2. a.How do you handle the Missing data?

NA handling methods

Argument	Description
dropna	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
fillna	Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.
isnull	Return like-type object containing boolean values indicating which values are missing / NA.
notnull	Negation of isnull.

Using dropna

```
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
   'A': [1, 2, np.nan],
   'B': [5, np.nan, np.nan],
   'C': [1, 2, 3]
})
# Use dropna
df_after_dropna = df.dropna()
print(df_after_dropna)
```

Output:

```
     A    B  C
0  1.0  5.0  1
```

Using fillna

```
# Create a simple dataframe
df = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [5, np.nan, np.nan],
    'C': [1, 2, 3]
})
# Use fillna
df_after_fillna = df.fillna(value=0)

print("DataFrame after using fillna:")
print(df_after_fillna)
```

OUTPUT


```
DataFrame after using fillna:
     A    B  C
0  1.0  5.0  1
1  2.0  0.0  2
2  0.0  0.0  3
```

Using isnull

```
import pandas as pd
import numpy as np

# Create a simple dataframe
df = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [5, np.nan, np.nan],
    'C': [1, 2, 3]
})
# Use isnull
df_isnull = df.isnull()

print("DataFrame after using isnull:")
print(df_isnull)
```

OUTPUT


```
DataFrame after using isnull:
       A      B      C
0  False  False  False
1  False   True  False
2   True   True  False
```

Using notnull

```
import pandas as pd
import numpy as np

# Create a simple dataframe
df = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [5, np.nan, np.nan],
    'C': [1, 2, 3]
})
# Use notnull
df_notnull = df.notnull()

print("DataFrame after using notnull:")
print(df_notnull)
```

OUTPUT


```
DataFrame after using notnull:
       A      B     C
0   True   True  True
1   True  False  True
2  False  False  True
```

2.b) Explain How do you filter out missing data?

In pandas, you can filter out missing data (NaN values) using the `dropna` function. Here's a simple example:

```
import pandas as pd
import numpy as np

# Create a simple dataframe
df = pd.DataFrame({
   'A': [1, 2, np.nan],
   'B': [5, np.nan, np.nan],
   'C': [1, 2, 3]
})
```

Output:

```
DataFrame after filtering out missing data:
     A    B  C
0  1.0  5.0  1
```

Filling in Missing Data

If you want to fill missing values instead of dropping them, you can use the `fillna`

```
# Create a simple dataframe
df = pd.DataFrame({
   'A': [1, 2, np.nan],
   'B': [5, np.nan, np.nan],
   'C': [1, 2, 3]
})

# Use fillna
df_after_fillna = df.fillna(value=0)

print("DataFrame after using fillna:")
print(df_after_fillna)
```

Output:

```
DataFrame after using fillna:
     A    B  C
0  1.0  5.0  1
1  2.0  0.0  2
2  0.0  0.0  3
```

3. Explain About:
a. Removing Duplicates:

```
import pandas as pd

# Create a simple dataframe with some duplicate
rows
df = pd.DataFrame({
   'Fruit': ['Apple', 'Banana', 'Apple',
'Banana', 'Apple'],
   'Color': ['Red', 'Yellow', 'Red', 'Yellow',
'Green']
})

print("Original DataFrame:")
print(df)
```

```
# Use drop_duplicates to remove duplicate rows
df_no_duplicates = df.drop_duplicates()

print("DataFrame after removing duplicates:")
print(df_no_duplicates)
```

Output:

```
Original DataFrame:
    Fruit   Color
0   Apple     Red
1  Banana  Yellow
2   Apple     Red
3  Banana  Yellow
4   Apple   Green

DataFrame after removing duplicates:
    Fruit   Color
0   Apple     Red
1  Banana  Yellow
4   Apple   Green
```

In this example, `drop_duplicates` removes all duplicate rows in the DataFrame based on all columns. The resulting DataFrame `df_no_duplicates` will only contain unique rows.

3. Explain About:
b. transforming data using a function or mapping

In pandas, you can transform data using functions or mappings. This is often used to modify a DataFrame or Series in a specific way, such as applying a mathematical operation to all elements, or changing the values based on a condition or a mapping.

Here's an example of transforming data using a function:

```
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
   'A': [1, 2, 3, 4, 5],
   'B': [10, 20, 30, 40, 50]
})

# Define a simple function to add 10 to a number
def add_ten(x):
    return x + 10

# Use the applymap function to apply add_ten to
each element of the dataframe
df_transformed = df.applymap(add_ten)

print(df_transformed)
```

In this example, the `applymap` function is used to apply the `add_ten` function to each element of the DataFrame.

