MASTER OF COMPUTER APPLICATION

SEMESTER 1

DATA VISUALISATION

Unit 11

Data Cleaning and Preparation in Python

Table of Contents

1. INTRODUCTION

Dealing with missing data is a fundamental aspect of preprocessing in data analytics and machine learning, crucial for ensuring data quality and integrity. Pandas, a powerful Python library for data manipulation and analysis, provides several functionalities to handle missing data effectively. This discussion focuses on understanding None and NaN values used in Pandas to represent missing or null data, their differences, and how they are treated in data frames and series. Additionally, we explore Pandas functions such as isnull(), notnull(), dropna(), fillna(), replace(), and interpolate(), which are instrumental in identifying, removing, or imputing missing values in datasets, thus preparing data for further analysis or modelling tasks. We also learn how to integrate databases in Python.

1.1 Learning Objectives

After studying this unit, you will be able to:

- *Recognise the importance of addressing missing data.*
- *Comprehend the fundamental principles of data transformation and its role in preparing data for analysis.*
- *Implement data transformation methods.*
- *Demonstrate the process of automating data cleaning.*
- *Interpret the aspects of feature engineering.*
- *Demonstrate the integration of databases.*

2. DEALING WITH MISSING DATA

In Pandas, missing data is commonly represented using two primary values: `None` and `NaN`. These values are placeholders for missing or null data within a DataFrame or Series. Both `None` and `NaN` are treated as essentially interchangeable for indicating missing or null values, but they have some critical differences in terms of their data types and behaviour.

1. None:

- `None` is a Python singleton object often used to find missing data in Python code. It is of type `NoneType`.
- In Pandas, `None` represents missing data in an object or string data type (i.e., `dtype('object')`).
- When you use `None` in a Pandas Series or DataFrame, it will be converted to the `object` data type.
- `None` is incompatible with numeric data types, so you can't use it to represent missing values in numerical columns.

2. NaN (Not a Number):

- `NaN` is a particular floating-point value recognised by all systems that use the standard IEEE floating-point representation. It is used to represent missing or undefined numerical data.
- `NaN` is used for columns with numeric data types (e.g., float64 or int64).
- When you use `NaN` in a Pandas Series or DataFrame, it will be converted to the appropriate numeric data type (typically float64).
- `NaN` is compatible with numeric data, so you can use it to represent missing values in numerical columns.

Now, let's explore some of the Pandas functions that help you work with missing data:

1. isnull() and notnull():

- `isnull()` function: This function returns a Boolean mask indicating where values are missing. It returns `True` for positions where data is missing (i.e., it is `NaN` or `None`)

and `False` where data is not missing. You can use this mask to filter or manipulate the data based on the presence or absence of missing values.

- `notnull()` function: This function is the opposite of `isnull()`. It returns `True` for positions where data is not missing and `False` where data is missing. It helps identify non-missing data.

Checking for missing values using isnull()

To check null values in Pandas DataFrame, we use the isnull() function. This function returns a dataframe of Boolean values, which are True for NaN values.

Code #1:

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
      'Second Score': [30, 45, 56, np.nan],
      'Third Score':[np.nan, 40, 80, 98]}
 # creating a dataframe from list
df = pd.DataFrame(dict)
 # using isnull() function
df.isnull()
```

Output:

	First Score	Second Score	Third Score
0	False	False	True
1	False	False	False
2	True	False	False
3	False	True	False

Checking for missing values using notnull()

To check null values in Pandas Dataframe, we use notnull() function. This function returns a dataframe of Boolean values, which are False for NaN values.

Code #3:

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
      'Second Score': [30, 45, 56, np.nan],
      'Third Score':[np.nan, 40, 80, 98]}
 # creating a dataframe using dictionary
df = pd.DataFrame(dict)
 # using notnull() function
df.notnull()
```

Output:

	First Score	Second Score	Third Score
0	True	True	False
1	True	True	True
2	False	True	True
3	True	False	True

2. dropna():

- The `dropna()` function removes rows or columns with missing data. You can specify whether to drop rows or columns using the `axis` parameter, with `axis=0` for rows and `axis=1` for columns.
- The `thresh` parameter allows you to specify how many non-null values are required to keep a row or column. This is useful for selectively removing rows or columns with significant missing data while retaining others.

3. fillna():

- The `fillna()` function fills missing values with specified values or methods. You can pass a constant value (e.g., 0) or use methods like forward fill (`ffill`) or backward fill (`bfill`) to impute missing values.
- For example, if you have a time series dataset, you might use `fillna(method='ffill')` to fill missing values by carrying forward the previous valid value. This is particularly useful in cases where data has a logical order.

4. replace():

- The `replace()` function allows you to replace values in a DataFrame with specified values. This can replace specific values, including missing ones, with other values of your choice.
- You can use `replace()` to customise the way missing values are imputed by specifying a dictionary that maps the values you want to replace to their replacement values.

5. interpolate():

- The `interpolate()` function is beneficial for time series data or data with a logical order. It fills in missing values by interpolating between existing values. This can help to fill in missing data in numerical columns smoothly.
- The method of interpolation can be specified using the `method` parameter, such as linear interpolation (`method='linear'`) or polynomial interpolation (`method='polynomial'`).

Filling missing values using fillna(), replace() and interpolate()

To fill null values in a dataset, we use fillna(), replace() and interpolate() functions these function replace NaN values with some value of their own. All these functions help fill null

values in datasets of a DataFrame. Interpolate() function is basically used to fill NA values in the dataframe but uses various interpolation techniques to fill the missing values rather than hard-coding the value.

