Python Exploratory Data Analysis of a Insurance Dataset

First we want to import our dataset into Jupyter Notebook, but for this time i imported raw dataset to SSMS first then by using python code i imported to Jupyter Notebook. I left all the fields as they originally were in order to show you some things you can do edit your dataset.

For Importing to Jupyter Notebook from SSMS i used below code

```
import pandas as pd
import pyodbc

# Define the connection string
conn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};'
                      'SERVER=Server name;'
                      'DATABASE=Database name'
                      'Trusted_Connection=yes;')


# SQL query to execute
```

```
query = "select * from table name"  # Use your actual table name in place of 'your_table_name'

# Fetch data using pandas
df = pd.read_sql(query, conn)

# Print the DataFrame to verify the output
print(df)

# Close the connection
conn.close()
```

First we need to verify the data

The Data Cleaning part

How to Handle Duplicates, Nulls, and Formatting in Python

```
df = df.drop_duplicates()  # for duplicated
df= df.dropna()            # for Null values
df['col'] = df['col'].str.capitalize # if u want first letter capital
#df['col'] = df['col'].str.Lower    # if u want lower case
#df['col'] = df['col'].str.upper    # if u want upper case
#df['col'] = df['col'].str.strip()  # it will remove extra spaces
```

Adding a New Column: `bmi_category` Based on Raw BMI Values(if you check raw dataset then you can figure it out)

This section explains how we classified individuals into BMI categories based on their `bmi` values in the dataset.

Objective

To categorize individuals into one of the following BMI categories based on their BMI value:

- **Underweight**: BMI ≤ 18
- **Normal weight**: 18 < BMI ≤ 24.9
- **Overweight**: 25 ≤ BMI ≤ 39.99
- **Obese**: BMI > 40

```
import numpy as np
import pandas as pd

# Define conditions and choices
conditions = [
    (df['bmi'] <= 18),
    (df['bmi'] > 18) & (df['bmi'] <= 24.9),
    (df['bmi'] >= 25) & (df['bmi'] <= 39.99),
    (df['bmi'] > 40)
]

choices = ['underweight', 'normal weight', 'overweight', 'obe
se']

# Create a new column 'bmi_category' based on 'bmi' values
df['bmi_category'] = np.select(conditions, choices, default
='unknown')

# Display the updated DataFrame
pd.set_option('display.max_rows', None)
print(df)
```

Heading: Categorizing Individuals into Age Groups Based on Age Data

Content:

This code categorizes individuals into distinct age groups (`youth`, `youngadult`, `middleaged`, and `olderadult`) based on their age values. The conditions for grouping are explicitly defined using logical operators to ensure that each individual falls into the correct category.

```
import numpy as np
import pandas as pd

# Define conditions for age groups
conditions = [
    (df['age'] >= 18) & (df['age'] <= 25),   # youth
    (df['age'] >= 26) & (df['age'] <= 35),   # young adult
    (df['age'] >= 36) & (df['age'] <= 45),   # middle-aged
    (df['age'] > 46)                        # older adult
]

# Define corresponding labels for the age groups
choices = ['youth', 'youngadult', 'middleaged', 'olderadult']

# Create a new column 'age_group' based on the conditions
df['age_group'] = np.select(conditions, choices, default='unk
nown')

# Display the entire DataFrame
pd.set_option('display.max_rows', None)
print(df)
```

Heading: Categorizing Insurance Charges into Tiers

Content:

This code divides individuals into distinct categories (`low`, `medium`, `high`) based on their insurance charges. The categorization is performed by evaluating specific ranges of the `charges` column and assigning appropriate labels.

```
import numpy as np
import pandas as pd

# Define conditions for age groups
charges_conditions = [
    (df['charges'] <= 5000),  # Low charges
    (df['charges'] > 5000) & (df['charges'] <= 15000),
    (df['charges'] > 15000)  # High charges
]

# Define corresponding labels for the age groups
choices = ['low', 'medium', 'high']

# Create a new column 'age_group' based on the conditions
df['charges_category'] = np.select(charges_conditions, choice
s, default='unknown')

# Display the entire DataFrame
pd.set_option('display.max_rows', None)
print(df)
```

Heading: Analyzing BMI Category and age_group

Content:

This code groups the dataset by age and BMI category to count the number of occurrences. It then identifies the age with the highest count for each BMI category (Underweight, Normal Weight, Overweight, Obese). The final result is sorted by BMI category, allowing us to easily see which age group corresponds to each BMI category.

```
# Group by 'age' and 'bmi_category' and count occurrences
result = (
    df.groupby(['age_group', 'bmi_category'])
    .size()
```

```
    .reset_index(name='count')  # Rename the size column to
'count'
)

# Find the age with the highest count for each BMI category
top_bmi_by_category = (
    result.loc[result.groupby('bmi_category')['count'].idxmax
()]
    .sort_values(by='bmi_category')  # Sort for readability
)

# Display the result
print(top_bmi_by_category)

----------------------------output--------------
        age_group   bmi_category  count
      olderadult  normal weight     65
      olderadult          obese     37
       olderadult     overweight    369
           youth    underweight      7
      middleaged        unknown      3
```

Heading: "**Identifying the Most Common Age Group per Age**"

Content: This code groups the data by `age` and `age_group`, counts the occurrences, and then identifies the most frequent `age_group` for each age. It sorts the results by `age_group` for better readability. The final output shows the top `age_group` for each age based on the count.

```
result = (
    df.groupby(['age','age_group'])
    .size()
    .reset_index(name='count')
)
top_age_group = (
    result.loc[result.groupby('age_group')['count'].idxmax()]
```

```
                              .sort_values(by='age_group')
)
print(top_age_group)

-----------------------output--------------
    age   age_group  count
   45  middleaged     29
   47  olderadult     29
   46     unknown     29
   26  youngadult     28
   18       youth     69
```

Heading: "**Top Charges by Age**"

Content: This code groups the data by `age` and `charges_category`, counts the occurrences, and finds the most frequent `charges_category` for each group. It then sorts the output by `charges_category` to make the results more interpretable. The output highlights the dominant charges category for different age groups based on the count.

```
result = (
    df.groupby(['age_group','charges'])
    .size()
    .reset_index(name='count')  # Add a 'count' column
)

# Find the age with the highest charge
top_age_with_charge = result.sort_values(by='charges', ascend
ing=False).head(1)

print(top_age_with_charge)

--------------output---------------
                age_group      charges  count
               olderadult  63770.42801      1
```

Heading: "**Top Charges Categories by Age, Gender, and Count**"

Content: This code groups the data by `age`, `sex`, and `charges_category`, counting the occurrences of each combination. It then identifies the most frequent `charges_category` across all groups and sorts the results by `charges_category`. The output provides insights into which age and gender groups dominate each charges category based on the highest count.

```
    result = (
    df.groupby(['age_group','sex','charges_category'])
    .size()
    .reset_index(name='count')
)
top_charges_category = (
    result.loc[result.groupby('charges_category')['count'].id
xmax()]
    .sort_values(by='charges_category')
)
print(top_charges_category)


     ----------------output-----------------

         age_group    sex    charges_category  count
        olderadult    male             high     73
          youth       male             low      110
        olderadult    female           medium    177
```

Heading: "**Top Regions by Age and Count**"

Content: This code groups the data by `age` and `region`, calculating the count of each combination. It identifies the most common `age` group in each `region` and sorts the results alphabetically by `region`. The output highlights the age group most prevalent in each region based on the highest count.

```
result = (
    df.groupby(['age_group','region'])
```

```
    .size()
    .reset_index(name='count')
)
top_region = (
    result.loc[result.groupby('region')['count'].idxmax()]
    .sort_values(by='region')
)
print(top_region)



---------------------output----------------
       age_group     region  count
    olderadult  northeast    115
    olderadult  northwest    114
    olderadult  southeast    126
    olderadult  southwest    116
```

Heading: "**Top Age Groups by Number of Children**"

Content: This code groups the data by `age` and the number of `children`, counting the occurrences of each combination. It then identifies the most common `age` group for each unique value of `children` and sorts the results in ascending order of `children`. The output shows which age group has the highest count for each number of children.

```
result = (
    df.groupby(['age_group','children'])
    .size()
    .reset_index(name='count')
)
top_children = (
    result.loc[result.groupby('children')['count'].idxmax()]
    .sort_values(by='children')
)
print(top_children)
```

```
-------------------output------------
          age_group  children  count
        olderadult         0    224
        olderadult         1    108
        middleaged         2     77
        olderadult         3     61
        olderadult         4     10
        middleaged         5      7
```

Heading: "**Top Age Groups by Smoking Status**"

Content: This code segments the data by `age` and `smoker` status, tallying the frequency of each combination. It determines the `age` group with the highest count for each smoking category (`smoker` or `non-smoker`) and organizes the output sorted by smoking status. This highlights the most prevalent age group for each smoking status.

```
result = (
    df.groupby(['age_group','smoker'])
    .size()
    .reset_index(name='count')
)
top_smoker = (
    result.loc[result.groupby('smoker')['count'].idxmax()]
    .sort_values(by='smoker')
)
print(top_smoker)

----------------output-----------------
       age_group smoker  count
     olderadult     no    384
     olderadult    yes     87
```

Heading: "**Highest Charges by Smoker Status**"

Content: This code groups the dataset by `smoker` status and `charges`, counting the frequency of each combination. It then identifies the charge value with the highest count for both smokers and non-smokers. The results are displayed in a sorted format based on smoking status, revealing the most common charge for each category.

```
# Group by 'smoker' and 'charges', and count occurrences
result = (
    df.groupby(['smoker', 'charges'])
    .size()
    .reset_index(name='count')  # Add a 'count' column
)

# Find the charge with the highest count for each smoker cate
gory
top_charges = (
    result.loc[result.groupby('smoker')['count'].idxmax()]
    .sort_values(by='smoker')  # Sort for readability
)

# Display the result
print(top_charges)

-------------output-------------
       smoker charges_category  count
         no           medium    614
        yes             high    267
```

Analyzing Smoking Habits by Gender

This code groups the dataset by **gender** (`sex`) and **smoking status** (`smoker`) and then calculates the number of occurrences in each group. The results are sorted first by **gender** and then by the **count** in descending order, showing the highest count of smokers or non-smokers for each gender.

```
result = (
    df.groupby(['sex','smoker'])
    .size()
```

```
    .reset_index(name = 'count')
)
top_smoker = (
    result.sort_values(by=['sex', 'count'], ascending=[True, 
False])

)

print(top_smoker)

--------------output------------
    sex   smoker  count
  Female     no    547
  Female    yes    115
    Male     no    516
    Male    yes    159
```

Analysis of Dependent Children by Gender

In this analysis, we examine which gender has more children, and among those, which has the highest count. The data is grouped by **sex** and **number of children** to identify the gender with the most dependent children.

Key Observations:

- **Sex** is sorted in ascending order (females first), while the **count** of children is sorted in descending order, so the gender with the highest number of children appears at the top.

```
result = (
    df.groupby(['sex','children'])
    .size()
    .reset_index(name = 'count')
)
top_dependent = (
    result.sort_values(by=['sex','count'], ascending= [True, 
False])
```

```
)

print(top_dependent)

----------------output---------------
       sex  children  count
0   Female         0    289
1   Female         1    158
2   Female         2    119
3   Female         3     77
4   Female         4     11
5   Female         5      8
6     Male         0    284
7     Male         1    166
8     Male         2    121
9     Male         3     80
10    Male         4     14
11    Male         5     10
```

Writing Data to SQL Server from Python

To write data from a Pandas DataFrame to a SQL Server database:

1. **Connect to SQL Server**: Use SQLAlchemy's `create_engine` to connect to the database.
2. **Write Data**: Use the `to_sql` function to insert or replace data in the target table.

```
from sqlalchemy import create_engine
import pandas as pd

# Connect to SQL Server using the new database
engine = create_engine(r'mssql+pyodbc://@servername/databasename?driver=ODBC+Driver+17+for+SQL+Server&Trusted_Connection=yes')
```

```
# Assuming df is your pandas DataFrame
df.to_sql('databasename', con=engine, if_exists='replace', in
dex=False)

print("Data written successfully to the new database and tabl
e.")
```