Documentation

1.Loading the data:

Pandas library is imported for data manipulation and analysis and a CSV file is read from the specified path into a pandas DataFrame (df).

Code :

```
import pandas as pd

df=pd.read_csv(r"C:\Users\usaan\Downloads\messy_data.csv")

df
```

2.Data Inspection:

Given dataset has 11000 rows and 8 columns. The dataset has the details of employees of a company with details being ID,Name,Age,Email id,Joining date ,Salary and Department.

Number of null values present in each fields:

Unnamed	0
ID	0
Name	2333
Age	1747
Email	1269
Join Date	2192
Salary	2239
Department	2255

Code used: df.isnull().sum()

3.QA Issues:

Fields	Issues
ID	
Name	i.Some values are missing ii.Some have incorrect format
Age	i.Some values are missing
Email ID	i.Mail id format is incorrect for some ii.Missing values
Join Date	i.Missing values ii.Wrong format
Salary	i.Missing values ii.Large decimal values for some and noises
Department	i.Missing values ii.Spelling mistakes

4.Handling of missing values:

Missing values in 'Age' column are replaced with the mean value obtained using sum of entire ages present in the column.

1.Null values were replaced with 0

2.Mean value was found out

3.0s in the Age column was replaced by this mean value

Code used:

```
df1.loc[:, 'Age'] = df1['Age'].fillna(value=0)

mean_value=int(df1["Age"].mean())

df1.loc[:,"Age"] = df1["Age"].replace(0, mean_value)

df1
```

Rows with excessive null vaues were removed.Using the code rows that had atleast 3 non null values were kept and others removed.

Code used:

df1=df1.dropna(thresh=3)

5.Removing duplicates

Duplicate datas were removed to make the record unique.

Code used:

df1=df.drop_duplicates()

6.Correcting Email Formats:

1. Regular expression module is imported.

2. A function correct_email is defined,that takes an email as input and ensures it follows a valid format. It first checks if the email is missing or invalid (i.e., None or NaN) and returns None if so.

3. It strips any extra spaces and converts the email to lowercase to avoid case sensitivity.

4.It uses a regular expression (email_regex) to validate whether the email is in the correct format.

5.If the email lacks an "@" symbol, the function attempts to insert it by splitting the string at the last period (assuming it indicates the domain). If the domain part of the email is incomplete (i.e., missing a period after the "@"), it adds ".com" as a default. Finally, it checks the corrected email against the regular expression again and returns the valid email or None if it remains invalid.

Code used:

```
Import re

def correct_email(email):
  # Regular expression for a valid email format
  email_regex = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
  # Check if the email is None or NaN
```

```
    if pd.isna(email) or email is None:
        return None

    # Clean the email (remove any trailing spaces or unnecessary characters)
    email = email.strip().lower()  # Convert to lowercase to avoid case-sensitivity issues

    # Check if the email matches the regular expression
    if re.match(email_regex, email):
        return email  # Already valid email

    # If no "@" is present, try to insert it
    if '@' not in email:
        # Split by the last period to guess the domain
        if '.' in email:
            email_parts = email.rsplit('.', 1) # This splits the email string into two parts, from the right, using the last occurrence of the period (.) as the splitting point. The rsplit() function ensures that only the last period is considered, resulting in a list with two elements: everything before the last period and everything after.
            email = email_parts[0] + '@' + email_parts[1] #The two parts are joined together with an "@" symbol between them, forming a potentially valid email format.
        else:
            return None  # Can't fix if there's no valid domain structure

    # If the domain part is missing, try adding ".com"
    if '@' in email and '.' not in email.split('@')[-1]:
        email += '.com'  # checks if an email contains the "@" symbol but is missing a top-level domain (TLD), like .com, after the "@" symbol. It does this by splitting the email at the "@" and checking if the domain part (everything after the "@") has a period. If the period is missing, it appends ".com" to the email to correct it, assuming ".com" as the default domain.
```

```
    # Check if it's now valid
    if re.match(email_regex, email):
        return email

    # If still invalid, return None
    return None


# Apply the email correction function
df1.loc[:,'Email'] = df1['Email'].apply(correct_email)
```

7.Cleaning Name Fields

1.A clean_name function is defined to clean and format names in a dataset. First, it checks if the input name is a string. If it is, the function removes any leading or trailing whitespace using strip(), and then removes any non-alphabetic characters (except spaces) using a regular expression.