Code #1: Filling null values with a single value

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
      'Second Score': [30, 45, 56, np.nan],
      'Third Score':[np.nan, 40, 80, 98]}
 # creating a dataframe from dictionary
df = pd.DataFrame(dict)
 # filling missing value using fillna()
df.fillna(0)
```

Output:

	First Score	Second Score	Third Score
0	100.0	30.0	0.0
1	90.0	45.0	40.0
2	0.0	56.0	80.0
3	95.0	0.0	98.0

Code #2: Filling null values with the previous ones

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
```

```
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
      'Second Score': [30, 45, 56, np.nan],
      'Third Score':[np.nan, 40, 80, 98]}
 # creating a dataframe from dictionary
df = pd.DataFrame(dict)
 # filling a missing value with
# previous ones
df.fillna(method ='pad')
```

Output:

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	90.0	56.0	80.0
3	95.0	56.0	98.0

Code #3: Filling null value with the next ones

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
      'Second Score': [30, 45, 56, np.nan],
```

```
    'Third Score':[np.nan, 40, 80, 98]}
 # creating a dataframe from dictionary
df = pd.DataFrame(dict)
 # filling  null value using fillna() function
df.fillna(method ='bfill')
```

Output:

	First Score	Second Score	Third Score
0	100.0	30.0	40.0
1	90.0	45.0	40.0
2	95.0	56.0	80.0
3	95.0	NaN	98.0

Code #4: Filling null values in CSV File

```
# importing pandas package
import pandas as pd
   # making data frame from csv file
data = pd.read_csv("employees.csv")
 # Printing the first 10 to 24 rows of
# the data frame for visualisation
data[10:25]
```

Output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance

15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN
24	John	Male	7/1/1992	10:08 PM	97950	13.873	False	Client Services

Now we are going to fill all the null values in Gender column with "No Gender"

```
# importing pandas package

import pandas as pd

   # making data frame from csv file

data = pd.read_csv("employees.csv")

 # filling a null values using fillna()

data["Gender"].fillna("No Gender", inplace = True)

data
```

Output:

10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	No Gender	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	No Gender	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN
24	John	Male	7/1/1992	10:08 PM	97950	13.873	False	Client Services

Code #5: Filling a null values using replace() method

```
# importing pandas package
import pandas as pd
   # making data frame from csv file
data = pd.read_csv("employees.csv")
 # Printing the first 10 to 24 rows of
# the data frame for visualisation
data[10:25]
```

Output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	NaN
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	NaN	Male	6/14/2012	4:19 PM	125792	5.042	NaN	NaN

Now, we will replace all Nan values in the data frame with a -99 value.

```
# importing pandas package
import pandas as pd
   # making data frame from csv file
data = pd.read_csv("employees.csv")
```

```
   # will replace  Nan value in dataframe with value -99
data.replace(to_replace = np.nan, value = -99)
```

Output:

10	Louise	Female	8/12/1980	9:01 AM	63241	15.132	True	-99
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
15	Lillian	Female	6/5/2016	6:09 AM	59414	1.256	False	Product
16	Jeremy	Male	9/21/2010	5:56 AM	90370	7.369	False	Human Resources
17	Shawn	Male	12/7/1986	7:45 PM	111737	6.414	False	Product
18	Diana	Female	10/23/1981	10:27 AM	132940	19.082	False	Client Services
19	Donna	Female	7/22/2010	3:48 AM	81014	1.894	False	Product
20	Lois	-99	4/22/1995	7:18 PM	64714	4.934	True	Legal
21	Matthew	Male	9/5/1995	2:12 AM	100612	13.645	False	Marketing
22	Joshua	-99	3/8/2012	1:58 AM	90816	18.816	True	Client Services
23	-99	Male	6/14/2012	4:19 PM	125792	5.042	-99	-99
24	John	Male	7/1/1992	10:08 PM	97950	13.873	False	Client Services

Code #6: Using interpolate() function to fill the missing values using the linear method.

```
# importing pandas as pd
import pandas as pd
   # Creating the dataframe
df = pd.DataFrame({"A":[12, 4, 5, None, 1],
                   "B":[None, 2, 54, 3, None],
                   "C":[20, 16, None, 3, 8],
                   "D":[14, 3, None, None, 6]})
   # Print the dataframe
df
```

Output:

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	NaN	NaN
3	NaN	3.0	3.0	NaN
4	1.0	NaN	8.0	6.0

Let's interpolate the missing values using the Linear method. The Linear method ignores the index and treats the values equally spaced.

```
# to interpolate the missing values
df.interpolate(method ='linear', limit_direction ='forward')
```

Output:

	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	9.5	4.0
3	3.0	3.0	3.0	5.0
4	1.0	3.0	8.0	6.0

As we can see in the output, values in the first row could not get filled as the direction of filling of values is forward and no previous value could have been used in interpolation.

Dropping missing values using dropna()

In order to drop a null value from a dataframe, we used dropna() function. This function drops Rows/Columns of datasets with Null values differently.

Code #1: Dropping rows with at least 1 null value.

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
```

```
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
       'Second Score': [30, np.nan, 45, 56],
       'Third Score':[52, 40, 80, 98],
       'Fourth Score':[np.nan, np.nan, np.nan, 65]}
 # creating a dataframe from dictionary
df = pd.DataFrame(dict)
df
```

Output:

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52	NaN
1	90.0	NaN	40	NaN
2	NaN	45.0	80	NaN
3	95.0	56.0	98	65.0

Now we drop rows with at least one Nan value (Null value)

