CMR
INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test III – April 2025

Sub:	Data Analytics using Python							Sub Code:	22MCA31
Date:	07/04/2025	Duration:	90 mins	Max Marks:	50	Sem:	III	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

	PART I	MARKS	OBE	
			CO	RBT
1	What is web scraping? Explain the tools and techniques used for web scraping. **OR**	3+7	CO1	L3
2	Explain the methods of beautiful soup with program examples for each.	10	CO3	L4
	PART II			
3	Demonstrate post() and how post() works in web scraping with a program. **OR**	10	CO3	L4
4	Explain the following in time series with program examples: a) Frequency and Date offsets b) Date ranges	5+5	CO2	L3
	PART III			
5	Discuss about conversion between strings and date time with program examples. **OR**	10	CO2	L2
6	Explain the different CSS selectors used in web scraping.	10	CO3	L4
	PART IV	5+5		
7	Explain the following with program examples: a) Web browser module b) requests module **OR**		CO3	L3
8	What is resampling?Explain the types of resampling with a program example	3+7	CO3	L3
	PART V			
9	Write a Python program to demonstrate the generation of linear regression models. **OR**	10	CO4	L4
10	Write a Python program to demonstrate Data Visualization using Seaborn	10	CO4	L4

1. Web scraping:

Web scraping in Python is the process of automatically extracting data from websites. It involves sending HTTP requests to web pages, parsing the HTML content, and retrieving the desired information. Web scraping is widely used for data collection in various domains like price comparison, news aggregation, sentiment analysis, and more.

□ Tools Used for Web Scraping in Python

Here are some of the most popular libraries and tools used:

1. **Requests**
   - Used to send HTTP/HTTPS requests to fetch web page content.
   - Simple and easy to use.
   - Example:
   - import requests
   - response = requests.get("https://example.com")
   - html = response.text
2. **BeautifulSoup**
   - A parsing library to extract data from HTML and XML documents.
   - Allows searching the HTML tree by tags, attributes, text, etc.
   - Example:
   - from bs4 import BeautifulSoup
   - soup = BeautifulSoup(html, 'html.parser')
   - title = soup.title.text
3. **lxml**
   - High-performance library for parsing HTML/XML using XPath.
   - Faster than BeautifulSoup for large documents.
   - Example:
   - from lxml import html
   - tree = html.fromstring(response.content)
   - titles = tree.xpath('//h1/text()')
4. **Selenium**
   - Used to scrape dynamic content that relies on JavaScript.
   - Simulates a real browser, useful when data is loaded asynchronously.
   - Example:
   - from selenium import webdriver
   - driver = webdriver.Chrome()
   - driver.get("https://example.com")
   - page = driver.page_source

□ Techniques in Web Scraping

1. **Understanding the website structure**
   - Use browser developer tools (Inspect Element) to analyze the HTML structure.
   - Identify tags, classes, or IDs where data is located.
2. **Fetching the web page**
   - Use requests or selenium to load the page content.
3. **Parsing the HTML content**

- Use BeautifulSoup or lxml to parse and navigate through the HTML.

4. **Extracting data**
   - Use selectors or XPath to extract relevant data.
5. **Storing data**
   - Save scraped data into formats like CSV, JSON, or databases.
6. **Handling dynamic content**
   - Use Selenium or Scrapy with Splash to handle JavaScript-rendered pages.

