C4 – Exploratory Data Analysis I

Read Carefully

- For each week's in-class exercises and/or homework, please create a new Jupyter notebook file with the following naming convention:

ID_Name_C(no.).ipynb

for example,

6619999_Johnwick_C4.ipynb

What is Exploration Data Analysis (EDA)

- 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!

Exploratory Data Analysis

The process of reviewing and cleaning data to...

- derive insights
- generate hypotheses

Remember from the last lecture (Lecture3), we can explore the data using various Pandas methods or attributes such as .head(), .info(), .describe(), .shape, etc.

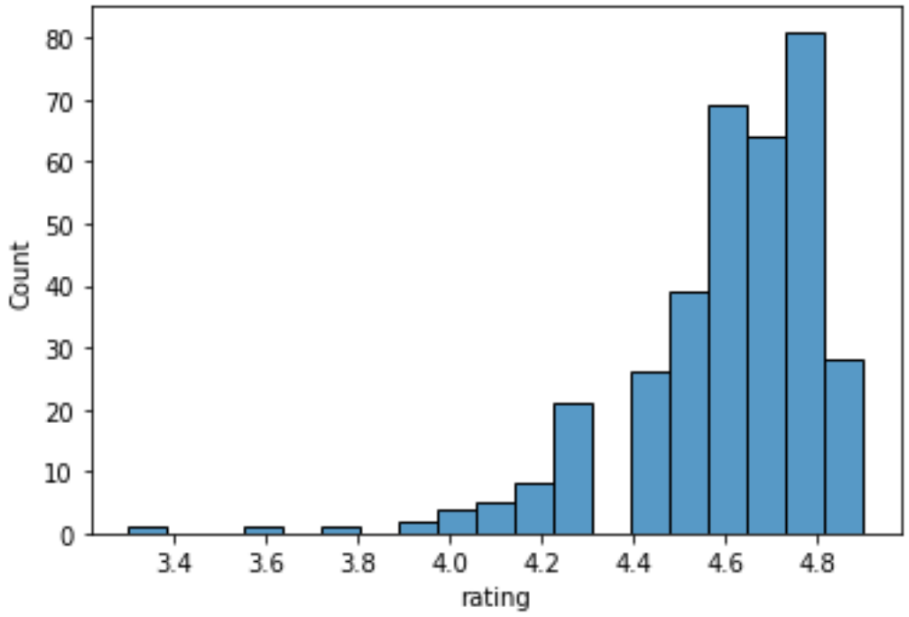
Gathering more .info()

```
books.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 350 entries, 0 to 349
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
--   ------  --------------  -----
 0   name    350 non-null    object
 1   author  350 non-null    object
 2   rating  350 non-null    float64
 3   year    350 non-null    int64
 4   genre   350 non-null    object
dtypes: float64(1), int64(1), object(3)
memory usage: 13.8+ KB
```

- Apart from exploring the initial statistics summary, it is also effective to visualize the data.
- Try the following codes.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
books = pd.read_csv('clean_books.csv')
sns.histplot(data=books, x='rating')
plt.show()
#You may add binwdith to histplot to change percentage point.
```



A closer look at categorical columns

```
books.value_counts("genre")
```

```
genre
Non Fiction    179
Fiction        131
Childrens       40
dtype: int64
```

For .value_counts(), remember we can select a specific column to get a closer look at the information. Try.

books['genre'].value_counts()

A new version of Pandas also supports

books.value_counts('genre')

1. **Load clean_umployment.csv to unemployment and complete the following tasks.**
   - **Import the seaborn visualization libraries.**
   - **Create a histogram of the distribution of 2021 unemployment percentages across all countries in unemployment; show a full percentage (binwidth = 0.1) point in each bin.**

Data Validation

- Data validation **is an important early step in EDA**. We want to understand whether data types and ranges are as expected before we progress too far in our analysis!

Validating data types

```
books.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 350 entries, 0 to 349
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
--   ------  --------------  -----
 0   name    350 non-null    object
 1   author  350 non-null    object
 2   rating  350 non-null    float64
 3   year    350 non-null    float64
 4   genre   350 non-null    object
dtypes: float64(1), int64(1), object(3)
memory usage: 13.8+ KB
```