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
       'Second Score': [30, np.nan, 45, 56],
       'Third Score':[52, 40, 80, 98],
       'Fourth Score':[np.nan, np.nan, np.nan, 65]}
 # creating a dataframe from dictionary
df = pd.DataFrame(dict)
 # using dropna() function
df.dropna()
```

Output:

	First Score	Second Score	Third Score	Fourth Score
3	95.0	56.0	98	65.0

Code #2: Dropping rows if all values in that row are missing.

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
 # dictionary of lists
dict = {'First Score':[100, np.nan, np.nan, 95],
       'Second Score': [30, np.nan, 45, 56],
       'Third Score':[52, np.nan, 80, 98],
        'Fourth Score':[np.nan, np.nan, np.nan, 65]}
 # creating a dataframe from dictionary
df = pd.DataFrame(dict)
df
```

Output:

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
1	NaN	NaN	NaN	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

Now we drop rows whose all data is missing or contain null values(NaN)

```
# importing pandas as pd
import pandas as pd
 # importing numpy as np
import numpy as np
```

```
  # dictionary of lists
dict = {'First Score':[100, np.nan, np.nan, 95],
        'Second Score': [30, np.nan, 45, 56],
        'Third Score':[52, np.nan, 80, 98],
        'Fourth Score':[np.nan, np.nan, np.nan, 65]}
  df = pd.DataFrame(dict)
  # using dropna() function
df.dropna(how = 'all')
```

Output:

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

Code #3: Dropping columns with at least 1 null value.

```
# importing pandas as pd
import pandas as pd
  # importing numpy as np
import numpy as np
  # dictionary of lists
dict = {'First Score':[100, np.nan, np.nan, 95],
        'Second Score': [30, np.nan, 45, 56],
        'Third Score':[52, np.nan, 80, 98],
        'Fourth Score':[60, 67, 68, 65]}
  # creating a dataframe from dictionary
df = pd.DataFrame(dict)
df
```

Output:

	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	60
1	NaN	NaN	NaN	67
2	NaN	45.0	80.0	68
3	95.0	56.0	98.0	65

Now we drop a column which have at least 1 missing value

```
# importing pandas as pd
import pandas as pd
  # importing numpy as np
import numpy as np
  # dictionary of lists
dict = {'First Score':[100, np.nan, np.nan, 95],
       'Second Score': [30, np.nan, 45, 56],
       'Third Score':[52, np.nan, 80, 98],
       'Fourth Score':[60, 67, 68, 65]}
# creating a dataframe from dictionary
df = pd.DataFrame(dict)
  # using dropna() function
df.dropna(axis = 1)
```

Output:

	Fourth Score
0	60
1	67
2	68
3	65

Code #4: Dropping Rows with at least 1 null value in CSV file

```
# importing pandas module
import pandas as pd
   # making data frame from csv file
data = pd.read_csv("employees.csv")
   # making new data frame with dropped NA values
new_data = data.dropna(axis = 0, how ='any')
new_data
```

Output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	8/6/1993	12:42 PM	97308	6.945	True	Marketing
2	Maria	Female	4/23/1993	11:17 AM	130590	11.858	False	Finance
3	Jerry	Male	3/4/2005	1:00 PM	138705	9.340	True	Finance
4	Larry	Male	1/24/1998	4:47 PM	101004	1.389	True	Client Services
5	Dennis	Male	4/18/1987	1:35 AM	115163	10.125	False	Legal
6	Ruby	Female	8/17/1987	4:20 PM	65476	10.012	True	Product
8	Angela	Female	11/22/2005	6:29 AM	95570	18.523	True	Engineering
9	Frances	Female	8/8/2002	6:51 AM	139852	7.524	True	Business Development
11	Julie	Female	10/26/1997	3:19 PM	102508	12.637	True	Legal
12	Brandon	Male	12/1/1980	1:08 AM	112807	17.492	True	Human Resources
13	Gary	Male	1/27/2008	11:40 PM	109831	5.831	False	Sales
14	Kimberly	Female	1/14/1999	7:13 AM	41426	14.543	True	Finance
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
993	Tina	Female	5/15/1997	3:53 PM	56450	19.040	True	Engineering
994	George	Male	6/21/2013	5:47 PM	98874	4.479	True	Marketing
996	Phillip	Male	1/31/1984	6:30 AM	42392	19.675	False	Finance
997	Russell	Male	5/20/2013	12:39 PM	96914	1.421	False	Product
998	Larry	Male	4/20/2013	4:45 PM	60500	11.985	False	Business Development
999	Albert	Male	5/15/2012	6:24 PM	129949	10.169	True	Sales

764 rows × 8 columns

Now we compare sizes of data frames so that we can come to know how many rows had at least 1 Null value

```
print("Old data frame length:", len(data))
print("New data frame length:", len(new_data))
print("Number of rows with at least 1 NA value: ", (len(data)-len(new_data)))
```

Output:

Old data frame length: 1000

New data frame length: 764

Number of rows with at least 1 NA value: 236

Since the difference is 236, there were 236 rows with at least 1 Null value in any column.

3. DATA TRANSFORMATION

Data transformation in Python refers to modifying, reorganising, or converting data to make it more suitable for analysis, visualisation, or modelling. It involves various operations that can help you improve data quality, structure, and compatibility with the specific tasks or algorithms you plan to apply. Data transformation is an essential step in the data preprocessing pipeline and is commonly performed using libraries such as NumPy, Pandas, and sci-kit-learn. Here are some everyday data transformation tasks in Python:

3.1. Reshaping and Restructuring Data

Reshaping and restructuring data are data manipulation processes commonly performed in data analysis and preparation. They involve changing the format or layout of your data to make it more suitable for analysis or to meet the requirements of a specific task. These processes often convert data from one structure to another, such as transforming broad data to long data or vice versa.

