April 30, 2023

7)a)Aim:-

To Selecting Columns or Rows Accessing sub data frames using pandas in python.

Description:-

When working with large datasets in pandas, it’s often necessary to select a subset of columns or rows, or to create a sub-DataFrame from a larger DataFrame. Pandas provides several methods for doing this.

One way to select columns in pandas is to use the bracket notation ([]). You can use this notation to select a single column by name, or multiple columns by passing in a list of column names. For example, to select a single column named my_column, you can use df[’my_column’]. To select multiple columns, you can use df[[’col1’, ’col2’]].

To select rows in pandas, you can use the iloc and loc accessors. The iloc accessor is used to select rows and columns based on their integer position. You can use df.iloc[row_index] to select a single row by its integer index, or df.iloc[start:end] to select multiple rows by their integer indices. You can also use df.iloc[:, col_index] to select a single column by its integer index, or df.iloc[:, start:end] to select multiple columns by their integer indices.

The loc accessor is used to select rows and columns based on their labels. You can use df.loc[row_label] to select a single row by its label, or df.loc[start:end] to select multiple rows by their labels. You can also use df.loc[:, col_label] to select a single column by its label, or df.loc[:, start:end] to select multiple columns by their labels.

You can also use boolean indexing to select rows based on a condition. For example, df[df[’my_column’] ¿ 5] will return all rows where the value in the my_column column is greater than 5.

Overall, pandas provides a variety of ways to select columns and rows, allowing you to easily manipulate and analyze your data in a flexible and powerful way.

Program:-

```
import pandas as pd
data = { ’name’: [’Alice’, ’Bob’, ’Charlie’, ’David’, ’Eva’],
’age’: [25, 30, 35, 40, 45],
’gender’: [’F’, ’M’, ’M’, ’M’, ’F’],
’city’: [’New York’, ’Paris’, ’Tokyo’, ’Berlin’, ’London’],
’salary’: [50000, 70000, 90000, 110000, 130000]
}
df = pd.DataFrame(data)
print(df[’name’])
print(df[[’name’, ’age’]])
print(df.iloc[2])
print(df.iloc[1:4])
print(df.loc[3])
print(df.loc[1:3, [’name’, ’city’]])
```

Expected output:-

```
Shell
0      Alice
1        Bob
2    Charlie
3      David
4        Eva
Name: name, dtype: object
name  age
0    Alice   25
1      Bob   30
2  Charlie   35
3    David   40
4      Eva   45
name      Charlie
age            35
gender          M
city        Tokyo
salary      90000
Name: 2, dtype: object
name  age gender    city  salary
1      Bob   30      M   Paris   70000
2  Charlie   35      M   Tokyo   90000
3    David   40      M  Berlin  110000
name       David
age           40
gender         M
city      Berlin
salary    110000
Name: 3, dtype: object
name    city
1      Bob   Paris
2  Charlie   Tokyo
3    David  Berlin
>
```

0.1 Observed output:-

```
Shell
0      Alice
1        Bob
2    Charlie
3      David
4        Eva
Name: name, dtype: object
name  age
0    Alice   25
1      Bob   30
2  Charlie   35
3    David   40
4      Eva   45
name      Charlie
age            35
gender          M
city        Tokyo
salary      90000
Name: 2, dtype: object
name  age gender    city  salary
1      Bob   30      M   Paris   70000
2  Charlie   35      M   Tokyo   90000
3    David   40      M  Berlin  110000
name       David
age           40
gender         M
city      Berlin
salary    110000
Name: 3, dtype: object
name    city
1      Bob   Paris
2  Charlie   Tokyo
3    David  Berlin
>
```

7)b)Aim:-

To Selecting Columns or Rows Filtering Records using pandas in python.

Description:-

Filtering records in a pandas DataFrame involves selecting a subset of the data based on one or more conditions. This can be useful when you want to focus on a particular subset of the data that meets certain criteria.

To filter records in pandas, you can use boolean indexing. This involves creating a boolean mask that indicates which rows meet the specified conditions, and then using that mask to select the corresponding rows.

