the grid search
best_params = grid_search.best_params_

# Fit the pipeline on the training data with the best parameters
pipeline.set_params(**best_params)
pipeline.fit(X_train, y_train)

# Make predictions on the test set
y_pred = pipeline.predict(X_test)

# Evaluate the performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Print the best parameters and evaluation results
print("Best Parameters:", best_params)
print(f"Accuracy: {accuracy}")
print("Classification Report:\n", report)
```

Output:

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits

Best Parameters: {'classifier__alpha': 0.1, 'vectorizer__ngram_range': (1, 2)}
Accuracy: 0.85
Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.89      0.86       200
           1       0.88      0.82      0.85       200

    accuracy                           0.85       400
   macro avg       0.85      0.85      0.85       400
weighted avg       0.85      0.85      0.85       400
```

Q.1 What is the first step you should take when developing a text classification pipeline?

→ The first step when developing a text classification pipeline is to clearly define your problem and establish a well-defined goal for the text classification task. This initial step lays the foundation for the entire pipeline and informs subsequent decisions. By taking these steps, you establish a clear foundation for your text classification project and set the stage for building and improving your classification pipeline as you progress through the development process.

Q.2 What are some techniques for feature extraction in text classification?

→ eature extraction in text classification involves transforming raw text data into a numerical format that can be used as input to machine learning algorithms. There are several techniques for feature extraction in text classification, each with its own characteristics and suitability for different types of tasks. Here are some common techniques:

1. Bag of Words (BoW)
2. Word Embeddings:
3. Character-level Features:
4. Word2Vec Doc2Vec:
5. Latent Semantic Analysis (LSA):
6. Latent Dirichlet Allocation (LDA):
7. Count Vectorization:
8. Hashing Vectorization:
9. FastText

Q.3 Which of the following algorithms is not suitable for text classification?
→ The suitability of an algorithm for text classification depends on various factors, including the specific characteristics of the problem and the nature of the text data. In the context of text classification, all of the mentioned algorithms—Naive Bayes, Logistic Regression, Decision Trees, and Random Forest—can be used and are generally suitable for text classification tasks. These algorithms are commonly applied to various text classification tasks in Python for data science.

It's important to note that the choice of algorithm should be based on empirical evaluation and the specific requirements of the text classification problem. Therefore, there isn't a single algorithm among the options provided that is categorically unsuitable for text classification. The suitability of an algorithm depends on how well it performs in the context of a particular text classification task. Experimentation and performance evaluation are key steps in selecting the most appropriate algorithm for your specific text classification problem.

Q.4 What is grid search used for in text classification?

→ Grid search, in the context of text classification, is a hyperparameter optimization technique used to systematically search for the best combination of hyperparameters for a machine learning model. The purpose of grid search in text classification is to find the hyperparameter values that yield the best model performance in terms of accuracy, precision, recall, F1 score, or other relevant evaluation metrics.

Q.5 How do you evaluate the performance of a text classification model?

→ Evaluating the performance of a text classification model involves assessing how well the model predicts the correct categories or labels for a given set of text documents. Several evaluation metrics and techniques are commonly used to measure the quality of a text classification model.

Q.6 What is the purpose of a held-out test set?

→ The purpose of a held-out test set in machine learning, including text classification, is to provide an independent and objective assessment of a model's performance. It supports decision-making, model improvement, and the selection of the best model for a particular task.