1. Reshaping Data:

Reshaping data typically involves changing the structure of your dataset to make it more amenable to analysis or modelling. Common reshaping operations include:

1. Pivoting Data (Wide to Long):

- Pivoting involves converting a wide-format dataset, where multiple variables are stored in separate columns, into a long-format dataset where the data is organised with fewer columns and more rows.
- For example, you might pivot a table where each year is a column into a long format where each row represents a year and a variable.

2. Melting Data:

- Melting is similar to pivoting, where you transform wide data into long data. In this process, you gather multiple columns into key-value pairs, making it more efficient for analysis.

3. Transposing Data:

- Transposing swaps rows and columns in your dataset, which is useful for various data manipulation tasks and reshaping operations.

Certainly! Let's explain these reshaping data operations using Python examples. We'll use the `pandas` library, a popular tool for data manipulation, to demonstrate these concepts.

Assuming you have a sample dataset in a tabular format like this:

```
import pandas as pd
data = {
   'Year': [2010, 2011, 2012],
   'Sales_A': [100, 120, 130],
   'Sales_B': [90, 110, 140],
}
df = pd.DataFrame(data)
print(df)
```

This dataset is in a wide format, where each year is a separate column, and sales data for different products are stored in these columns.

To pivot the data from a wide format to a long format, you can use the `pd.melt()` function:

melted_df = pd.melt(df, id_vars='Year', var_name='Product', value_name='Sales')

print(melted_df)

The resulting `melted_df` will look like this:

	Year	Product	Sales
0	2010	Sales A	100
1	2011	Sales A	120
2	2012	Sales A	130
3	2010	Sales B	90
4	2011	Sales B	110
5	2012	Sales B	140

Now, the data is organised in a long format with fewer columns and more rows.

2. Melting Data:

The previous example demonstrates melting data as part of the pivot operation. In the `pd.melt()` function, you specify which columns should remain as identifiers (in this case, 'Year'), and all other columns are melted into key-value pairs ('Product' and 'Sales').

3. Transposing Data:

Transposing swaps rows and columns in your dataset. You can use the `.T` attribute of a DataFrame to achieve this:

```
transposed_df = df.T
print(transposed_df)
```

The transposed DataFrame, `transposed_df`, will look like this:

	0	1	2
Year	2010	2011	2012
Sales A	100	120	130
Sales B	90	110	140

Here, the rows have become columns, and the columns have become rows.

These examples demonstrate pivoting, melting, and transposing operations using Python and the `pandas` library. These operations are essential for reshaping data to suit your analysis and modelling needs.

3.2. Restructuring Data

Restructuring data involves organising your data in a way that is better suited for analysis, visualisation, or modelling. Common restructuring operations include:

1. Aggregating Data:

- Aggregation involves combining multiple rows of data to create summary statistics or observations. For example, calculating the average sales per month from daily sales data.

2. Grouping Data:

- Grouping data involves creating groups or subsets based on specific criteria (e.g., grouping data by category or location). Grouping is often a prerequisite for aggregation.

3. Splitting Data:

- Splitting data is the opposite of aggregation. It involves dividing data into smaller subsets or separate datasets, which can be helpful for parallel processing or dividing data for different purposes.

4. Filtering Data:

- Filtering data involves selecting or excluding rows based on specific conditions. Filtering is used to focus on a subset of your data that is relevant to your analysis or task.

5. Merging or Joining Data:

- Merging or joining data involves combining multiple datasets based on common keys or identifiers, often used to combine information from different sources into a single dataset.

Data restructuring and manipulation operations in detail with Python examples using the `pandas` library.

1. Aggregating Data:

Aggregating data involves combining multiple rows of data to create summary statistics or observations. For example, the average sales per month can be calculated from daily sales data.

```
import pandas as pd
# Sample daily sales data
data = {
   'Date': ['2023-01-01', '2023-01-02', '2023-01-03', '2023-02-01', '2023-02-02'],
   'Sales': [100, 120, 90, 80, 110]
}
df = pd.DataFrame(data)
# Convert 'Date' column to datetime
df['Date'] = pd.to_datetime(df['Date'])
# Group data by month and calculate the average sales
monthly_avg_sales = df.groupby(df['Date'].dt.to_period('M')).agg({'Sales': 'mean'})
print(monthly_avg_sales)
```

Output:

```
              Sales
Date
2023-01  103.333333
2023-02   95.000000
```

2. Grouping Data:

Grouping data involves creating groups or subsets based on specific criteria, such as grouping data by category or location.

```
import pandas as pd
# Sample data
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Value': [10, 20, 15, 25, 30]
}
df = pd.DataFrame(data)
# Group data by 'Category'
grouped = df.groupby('Category')
# Calculate the sum for each group
group_sum = grouped['Value'].sum()
print(group_sum)
```

Output:

```
Category
A    55
B    45
Name: Value, dtype: int64
```

3. Splitting Data:

Splitting data involves dividing data into smaller subsets or separate datasets. Here's a simple example where we split a dataset into two subsets based on a condition:

```
import pandas as pd
# Sample data
data = {
```

```
    'Category': ['A', 'B', 'A', 'B', 'A'],
    'Value': [10, 20, 15, 25, 30]
}
df = pd.DataFrame(data)
# Split data into two subsets based on 'Category'
subset_A = df[df['Category'] == 'A']
subset_B = df[df['Category'] == 'B']
print("Subset A:")
print(subset_A)
print("Subset B:")
print(subset_B)
```

Output:

```
Subset A:
  Category  Value
0        A     10
2        A     15
4        A     30
Subset B:
  Category  Value
1        B     20
3        B     25
```

4. Merging or Joining Data:

Merging or joining data involves combining datasets based on common keys or identifiers. Here's an example of merging two DataFrames:

```
import pandas as pd
# Sample data
data1 = {
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
}
data2 = {
    'ID': [2, 3, 4],
    'Age': [25, 30, 22]
```

```
}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)
# Merge data based on the 'ID' column
merged_data = pd.merge(df1, df2, on='ID', how='inner')
print(merged_data)
```

In this example, we merge two DataFrames based on the common 'ID' column.