2. Methods of beautiful soup

1. soup.title

Gets the <title> tag.

```
print(soup.title)        # <title>Example Page</title>
print(soup.title.text)   # Example Page
```

2. soup.find()

Finds the **first occurrence** of a tag.

```
first_p = soup.find('p')
print(first_p.text)      # This is a paragraph.
```

You can also search by attributes:

```
print(soup.find('a', id='link2').text)  # Second link
```

3. soup.find_all()

Finds **all matching tags**.

```
all_paragraphs = soup.find_all('p')
for p in all_paragraphs:
    print(p.text)
```

Output:

```
This is a paragraph.
This is another paragraph.
```

4. soup.select()

Use **CSS selectors** to find elements.

```
links = soup.select('a')
for link in links:
    print(link['href'])
```

5. .get()

Used to get an attribute value of a tag.

```
a_tag = soup.find('a')
print(a_tag.get('href'))  # https://example.com/page1
```

6. .text or .get_text()

Extracts only the **text** from a tag (no HTML).

```
print(soup.h1.text)       # Welcome to Web Scraping
```

7. .attrs

Returns a dictionary of all attributes of a tag.

```
print(soup.a.attrs)
# {'href': 'https://example.com/page1', 'id': 'link1'}
```

8. .parent / .children / .descendants

Used for **navigating the DOM tree**.

```
print(soup.h1.parent.name)  # body

# All direct children of body
for child in soup.body.children:
    print(child.name)

# All descendants (deep search)
for desc in soup.body.descendants:
    if desc.name:
        print(desc.name)
```

9. .find_next() and .find_previous()

Used for finding tags relative to the current tag.

```
p_tag = soup.find('p')
next_link = p_tag.find_next('a')
print(next_link.text)  # First link
```

10. .string

Gets the direct string inside a tag.

```
print(soup.title.string)  # Example Page
```

3. post()

```
import requests

# Step 1: Define the target URL for the POST request
url = 'https://httpbin.org/post'

# Step 2: Create the payload (form data you want to send)
form_data = {
    'username': 'testuser',
    'password': 'mypassword123'
}

# Step 3: Send a POST request with form data
response = requests.post(url, data=form_data)

# Step 4: Check if the request was successful
if response.status_code == 200:
    print("POST request successful!\n")

    # Step 5: Print the response data (JSON format)
    result = response.json()
    print("Form data sent:")
    print(result['form'])  # Shows the data we posted
else:
    print("Failed to send POST request. Status code:", response.status_code)
```

How This Works

- requests.post() sends form data to the server.
- data=form_data attaches key-value pairs like a login form.
- The server at httpbin.org/post responds with a JSON object showing what was sent.

Output (example)

```
POST request successful!

Form data sent:
{'username': 'testuser', 'password': 'mypassword123'}
```

Use in Web Scraping

In real-world scraping, you use post() to:

- Submit login forms
- Send search queries to a server
- Interact with websites that use forms or APIs

4. a) **Frequency and Date Offsets**

 What is Frequency?

- **Frequency** refers to how often data points occur in a time series (e.g., daily, hourly, monthly).
- In Pandas, you set frequency using freq parameter in date_range() or when resampling.

 What is DateOffset?

- **DateOffset** is used to define custom or complex date shifting (like "add 3 business days").

☑Example: Frequency and Date Offsets

```
import pandas as pd

# Creating a date range with daily frequency
date_range = pd.date_range(start='2025-01-01', periods=5, freq='D')
print("Daily Frequency:\n", date_range)

# Frequency with business days
business_days = pd.date_range(start='2025-01-01', periods=5, freq='B')
print("\nBusiness Days:\n", business_days)

# Using DateOffset: Add 2 months to a date
from pandas.tseries.offsets import DateOffset

date = pd.Timestamp('2025-01-01')
new_date = date + DateOffset(months=2)
print("\nOriginal Date:", date)
print("Date + 2 months:", new_date)
```

 Common Frequencies

Code	Description
D	Daily
B	Business day
W	Weekly
M	Month end
MS	Month start
H	Hourly
T or min	Minutely
S	Secondly

 b) Date Ranges

 What is a Date Range?

- A sequence of dates generated by pd.date_range().
- Useful for creating time indexes for time series.