```
books.dtypes
```

```
name       object
author     object
rating    float64
year      float64
genre      object
dtype: object
```

Validating Categorial Data

- We can validate categorical data by comparing values in a column to a list of expected values using .isin(), which can either be applied to a Series as we'll show here or to an entire DataFrame.
- For example, if the values in the genre column are limited to "Fiction" and "Non Fiction" by passing these genres as a list of strings to .isin(). The function returns a Series of the same size and shape as the original but with True and False in place of all values, depending on whether the value from the original Series was included in the list passed to .isin().
- We can see that some values are False.

Validating categorical data

```
books["genre"].isin(["Fiction", "Non Fiction"])
```

```
0       True
1       True
2       True
3       True
4      False
       ...
345     True
346     True
347     True
348     True
349    False
Name: genre, Length: 350, dtype: bool
```

Note we can use ~ to express it is not in … For example,

~books['genre'].isin(['Fiction','Non Ffiction'])

Validating categorical data

```
books[books["genre"].isin(["Fiction", "Non Fiction"])].head()
```

	name	author	rating	year	genre
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	2016	Non Fiction
1	11/22/63: A Novel	Stephen King	4.6	2011	Fiction
2	12 Rules for Life	Jordan B. Peterson	4.7	2018	Non Fiction
3	1984 (Signet Classics)	George Orwell	4.7	2017	Fiction
5	A Dance with Dragons	George R. R. Martin	4.4	2011	Fiction

Working with Numerical Data

- We can select and view only the numerical columns in a DataFrame by calling the select_dtypes method and passing "number" as the argument.

Validating numerical data

```
books.select_dtypes("number").head()
```

	rating	year
0	4.7	2016
1	4.6	2011
2	4.7	2018
3	4.7	2017
4	4.8	2019

Working with Numerical Data

- And we can view a more detailed picture of the distribution of year data using Seaborn's boxplot function. The boxplot shows the boundaries of each quartile of year data: as we saw using min and max, the lowest year is 2009 and the highest year is 2019. The 25th and 75th percentiles are 2010 and 2016 and the median year is 2013.

Validating numerical data

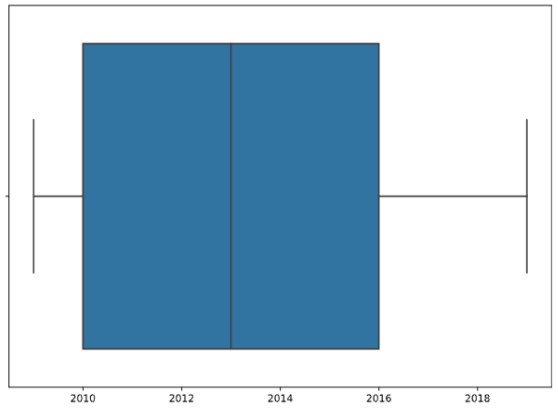
```
books["year"].min()
```

```
2009
```

```
books["year"].max()
```

```
2019
```

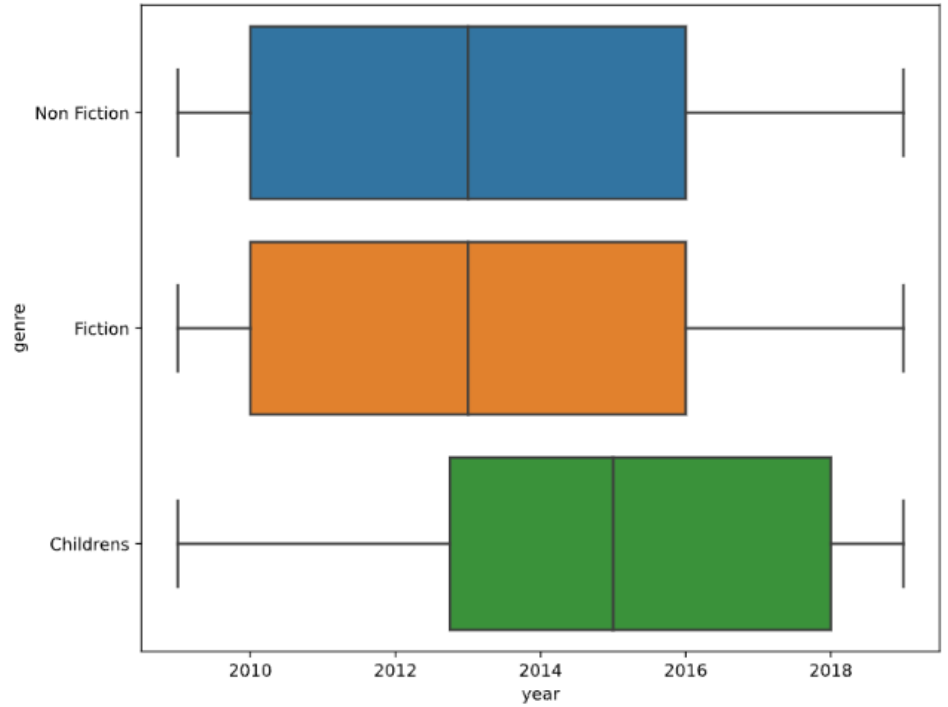
```
sns.boxplot(data=books, x="year")
plt.show()
```



It is possible to view the year data grouped by a categorical variable such as genre by setting the y keyword argument. It looks like the children's books in our dataset have slightly later publishing years in general, but the range of years is the same for all genres.

Validating numerical data