Output:

```
   ID     Name  Age
0   2      Bob   25
1   3  Charlie   30
```

Both reshaping and restructuring are essential techniques in data preparation and data analysis. The choice of operation depends on your specific data needs and the requirements of your analysis or modelling tasks. These operations can help make your data more accessible and meaningful, enabling you to derive insights and make informed decisions.

4. APPLYING FILTERS

4.1. Filtering Data in Python

Filtering data is a fundamental data manipulation operation that involves selecting or excluding rows from a dataset based on specific conditions or criteria. It allows you to focus on a subset of your data that meets your criteria or is relevant to your analysis. Filtering is often used in data analysis, cleaning, and preprocessing to extract the data that is of interest for further examination or analysis. Here's a detailed explanation of filtering data:

Filtering Data in Python with pandas:

In Python, the `pandas` library is commonly used for data manipulation, including filtering data. The `pandas` library provides a flexible and efficient way to filter rows based on conditions.

Basic Syntax:

filtered_data = dataframe[condition]

- `dataframe`: The DataFrame or dataset you want to filter.
- `condition`: A Boolean or expression defining the filtering criteria. Rows that satisfy this condition will be included in the resulting filtered dataset.

Example:

Let's consider a simple example with a dataset of student grades:

```
import pandas as pd
data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David'],
   'Math Score': [85, 92, 78, 90],
   'Science Score': [89, 88, 92, 85]
}
df = pd.DataFrame(data)
# Filter students with Math scores greater than or equal to 90
high_math_scores = df[df['Math Score'] >= 90]
print(high_math_scores)
```

In this example, we filter the dataset to select students with Math scores greater than or equal to 90. The resulting `high_math_scores` DataFrame will include rows only for students who meet this condition.

Output:

	Name	Math Score	Science Score
1	Bob	92	88
3	David	90	85

Detailed Steps in Filtering Data:

1. **Define the Condition:** Identify the condition or criteria that you want to use for filtering. This condition should be expressed as a Boolean expression that evaluates to `True` for rows you want to keep and `False` for rows you wish to exclude.
2. **Apply the Condition:** Use the condition inside square brackets `[]` to filter the data. The condition is applied to each row in the DataFrame, and rows that satisfy the condition are included in the filtered dataset.
3. **Result:** The filtered data is stored in a new DataFrame or variable, which can be used for further analysis, visualisation, or other data-related tasks.

Common Use Cases for Data Filtering:

- **Selecting Rows:** You can filter data to select specific rows that meet your criteria, such as selecting students with high scores, customers who made a purchase, or products with low inventory.
- **Excluding Rows:** Filtering can also exclude rows that do not meet your criteria, such as excluding missing data or outliers from your analysis.
- **Complex Conditions:** You can create complex conditions by combining multiple criteria using logical operators (e.g., `&` for "and," `|` for "or"). This allows you to express more intricate filtering conditions.
- **Chaining Filters:** You can chain multiple filter conditions to apply a sequence of filters to the data. This allows for more fine-grained data extraction.

Filtering data is a fundamental technique in data analysis and cleaning, enabling you to work with subsets of your data relevant to your research or analysis goals. It's a powerful tool for extracting valuable insights from your datasets.

To demonstrate this, we'll use a practical example using a dataset of student exam scores.

Example: Filtering Data in Python with pandas

Suppose we have a dataset of student exam scores like this:

```
import pandas as pd
data = {
   'StudentID': [1, 2, 3, 4, 5],
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
   'Math_Score': [85, 92, 78, 90, 88],
   'Science_Score': [89, 88, 92, 85, 91]
}
df = pd.DataFrame(data)
print(df)
```

This dataset contains columns for student IDs, names, math scores, and science scores.

Output:

	StudentID	Name	Math_Score	Science_Score
0	1	Alice	85	89
1	2	Bob	92	88
2	3	Charlie	78	92
3	4	David	90	85
4	5	Eva	88	91

Filtering by Math Score:

Let's say we want to filter the dataset to include only students who scored 90 or above in math. Here's how you can do it:

```
# Filter data to select students with Math scores >= 90
high_math_scores = df[df['Math_Score'] >= 90]
print(high_math_scores)
```

The result will be a new DataFrame, `high_math_scores`, containing only the students who scored 90 or above in math.

Output:

```
   StudentID     Name  Math_Score  Science_Score
0          1    Alice          85             89
1          2      Bob          92             88
2          3  Charlie          78             92
3          4    David          90             85
4          5      Eva          88             91
   StudentID   Name  Math_Score  Science_Score
1          2    Bob          92             88
3          4  David          90             85
```

Filtering by Multiple Conditions:

You can filter data based on multiple conditions using logical operators (`&` for "and," `|` for "or"). For instance, let's filter the students who scored 90 or above in math and 90 or above in science:

```
# Filter data to select students with Math scores and Science scores both >= 90
high_math_and_science = df[(df['Math_Score'] >= 90) & (df['Science_Score'] >= 90)]
print(high_math_and_science)
```

This code selects students who scored 90 or above in both math and science.