For example, suppose you have a DataFrame with columns for name, age, gender, and salary, and you want to filter the data to only include records where the age is greater than 30. Overall, filtering records in pandas allows you to easily focus on a particular subset of the data based on one or more conditions. This can be useful when you want to analyze or manipulate a specific subset of your data.

Program:-

```
import pandas as pd
data = { ’name’: [’Alice’, ’Bob’, ’Charlie’, ’David’, ’Eva’],
’age’: [25, 30, 35, 40, 45],
’gender’: [’F’, ’M’, ’M’, ’M’, ’F’],
’city’: [’New York’, ’Paris’, ’Tokyo’, ’Berlin’, ’London’],
’salary’: [50000, 70000, 90000, 110000, 130000]
}
df = pd.DataFrame(data)
df_filtered = df[df[’age’] >30]
df_filtered2 = df[(df[’gender’] == ’F’) & (df[’salary’] >60000)]
print(df_filtered)
print(df_filtered2)
```


Expected output:-

```
Shell                                                    Clear
name  age gender    city  salary
2  Charlie   35      M    Tokyo   90000
3    David   40      M  Berlin  110000
4      Eva   45      F  London  130000name  age gender    city  salary
4  Eva   45      F  London  130000
>
```

Observed output:-

```
Shell                                                    Clear
name  age gender    city  salary
2  Charlie   35      M    Tokyo   90000
3    David   40      M  Berlin  110000
4      Eva   45      F  London  130000name  age gender    city  salary
4  Eva   45      F  London  130000
>
```

8)a)Aim:-

wite a python handling missing values dropna using pandas

Description:-

In data analysis and machine learning, it's common to have missing values in datasets. Missing values can arise due to a variety of reasons, such as data entry errors or incomplete data collection. However, most machine learning algorithms cannot handle missing values, and it's important to preprocess the data by handling the missing values appropriately.

In Python, the Pandas library provides a convenient way to handle missing values using the dropna() function. This function is used to remove rows or columns from a DataFrame that contain missing values.

The dropna() function has several parameters that allow you to customize how missing values are handled. Some of the important parameters include:

axis: specifies whether to remove rows (axis=0) or columns (axis=1) that contain missing values. how: specifies how to determine if a row or column contains missing values. Possible values include any (remove any row or column with at least one missing value) or all (remove only rows or columns where all values are missing). thresh: specifies the minimum number of non-missing values required to keep a row or column. For example, thresh=2 means that a row or column must have at least 2 non-missing values to be kept. subset: specifies the columns or rows to consider for missing values. For example, subset=['col1', 'col2'] will only consider missing values in columns col1 and col2.

The dropna() function modifies the original DataFrame by default, but you can use the inplace=True parameter to modify the DataFrame in place instead of creating a new one.

Overall, using the dropna() function in Pandas is a straightforward way to handle missing values in Python, allowing you to preprocess your data for further analysis or machine learning tasks.

Program:-

```
import pandas as pd
df = pd.DataFrame({ 'col1': [1, 2, None, 4],
'col2': [5, 6, 7, None],
'col3': [8, None, 10, 11]
})
print("Original DataFrame:")
print(df)
df.dropna(inplace=True)
print("Updated DataFrame after dropping rows with missing values:")
print(df)
```

Expected output:-

```
Original DataFrame:
col1  col2  col3
0   1.0   5.0   8.0
1   2.0   6.0   NaN
2   NaN   7.0  10.0
3   4.0   NaN  11.0
Updated DataFrame after dropping rows with missing values:
col1  col2  col3
0   1.0   5.0   8.0
>
```

Observed output:-

```
Original DataFrame:
col1  col2  col3
0   1.0   5.0   8.0
1   2.0   6.0   NaN
2   NaN   7.0  10.0
3   4.0   NaN  11.0
Updated DataFrame after dropping rows with missing values:
col1  col2  col3
0   1.0   5.0   8.0
>
```

8)b)Aim:-

wite a python handling missing values fillna using pandas

Description:-

Missing data is a common problem in real-world datasets. It is important to handle missing data appropriately because it can affect the accuracy of data analysis and machine learning models.