```
sns.boxplot(data=books, x="year", y="genre")
```



Let's Practice.
Validating continents

Your colleague has informed you that the data on unemployment from countries in Oceania is not reliable, and you'd like to identify and exclude these countries from your unemployment data. The .isin() function can help with that!

Your task is to use .isin() to 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.

Load clearn_unemployment.csv to unemployment

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

  Use Boolean indexing to print the unemployment DataFrame without any of the data related to countries in Oceania.

- **Print the minimum and maximum unemployment rates, in that order, during 2021.**

  Create a boxplot of 2021 unemployment rates, broken down by continent.

- We can explore the characteristics of subsets of data further with the help of the .groupby() function, which groups data by a given category, allowing the user to chain an aggregating function like .mean() or .count() to describe the data within each group.
- The .agg() function, short for aggregate, allows us to apply aggregating functions. By default, it aggregates data across all rows in a given column and is typically used when we want to apply more than one function.

Exploring groups of data

- `.groupby()` groups data by category
- Aggregating function indicates how to summarize grouped data

```
books.groupby("genre").mean()
```

```
|       genre |   rating |        year |
|-------------|----------|-------------|
|   Childrens | 4.780000 | 2015.075000 |
|     Fiction | 4.570229 | 2013.022901 |
| Non Fiction | 4.598324 | 2013.513966 |
```

Aggregating ungrouped data

- `.agg()` applies aggregating functions across a DataFrame

```
books.agg(["mean", "std"])
```

```
|      |   rating |        year |
|------|----------|-------------|
| mean | 4.608571 | 2013.508571 |
|  std | 0.226941 |     3.28471 |
```

books.groupby('genre')[['rating', 'year']].mean()

books.groupby('genre').mean(numeric_only = True)

books[['rating', 'year']].agg(['std','mean',])

numeric_cols = list(books.select_dtypes(include=np.number).columns)
books[numeric_cols].agg(['std','mean',])

- We can even use a dictionary to specify which aggregation functions to apply to which columns. The keys in the dictionary are the columns to apply the aggregation, and each value is a list of the specific aggregating functions to apply to that column.

Specifying aggregations for columns

```
books.agg({"rating": ["mean", "std"], "year": ["median"]})
```

```
|        |   rating |   year |
|--------|----------|--------|
|   mean | 4.608571 |    NaN |
|    std | 0.226941 |    NaN |
| median |      NaN | 2013.0 |
```

Named summary columns

- By combining .agg() and .groupby(), we can apply these new exploration skills to grouped data. Maybe we'd like to show the mean and standard deviation of rating for each book genre along with the median year. We can create named columns with our desired aggregations by using the .agg() function and creating named tuples inside it.

Named summary columns

```
books.groupby("genre").agg(
    mean_rating=("rating", "mean"),
    std_rating=("rating", "std"),
    median_year=("year", "median")
)
```

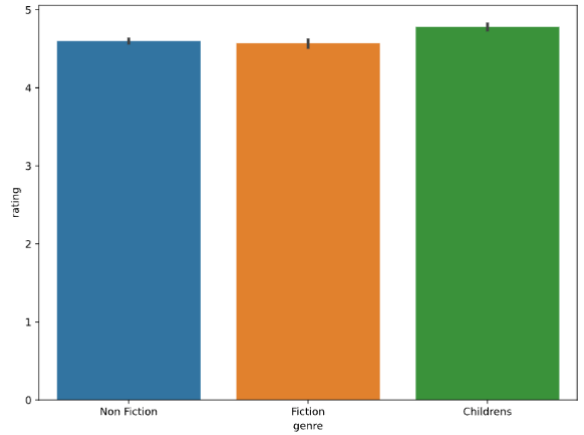
genre	mean_rating	std_rating	median_year
Childrens	4.780000	0.122370	2015.0
Fiction	4.570229	0.281123	2013.0
Non Fiction	4.598324	0.179411	2013.0

Visualizing categorical summaries

We can display similar information visually using a barplot. In Seaborn, bar plots will automatically calculate the mean of a quantitative variable like rating across grouped categorical data, such as the genre category we've been looking at. In Seaborn, bar plots also show a 95% confidence interval for the mean as a vertical line on the top of each bar.

Visualizing categorical summaries