Output:

```
Empty DataFrame
Columns: [StudentID, Name, Math_Score, Science_Score]
Index: []
```

Filtering by Name:

You can also filter data based on text data, like names. For example, to select students whose names start with the letter 'A', you can use the `str.startswith()` method:

```
# Filter data to select students with names starting with 'A'
names_starting_with_A = df[df['Name'].str.startswith('A')]
print(names_starting_with_A)
```

This code filters students whose names start with 'A'.

Output:

```
   StudentID   Name  Math_Score  Science_Score
0          1  Alice          85             89
```

Filtering with Multiple Criteria:

You can create more complex filters by combining multiple conditions. For instance, let's filter students who scored 90 or above in math and have names starting with 'A':

```
# Filter data to select students with Math scores >= 90 and names starting with 'A'
complex_filter = df[(df['Math_Score'] >= 90) & (df['Name'].str.startswith('A'))]
print(complex_filter)
```

This code combines a numerical condition with a text-based condition.

Output:

```
Empty DataFrame
Columns: [StudentID, Name, Math_Score, Science_Score]
Index: []
```

4.2. Querying and Subsetting

Querying and subsetting are fundamental data manipulation operations used in data analysis to extract specific portions of data from a dataset based on certain conditions or criteria. These operations help you focus on relevant data for further analysis. In this explanation, I'll provide details and examples for querying and subsetting data using Python and the `pandas` library.

Querying Data:

Querying data selects rows from a dataset that meet specific conditions or criteria. You can think of it as a way to filter data based on logical conditions.

Basic Syntax:

query_result = dataframe[condition]

- **`dataframe`:** The DataFrame or dataset you want to query.
- **`condition`:** A Boolean or expression specifying the filtering criteria. Rows that satisfy this condition are included in the query result.

Example: Querying Data in Python

Suppose we have a dataset of sales transactions:

```
import pandas as pd
data = {
   'Product': ['A', 'B', 'A', 'C', 'B'],
   'Sales': [100, 120, 90, 80, 110]
}
df = pd.DataFrame(data)
# Query data to select rows where Sales are greater than 100
high_sales = df[df['Sales'] > 100]
print(high_sales)
```

In this example, the query selects rows where sales are greater than 100.

Output:

	Product	Sales
1	B	120
4	B	110

Subsetting Data:

Subsetting data is selecting specific columns or variables from a dataset. It allows you to extract only the variables of interest for your analysis, leaving out the rest.

Basic Syntax:

subset = dataframe[['column1', 'column2', ...]]

- **`dataframe`:** The DataFrame or dataset you want to subset.
- **`['column1', 'column2', ...]`:** A list of column names you want to include in the subset.

Example: Subsetting Data in Python

Let's continue with the same sales dataset:

```
# Subset data to select only the 'Product' column
product_subset = df[['Product']]
print(product_subset)
```

Output:

```
  Product
0       A
1       B
2       A
3       C
4       B
```

Combined Querying and Subsetting:

You can combine querying and subsetting to filter rows based on specific conditions and then select particular columns from the filtered rows.

Example: Combined Querying and Subsetting in Python

Suppose you want to select products with sales greater than 100 and only retain the 'Product' column:

```
# Combined Query and Subset: Select 'Product' where Sales > 100
filtered_and_subsetting = df[df['Sales'] > 100][['Product']]
print(filtered_and_subsetting)
```

In this example, you first query rows with sales > 100 and then subset the result to include only the 'Product' column.

Output:

```
  Product
0       A
1       B
2       A
3       C
4       B
  Product
1       B
4       B
```

These examples illustrate how to use querying and sub-setting in Python with `pandas` for data manipulation. These operations are essential for selecting, filtering, and organising data to perform more focused and specific analyses.

5. AUTOMATING DATA CLEANING

Automating data cleaning is a process that streamlines data preparation for analysis, ensuring that it is accurate, consistent, and ready for use. This is especially important in data analytics and machine learning projects, where data quality directly impacts outcomes' accuracy. Automating these tasks can save time, reduce errors, and allow data scientists to focus on higher-level analysis.

Critical Concepts of Automating Data Cleaning

1. **Data Cleaning:** The process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database.
2. **Automation:** Using software tools or scripts to perform tasks without human intervention increases efficiency and consistency.

Benefits of Automating Data Cleaning

- **Efficiency:** Automated processes can run 24/7, processing large volumes of data faster than manual methods.
- **Accuracy:** Reduces human error, ensuring data quality and consistency.
- **Scalability:** Easily handles increasing volumes of data without the need to increase resources linearly.
- **Focus on Analysis:** Frees up data professionals to concentrate on extracting insights rather than cleaning data.

Tools and Technologies

Various tools and libraries are available for automating data cleaning, especially within the Python ecosystem, such as Pandas for data manipulation, NumPy for numerical data, and more specialised libraries like DataCleaner and Pyjanitor.

Example:

There are missing values, duplicates, and inconsistent email formats. The dataset used here is the data.csv file with columns ID, Name, Age, and Email.

```
import pandas as pd
# Load the dataset
df = pd.read_csv('data.csv')
```

```
# 1. Remove duplicates
df = df.drop_duplicates()
# 2. Fill missing values for 'Age'
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Replace missing ages with the mean age
# 3. Standardise Email format and handle NaN values in 'Email'
df['Email'] = df['Email'].str.lower()
# 4. Remove rows with invalid emails (simple validation), handling NaN values in the condition
df = df[df['Email'].str.contains('@', na=False)]  # The na=False parameter treats NaN values as False in the condition
# Save the cleaned data
df.to_csv('cleaned_data.csv', index=False)


print("Data cleaning completed.")
```

Output:

```
Data cleaning completed.
```

6. FEATURE ENGINEERING

Feature Engineering is a crucial step in the data preprocessing phase of a machine learning pipeline, involving the creation, modification, or selection of features that improve the performance of a model. It's essentially about transforming raw data into a more suitable dataset for modelling. Through feature engineering, data scientists can leverage domain knowledge to extract more information from the data, making it possible for algorithms to discern patterns or insights they otherwise might miss.