2.Then, it converts the cleaned name to title case (i.e., capitalizing the first letter of each word).

3.The cleaned name is returned. If the input is not a string, the original value is returned unchanged. This function is applied to the 'Name' column of the DataFrame using apply().

Code used:

```
def clean_name(name):
    if isinstance(name, str): # checks if input name is a string
        name = name.strip()    #strips any leading or trailing spaces
        name = re.sub(r'[^a-zA-Z\s]', '', name) # removes all characters from the name string that are not letters or spaces. It ensures that only alphabetic characters and spaces remain by replacing unwanted characters with an empty string.
        name = name.title() # converts the cleaned name to title case
```

```
    return name
  else:
    return name
df1.loc[:,'Name'] = df1['Name'].apply(clean_name) # Apply the clean_name function
```

8.Standardising date format:

The 'Join Date' column is converted to datetime format, expecting the format YYYYMMDD.

Code used:

```
df1.loc[:,'Join Date'] = pd.to_datetime(df1['Join Date'], format='%Y%m%d', errors='coerce').fillna(df1['Join Date'])   #converts the 'Join Date' column in the df1 DataFrame to datetime format, expecting the format YYYYMMDD. If any dates don't match the format, it coerces those values to NaT (Not a Time)
```

9.Correcting department names:

1.The difflib module is imported for comparing sequences, finding similarities between strings or lists for tasks such as string matching, finding closest matches, or comparing the similarity between sequences. The function difflib enables get_close_matches(), which helps find the best matches for a string from a list based on a similarity ratio.

2.A list of valid department names (correct_departments) is defined and applies the correct_department function to each value in the 'Department' column.

3.If a department name is a string, it uses difflib.get_close_matches to find the closest match from the predefined list with a similarity threshold of 60%. If a match is found, it replaces the value with the closest match; otherwise, it keeps the original value. Non-string entries are set to 'Unknown'.

Code used:

```
import difflib
correct_departments = ['HR', 'Engineering', 'Marketing', 'Sales', 'Support']#list of valid department names
def correct_department(dept_name):
  if isinstance(dept_name, str): # checks if the department names are of string type
    closest_match = difflib.get_close_matches(dept_name, correct_departments, n=1, cutoff=0.6) # Limits the number of matches returned to 1 (the closest match) and sets a
```

threshold for similarity. Only matches with a similarity score of 60% or higher are considered valid.

```
        return closest_match[0] if closest_match else dept_name
    else:
        return 'Unknown'
df1.loc[:,'Department'] = df1['Department'].apply(correct_department)
```

10.Handling Salary noises:

1. A summary of statistics for the "Salary" column is obtained.

Code used: df1['Salary'].describe()

2. 1st and 95th percentiles of the "Salary" column in df1 is found and the corresponding values are printed, providing an estimate of reasonable salary ranges

Code used:

```
min_salary = df1['Salary'].quantile(0.01)  # 1st percentile
max_salary = df1['Salary'].quantile(0.95)  # 95th percentile
print(f"Minimum reasonable salary: {min_salary}")
print(f"Maximum reasonable salary: {max_salary}")
```

3.A new DataFrame outliers is created that contains rows where the "Salary" is either below the 1st percentile (min_salary) or above the 95th percentile (max_salary), which are considered extreme values or outliers.

Code used:

```
outliers = df1[(df1['Salary'] < min_salary) | (df1['Salary'] > max_salary)]
```

4.A new DataFrame df_cleaned is created that contains only the rows where the "Salary" is within the 1st and 95th percentiles, effectively removing the outliers.

Code used:

```
df_cleaned = df1[(df1['Salary'] >= min_salary) & (df1['Salary'] <= max_salary)]
print("Data after removing outliers:")
print(df_cleaned)
```

5. The median salary is calculated.

Code used:

```
median_salary = df1['Salary'].median()
```

6. A lambda function is applied across the DataFrame using apply() to check each salary value. If the salary is below the min_salary or above the max_salary, it replaces that salary with the median_salary. Otherwise, the original salary is retained and axis=1ensures the function is applied row-wise across the DataFrame.

Code used:

```
df1.loc[:,'Salary'] = df1.apply(lambda x: median_salary if x['Salary'] < min_salary or x['Salary'] > max_salary else x['Salary'], axis=1)
```