```
sns.barplot(data=books, x="genre", y="rating")
plt.show()
```



Let's Practice.

Print the mean and standard deviation of the unemployment rates for each year. (using .agg() as shown in the example above)

	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
mean	8.409286	8.315440	8.317967	8.344780	8.179670	8.058901	7.925879	7.668626	7.426429	7.243736	8.420934	8.390879
std	6.248887	6.266795	6.367270	6.416041	6.284241	6.161170	6.045439	5.902152	5.818915	5.696573	6.040915	6.067192

Print the mean and standard deviation of the unemployment rates for each year (not all data is shown below).

	2010		2011		2012		2013		2014		...	2017		2018	
	mean	std	mean	std	mean	std	mean	std	mean	std	...	mean	std	mean	std
continent															
Africa	9.343585	7.411259	9.369245	7.401556	9.240755	7.264542	9.132453	7.309285	9.121321	7.291359	...	9.284528	7.407620	9.237925	7.358425
Asia	6.240638	5.146175	5.942128	4.779575	5.835319	4.756904	5.852128	4.668405	5.853191	4.681301	...	6.171277	5.277201	6.090213	5.409128
Europe	11.008205	6.392063	10.947949	6.539538	11.325641	7.003527	11.466667	6.969209	10.971282	6.759765	...	8.359744	5.177845	7.427436	4.738206
North America	8.663333	5.115805	8.563333	5.377041	8.448889	5.495819	8.840556	6.081829	8.512222	5.801927	...	7.391111	5.326446	7.281111	5.253180
Oceania	3.622500	2.054721	3.647500	2.008466	4.103750	2.723118	3.980000	2.640119	3.976250	2.659205	...	3.872500	2.492834	3.851250	2.455893
South America	6.870833	2.807058	6.518333	2.801577	6.410833	2.936508	6.335000	2.808780	6.347500	2.834332	...	7.281667	3.398994	7.496667	3.408856

Named aggregations.
You've seen how .groupby() and .agg() can be combined to show summaries across categories. 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.

Your task is to 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. And of course, you'll rename the columns so that their contents are clear!

From the given code –