Output:

→

	A	B
0	11	20
1	12	30
2	13	40
3	14	50
4	15	60

Here’s an example of transforming data using a mapping:

```python
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
   'A': ['apple', 'banana', 'cherry'],
})

# Define a mapping from fruit names to colors
fruit_color = {
    'apple': 'red',
    'banana': 'yellow',
    'cherry': 'red'
}

# Use the map function to replace the fruit
names with their colors
df['A'] = df['A'].map(fruit_color)

print(df)
```

Output:

	A
0	red
1	yellow
2	red

Q4. **a)Explain about replacing values**

b)Explain about renaming index

a)

In pandas, you can replace values using the `replace` function. This function is used to replace specific values in a DataFrame or Series.

Here's a simple example:

```
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
   'A': ['apple', 'banana', 'cherry'],
})

print("Original DataFrame:")
print(df)

# Use replace to change 'apple' to 'orange'
df_replaced = df.replace('apple', 'orange')

print("DataFrame after replacing values:")
print(df_replaced)
```

Output:

```
Original DataFrame:
        A
0   apple
1  banana
2  cherry

DataFrame after replacing values:
        A
0  orange
1  banana
2  cherry
```

In this example, the `replace` function is used to replace all occurrences of 'apple' with 'orange' in the DataFrame. The resulting DataFrame `df_replaced` will have 'orange' instead of 'apple'. Please note that `replace` does not modify the original DataFrame `df` unless you use it with the `inplace=True` argument.

b) Explain about renaming index

In pandas, you can rename the indices (row labels) of a Data Frame or Series using the rename function. This can be useful when you want to change the labels of your data for better readability or for consistency with other data.

```
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
   'A': [1, 2, 3],
   'B': [4, 5, 6]
}, index=['row1', 'row2', 'row3'])

print("Original DataFrame:")
print(df)

# Use rename to change the index names
df_renamed = df.rename(index={'row1': 'r1',
'row2': 'r2', 'row3': 'r3'})

print("DataFrame after renaming indices:")
print(df_renamed)
```

Output:

```
Original DataFrame:
      A  B
row1  1  4
row2  2  5
row3  3  6

DataFrame after renaming indices:
    A  B
r1  1  4
r2  2  5
r3  3  6
```

In this example, the `rename` function is used to rename the indices of the DataFrame. The resulting DataFrame `df_renamed` will have the indices 'r1', 'r2', and 'r3' instead of 'row1', 'row2', and 'row3'. Please note that `rename` does not modify the original DataFrame `df` unless you use it with the `inplace=True` argument.

Q7. Write about regular expression and write the code to retrieve pattern on email address?

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

The findall() Function

The `findall()` function returns a list containing all matches.

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

Output: → `['ai', 'ai']`

The search() Function

The `search()` function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example:

```
import re

txt = "The rain in Spain"
x = re.search("rain", txt)

print("rain is located at position:", x.start())
```

Output:

```
rain is located at position: 4
```

The split() Function

The `split()` function returns a list where the string has been split at each match:

Example:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

Output:

```
['The', 'rain', 'in', 'Spain']
```

The sub() Function

The `sub()` function replaces the matches with the text of your choice:

Example:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

Output:

```
The9rain9in9Spain
```

It replaces all the spaces with 9

Extracting email addresses using regular expressions

```
import re
# below is the input string
text = 'This email is from xyz@gmail.com to abc@gmail.com'
# finding pattern
lst = re.findall('\S+@\S+', text)
print(lst)
```

Output:

```
['xyz@gmail.com', 'abc@gmail.com']
```

Pattern explanation:

```
# \S matches any non-whitespace character
# @ for as in the Email
# + for Repeats a character one or more times
```

9. Explain about the rename function with example?

Pandas rename() method is used to rename any index, column or row wise.

rename() Syntax

The syntax of the `rename()` method in Pandas is:

```
df.rename(columns=None, index=None, inplace=False)
```

rename() Arguments

The `rename()` method takes following arguments:

- `columns` (optional) - a dictionary that specifies the new names for columns
- `index` (optional) - a dictionary that specifies the new names for index labels
- `inplace` (optional) - if `True`, modifies the original DataFrame in place; if `False`, returns a new DataFrame.