Pandas is a popular data analysis library in Python that provides many functions to handle missing data. The fillna() function is one such function that is used to fill missing values in a pandas DataFrame.

The fillna() function takes one or more arguments to specify how to fill missing values. The most common argument is the value to be used for filling missing values, which can be a scalar value, a dictionary mapping column names to values, or a pandas Series. The inplace=True parameter is used to modify the original DataFrame instead of creating a new one.

You can also use other methods to fill missing values, such as forward-fill or backward-fill, which fill missing values with the nearest non-missing value in the same column Similarly, you can use bfill method for backward-fill.

In conclusion, the fillna() function is a powerful tool for handling missing values in pandas. By choosing the appropriate fill method and value, you can clean up your data and ensure that your analysis and models are based on accurate data.

Program:-

```
import pandas as pd
data = {’name’: [’John’, ’Sara’, ’Peter’, ’Emily’, ’Mike’],
’age’: [32, 21, None, 25, 28],
’gender’: [’M’, ’F’, ’M’, None, ’M’],
’salary’: [45000, 55000, 65000, None, 75000]}
df = pd.DataFrame(data)
print(”Original Dataframe:”)
print(df)
df.fillna(0, inplace=True)
print(”Dataframe after filling missing values:”)
print(df)
```

Expected output:-

Shell

```
Original Dataframe:
name    age gender    salary
0    John   32.0       M   45000.0
1    Sara   21.0       F   55000.0
2   Peter    NaN       M   65000.0
3   Emily   25.0    None       NaN
4    Mike   28.0       M   75000.0
Dataframe after filling missing values:
name    age gender    salary
0    John   32.0       M   45000.0
1    Sara   21.0       F   55000.0
2   Peter    0.0       M   65000.0
3   Emily   25.0       0       0.0
4    Mike   28.0       M   75000.0
>
```

Observed output:-

Shell

```
Original Dataframe:
name    age gender    salary
0    John   32.0       M   45000.0
1    Sara   21.0       F   55000.0
2   Peter    NaN       M   65000.0
3   Emily   25.0    None       NaN
4    Mike   28.0       M   75000.0
Dataframe after filling missing values:
name    age gender    salary
0    John   32.0       M   45000.0
1    Sara   21.0       F   55000.0
2   Peter    0.0       M   65000.0
3   Emily   25.0       0       0.0
4    Mike   28.0       M   75000.0
>
```

8)c)Aim:-

handling missing values recognize and treat missing values and outliers using pandas

Description:-

Handling missing values and outliers is an important step in data cleaning and preprocessing. In Python, pandas is a popular library for handling and manipulating data frames. Here's how you can recognize and treat missing values and outliers in pandas: Recognizing Missing Values

Pandas represent missing values as NaN (Not a Number). You can recognize missing values in a data frame using the isna() method. There are several ways to treat missing values in pandas. You can drop rows or columns containing missing values using the dropna() method. You can also fill missing values with a specified value using the fillna() method. Outliers are extreme values that fall outside of the normal range of values in a data set. You can recognize outliers using statistical methods such as the z-score or the interquartile range (IQR).

Program:-

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, None, 4], 'B': [5, None, 7, 8]})
print("Original Data Frame:")
print(df)
print("Missing Values:")
print(df.isna())
df_dropped = df.dropna()
print("Data Frame after dropping rows with missing values:")
print(df_dropped)
df_filled = df.fillna(0)
print("Data Frame after filling missing values with 0:")
print(df_filled)
df_outliers = pd.DataFrame({'A': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20]})
print("Data Frame with Outliers:")
print(df_outliers)
Q1 = df_outliers['A'].quantile(0.25)
Q3 = df_outliers['A'].quantile(0.75)
IQR = Q3 - Q1
outliers = df_outliers[(df_outliers['A'] >Q1 - 1.5*IQR) || (df_outliers['A']¿Q3 + 1.5 ∗ IQR)]
print("Outliers : ")
print(outliers)
forindex, rowinoutliers.iterrows() :
ifrow['A']l¡Q1 − 1.5 ∗ IQR :
df_outliers.loc[index,' A'] = Q1
elifrow['A']¿Q3 + 1.5 ∗ IQR :
df_outliers.loc[index,' A'] = Q3
print("DataFrameafterreplacingoutlierswithnearestnon − outliervalues : ")
print(df_outliers)
```

Expected output:-

