Pandas Data Cleaning

Data cleaning means fixing and organizing messy data. Pandas offers a wide range of tools and functions to help us clean and preprocess our data effectively.

Data cleaning often involves:

1. Dropping irrelevant columns.
2. Renaming column names to meaningful names.
3. Making data values consistent.
4. Replacing or filling in missing values.

Pandas Handling Missing Values

In Pandas, missing values, often represented as `NaN` (Not a Number), can cause problems during data processing and analysis. These gaps in data can lead to incorrect analysis and misleading conclusions.

Pandas provides a host of functions like `dropna()`, `fillna()` and `combine_first()` to handle missing values.

Let's consider the following DataFrame to illustrate various techniques on handling missing data:

```
import pandas as pd
import numpy as np

# create dataframe with missing values
data = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, 4, 5],
    'C': [1, 2, 3, np.nan, 5],
    'D': [1, 2, 3, 4, 5]
}
df = pd.DataFrame(data)

print(df)
```

```
     A    B    C  D
0  1.0  NaN  1.0  1
1  2.0  2.0  2.0  2
2  NaN  3.0  3.0  3
3  4.0  4.0  NaN  4
4  5.0  5.0  5.0  5
```

Here, we have used the NumPy library to generate `NaN` values in the DataFrame.

Drop Rows With Missing Values with dropna()

In Pandas, we can drop rows with missing values using the `dropna()` function. For example,

```
import pandas as pd
import numpy as np

# create a dataframe with missing values
data = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, 4, 5],
    'C': [1, 2, 3, np.nan, 5],
    'D': [1, 2, 3, 4, 5]
}
df = pd.DataFrame(data)
print('Original data\n',df)
print('\n')
# remove rows with missing values
df.dropna(inplace=True)

print('Cleaned data \n',df)
```

```
Original data
      A    B    C  D
0  1.0  NaN  1.0  1
1  2.0  2.0  2.0  2
2  NaN  3.0  3.0  3
3  4.0  4.0  NaN  4
4  5.0  5.0  5.0  5


Cleaned data
      A    B    C  D
1  2.0  2.0  2.0  2
4  5.0  5.0  5.0  5
```

```
import pandas as pd

# define a dictionary with sample data which includes some missing
values
data = {
    'A': [1, 2, 3, None, 5],
    'B': [None, 2, 3, 4, 5],
    'C': [1, 2, None, None, 5]
}

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

```
# use dropna() to remove rows with any missing values
df_cleaned = df.dropna()

print("Cleaned Data:\n",df_cleaned)
```

```
Original Data:
     A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  2.0
2  3.0  3.0  NaN
3  NaN  4.0  NaN
4  5.0  5.0  5.0

Cleaned Data:
     A    B    C
1  2.0  2.0  2.0
4  5.0  5.0  5.0
```

Here, we have used the `dropna()` method to remove rows with any missing values. The resulting DataFrame `df_cleaned` will only contain rows without any missing values

Fill Missing Values with fillna()

To fill the missing values in Pandas, we use the `fillna()` function. For example,

```
import pandas as pd

# define a dictionary with sample data which includes some missing
values
data = {
    'A': [1, 2, 3, None, 5],
    'B': [None, 2, 3, 4, 5],
    'C': [1, 2, None, None, 5]
}

df = pd.DataFrame(data)

print("Original Data:\n", df)

# filling NaN values with 0
df.fillna(0, inplace=True)

print("\nData after filling NaN with 0:\n", df)
```

```
Original Data:
     A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  2.0
2  3.0  3.0  NaN
3  NaN  4.0  NaN
```

```
4  5.0  5.0  5.0

Data after filling NaN with 0:
     A    B    C
0  1.0  0.0  1.0
1  2.0  2.0  2.0
2  3.0  3.0  0.0
3  0.0  4.0  0.0
4  5.0  5.0  5.0
```

```
import pandas as pd
import numpy as np

# create a dataframe with missing values
data = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, 4, 5],
    'C': [1, 2, 3, np.nan, 5],
    'D': [1, 2, 3, 4, 5]
}
df = pd.DataFrame(data)

# replace missing values with 0
df.fillna(value=0, inplace=True)

print(df)
```

```
     A    B    C  D
0  1.0  0.0  1.0  1
1  2.0  2.0  2.0  2
2  0.0  3.0  3.0  3
3  4.0  4.0  0.0  4
4  5.0  5.0  5.0  5
```

Here, we used `data.fillna()` to fill the missing values(`NaN`) in each column with **0**.

Note: The `inplace=True` argument here means that the operation will modify the DataFrame directly, rather than returning a new DataFrame with the modifications.

Use Aggregate Functions to Fill Missing Values

Instead of filling with **0**, we can also use aggregate functions to fill missing values.

Let's look at an example to fill missing values with the mean of each column.