Importance of Feature Engineering

Improves Model Accuracy: Properly engineered features can significantly improve the accuracy of predictive models by providing them with relevant information in an easily digestible form.

Reduces Model Complexity: Feature engineering can reduce model complexity by capturing essential information in fewer features, leading to faster training times and less overfitting.

Enhances Model Interpretability: Well-crafted features can make models more interpretable, helping stakeholders understand the factors driving predictions.

Creating New Data Points for Better Insights

Creating new data points, often referred to as feature construction or generation, involves creating new features from existing ones, providing additional insights into the dataset. This process can unveil relationships between variables that weren't initially apparent, improving model performance by adding valuable information.

Techniques for Creating New Data Points

- **Combining Features:** Mathematical operations (addition, subtraction, multiplication, division) can combine two or more features to create a new one that captures their interaction or relationship.

  Example: In real estate, combining 'total_rooms' and 'total_bedrooms' into a new feature ', bedrooms_per_room', can provide more nuanced insight into house size and layout preferences.

- **Binning:** Converting continuous variables into categorical variables (bins) can sometimes improve a model's ability to capture non-linear relationships.

  Example: Age groups (e.g., 0-18, 19-35, etc.) instead of continuous age values can better capture demographic trends in data.
- **Extraction from Date/Time:** Dates and times can be decomposed into multiple features like year, month, day, day of the week, and time of day, each of which might carry predictive power.

  Example: E-commerce sales might peak on certain days of the week or times of the day, making these extracted features valuable for predicting sales volumes.
- **Text Data Transformation:** Natural Language Processing (NLP) techniques can transform text into features like sentiment scores, topic classifications, or term frequency-inverse document frequency (TF-IDF) vectors.

  Example: Customer reviews can be transformed into sentiment scores, indicating overall customer satisfaction.
- **Geospatial Feature Creation:** When dealing with location data, new features can be derived from geographical coordinates, such as distances from points of interest, or encoded through techniques like geohashing.

  Example: Distance from a property to the nearest metro station might significantly impact its price.
- **Polynomial Features:** Generating polynomial and interaction features from existing data can help capture the interaction between variables, especially for models that assume linear relationships between features.

  Example: If studying the effect of education level and work experience on salary, an interaction feature might capture the combined impact of these variables more effectively than considering them separately.

Challenges and Considerations

While feature engineering can dramatically enhance model performance, avoiding pitfalls like overfitting, where the model learns noise in the training data too well, reducing its generalisation to new data is essential. Balancing the creation of insightful features with the risk of overfitting requires careful consideration and domain knowledge. Furthermore, the

process can be time-consuming and requires iterative experimentation to identify the most effective features.

Example:

```
import pandas as pd
import numpy as np
# Sample dataset creation
data = {
   'customer_id': range(1, 6),
   'age': [25, 45, 35, 55, 20],
   'annual_income': [30000, 60000, 45000, 80000, 20000],
   'total_transactions': [12, 15, 10, 8, 20],
   'total_spent': [1200, 1500, 1000, 1600, 2000],
   'signup_date': ['2019-06-01', '2018-07-15', '2019-08-09', '2017-06-24', '2020-03-02']
}
df = pd.DataFrame(data)
# Convert 'signup_date' to datetime
df['signup_date'] = pd.to_datetime(df['signup_date'])
# Feature Engineering
# 1. Spending per transaction
df['spending_per_transaction'] = df['total_spent'] / df['total_transactions']
# 2. Customer tenure (in years)
df['customer_tenure'] = (pd.Timestamp('now') - df['signup_date']).dt.days / 365.25
# 3. Age group
df['age_group'] = pd.cut(df['age'], bins=[0, 24, 34, 44, 54, 64, 100], labels=['0-24', '25-34', '35-44', '45-54', '55-64', '65+'])
# 4. Income to spending ratio
df['income_to_spending_ratio'] = df['annual_income'] / df['total_spent']
# Display the DataFrame with new features
print(df)
```

Output:

```
   customer_id  age  annual_income  total_transactions  total_spent  \
0            1   25          30000                  12         1200
1            2   45          60000                  15         1500
2            3   35          45000                  10         1000
3            4   55          80000                   8         1600
4            5   20          20000                  20         2000

  signup_date  spending_per_transaction  customer_tenure age_group  \
0  2019-06-01                     100.0         4.804928     25-34
1  2018-07-15                     100.0         5.683778     45-54
2  2019-08-09                     100.0         4.616016     35-44
3  2017-06-24                     200.0         6.740589     55-64
4  2020-03-02                     100.0         4.052019      0-24

   income_to_spending_ratio
0                      25.0
1                      40.0
2                      45.0
3                      50.0
4                      10.0
```

7. INTEGRATION WITH DATABASES

In the realm of software development and data science, integrating applications with databases is a critical operation. This integration enables applications to query, manipulate, and manage data efficiently. Databases, whether SQL (like MySQL, PostgreSQL, SQLite) or NoSQL (like MongoDB, Cassandra), serve as the backbone for storing, retrieving, and organising data in a structured way.

Importance of Database Integration

- **Data Persistence:** Ensures data is stored permanently in databases, allowing for retrieval and manipulation as needed.
- **Scalability:** Databases are designed to handle large volumes of data, making them essential for scaling applications.
- **Data Integrity:** Provides mechanisms to maintain accuracy and consistency of data through constraints, transactions, and rollback capabilities.
- **Security:** Offers robust security features to protect data, including access controls and encryption.