☑Example: Creating Date Ranges

```
# Generate 10 days starting from 2025-01-01
date_range = pd.date_range(start='2025-01-01', periods=10, freq='D')
print("Date Range:\n", date_range)

# Monthly frequency, 6 periods
monthly_range = pd.date_range(start='2025-01-01', periods=6, freq='M')
print("\nMonthly Range:\n", monthly_range)

# Custom time range with hourly frequency
hourly_range = pd.date_range(start='2025-01-01 08:00', end='2025-01-01 12:00', freq='H')
print("\nHourly Range:\n", hourly_range)
```

5. String to **datetime and datetime to string conversions**

Use **pd.to_datetime()** or the datetime.strptime() method to convert a string into a datetime object.

☑Example using pandas.to_datetime()

```
import pandas as pd

date_str = '2025-04-07'
dt = pd.to_datetime(date_str)
print("Converted to datetime:", dt)
```

☑Example using datetime.strptime()

```
from datetime import datetime

date_str = '07/04/2025'
dt_obj = datetime.strptime(date_str, '%d/%m/%Y')
print("Converted to datetime:", dt_obj)
```

□ Common Format Codes

Format Code	Meaning	Example
%Y	Year (4-digit)	2025
%y	Year (2-digit)	25
%m	Month (01–12)	04
%d	Day (01–31)	07
%H	Hour (00–23)	14
%M	Minute (00–59)	30
%S	Second (00–59)	45

□ 2. datetime to String

Use .strftime() to format a datetime object as a string.

☑Example

```
from datetime import datetime

now = datetime.now()
```

```
formatted_date = now.strftime('%d-%b-%Y %H:%M:%S')
print("Formatted date string:", formatted_date)
```

□ 3. List of Dates or Series Conversion

☑String series to datetime (like a date column in CSV)

```
import pandas as pd

date_series = pd.Series(['2025-01-01', '2025-02-01', '2025-03-01'])
converted = pd.to_datetime(date_series)
print("Converted Series:\n", converted)
```

6. CSS Selectors

Tag Selector

Selects all elements with a specific tag.

```
soup.select('p')  # selects all <p> tags
```

□ 2. ID Selector

Selects a tag by its unique id.

```
soup.select('#title')  # selects the element with id="title"
```

□ 3. Class Selector

Selects elements with a specific class.

```
soup.select('.description')     # selects elements with class="description"
soup.select('.item')            # selects all <li> tags with class="item"
```

□ 4. Attribute Selector

```
soup.select('a[href]')  # selects all <a> tags with href attribute
soup.select('a[href="https://example.com"]')  # exact match
```

□ 5. Descendant Selector (Space)

Selects nested elements.

```
soup.select('div p')  # selects all <p> inside <div>
```

□ 6. Direct Child Selector (>)

```
soup.select('div > h1')  # selects <h1> directly under <div>
```

□ 7. Multiple Selectors (,)

soup.select('h1, p') # selects both <h1> and <p>

□ 8. Nth-of-type / Pseudo Selectors (limited support in BeautifulSoup)

soup.select('li:nth-of-type(2)') # selects the 2nd

□ Example: Extracting Data

```
titles = soup.select('h1')
for title in titles:
    print(title.text)
```

7. a)Webbrowser module

The `webbrowser` and `requests` modules are both Python libraries commonly used in web development and automation tasks, but they serve different purposes.

`webbrowser` Module:

The `webbrowser` module allows you to launch and control a web browser from your Python script. It provides a simple interface to open web pages in a browser window. Below is a simple program demonstrating how to use the `webbrowser` module:

```python
import webbrowser

# URL to open in the browser
url = "https://www.example.com"

# Open the URL in a web browser
webbrowser.open(url)
```

This program will open the specified URL in the default web browser on your system.

`requests` Module:

The `requests` module allows you to send HTTP requests easily in Python. It provides methods to make GET, POST, PUT, DELETE, and other types of HTTP requests, and handle responses from web servers. Below is a simple program demonstrating how to use the `requests` module to make a GET request:

```python
import requests

# URL to send GET request
url = "https://jsonplaceholder.typicode.com/posts/1"

# Send GET request to the URL
response = requests.get(url)

# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Print the response content (usually JSON or HTML)
    print(response.json())
else:
    # Print an error message if the request was unsuccessful
    print("Error:", response.status_code)
```

This program sends a GET request to the specified URL and prints the response content if the request was successful.