```
import pandas as pd
print(pd.__version__)
```

```
# define a dictionary with sample data which includes some missing
values
data = {
    'A': [1, 2, 3, None, 5],
    'B': [None, 2, 3, 4, 5],
    'C': [1, 2, None, None, 5]
}

df = pd.DataFrame(data)

print("Original Data:\n", df)

# filling NaN values with the mean of each column
df.fillna(df.sum(), inplace=True)

print("\nData after filling NaN with mean:\n", df)
```

```
2.1.1
Original Data:
      A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  2.0
2  3.0  3.0  NaN
3  NaN  4.0  NaN
4  5.0  5.0  5.0

Data after filling NaN with mean:
       A     B    C
0   1.0  14.0  1.0
1   2.0   2.0  2.0
2   3.0   3.0  8.0
3  11.0   4.0  8.0
4   5.0   5.0  5.0
```

Here, the `df.mean()` calculates the mean for each column, and the `fillna()` method then replaces `NaN` values in each column with the respective mean.

Replace Missing Values With Mean, Median and Mode

A more refined approach is to replace missing values with the mean, median, or mode of the remaining values in the column. This can give a more accurate representation than just replacing it with a default value.

We can use the `fillna()` function with aggregate functions to replace missing values with mean, median or mode.

Let's look at an example.

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

```
# create a dataframe with missing values
data = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, 4, 5],
    'C': [1, 2, 3, np.nan, 5],
    'D': [1, 2, 3, 4, 5]
}
df = pd.DataFrame(data)

# replace missing values with mean
df['A'].fillna(value=df['A'].mean(), inplace=True)

# replace missing values with median
df['B'].fillna(value=df['B'].median(), inplace=True)

# replace missing values with mode
df['C'].fillna(value=df['C'].mode()[0], inplace=True)

print(df)
```

```
     A    B    C  D
0  1.0  3.5  1.0  1
1  2.0  2.0  2.0  2
2  3.0  3.0  3.0  3
3  4.0  4.0  1.0  4
4  5.0  5.0  5.0  5
```

Replace Values Using Another DataFrame

We can replace missing values in one DataFrame using another DataFrame using the `fillna()` method.

Let's look at an example.

```
import pandas as pd
import numpy as np

# create a dataframe with missing values
data1 = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [np.nan, 2, 3, 4, 5],
    'C': [1, 2, 3, np.nan, 5],
    'D': [1, 2, 3, 4, 5]
}
df1 = pd.DataFrame(data1)

# create datframe to fill the missing values with
data2 = {
    'A': [10, 20, 30, 40, 50],
```

```
    'B': [10, 20, 30, 40, 50],
    'C': [10, 20, 30, 40, 50],
    'D': [10, 20, 30, 40, 50]
}
df2 = pd.DataFrame(data2)

# replace missing values
df1.fillna(df2, inplace=True)

print(df1)
```

```
      A     B     C  D
0   1.0  10.0   1.0  1
1   2.0   2.0   2.0  2
2  30.0   3.0   3.0  3
3   4.0   4.0  40.0  4
4   5.0   5.0   5.0  5
```

Here, we've two dataframes `df1` and `df2`. The `fillna()` replaces missing values in `df1` with corresponding values from `df2`.

Handle Duplicates Values

In Pandas, to handle duplicate rows, we can use the `duplicated()` and the `drop_duplicates()` function.

`duplicated()` - to check for duplicates `drop_duplicates()` - remove duplicate rows

```
import pandas as pd

# sample data
data = {
    'A': [1, 2, 2, 3, 3, 4],
    'B': [5, 6, 6, 7, 8, 8]
}
df = pd.DataFrame(data)

print("Original DataFrame:\n", df.to_string(index=False))

# detect duplicates
print("\nDuplicate Rows:\n",
df[df.duplicated()].to_string(index=False))

# remove duplicates based on column 'A'
df.drop_duplicates(subset=['A'], keep='first', inplace=True)
#keep='last'

print("\nDataFrame after removing duplicates based on column 'A':\n",
df.to_string(index=False))
```

```
Original DataFrame:
  A  B
 1  5
 2  6
 2  6
 3  7
 3  8
 4  8

Duplicate Rows:
  A  B
 2  6

DataFrame after removing duplicates based on column 'A':
  A  B
 1  5
 2  6
 3  8
 4  8
```

Here,

`df[df.duplicated()]` produces a boolean Series to identify duplicate rows. `df.drop_duplicates(subset=['A'], keep='first', inplace=True)`, removes duplicates based on column `A`, retaining only the first occurrence of each duplicate directly in the original DataFrame.

`subset=['B']`: This parameter specifies that duplicates should be identified based on the values in column 'B'. It means the code looks for rows with duplicate values in column 'B'.

`keep='first'`: This specifies which duplicates (if any) to keep. 'first' means that it will keep the first occurrence of each unique value and drop all subsequent duplicates.

`inplace=True`: This means that the DataFrame df will be modified in place. The operation will be performed on the original DataFrame, and no new DataFrame is returned.

Rename Column Names to Meaningful Names

To rename column names to more meaningful names in Pandas, we can use the `rename()` function. For example,

```
import pandas as pd

# sample data
data = {
    'A': [25, 30, 35],
    'B': ['John', 'Doe', 'Smith'],
    'C': [50000, 60000, 70000]
}
```

```
df = pd.DataFrame(data)

# rename columns
df.rename(columns={'A': 'Age', 'B': 'Name', 'C': 'Salary'},
inplace=True)

print(df.to_string(index=False))
```

```
 Age  Name  Salary
  25  John   50000
  30   Doe   60000
  35 Smith   70000
```

Here, the columns of `df` are renamed from `A`, `B`, and `C` to more meaningful names `Age`, `Name`, and `Salary` respectively.

```
```