```
Original Data Frame:
A    B
0  1.0  5.0
1  2.0  NaN
2  NaN  7.0
3  4.0  8.0

Missing Values:
A      B
0  False  False
1  False   True
2   True  False
3  False  False
Data Frame after dropping rows with missing values:
A    B
0  1.0  5.0
3  4.0  8.0
Data Frame after filling missing values with 0:
A    B
0  1.0  5.0
1  2.0  0.0
2  0.0  7.0
3  4.0  8.0
Data Frame with Outliers:
A
0    1
1    2
2    3
3    4
4    5
5    6
6    7
7    8
8    9
9   10
10  20
Outliers:
A
10  20
Data Frame after replacing outliers with nearest non-outlier values:
A
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
6    7.0
7    8.0
8    9.0
9   10.0
10   8.5
>
```

Observed output:-

```
Original Data Frame:
A    B
0  1.0  5.0
1  2.0  NaN
2  NaN  7.0
3  4.0  8.0

Missing Values:
A      B
0  False  False
1  False   True
2   True  False
3  False  False
Data Frame after dropping rows with missing values:
A    B
0  1.0  5.0
3  4.0  8.0
Data Frame after filling missing values with 0:
A    B
0  1.0  5.0
1  2.0  0.0
2  0.0  7.0
3  4.0  8.0
Data Frame with Outliers:
A
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10
10   20
Outliers:
A
10   20
Data Frame after replacing outliers with nearest non-outlier values:
A
0     1.0
1     2.0
2     3.0
3     4.0
4     5.0
5     6.0
6     7.0
7     8.0
8     9.0
9    10.0
10    8.5
>
```

9)a)i)AIM :- Write a program for Splitting the data into groups using python

DESCRIPTION :-

In pandas, data can be split into groups using the groupby() function. This function groups rows based on a specified column or multiple columns and creates a GroupBy object. The GroupBy object can then be used to perform various aggregation functions such as sum, mean, max, min, and count, among others. The groupby() function can also be used with the apply() function to apply a custom function to each group. Overall, splitting data into groups using pandas is an essential technique that allows users to perform in-depth analysis and gain insights into the relationships between variables in a dataset.

PROGRAM :-

```
import itertools
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
groups = itertools.groupby(data, lambda x: (x-1)//3)
for key, group in groups:
print("Group : ".format(key+1, list(group)))
```

EXPECTED OUTPUT :-

```
Group 1: [1, 2, 3]
Group 2: [4, 5, 6]
Group 3: [7, 8, 9]
Group 4: [10]
```

OBSERVED OUTPUT :-

```
Group 1: [1, 2, 3]
Group 2: [4, 5, 6]
Group 3: [7, 8, 9]
Group 4: [10]
```

9)a)ii)AIM :- Write a program for Applying a function to each group individually using python

DESCRIPTION :-
In pandas, applying a function to each group individually can be done using the apply() function. The apply() function is used to apply a specified function to each group of a GroupBy object. The function can be a built-in function or a custom function created by the user. The apply() function can also be used with lambda functions for quick and simple operations. Overall, applying a function to each group individually using pandas is an essential technique that allows users to perform complex analysis and gain deeper insights into the relationships between variables in a dataset.

PROGRAM :-

```
import itertools
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def process_group(key, group):
print("Processing group {}: {}".format(key+1, list(group)))
groups = itertools.groupby(data, lambda x: (x-1)//3)
for key, group in groups:
process_group(key, group)
```

EXPECTED OUTPUT :-

