EDA in Python

Author: Samuel Ndegwa July, 2023

- Exploratory Data Analysis, or EDA for short, is the process of cleaning and reviewing data to derive insights such as descriptive statistics and correlation and generate hypotheses for experiments
- EDA results often inform the next steps for the dataset, whether that be generating hypotheses, preparing the data for use in a machine learning model, or even throwing the data out and gathering new data!
- We are going to utilize the pandas library to analyze our data

Functions for initial Exploration

Import the required python package, in this case pandas as pd.

```
import pandas as pd # importing pandas as pd
```

To download the CSV file, what is needed in this example, console/command line is enough:

```
curl -O https://raw.githubusercontent.com/Asabeneh/30-Days-Of-Python/master/data/weight-height.csv
```

Inspecting a DataFrame

When you get a new DataFrame to work with, the first thing you need to do is explore it and see what it contains. There are several useful methods and attributes for this.

- .head() returns the first few rows (the “head” of the DataFrame).
- .info() shows information on each of the columns, such as the data type and number of missing values.
- .shape returns the number of rows and columns of the DataFrame.
- .describe() calculates a few summary statistics for each column.

```
import pandas as pd

df = pd.read_csv('weight-height.csv')
print(df)
```

This line prints the DataFrame *df* to the console. When you run this code, it will display the contents of the CSV file "weight-height.csv" in tabular form.

Data Exploration Let us read only the first 5 rows using head()

```
print(df.head(10)) # give five rows we can increase the number of rows by passing argument to the head() method
```

- Use a pandas function to print a summary of column non-missing values and data types from the *df* DataFrame.

```
print(df.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Gender  10000 non-null  object
 1   Height  10000 non-null  float64
 2   Weight  10000 non-null  float64
dtypes: float64(2), object(1)
memory usage: 234.5+ KB
None
```

Explanation The *info()* function provides a concise summary of the DataFrame, including the number of non-null values in each column and the data types of those columns. It also gives information about the memory usage of the DataFrame.

- Print the summary statistics (count, mean, standard deviation, min, max, and quartile values) of each numerical column in unemployment.

```
# Using the describe() function to get the summary statistics of numerical columns
summary_stats = df.describe()

# Printing the summary statistics
print(summary_stats)

             Height        Weight
count  10000.000000  10000.000000
mean      66.367560    161.440357
std        3.847528     32.108439
min       54.263133     64.700127
25%       63.505620    135.818051
50%       66.318070    161.212928
75%       69.174262    187.169525
max       78.998742    269.989699
```

Types of Data in Statistics

1. Numerical Data
   - This data has a sense of measurement involved in it; for example, a person's age, height, weight, blood pressure, heart rate, temperature, number of teeth, number of bones, and the number of family members. This data is often referred to as quantitative data in statistics. The numerical dataset can be either discrete or continuous types.
2. Categorical data:
   - This type of data represents the characteristics of an object; for example, gender, marital status, type of address, or categories of the movies. This data is often

referred to as qualitative datasets in statistics. To understand clearly, here are some of the most common types of categorical data you can find in data:

- Gender (Male, Female, Other, or Unknown)
- Marital Status (Annulled, Divorced, Interlocutory, Legally Separated, Married, Polygamous, Never Married, Domestic Partner, Unmarried, Widowed, or Unknown)
- Movie genres (Action, Adventure, Comedy, Crime, Drama, Fantasy, Historical, Horror, Mystery, Philosophical, Political, Romance, Saga, Satire, Science Fiction, Social, Thriller, Urban, or Western)
- Blood type (A, B, AB, or O)

Types of measurement scales in statistics

1. Norminal
2. Ordinal
3. Interval
4. Ratio

Counting categorical values You'd now like to explore the categorical data contained in unemployment to understand the data that it contains related to each continent.

```
print(df['Gender'].value_counts())

Male      5000
Female    5000
Name: Gender, dtype: int64
```

Explanation The value_counts() function calculates the frequency of unique values in a Series (in this case, the 'Gender' column) and returns a new Series with the counts. The result will show the number of occurrences for each Gender in the 'df' DataFrame.