Both of these modules are useful in different scenarios. The `webbrowser` module is handy for tasks where you need to interact with a web browser, such as opening a URL for user interaction. On the other hand, the `requests` module is useful for tasks involving HTTP requests, such as web scraping, API interactions, etc.

8. Resampling:

Resampling is a technique used in time series analysis to change the frequency of the time series data. It involves aggregating or interpolating the data over different time intervals. Resampling is useful for various purposes such as downsampling (reducing the frequency of data points) or upsampling (increasing the frequency of data points), as well as for adjusting the time periods to align with specific requirements.

There are two main types of resampling:

1. **Upsampling**: Increasing the frequency of data points by interpolating or filling in missing values for the new time intervals.
2. **Downsampling**: Decreasing the frequency of data points by aggregating multiple data points into a single data point for the new time intervals.

Here's a program example demonstrating both types of resampling using pandas:

```python
import pandas as pd
import numpy as np

# Create a sample time series DataFrame
np.random.seed(0)
dates = pd.date_range(start='2022-01-01', periods=10, freq='D')
data = {'value': np.random.randint(1, 100, size=10)}
ts = pd.DataFrame(data, index=dates)

# Upsample to hourly frequency and interpolate missing values
upsampled = ts.resample('H').asfreq()
interpolated = upsampled.interpolate(method='linear')

print("Upsampled and Interpolated Data:")
print(interpolated)

# Downsample to weekly frequency and aggregate with mean
downsampled = ts.resample('W').mean()

print("\nDownsampled Data (Mean Aggregation):")
print(downsampled)
```

Output:
```
Upsampled and Interpolated Data:
                     value
2022-01-01 00:00:00   47.000000
2022-01-01 01:00:00   47.400000
2022-01-01 02:00:00   47.800000
2022-01-01 03:00:00   48.200000
...
2022-01-09 20:00:00   85.714286
2022-01-09 21:00:00   86.142857
2022-01-09 22:00:00   86.571429
2022-01-09 23:00:00   87.000000

Downsampled Data (Mean Aggregation):
                value
2022-01-02  50.333333
2022-01-09  66.571429
```

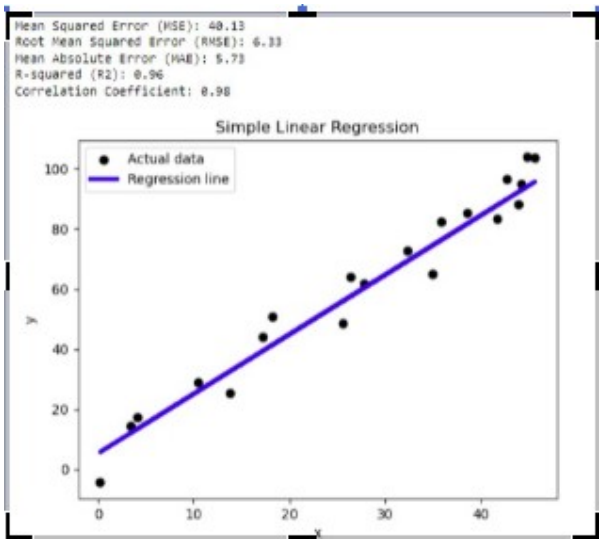
In this example:

- We create a sample time series DataFrame `ts` with 10 data points spanning 10 days.
- We upsample the data to hourly frequency using the `resample()` method with a frequency of 'H', and then interpolate the missing values using linear interpolation.
- We downsample the data to weekly frequency using the `resample()` method with a frequency of 'W', and aggregate the data using the mean function to compute the average value for each week.