```
Processing group 1: [1, 2, 3]
Processing group 2: [4, 5, 6]
Processing group 3: [7, 8, 9]
Processing group 4: [10]
```

OBSERVED OUTPUT :-

```
Processing group 1: [1, 2, 3]
Processing group 2: [4, 5, 6]
Processing group 3: [7, 8, 9]
Processing group 4: [10]
```

9)a)iii)AIM :- Write a program for Combining the result into a data structure using python

DESCRIPTION :-

In pandas, combining the results of multiple operations into a single data structure can be done using the concat(), merge(), and join() functions. The concat() function is used to combine DataFrames vertically or horizontally, while the merge() function is used to combine DataFrames based on a specified column or index. The join() function is used to join DataFrames based on a specified index. These functions allow users to combine data from different sources and perform more complex analysis and modeling. Overall, combining the results into a data structure using pandas is an essential technique that allows users to work with larger datasets and gain deeper insights into the data.

PROGRAM :-

```
import itertools
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def process_group(key, group):
return sum(group)
groups = itertools.groupby(data, lambda x: (x-1)//3)
results = [process_group(key, group) for key, group in groups]
print("Results:", results)
```

EXPECTED OUTPUT :-

```
Results: [6, 15, 24, 10]
```

OBSERVED OUTPUT :-

```
Results: [6, 15, 24, 10]
```

9)b)AIM :- Write a program for Pivot thable using python

DESCRIPTION :-

In pandas, a pivot table is a way to summarize and aggregate data in a DataFrame by grouping data according to multiple variables and calculating summary statistics for each group. The pivot_table() function in pandas is used to create a pivot table from a DataFrame. The pivot_table() function allows users to specify which variables to use for the rows, columns, and values in the pivot table. Users can also specify how to aggregate the data using functions such as sum(), mean(), count(), and others. Pivot tables are useful for analyzing and visualizing complex datasets and can provide insights into relationships between variables. Overall, pivot tables are an essential tool in pandas for data analysis and modeling.

PROGRAM :-

```
import pandas as pd
df = pd.DataFrame({
'Name': ['Alice', 'Bob', 'Charlie', 'Alice', 'Bob', 'Charlie'],
'Month': ['Jan', 'Jan', 'Jan', 'Feb', 'Feb', 'Feb'],
'Sales': [100, 200, 150, 300, 250, 200]
})
pivot_table = pd.pivot_table(df, values='Sales', index='Name', columns='Month')
print(pivot_table)
```

EXPECTED OUTPUT:-

```
Month     Feb  Jan
Name
Alice     300  100
Bob       250  200
Charlie   200  150
```

OBSERVED OUTPUT:- :

```
Month     Feb  Jan
Name
Alice     300  100
Bob       250  200
Charlie   200  150
```

9)c)AIM :- Write a program for Cross tab using python

DESCRIPTION :-

In pandas, a cross tabulation table or crosstab is a way to summarize and compare the frequency or count of two or more variables. The crosstab() function in pandas is used to create a cross tabulation table from a DataFrame. The crosstab() function allows users to specify the row and column variables to use in the table, as well as any additional options such as normalization or aggregation functions. Cross tabulation tables are useful for analyzing and comparing categorical data, identifying patterns and trends, and gaining insights into relationships between variables. Overall, crosstabs are an essential tool in pandas for data analysis and visualization.

PROGRAM :-

```
import pandas as pd
df = pd.DataFrame({
'Name': ['Alice', 'Bob', 'Charlie', 'Alice', 'Bob', 'Charlie'],
'Gender': ['F', 'M', 'M', 'F', 'M', 'F'],
'Sales': [100, 200, 150, 300, 250, 200]
})
cross_tab = pd.crosstab(df['Name'], df['Gender'],
values=df['Sales'], aggfunc='sum')
print(cross_tab)
```

EXPECTED OUTPUT :-

```
Month    Feb  Jan
Name
Alice    300  100
Bob      250  200
Charlie  200  150
```


OBSERVED OUTPUT :-

```
Month    Feb  Jan
Name
Alice    300  100
Bob      250  200
Charlie  200  150
```