A little bit of Visualization

1. We are going to use our unemployment dataset to view typical unemployment in a given year.
    - Our task in this exercise is to create a histogram showing the distribution of global unemployment rates in 2021.

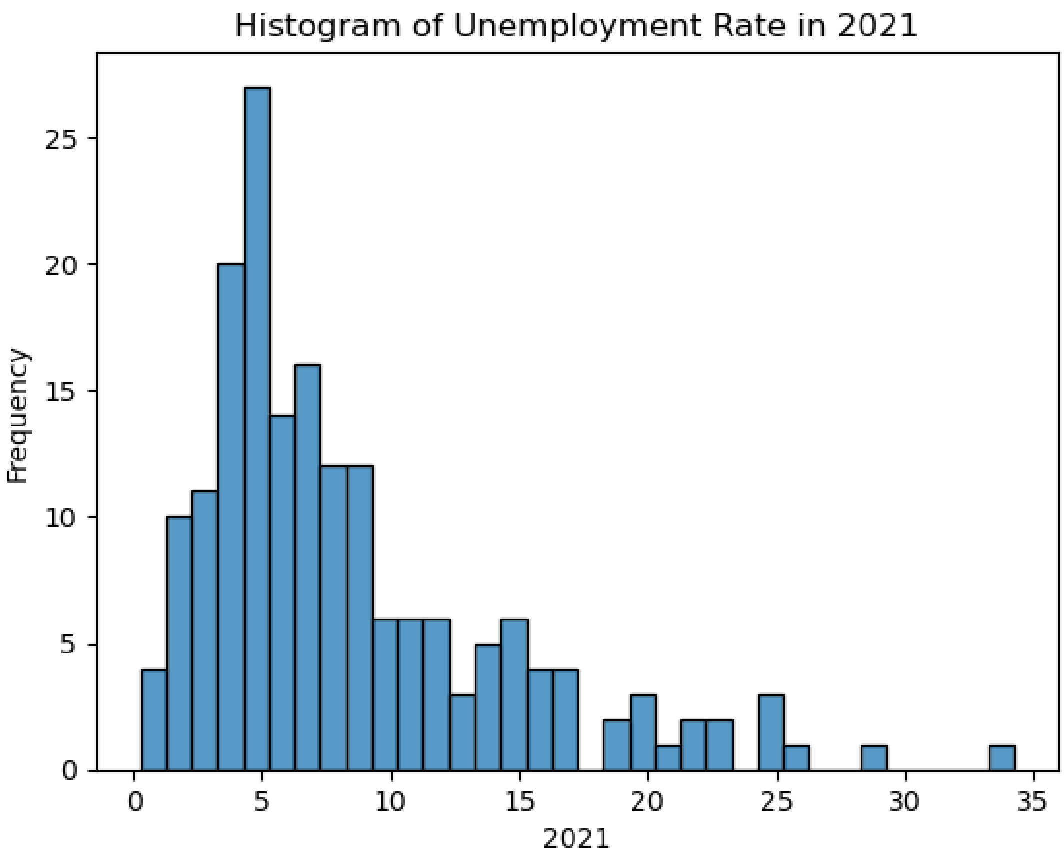
```
#import required python packages
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

unemployment = df = pd.read_csv('clean_unemployment.csv')

# Create a histogram of 2021 unemployment; show a full percent in each bin
sns.histplot(data=unemployment, x="2021", binwidth=1)

# Add labels and title
```

```
plt.ylabel('Frequency')
plt.title('Histogram of Unemployment Rate in 2021')
plt.show()
```



Detecting Data types Let us check the data types in our Unemployment DataFrame

```
print(unemployment.info())
```

Change between data types

```
#change our 2019 Column data type from float to integer
unemployment['2019'] = unemployment['2019'].astype(float)

print(unemployment.info())
```

Data Validation using the *.isin* function The *.isin()* function in Python is a powerful tool used in data validation to check whether elements in a Series or DataFrame column are present in a specified list or another DataFrame column. It returns a boolean Series or DataFrame, indicating whether each element in the Series or DataFrame column is found in the specified list or column.