Renaming Column

```python
import pandas as pd

# Create a dataframe
df = pd.DataFrame({'old_name': [1, 2, 3]})
print(df)

# Rename the column
df.rename(columns={'old_name': 'new_name'}, inplace=True)
print(df)
```

Output:

original data frame

	old_name
0	1
1	2
2	3

After Renaming

	new_name
0	1
1	2
2	3

Renaming Row/Index

```python
import pandas as pd

# Create a simple dataframe
df = pd.DataFrame({
   'A': [1, 2, 3],
   'B': [4, 5, 6]
}, index=['row1', 'row2', 'row3'])

print("Original DataFrame:")
print(df)

# Use rename to change the index names
df_renamed = df.rename(index={'row1': 'r1',
'row2': 'r2', 'row3': 'r3'})

print("DataFrame after renaming indices:")
print(df_renamed)
```

Output:

```
Original DataFrame:
      A  B
row1  1  4
row2  2  5
row3  3  6

DataFrame after renaming indices:
    A  B
r1  1  4
r2  2  5
r3  3  6
```

10. Difference between join() and merge()?

- `join()` is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.
- The default type of join performed by `join()` is a left join, which means it combines the DataFrames based on their index, and it keeps all the rows from the left DataFrame (`df1` in this case).

```python
import pandas as pd

# Creating two DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3],
                    'B': ['a', 'b', 'c']},
                   index=[10, 20, 30])

df2 = pd.DataFrame({'C': ['x', 'y', 'z']},
                   index=[20, 30, 40])

# Using join() with default settings (left join)
result_join = df1.join(df2)

print("DataFrame 1:")
print(df1)
print("\nDataFrame 2:")
print(df2)
print("\nResult after join():")
print(result_join)
```

Output:

```
DataFrame 1:
    A  B
10  1  a
20  2  b
30  3  c

DataFrame 2:
    C
20  x
30  y
40  z

Result after join():
    A  B    C
10  1  a  NaN
20  2  b    x
30  3  c    y
```