Connecting Python to Databases

Python, with its simplicity and vast ecosystem of libraries, is a popular choice for interacting with databases. The standard way to connect Python to a database is through a database driver or an ORM (Object Relational Mapping) library. While drivers like PyMySQL (for MySQL) and psycopg2 (for PostgreSQL) provide low-level access to execute SQL queries, ORMs like SQLAlchemy offer a high-level abstraction over these databases, allowing developers to interact with the database using Python objects instead of writing SQL queries directly.

Steps to Connect Python to a Database:

- **Install Database Driver/ORM Library:** This is typically done via pip. For example, to install SQLAlchemy, you would run pip install SQLAlchemy.
- **Create Database Connection:** Establish a connection to the database using connection parameters like host, database name, username, and password.
- **Execute Queries:** Once connected, you can execute SQL queries or use ORM methods to interact with the database.

- **Close Connection:** After the operations are complete, closing the database connection is essential to free up resources.

Simple Python Code Example: Connecting to SQLite Database

SQLite is a lightweight, disk-based database that doesn't require a separate server process. Python's standard library includes support for SQLite, making it an excellent choice for demonstrations.

```
import sqlite3
# Create a new SQLite database (or connect to an existing one)
conn = sqlite3.connect('example.db')
# Create a cursor object using the cursor() method
cursor = conn.cursor()
# Create table
cursor.execute('''CREATE TABLE IF NOT EXISTS users
             (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')
# Insert a row of data
cursor.execute("INSERT INTO users (name, age) VALUES ('Alice', 30)")
# Save (commit) the changes
conn.commit()
# Query the database
cursor.execute("SELECT * FROM users")
print(cursor.fetchall())
# Close the connection when done
conn.close()
```

Output:

```
[(1, 'Alice', 30)]
```

In this example, we connect to an SQLite database called example.db, create a table called users, insert a single row of data into it, query all the data from the table, and finally close

the connection. This basic workflow illustrates how Python can interact with databases to perform CRUD (Create, Read, Update, Delete) operations.

8. SUMMARY

The exploration of handling missing data in Pandas demonstrated the distinction between None and NaN values, which serve as placeholders for absent information in a dataset. While None is utilised for missing data in object data types, NaN is employed for numeric data types, each with specific behaviors in Pandas Series and DataFrames. We delved into functions like isnull() and notnull() for identifying missing values, dropna() for removing them, and fillna(), replace(), and interpolate() for filling in missing data. The concepts of checking, removing, and imputing missing values were illustrated through practical Python examples, emphasising the importance of cleaning data as a precursor to practical data analysis and model building. This knowledge equips practitioners to handle missing data efficiently, ensuring data reliability and robustness in analytical outcomes.

9. QUESTIONS

Self-Assessment Questions

1. What is the primary challenge associated with handling missing values in data analysis?
2. In Python, which library is commonly used to manipulate and preprocess data?
3. Name one common imputation technique for handling missing values in a dataset.
4. What is the purpose of reshaping and restructuring data in data analysis?
5. How can you convert wide-format data into long-format data using pandas in Python?
6. What is aggregation, and how is it used in data analysis?
7. Provide an example of an aggregation function that can be applied to a dataset.
8. Explain the significance of filtering data in Python during data analysis.
9. What is the primary difference between querying and sub setting data in Python?
10. How can you select rows from a dataset in Python using a filtering condition?
11. When would you use the replace() function in the context of missing data?
12. Which method helps fill in missing values in time series data?

Terminal questions

1. Explain the importance of addressing missing values in a dataset.
2. Describe the common methods for detecting and handling missing data, providing examples for each.
3. Discuss the concept of data imputation in data analysis.
4. Explain the significance of reshaping and restructuring data in the context of data analysis.
5. Describe the role of aggregation and grouping in data analysis.
6. Describe the process of filtering data in Python.
7. Explain the fundamental differences between querying and sub setting data in Python.
8. What does the dropna() function do, and how can it be customised?
9. Explain the use of the fillna() function with an example.
10. How does the databases are integrated in Python?

10. ANSWERS

Self-Assessment Questions

1. The primary challenge is that missing values can lead to biased or incomplete analyses, potentially resulting in erroneous conclusions.
2. `pandas` is a popular Python library used for data manipulation and preprocessing tasks.
3. Mean imputation is a common technique where missing values are replaced with the mean of the available values in that column.
4. Reshaping and restructuring data are used to change the format or layout of datasets to make them more suitable for analysis, visualisation, or modelling.
5. You can use the `pd.melt()` function in `pandas` to convert wide-format data to long-format data.
6. Aggregation involves summarising data to create new features or generate summary statistics. It's used to gain insights into the data and reduce data complexity.
7. The `sum()` function can be used to calculate the sum of values in a dataset, which is a common aggregation operation.
8. Filtering data allows you to focus on specific subsets of data that meet certain conditions, making it essential for targeted analysis.
9. Querying is selecting rows based on specific conditions, whereas sub setting is selecting specific columns or variables from the dataset.
10. You can use conditional statements within square brackets (`[]`) to select rows from a dataset in Python based on specific filtering conditions.
11. To replace specific values in a DataFrame, including missing values
12. interpolate()

Terminal Questions

1. Refer section 3.1
2. Refer section 3.1
3. Refer section 3.2
4. Refer section 4
5. Refer section 4

6. Refer section 5
7. Refer section 5
8. Refer section 3
9. Refer section 3
10. Refer section 7