Resampling allows us to adjust the frequency of time series data according to our analysis requirements and can be useful for various applications such as data visualization, forecasting, and modeling.

9. Linear regression:

```
import matplotlib.pyplot as plt
from scipy import stats
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
slope, intercept,r,p,std_err = stats.linregress(x, y)
def myfunc(x):
return slope * x + intercept
#code
mymodel = list(map(myfunc, x))
plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
# Load the dataset from the CSV file
df = pd.read_csv('sample_dataset.csv')
# Extract 'x' and 'y' columns
X = df[['x']]
y = df[['y']]
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Initialize the Linear Regression model
model = LinearRegression()
# Train the model
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False) # RMSE is the square root of MSE
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
cor_coe=df['x'].corr(df['y'])
# Print evaluation metrics
print(f'Mean Squared Error (MSE): {mse:.2f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.2f}')
print(f'Mean Absolute Error (MAE): {mae:.2f}')
print(f'R-squared (R2): {r2:.2f}')
print(f'Coefficient_coefficent:{cor_coe:.2f}')
# Visualize the regression line
plt.scatter(X_test, y_test, color='black', label='Actual data')
plt.plot(X_test, y_pred, color='blue', linewidth=3, label='Regression line')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Simple Linear Regression')
plt.legend()
plt.show()
```

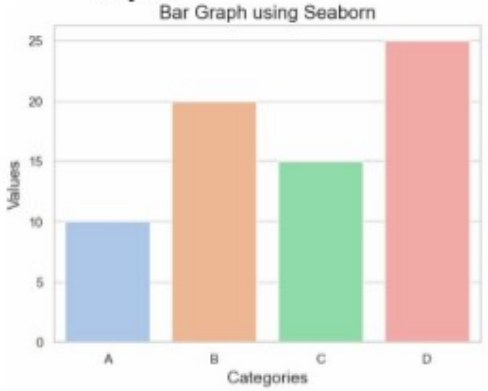


10.Data visualization in python using seaborn

Bargraph

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
# Sample data
data = {
'Category': ['A', 'B', 'C', 'D'],
'Values': [10, 20, 15, 25]
}
# Convert the data into a DataFrame
df = pd.DataFrame(data)
# Create a bar plot using Seaborn
sns.barplot(x='Category', y='Values', data=df, palette='pastel')
# Add titles and labels
plt.title('Bar Graph using Seaborn', fontsize=16)
plt.xlabel('Categories', fontsize=14)
plt.ylabel('Values', fontsize=14)
# Show the plot
plt.show()
```

Output:

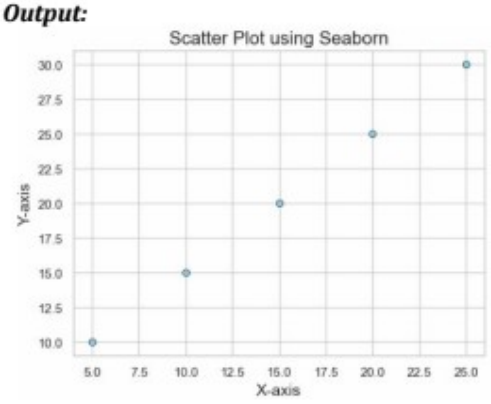


Scatter plot

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
# Sample data
data = {
'X': [5, 10, 15, 20, 25],
'Y': [10, 15, 20, 25, 30]
}
# Convert the data into a DataFrame
df = pd.DataFrame(data)


# Create a scatter plot using Seaborn
sns.scatterplot(x='X', y='Y', data=df, color='skyblue',
edgecolor='black')
# Add titles and labels
plt.title('Scatter Plot using Seaborn', fontsize=16)
plt.xlabel('X-axis', fontsize=14)
plt.ylabel('Y-axis', fontsize=14)
plt.show()
```