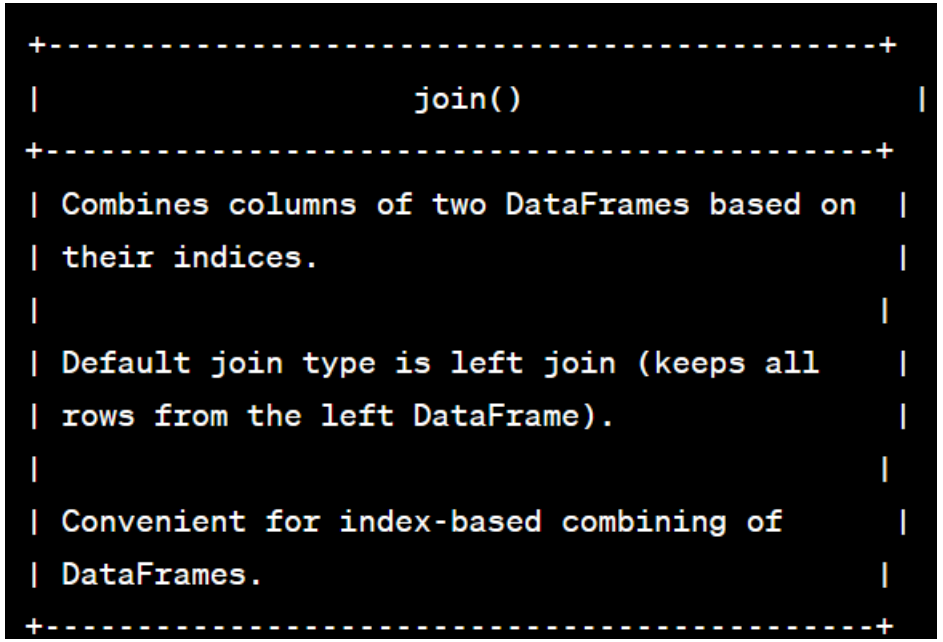
Merge

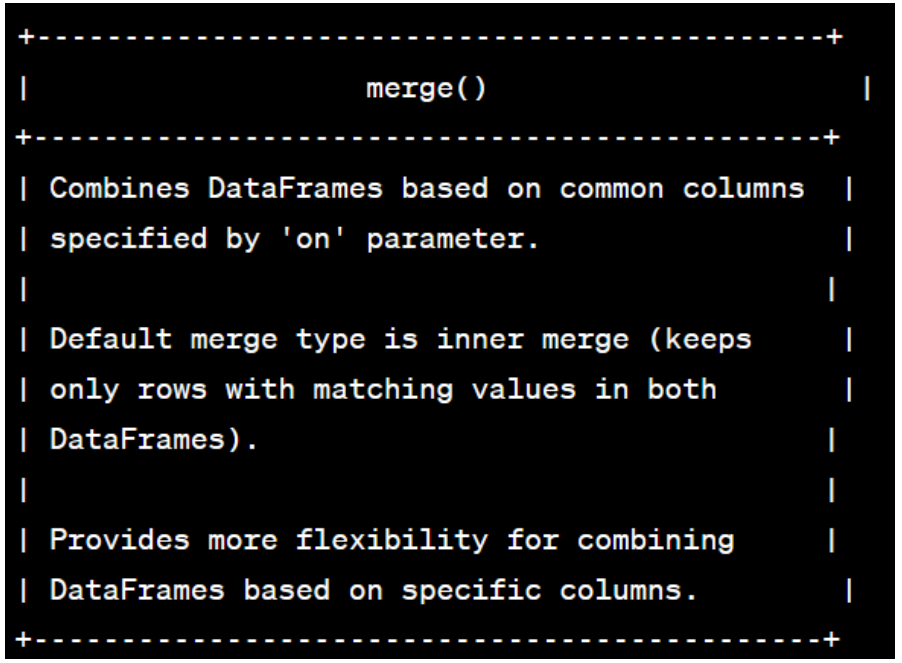
- `merge()` is a more flexible method for combining two DataFrames based on common columns (in this case, the 'key' column).
- The default type of merge performed by `merge()` is an inner merge, which means it only includes the rows with matching values in both DataFrames.

```python
import pandas as pd

# Creating two DataFrames
df1 = pd.DataFrame({'key': [1, 2, 3], 'value': ['A', 'B', 'C']})
df2 = pd.DataFrame({'key': [2, 3, 4], 'value': ['X', 'Y', 'Z']})

# Using merge() with default settings (inner merge)
result_merge = pd.merge(df1, df2, on='key')

print("DataFrame 1:")
print(df1)
print("\nDataFrame 2:")
print(df2)
print("\nResult after merge():")
print(result_merge)
```

Output:

```
DataFrame 1:
   key value
0    1     A
1    2     B
2    3     C

DataFrame 2:
   key value
0    2     X
1    3     Y
2    4     Z

Result after merge():
   key value_x value_y
0    2       B       X
1    3       C       Y
```





---

8. Explain about the different buit -in string methods?

Certainly! There are numerous built-in string methods in Python, and they offer a variety of functionalities to manipulate and work with strings. Here are some common and useful string methods:

1. `capitalize()`
   - Converts the first character of a string to uppercase.

```
text = "hello world"
result = text.capitalize()
print(result)  # Output: "Hello world"
```

2. `upper()` and `lower()`
   - `upper()`: Converts all characters in a string to uppercase.
   - `lower()`: Converts all characters in a string to lowercase.

```
text = "Hello World"
upper_result = text.upper()
lower_result = text.lower()
print("Output:")
print(upper_result)
print(lower_result)
```

```
Output:
HELLO WORLD
hello world
```

3. `title()`
   - Converts the first character of each word to uppercase.

Example:

```
text = "hello world"
result = text.title()
print(result)

# Output
```

```
Hello World
```

4. `strip()`
   - Removes leading and trailing whitespaces from a string.

Example:

```
text = "     hello world    "
result = text.strip()
print(result)
```

Output:

```
hello world
```

5. `replace()`
   - Replaces a specified substring with another substring.

Example:

```
text = "I like apples"
result = text.replace("apples", "oranges")
print(result)
```

Output:

```
I like oranges
```

6. `find()` and `index()`
   - `find()`: Returns the lowest index of the substring. If not found, returns -1.
   - `index()`: Returns the lowest index of the substring. Raises an error if not found.

Example:

```
text = "Hello World"
find_result = text.find("World")
index_result = text.index("World")
print(find_result)
print(index_result)
```

Output:

```
6
6
```

7. `startswith()` and `endswith()`
   - `startswith()`: Returns True if the string starts with the specified prefix.
   - `endswith()`: Returns True if the string ends with the specified suffix.

Example:

```
text = "Hello World"
starts_with_result = text.startswith("Hello")
ends_with_result = text.endswith("World")
print(starts_with_result)
print(ends_with_result)
```

Output:

```
True
True
```

8. `split()`
   - Splits a string into a list of substrings based on a specified delimiter.

Example:

```
text = "apple,orange,banana"
result = text.split(',')
print(result)
```

Output:

```
['apple', 'orange', 'banana']
```

9. `join()`
   - Joins the elements of an iterable (e.g., a list) into a string using the specified separator.

```
fruits = ['apple', 'orange', 'banana']
result = ', '.join(fruits)
print(result)
```

```
apple, orange, banana
```

These are just a few examples of the many string methods available in Python. Each method serves a specific purpose, and understanding them can be highly beneficial when working with strings in Python.