Example: Using our **unemployment** dataframe, let us identify countries that are not in Oceania. These countries should return True while countries in Oceania should return False. This will set you up to use the results of .isin() to quickly filter out Oceania countries using Boolean indexing.

- Define a Series of Booleans describing whether or not each continent is outside of Oceania; call this Series not_oceania.

```
not_oceania = ~unemployment["continent"].isin(["Oceania"])

# Print unemployment without records related to countries in Oceania
print(unemployment[not_oceania])
```

Summaries with .groupby and .agg These functions are particularly useful when you want to calculate summary statistics for different groups in your dataset.

```
import pandas as pd

df = pd.read_csv('weight-height.csv')
##height_weight_summary = df.groupby('Gender')['Weight',
'Height'].agg(['mean', 'std']) --previous way of indexing with
multiple keys
height_weight_summary = df.groupby('Gender')[['Weight',
'Height']].agg(['mean', 'std'])
print(height_weight_summary)
```

Example 2 Using our unemployment DataFrame; Print the mean and standard deviation of the unemployment rates for each year, grouped by continent.

```
#Old method
print(unemployment.groupby("continent").agg(["mean", "std"]))
```

Print the mean and standard deviation of the unemployment rates for each year, without grouping.

```
#import required python packages
import pandas as pd

unemployment = df = pd.read_csv('clean_unemployment.csv')

#get a concise summary of a DataFrame
##print(unemployment.info())
#drop columns that cannot be aggregated
#specify which columns to drop
columns_to_exclude = ['country_code', 'country_name']

#return a boolean array where True represents the columns that are not
in columns_to_exclude
inverse_columns = df.loc[:, ~df.columns.isin(columns_to_exclude)]
#we use the .loc[] indexer to select the columns based on the boolean
mask.
```

```python
#print the mean and standard deviation of columns that can be aggregated
print(inverse_columns.groupby("continent").agg(["mean", "std"]))
```

Named Aggregation

Sometimes, it's helpful to name new columns when aggregating so that it's clear in the code output what aggregations are being applied and where In this example we are going to:

1. Create a DataFrame called continent_summary which shows a row for each continent. The DataFrame columns will contain the mean unemployment rate for each continent in 2021 as well as the standard deviation of the 2021 employment rate.
2. Rename the columns

```python
continent_summary = unemployment.groupby("continent").agg(
    # Create the mean_rate_2021 column
    mean_rate_2021=("2021", "mean"),
    # Create the std_rate_2021 column
    std_rate_2021=("2021", "std")
)
print(continent_summary)
```

Explanation

1. **unemployment.groupby("continent"):** This part groups the "unemployment" DataFrame by the "continent" column, creating separate groups for each unique continent.
2. **.agg(...):** The .agg() function is used to perform aggregation on the grouped data.
3. **mean_rate_2021=("2021", "mean"):** This line creates a new column in the resulting DataFrame called "mean_rate_2021." It calculates the mean of the "2021" column within each continent group and stores the result in this new column.
4. **std_rate_2021=("2021", "std"):** Similarly, this line creates another new column called "std_rate_2021." It calculates the standard deviation of the "2021" column within each continent group and stores the result in this new column.

Visualizing Categorical Summaries

- Seaborn has many great visualizations for exploration, including a bar plot for displaying an aggregated average value by category of data.
- In Seaborn, bar plots include a vertical bar indicating the 95% confidence interval for the categorical mean. Since confidence intervals are calculated using both the number of values and the variability of those values, they give a helpful indication of how much data can be relied upon.

```python
#import required python packages
import pandas as pd
import seaborn as sns
```

```python
import matplotlib.pyplot as plt

# Create a bar plot of continents and their 2021 average unemployment
sns.barplot(data=unemployment, x="continent", y="2021")
plt.show()
```

Data Manipulation with Pandas

Sorting rows Finding interesting bits of data in a DataFrame is often easier if you change the order of the rows. You can sort the rows by passing a column name to .sort_values().

In cases where rows have the same value (this is common if you sort on a categorical variable), you may wish to break the ties by sorting on another column. You can sort on multiple columns in this way by passing a list of column names.

Sort on … Syntax one column df.sort_values("breed") multiple columns df.sort_values(["breed", "weight_kg"])

Sort on …	Syntax
one column	df.sort_values("breed")
multiple columns	df.sort_values(["breed", "weight_kg"])

By combining .sort_values() with .head(), you can answer questions in the form, "What are the top cases where…?".

Sorting General Syntax

1. **sort_values():** This method is used to sort the DataFrame by the values of one or more columns. You can sort in ascending or descending order. By default, it sorts in ascending order. Syntax: df.sort_values(by, ascending=True/False)
   - by: Specifies the column(s) to sort by. It can be a single column name or a list of column names if you want to sort by multiple columns.
   - ascending: Determines the sorting order. If True, the data will be sorted in ascending order; if False, it will be sorted in descending order.
2. **sort_index():** This method is used to sort the DataFrame based on the row index. Syntax: df.sort_index(axis=0, ascending=True/False)
   - axis: Specifies whether to sort the rows (axis=0) or columns (axis=1). The default is axis=0 (sorting rows).
   - ascending: Determines the sorting order. If True, the data will be sorted in ascending order of the index; if False, it will be sorted in descending order.

Example 1 Sort homelessness by the number of homeless individuals, from smallest to largest, and save this as homelessness_ind. Print the head of the sorted DataFrame.

```python
import pandas as pd

homelessness = pd.read_csv('homelessness.csv')
```

```
# Sort homelessness by individuals
homelessness_ind = homelessness.sort_values("individuals")

# Print the top few rows
print(homelessness_ind.head(10))
```

```
    Unnamed: 0              region          state  individuals  \
50          50            Mountain        Wyoming        434.0
34          34  West North Central   North Dakota        467.0
7            7      South Atlantic       Delaware        708.0
39          39         New England   Rhode Island        747.0
45          45         New England        Vermont        780.0
29          29         New England  New Hampshire        835.0
41          41  West North Central   South Dakota        836.0
26          26            Mountain        Montana        983.0
48          48      South Atlantic  West Virginia       1021.0
24          24  East South Central    Mississippi       1024.0

    family_members  state_pop
50           205.0     577601
34            75.0     758080
7            374.0     965479
39           354.0    1058287
45           511.0     624358
29           615.0    1353465
41           323.0     878698
26           422.0    1060665
48           222.0    1804291
24           328.0    2981020
```

Example 2 Sort homelessness by the number of homeless family_members in descending order, and save this as homelessness_fam. Print the head of the sorted DataFrame.

```
# Sort homelessness by descending family members
homelessness_fam = homelessness.sort_values("family_members",
ascending=False)

# Print the top few rows
print(homelessness_fam.head(10))
```

Example 3 Sort homelessness first by region (ascending), and then by number of family members (descending). Save this as homelessness_reg_fam. Print the head of the sorted DataFrame.

```
# Sort homelessness by region, then descending family members
homelessness_reg_fam = homelessness.sort_values(["region",
"family_members"], ascending=[True, False])

# Print the top few rows
```

```
print(homelessness_reg_fam.head(15))

'''Data Structures in Python
1. DataFrame
2. List
3. Tuple
4. Dictionary
5. Set
'''
```

```
    Unnamed: 0              region         state  individuals
family_members  \
13          13  East North Central      Illinois       6752.0
3891.0
35          35  East North Central          Ohio       6929.0
3320.0
22          22  East North Central      Michigan       5209.0
3142.0
49          49  East North Central     Wisconsin       2740.0
2167.0
14          14  East North Central       Indiana       3776.0
1482.0
42          42  East South Central     Tennessee       6139.0
1744.0
17          17  East South Central      Kentucky       2735.0
953.0
0            0  East South Central       Alabama       2570.0
864.0
24          24  East South Central   Mississippi       1024.0
328.0
32          32        Mid-Atlantic      New York      39827.0
52070.0
38          38        Mid-Atlantic  Pennsylvania       8163.0
5349.0
30          30        Mid-Atlantic    New Jersey       6048.0
3350.0
5            5            Mountain      Colorado       7607.0
3250.0
2            2            Mountain       Arizona       7259.0
2606.0
44          44            Mountain          Utah       1904.0
972.0

    state_pop
13   12723071
35   11676341
22    9984072
49    5807406
14    6695497
42    6771631
```

```
17     4461153
0      4887681
24     2981020
32    19530351
38    12800922
30     8886025
5      5691287
2      7158024
44     3153550

'Data Structures in Python\n1. DataFrame\n2. List\n3. Tuple\n4.
Dictionary\n5. Set\n'
```

Subsetting Columns In pandas, subsetting columns refers to selecting and extracting specific columns from a DataFrame. When working with data, you may not need all of the variables in your dataset. Square brackets ([]) can be used to select only the columns that matter to you in an order that makes sense to you. To select only "col_a" of the DataFrame df, use *df["col_a"]*

```
#To select "col_a" and "col_b" of df, use

df[["col_a", "col_b"]]
```

Example 1

```
'''Create a DataFrame called individuals that contains only the
individuals column of homelessness.
Print the head of the result.'''

# Select the individuals column
individuals = homelessness["individuals"]

# Print the head of the result
print(individuals.head())

'''Create a DataFrame called state_fam that contains only the state
and family_members columns of homelessness, in that order.
Print the head of the result.'''

# Select the state and family_members columns
state_fam = homelessness[["state", "family_members"]]

# Print the head of the result
print(state_fam.head())

'''Create a DataFrame called ind_state that contains the individuals
and state columns of homelessness, in that order.
Print the head of the result.'''

# Select only the individuals and state columns, in that order
ind_state = homelessness[["individuals", "state"]]
```

```
# Print the head of the result
print(ind_state.head())
```

Subsetting rows A large part of data science is about finding which bits of your dataset are interesting. One of the simplest techniques for this is to find a subset of rows that match some criteria. This is sometimes known as filtering rows or selecting rows.

How to subset rows in Pandas

1. Using Boolean Indexing
2. Using the *isin()* method
3. Using the *query()* method
4. Using the *loc* or *iloc* accessor

The most common is to use relational operators to return True or False for each row, then pass that inside square brackets as shown below

```
dogs[dogs["height_cm"] > 60]
dogs[dogs["color"] == "tan"]
```

Substet rows using Boolean Indexing You can filter for multiple conditions at once by using the "bitwise and" operator, &.

```
dogs[(dogs["height_cm"] > 60) & (dogs["color"] == "tan")]
```

Example 1

```
'''Filter homelessness for cases where the number of individuals is
greater than ten thousand, assigning to ind_gt_10k.
View the printed result.'''

# Filter for rows where individuals is greater than 10000
ind_gt_10k = homelessness[homelessness["individuals"] > 10000]

# See the result
print(ind_gt_10k)
```

Example 2

```
'''
Filter homelessness for cases where the USA Census region is
"Mountain", assigning to mountain_reg.
View the printed result.'''

# Filter for rows where region is Mountain
mountain_reg = homelessness[homelessness["region"] == "Mountain"]
```

```python
# See the result
print(mountain_reg)
```

Example 3

```python
'''Filter homelessness for cases where the number of family_members is
less than one thousand and the region is "Pacific", assigning to
fam_lt_1k_pac.
View the printed result.'''
# Filter for rows where family_members is less than 1000
# and region is Pacific
fam_lt_1k_pac = homelessness[(homelessness["family_members"] < 1000) &
(homelessness["region"] == "Pacific")]

# See the result
print(fam_lt_1k_pac)
```

Subset rows by using .isin() method You can use the isin() method to filter rows based on whether their values are present in a specified list.

Example 1

```python
'''
Filter homelessness for cases where the USA census region is "South
Atlantic" or it is "Mid-Atlantic", assigning to south_mid_atlantic.
View the printed result.'''

# Subset for rows in South Atlantic or Mid-Atlantic regions
south_mid_atlantic = homelessness[(homelessness["region"] == "South
Atlantic") | (homelessness["region"] == "Mid-Atlantic")]

# See the result
print(south_mid_atlantic)
```

Explanation

1. **homelessness["region"] == "South Atlantic":** This is a boolean expression that creates a boolean Series with True for rows where the value in the "region" column is equal to "South Atlantic" and False for all other rows.
2. **homelessness["region"] == "Mid-Atlantic":** Similarly, this boolean expression creates another boolean Series with True for rows where the value in the "region" column is equal to "Mid-Atlantic" and False for all other rows.
3. **(homelessness["region"] == "South Atlantic") | (homelessness["region"] == "Mid-Atlantic"):** The | operator performs element-wise OR operation between the two boolean Series created above. It results in a new boolean Series where each element represents the result of the OR operation for the corresponding rows.
4. **homelessness[(...) | (...)]:** This syntax uses the boolean Series from the previous step to subset the rows in the "homelessness" DataFrame. It selects rows where either of the

conditions is True, meaning rows where the "region" is either "South Atlantic" or "Mid-Atlantic."

5. **south_mid_atlantic:** The result of the above subsetting operation is assigned to a new DataFrame called "south_mid_atlantic," which contains only the rows from the "homelessness" DataFrame where the region is "South Atlantic" or "Mid-Atlantic."
6. The code then prints the "south_mid_atlantic" DataFrame, which will display only the rows from the original "homelessness" DataFrame where the region is either "South Atlantic" or "Mid-Atlantic." This allows you to focus on and analyze data related to these specific regions.

Example 2

```
'''Filter homelessness for cases where the USA census state is in the
list of Mojave states, canu, assigning to mojave_homelessness.
View the printed result.'''

# The Mojave Desert states
canu = ["California", "Arizona", "Nevada", "Utah"]

# Filter for rows in the Mojave Desert states
mojave_homelessness = homelessness[homelessness["state"].isin(canu)]

# See the result
print(mojave_homelessness)
```

Adding new Columns You can create new columns from scratch, but it is also common to derive them from other columns, for example, by adding columns together or by changing their units. This is called transformation or feature engineering

Example 1

```
'''Add a new column to homelessness, named total, containing the sum
of the individuals and family_members columns.'''

# Add total col as sum of individuals and family_members
homelessness["total"] = homelessness["individuals"] +
homelessness["family_members"]

'''Add another column to homelessness, named p_individuals, containing
the proportion of homeless people in each state who are
individuals.'''

# Add p_individuals col as proportion of total that are individuals
homelessness["p_individuals"] = (homelessness["individuals"] /
homelessness["total"]) * 100

# See the result
print(homelessness)
```

What we have learnt

1. Sorting rows
2. Subsetting columns
3. Subsetting rows
4. Adding new columns

Exercise You will answer the question " Which state has the highest number of homeless individuals per 10,000 people in the state?" **Instructions**

- Add a column to homelessness, indiv_per_10k, containing the number of homeless individuals per ten thousand people in each state.
- Subset rows where indiv_per_10k is higher than 20, assigning to high_homelessness.
- Sort high_homelessness by descending indiv_per_10k, assigning to high_homelessness_srt.
- Select only the state and indiv_per_10k columns of high_homelessness_srt and save as result. Look at the result.

```
import pandas as pd
```