```
continent_summary = unemployment.groupby("continent").agg()
```

Try

- **Create a column called mean_rate_2021, which shows the mean 2021 unemployment rate for each continent.**

- **Create a column called std_rate_2021, which shows the standard deviation of the 2021 unemployment rate for each continent.**

Visualizing categorical summaries

As you've learned in this lecture, 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.

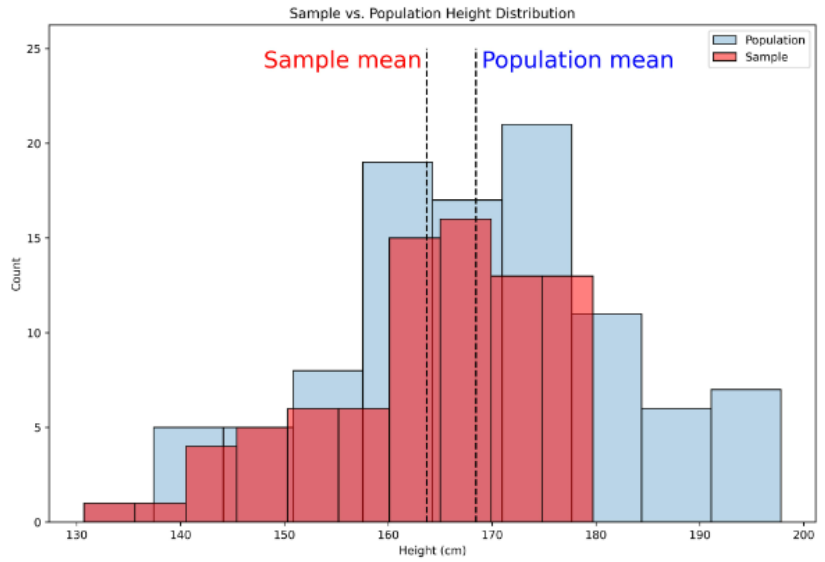
Your task is to create a bar plot to visualize the means and confidence intervals of unemployment rates across the different continents.

- **Create a bar plot showing continents on the x-axis and their respective average 2021 unemployment rates on the y-axis.**

- So, why is it important to deal with missing data? Well, it can affect distributions. As an example, we collect the heights of students at a high school. If we fail to collect the heights of the oldest students, who were taller than most of our sample, then our sample mean will be lower than the population mean.
- Put another way, our data is less representative of the underlying population. In this case, parts of our population aren't proportionately represented. This misrepresentation can lead us to draw incorrect conclusions, like thinking that, on average, students are shorter than they really are.

Why is missing data a problem?

- Affects distributions
  - Missing heights of taller students
- Less representative of the population
  - Certain groups disproportionately represented, e.g., lacking data on oldest students
- Can result in drawing incorrect conclusions

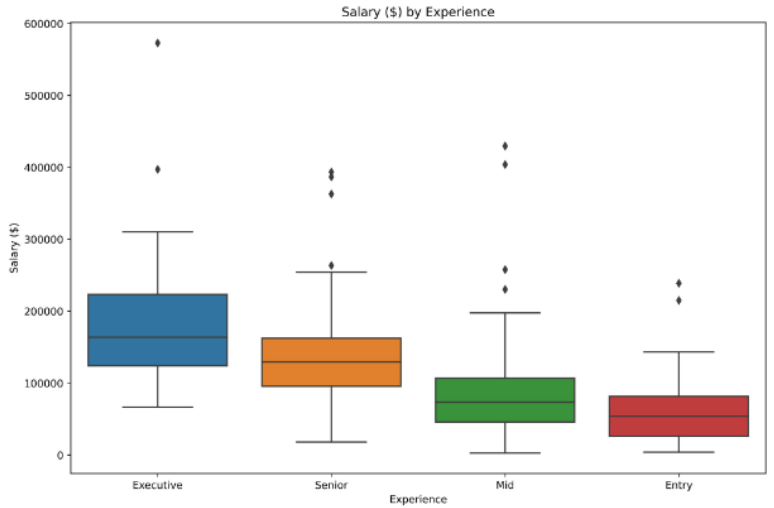


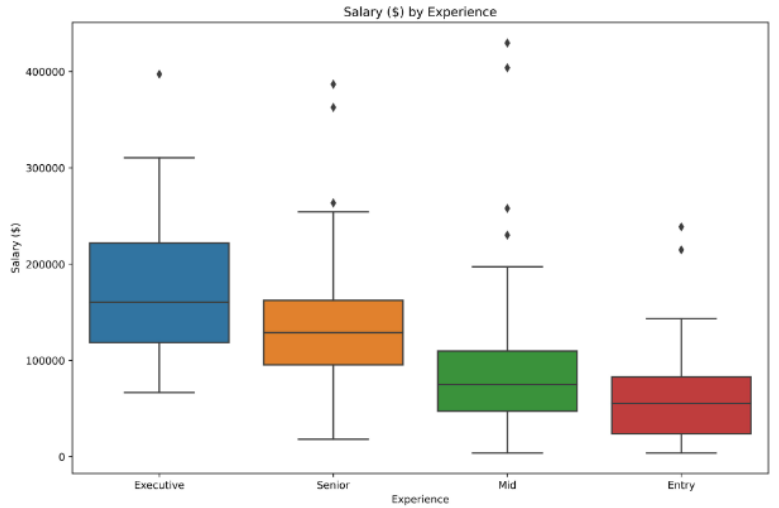
Data professionals' job data

Column	Description	Data type
`Working_Year`	Year the data was obtained	Float
`Designation`	Job title	String
`Experience`	Experience level e.g., `"Mid"`, `"Senior"`	String
`Employment_Status`	Type of employment contract e.g., `"FT"`, `"PT"`	String
`Employee_Location`	Country of employment	String
`Company_Size`	Labels for company size e.g., `"S"`, `"M"`, `"L"`	String
`Remote_Working_Ratio`	Percentage of time working remotely	Integer
`Salary_USD`	Salary in US dollars	Float

To highlight the impact of missing values, let's look at salaries by experience level. Now, let's compare this to the same data with some missing values. The y-axis shows that the largest salary is around 150000 dollars less in the second plot!

Salary by experience level





Checking for missing values