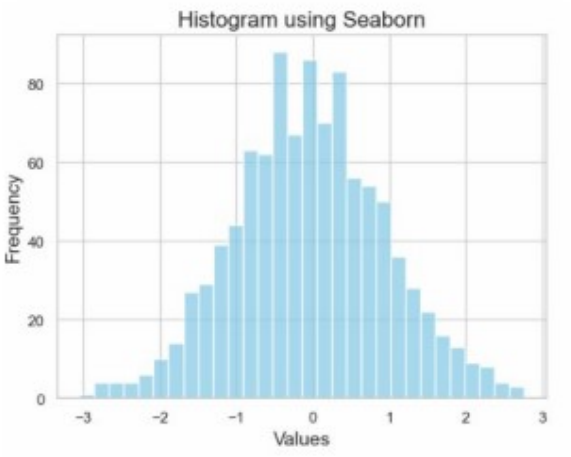
Output:



Histogram

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
# Generate random data (for demonstration purposes)
np.random.seed(0)
data = np.random.randn(1000) # Generate 1000 random numbers from a standard normal distribution
# Create a histogram using Seaborn
sns.histplot(data,color='skyblue', bins=30)
# Add titles and labels
plt.title('Histogram using Seaborn', fontsize=16)
plt.xlabel('Values', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.show()
```

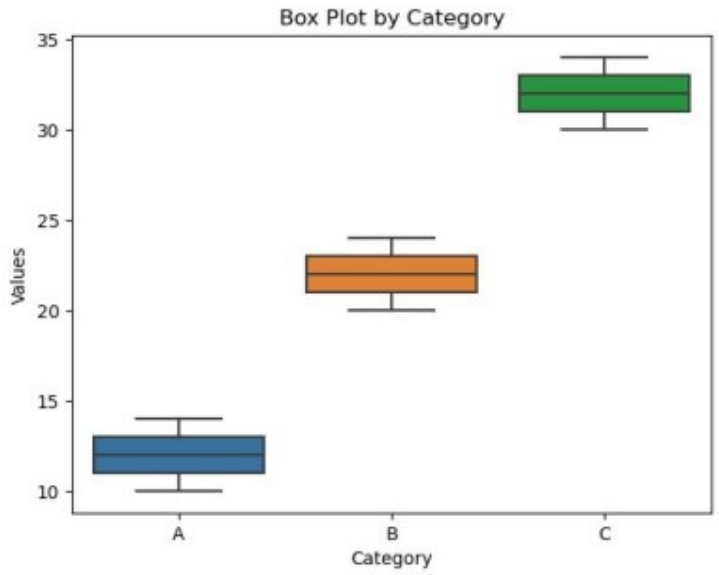
Output:



Box Plot

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
# Example dataset
data = pd.DataFrame({
"Category": ["A", "A", "A", "B", "B", "B", "C", "C", "C"],
"Values": [10, 12, 14, 20, 22, 24, 30, 32, 34]
})
# Create a box plot
sns.boxplot(x="Category", y="Values", data=data)
# Add a title and labels
plt.title('Box Plot by Category')
plt.xlabel('Category')
plt.ylabel('Values')
# Show the plot
plt.show()
```

Output:



Violin Plot

```
import seaborn as sns
```

```
import matplotlib.pyplot as plt
import pandas as pd
# Example dataset
data = pd.DataFrame({
"Category": ["A", "A", "A", "B", "B", "B", "C", "C", "C"],
"Values": [10, 12, 14, 20, 22, 24, 30, 32, 34],

"Subcategory": ["X", "Y", "X", "Y", "X", "Y", "X", "Y", "X"]
})
# Create a Split Violin Plot
plt.figure(figsize=(8, 6))
sns.violinplot(x="Category", y="Values", hue="Subcategory", data=data, split=True)
# Customize the graph
plt.title('Split Violin Plot of Values by Category and Subcategory')
plt.xlabel('Category')
plt.ylabel('Values')
# Display the plot
plt.show()
```

Output: