CMR
INSTITUTE OF TECHNOLOGY

USN

Internal Assessment Test 2 – March 2025

Sub:	**Data Analytics using Python**							**Sub Code:**	**22MCA31**
Date:	**10/03/2025**	**Duration:**	**90 mins**	**Max Marks:**	**50**	**Sem:**	**III**	**Branch:**	**MCA**

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

		MARKS	OBE	
	PART I		CO	RBT
1	Explain about the pandas data structure in detail. **OR**	5+5	CO2	L2
2	Write a note on universal functions with a program example.		CO2	L2
3	**PART II** Explain reordering and sorting levels in pandas with a program example. **OR**	5+5	CO2	L3
4	Discuss about regular expressions in python with a program example.	10	CO3	L3
5	**PART III** Write a python program to demonstrate the steps involved in data pre-processing. **OR**	10	CO4	L4
6	What is a join? Explain the types of joins with a program example.	3+7	CO2	L3
7	**PART IV** How to read and write into a file using a CSV library?Explain with a program. **OR**	5+5	CO2	L3
8	Explain the following with a program example A) Hierarchical indexing B) Fancy indexing	5+5	CO2	L2
9	**PART V** Write a python program to demonstrate the different attributes of arrays in numpy **OR**	10	CO3	L4
10	Explain the following with a program example A) Splitting of the arrays B) Searching in an array	5+5	CO3	L3

1. Pandas

Pandas Data Structures – Detailed Explanation

Pandas is a popular Python library used for data manipulation and analysis. It provides powerful data structures designed to handle structured data efficiently. The two main data structures in Pandas are:

1. **Series** (1D labeled array)
2. **DataFrame** (2D labeled table)

1. Pandas Series

A **Series** is a one-dimensional array-like object that can store various data types (integers, floats, strings, or even Python objects). Each value in a Series is associated with an index, which is similar to row labels in a spreadsheet.

Features of Series:

- Homogeneous (stores data of the same type).
- Has an **index** (default or custom).
- Can be created from lists, dictionaries, or NumPy arrays.
- Supports vectorized operations (like NumPy arrays).

Creating a Pandas Series

A Series can be created using the pd.Series() function.

Example 1: Creating a Series with Default Index

```
import pandas as pd

# Creating a Series from a list
s = pd.Series([10, 20, 30, 40, 50])

# Displaying the Series
print("Series with Default Index:\n", s)
```

Output:

```
Series with Default Index:
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

- The numbers on the left (0, 1, 2, ...) represent the **index**.
- The values (10, 20, 30, ...) are the actual data stored in the Series.

2. Pandas DataFrame

A **DataFrame** is a two-dimensional tabular structure with labeled rows and columns. It is similar to an Excel spreadsheet or an SQL table.

Features of DataFrame:

- Can hold multiple data types (integers, floats, strings, etc.).
- Has labeled rows (index) and columns.
- Can be created from dictionaries, lists, NumPy arrays, CSV files, etc.
- Supports operations like filtering, sorting, and aggregation.

Creating a Pandas DataFrame

Example 1: Creating a DataFrame from a Dictionary

```
# Creating a DataFrame using a dictionary
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'Los Angeles', 'San Francisco']
}

df = pd.DataFrame(data)

print("DataFrame:\n", df)
```

Output:

```
DataFrame:
      Name  Age           City
0    Alice   25       New York
1      Bob   30    Los Angeles
2  Charlie   35  San Francisco
```

- The **columns** (Name, Age, City) represent different features.
- The **rows** (0, 1, 2) represent different records.

2. Universal functions

NumPy Universal Functions (ufuncs) – Detailed Explanation

NumPy provides a variety of **universal functions (ufuncs)** that allow us to perform operations efficiently on arrays. These functions are categorized into different types based on their functionality.

1. Arithmetic Functions

Arithmetic functions perform element-wise mathematical operations on arrays.

Common Arithmetic ufuncs

Function	Description
np.add(x, y)	Element-wise addition of two arrays
np.subtract(x, y)	Element-wise subtraction
np.multiply(x, y)	Element-wise multiplication

np.divide(x, y)	Element-wise division
np.mod(x, y)	Element-wise modulus (remainder)
np.power(x, y)	Element-wise exponentiation
np.sqrt(x)	Element-wise square root

Example

```
import numpy as np

# Creating two NumPy arrays
a = np.array([2, 4, 6, 8])
b = np.array([1, 2, 3, 4])

# Performing arithmetic operations
print("Addition:", np.add(a, b))       # [3 6 9 12]
print("Subtraction:", np.subtract(a, b))  # [1 2 3 4]
print("Multiplication:", np.multiply(a, b))  # [2 8 18 32]
print("Division:", np.divide(a, b))  # [2. 2. 2. 2.]
```

2. Mathematical Functions

These functions are used for mathematical computations, such as exponentiation, logarithm, and trigonometric calculations.

Common mathematical ufuncs

Function	Description
np.exp(x)	Exponential function e^x
np.log(x)	Natural logarithm ln(x)
np.log10(x)	Base-10 logarithm
np.log2(x)	Base-2 logarithm
np.abs(x)	Absolute value of elements
np.round(x, decimals=n)	Round elements to n decimal places

Example

```
# Creating a NumPy array
arr = np.array([1, 4, 9, 16])

print("Square Root:", np.sqrt(arr))  # [1. 2. 3. 4.]
print("Exponential:", np.exp(arr))  # [2.718... 54.59... 8103.08...]
print("Logarithm:", np.log(arr))  # [0. 1.386 2.197 2.772]
```

3. Statistical Functions

Statistical functions help in analyzing data by calculating mean, median, standard deviation, and more.

Common Statistical ufuncs

Function	Description
np.mean(x)	Mean (average) of elements
np.median(x)	Median of elements

np.std(x)	Standard deviation
np.var(x)	Variance
np.min(x)	Minimum value
np.max(x)	Maximum value
np.percentile(x, q)	q-th percentile of the array

Example

```
# Creating a NumPy array
data = np.array([5, 10, 15, 20, 25])

print("Mean:", np.mean(data))  # 15.0
print("Median:", np.median(data))  # 15.0
print("Standard Deviation:", np.std(data))  # 7.07
print("Variance:", np.var(data))  # 50.0
print("Min:", np.min(data))  # 5
print("Max:", np.max(data))  # 25
```

5. Broadcasting in NumPy

What is Broadcasting?

Broadcasting allows NumPy to perform element-wise operations on arrays of **different shapes and sizes** without explicitly reshaping them.

Broadcasting Rules

1. If the two arrays have **different dimensions**, NumPy **automatically expands** the smaller array to match the larger array.
2. If one array has **size 1 in any dimension**, it is **stretched** to match the corresponding dimension of the larger array.

Example 1: Broadcasting in Addition

```
# Creating a 1D array
arr1 = np.array([1, 2, 3])

# Creating a scalar (single number)
scalar = 5

# Broadcasting allows us to add a scalar to an array
result = arr1 + scalar
print("Broadcasted Addition:", result)  # [6 7 8]
```

Example 2: Broadcasting in a 2D Array

```
# Creating a 2D array (3x3)
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Creating a 1D array
row_vector = np.array([1, 0, -1])

# Broadcasting will apply the row_vector to each row
result = matrix + row_vector
```

```
print("Broadcasted Result:\n", result)
```

Output:

```
Broadcasted Result:
 [[ 2  2  2]
  [ 5  5  5]
  [ 8  8  8]]
```

Here, row_vector is broadcasted across each row of matrix.

3. Reordering ans sorting levels

Reordering and sorting levels in hierarchical indexing(MultiIndex) is a common operation in data cleaning and analysis tasks using the pandas library in Python.

Hierarchical indexing allows you to work withmulti-dimensional datasets efficiently, but sometimes you may need to reorder or sort the levels of the index to better organize, analyze, or present the data.

Reordering levels refers to changing the order of the hierarchical index levels. This is useful when dealing with **MultiIndex DataFrames** where we need to swap or rearrange levels for better organization or processing.

Reordering:

```
import pandas as pd



# Creating a MultiIndex DataFrame

arrays = [

    ['USA', 'USA', 'India', 'India'],

    ['New York', 'California', 'Delhi', 'Mumbai']

]



index = pd.MultiIndex.from_arrays(arrays, names=('Country', 'City'))



data = {'Population': [8500000, 4000000, 18000000, 20000000]}



df = pd.DataFrame(data, index=index)
```

```
# Display original DataFrame

print("Original MultiIndex DataFrame:\n", df)


# Swapping levels (City and Country)

df_swapped = df.swaplevel('Country', 'City')


print("\nAfter Swapping Levels:\n", df_swapped)


# Reordering levels explicitly

df_reordered = df.reorder_levels(['City', 'Country'])


print("\nAfter Reordering Levels:\n", df_reordered)
```

Sorting:

```
# Sorting by 'Country' level
df_sorted = df.sort_index(level='Country')

print("\nAfter Sorting by Country:\n", df_sorted)

# Sorting by 'City' level
df_sorted_city = df.sort_index(level='City')

    print("\nAfter Sorting by City:\n", df_sorted_city)
```

4. Regular expressions

Regular Expressions in Python

What are Regular Expressions?

Regular Expressions (**regex**) are patterns used for searching, matching, and manipulating strings. Python provides the re module, which allows us to perform pattern-based operations like searching and replacing text.

Common Regex Functions in Python

1. **re.match(pattern, string)** – Checks if the pattern matches at the beginning of the string.
2. **re.search(pattern, string)** – Searches for the first occurrence of the pattern anywhere in the string.

3. **re.findall(pattern, string)** – Returns all occurrences of the pattern as a list.
4. **re.sub(pattern, replace, string)** – Replaces occurrences of the pattern with the given text.
5. **re.split(pattern, string)** – Splits the string wherever the pattern matches.
6. **re.compile(pattern)** – Precompiles a regex pattern for repeated use.

Example Program Using Regex in Python

```
import re

# Sample text
text = "The price of the product is $50 and the discount is 20%."

# Finding all numbers in the text
numbers = re.findall(r'\d+', text)
print("Extracted Numbers:", numbers)

# Checking if text starts with "The"
if re.match(r'^The', text):
    print("The text starts with 'The'.")

# Searching for a word in the text
match = re.search(r'product', text)
if match:
    print(f"Found the word 'product' at index {match.start()}")

# Replacing numbers with "[NUMBER]"
modified_text = re.sub(r'\d+', '[NUMBER]', text)
print("Modified Text:", modified_text)
```

Output of the Above Code

```
Extracted Numbers: ['50', '20']
The text starts with 'The'.
Found the word 'product' at index 20
Modified Text: The price of the product is $[NUMBER] and the discount is [NUMBER]%.
```

5. Program on data pre processing

Python Program Demonstrating Complete Data Preprocessing

This program covers all the essential data preprocessing steps:
Importing Libraries
Loading Data
Identifying & Handling Missing Values
Handling Duplicates
Data Transformation
Replacing Values
Handling Outliers
Data Validation

Python Program

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Step 1: Importing Necessary Libraries
print("Libraries Imported Successfully!")

# Step 2: Loading Data (Creating Sample Data)
data = {
   'ID': [1, 2, 3, 4, 5, 6, 7, 8, 8],  # Duplicate ID
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank', 'Grace', 'Henry', 'Henry'],
   'Age': [25, np.nan, 30, 35, 40, 45, np.nan, 50, 50],  # Missing values
   'Salary': [50000, 60000, np.nan, 80000, 90000, 100000, 110000, 120000, 120000],  # Missing value
   'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago', 'Los Angeles', 'Chicago', 'New York', 'New
York'],
   'Experience': [2, 5, 7, 10, 15, 20, 25, 30, 30],
   'Department': ['HR', 'IT', 'HR', 'Finance', 'IT', 'HR', 'Finance', 'IT', 'IT']
}

df = pd.DataFrame(data)
print("\nOriginal Data:\n", df)

# Step 3: Identifying Missing Values
print("\nMissing Values in Each Column:\n", df.isnull().sum())

# Step 4: Handling Missing Values (Replacing with Mean)
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].mean(), inplace=True)

# Step 5: Handling Duplicates
df.drop_duplicates(inplace=True)

# Step 6: Data Transformation (Encoding Categorical Data)
df['City'] = df['City'].astype('category').cat.codes  # Converts categories to numerical codes
df['Department'] = df['Department'].astype('category').cat.codes

# Step 7: Replacing Values (Replacing IT with Information Technology)
df['Department'].replace({1: 'IT'}, inplace=True)  # Example: Replace department codes

# Step 8: Handling Outliers (Using IQR Method)
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
df = df[(df['Salary'] >= lower_bound) & (df['Salary'] <= upper_bound)]

# Step 9: Data Validation (Checking Final Data)
print("\nFinal Processed Data:\n", df)
```

Explanation of Steps

1. **Importing Libraries** – We use **pandas, numpy, and sklearn** for processing.
2. **Loading Data** – Creating a sample dataset with missing values, duplicates, and categorical data.
3. **Identifying Missing Values** – Using df.isnull().sum() to find missing data.

4. **Handling Missing Values** – Filling missing **Age** and **Salary** with their mean values.
5. **Handling Duplicates** – Removing duplicate rows with df.drop_duplicates().
6. **Data Transformation** – Converting categorical columns (**City, Department**) into numerical values.
7. **Replacing Values** – Modifying values (e.g., replacing IT with Information Technology).
8. **Handling Outliers** – Using **IQR (Interquartile Range)** to remove extreme salary values.
9. **Data Validation** – Displaying the cleaned, processed dataset.

Sample Output

```
Libraries Imported Successfully!

Original Data:
   ID    Name   Age   Salary        City  Experience Department
0  1   Alice  25.0  50000.0    New York         2        HR
1  2     Bob   NaN  60000.0  Los Angeles         5        IT
2  3  Charlie  30.0      NaN     Chicago         7       HR
3  4   David  35.0  80000.0    New York        10    Finance
4  5     Eve  40.0  90000.0     Chicago        15       IT

Missing Values in Each Column:
 Age       2
Salary     1
dtype: int64

Final Processed Data:
   ID    Name   Age   Salary  City  Experience    Department
0  1   Alice  25.0  50000.0    2         2          HR
1  2     Bob  32.0  60000.0    1         5  Information Technology
3  4   David  35.0  80000.0    2        10     Finance
4  5     Eve  40.0  90000.0    0        15  Information Technology
```

6. Joins

Joins in Python (Pandas)

A **join** in Pandas is used to combine two or more DataFrames based on a common column or index. It is similar to SQL joins and helps in merging datasets efficiently.

Types of Joins in Pandas

There are four main types of joins in Pandas:

1. **Inner Join** – Returns only matching records from both DataFrames.
2. **Left Join** – Returns all records from the left DataFrame and matching records from the right.
3. **Right Join** – Returns all records from the right DataFrame and matching records from the left.
4. **Outer Join** – Returns all records from both DataFrames, filling missing values with NaN.

Example: Demonstrating Different Joins in Pandas

```
import pandas as pd

# Creating first DataFrame
df1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40]
})

# Creating second DataFrame
df2 = pd.DataFrame({
    'ID': [3, 4, 5, 6],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston'],
    'Salary': [70000, 80000, 90000, 60000]
})

# Inner Join (Only common IDs)
inner_join = pd.merge(df1, df2, on='ID', how='inner')

# Left Join (All IDs from df1, matching from df2)
left_join = pd.merge(df1, df2, on='ID', how='left')

# Right Join (All IDs from df2, matching from df1)
right_join = pd.merge(df1, df2, on='ID', how='right')

# Outer Join (All IDs from both DataFrames)
outer_join = pd.merge(df1, df2, on='ID', how='outer')

# Displaying results
print("\nInner Join:\n", inner_join)
print("\nLeft Join:\n", left_join)
print("\nRight Join:\n", right_join)
print("\nOuter Join:\n", outer_join)
```

Output

```
Inner Join:
   ID    Name  Age         City  Salary
2   3  Charlie   35     New York   70000
3   4    David   40  Los Angeles   80000

Left Join:
   ID    Name  Age         City  Salary
0   1    Alice   25          NaN     NaN
1   2      Bob   30          NaN     NaN
2   3  Charlie   35     New York  70000
3   4    David   40  Los Angeles  80000

Right Join:
   ID    Name   Age         City  Salary
0   3  Charlie  35.0     New York  70000
1   4    David  40.0  Los Angeles  80000
2   5      NaN   NaN      Chicago  90000
3   6      NaN   NaN      Houston  60000
```

```
Outer Join:
   ID    Name   Age         City  Salary
0   1   Alice  25.0          NaN     NaN
1   2     Bob  30.0          NaN     NaN
2   3  Charlie  35.0    New York   70000
3   4   David  40.0  Los Angeles   80000
4   5     NaN   NaN     Chicago   90000
5   6     NaN   NaN     Houston   60000
```

Explanation

1. **Inner Join** – Keeps only rows where **ID** is present in both DataFrames (IDs 3 and 4).
2. **Left Join** – Keeps all rows from df1, filling missing values from df2 with NaN (IDs 1 and 2 have no matches).
3. **Right Join** – Keeps all rows from df2, filling missing values from df1 with NaN (IDs 5 and 6 have no matches).
4. **Outer Join** – Keeps all rows from both DataFrames, filling missing values where necessary.

7. Reading and writing into file using csv library

Reading and Writing CSV Files in Python using Pandas

1. Introduction

The **pandas** library provides functions to easily read and write CSV (Comma-Separated Values) files:

- pd.read_csv("filename.csv") → Reads a CSV file into a **DataFrame**.
- df.to_csv("filename.csv", index=False) → Writes a **DataFrame** to a CSV file.

Python Program for Reading and Writing CSV Files

```
import pandas as pd

# Step 1: Creating a Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'Salary': [50000, 60000, 70000, 80000]
}

df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Step 2: Writing Data to a CSV File
df.to_csv("employees.csv", index=False)  # Writing without the index column
print("\nData has been written to employees.csv")

# Step 3: Reading Data from the CSV File
df_read = pd.read_csv("employees.csv")
print("\nData Read from CSV:\n", df_read)
```

Explanation

Writing to a CSV File

- The function to_csv("filename.csv", index=False) saves the DataFrame to a CSV file.
- The index=False argument prevents pandas from writing row indices to the file.

Reading from a CSV File

- The function read_csv("filename.csv") reads the CSV file and loads it into a **DataFrame**.

Output

```
Original DataFrame:
     Name  Age  Salary
0  Alice   25  50000
1    Bob   30  60000
2 Charlie   35  70000
3  David   40  80000

Data has been written to employees.csv

Data Read from CSV:
     Name  Age  Salary
0  Alice   25  50000
1    Bob   30  60000
2 Charlie   35  70000
3  David   40  80000
```

8. Hirearchical indexing and fancy indexing

A) Hierarchical Indexing in Pandas

What is Hierarchical Indexing?

Hierarchical indexing (MultiIndex) allows multiple index levels in a DataFrame, making it easier to work with multi-dimensional data.

Example: Hierarchical Indexing in Pandas

```
import pandas as pd

# Creating a MultiIndex DataFrame
arrays = [
   ['USA', 'USA', 'India', 'India'],
   ['New York', 'California', 'Delhi', 'Mumbai']
]

index = pd.MultiIndex.from_arrays(arrays, names=('Country', 'City'))

data = {'Population': [8500000, 4000000, 18000000, 20000000]}
```

```
df = pd.DataFrame(data, index=index)

# Display DataFrame with hierarchical index
print("Hierarchical Index DataFrame:\n", df)

# Accessing data using hierarchical indexing
print("\nPopulation of Delhi:\n", df.loc[('India', 'Delhi')])
```

Output

```
Hierarchical Index DataFrame:
                Population
Country City
USA     New York     8500000
        California   4000000
India   Delhi        18000000
        Mumbai       20000000

Population of Delhi:
 Population    18000000
Name: (India, Delhi), dtype: int64
```

B) Fancy Indexing in NumPy

What is Fancy Indexing?

Fancy indexing allows us to retrieve specific elements from an array using an array of indices.

Example: Fancy Indexing in NumPy

```
import numpy as np

# Creating a NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Selecting elements using a list of indices
indices = [0, 2, 4]  # Selecting the 1st, 3rd, and 5th elements
fancy_result = arr[indices]

print("Original Array:", arr)
print("Fancy Indexed Result:", fancy_result)
```

Output

```
Original Array: [10 20 30 40 50]
Fancy Indexed Result: [10 30 50]
```

9. Attributes of numpy:

Python Program to Demonstrate Different Attributes of NumPy Arrays

NumPy arrays have various attributes that help us understand their structure and properties. Some of the important attributes are:

- ndim (Number of dimensions)
- shape (Shape of the array)
- size (Total number of elements)

- dtype (Data type of elements)
- itemsize (Size of each element in bytes)
- nbytes (Total memory consumed by the array)

Python Program

```
import numpy as np

# Creating a NumPy array
arr = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32)

# Displaying different attributes of the array
print("Array:\n", arr)
print("\nNumber of Dimensions:", arr.ndim)
print("Shape of Array:", arr.shape)
print("Total Elements:", arr.size)
print("Data Type of Elements:", arr.dtype)
print("Size of Each Element (in bytes):", arr.itemsize)
print("Total Memory Used (in bytes):", arr.nbytes)
```

Output

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

Number of Dimensions: 2
Shape of Array: (2, 3)
Total Elements: 6
Data Type of Elements: int32
Size of Each Element (in bytes): 4
Total Memory Used (in bytes): 24
```

Explanation

1. **ndim** – The array has **2 dimensions** (rows and columns).
2. **shape** – The shape **(2, 3)** means **2 rows and 3 columns**.
3. **size** – The total number of elements is **6** (2×3).
4. **dtype** – The elements are of type **int32**.
5. **itemsize** – Each element takes **4 bytes** in memory.
6. **nbytes** – The total memory occupied is **24 bytes** (6 elements × 4 bytes).

10. Splitting and searching

A) Splitting of Arrays in NumPy

Array splitting in NumPy allows us to divide a large array into multiple smaller arrays. The functions used for splitting are:

- np.split(array, sections, axis) – Splits an array into equal-sized parts.
- np.array_split(array, sections, axis) – Splits an array into **unequal** parts.
- np.hsplit(array, sections) – Splits an array horizontally (column-wise).
- np.vsplit(array, sections) – Splits an array vertically (row-wise).

Example: Splitting an Array

```
import numpy as np
```

```
# Creating an array
arr = np.array([10, 20, 30, 40, 50, 60])

# Splitting into 3 equal parts
split_arr = np.split(arr, 3)
print("Equal Splitting:", split_arr)

# Splitting into 4 unequal parts
unequal_split = np.array_split(arr, 4)
print("Unequal Splitting:", unequal_split)
```

Output

```
Equal Splitting: [array([10, 20]), array([30, 40]), array([50, 60])]
Unequal Splitting: [array([10, 20]), array([30, 40]), array([50]), array([60])]
```

B) Searching in an Array

Searching in NumPy is used to find elements in an array based on conditions. The functions used for searching are:

- np.where(condition) – Returns the indices where the condition is **True**.
- np.searchsorted(sorted_array, value) – Finds the index where a value should be inserted in a **sorted** array.
- np.nonzero(array) – Returns indices of all **non-zero** elements.

Example: Searching in an Array

```
# Creating an array
arr = np.array([10, 20, 30, 40, 50, 60])

# Finding indices where values are greater than 30
indices = np.where(arr > 30)
print("Indices where values > 30:", indices)

# Searching for the position to insert 35 in a sorted array
sorted_arr = np.array([10, 20, 30, 40, 50])
insert_index = np.searchsorted(sorted_arr, 35)
print("Index to insert 35 in sorted array:", insert_index)
```

Output

```
Indices where values > 30: (array([3, 4, 5]),)
Index to insert 35 in sorted array: 3
```