```
print(salaries.isna().sum())
```

```
Working_Year            12
Designation             27
Experience              33
Employment_Status       31
Employee_Location       28
Company_Size            40
Remote_Working_Ratio    24
Salary_USD              60
dtype: int64
```

Strategies for addressing missing data

- Drop missing values
  - 5% or less of total values
- Impute mean, median, mode
  - Depends on distribution and context
- Impute by sub-group
  - Different experience levels have different median salary

To calculate our missing values threshold, we multiply the length of our DataFrame by five percent, giving us an upper limit of 30, as shown in the example below.

Dropping missing values

```
threshold = len(salaries) * 0.05
print(threshold)
```

```
30
```

We can use Boolean indexing to filter for columns with missing values less than or equal to this threshold, storing them as a variable called cols_to_drop. Printing cols_to_drop shows four columns. We drop missing values by calling .dropna(), passing cols_to_drop to the subset argument. We set inplace to True so the DataFrame is updated.

Dropping missing values

```
cols_to_drop = salaries.columns[salaries.isna().sum() <= threshold]
print(cols_to_drop)
```

```
Index(['Working_Year', 'Designation', 'Employee_Location',
       'Remote_Working_Ratio'],
      dtype='object')
```

```
salaries.dropna(subset=cols_to_drop, inplace=True)
```

We then filter for the remaining columns with missing values, giving us four columns. To impute the mode for the first three columns, we loop through them and call the .fillna() method, passing the respective column's mode and indexing the first item, which contains the mode, in square brackets.

Imputing a summary statistic

```
cols_with_missing_values = salaries.columns[salaries.isna().sum() > 0]
print(cols_with_missing_values)
```

```
Index(['Experience', 'Employment_Status', 'Company_Size', 'Salary_USD'],
    dtype='object')
```

```
for col in cols_with_missing_values[:-1]:
    salaries[col].fillna(salaries[col].mode()[0])
```

Checking for missing values again, we see salary_USD is now the only column with missing values and the volume has changed from 60 missing values to 41. This is because some rows may have contained missing values for our subset columns as well as salary, so they were dropped.

Checking the remaining missing values

```
print(salaries.isna().sum())
```

```
Working_Year             0
Designation              0
Experience               0
Employment_Status        0
Employee_Location        0
Company_Size             0
Remote_Working_Ratio     0
Salary_USD              41
```

We'll impute median salary by experience level by grouping salaries by experience and calculating the median. We use the .to_dict() method, storing the grouped data as a dictionary. Printing the dictionary returns the median salary for each experience level, with executives earning the big bucks!

Imputing by sub-group

```
salaries_dict = salaries.groupby("Experience")["Salary_USD"].median().to_dict()
print(salaries_dict)
```

```
{'Entry': 55380.0, 'Executive': 135439.0, 'Mid': 74173.5, 'Senior': 128903.0}
```

We then impute using the .fillna() method, providing the Experience column and calling the .map() method, inside which we pass the salaries dictionary.

Imputing by sub-group

```
salaries["Salary_USD"] = salaries["Salary_USD"].fillna(salaries["Experience"].map(salaries_dict))
```

No more missing values!

```
print(salaries.isna().sum())
```

```
Working_Year            0
Designation             0
Experience              0
Employment_Status       0
Employee_Location       0
Company_Size            0
Remote_Working_Ratio    0
Salary_USD              0
dtype: int64
```

Let’s Practice. Load airline_unclean.csv to planes with index_col = 0 argument.

- Print the number of missing values in each column of the DataFrame.
  - ✓ Calculate how many observations five percent of the planes DataFrame is equal to.
  - ✓ Create cols_to_drop by applying boolean indexing to columns of the DataFrame with missing values less than or equal to the threshold.
  - ✓ Use this filter to remove missing values and save the updated DataFrame.

```
Airline            427
Date_of_Journey    322
Source             187
Destination        347
Route              256
Dep_Time           260
Arrival_Time       194
Duration           214
Total_Stops        212
Additional_Info    589
Price              616
dtype: int64
```

```
Airline              0
Date_of_Journey      0
Source               0
Destination          0
Route                0
Dep_Time             0
Arrival_Time         0
Duration             0
Total_Stops          0
Additional_Info    300
Price              368
dtype: int64
```

Strategies for remaining missing data.

So far, the five percent rule has worked nicely for the planes dataset, eliminating missing values from nine out of 11 columns!

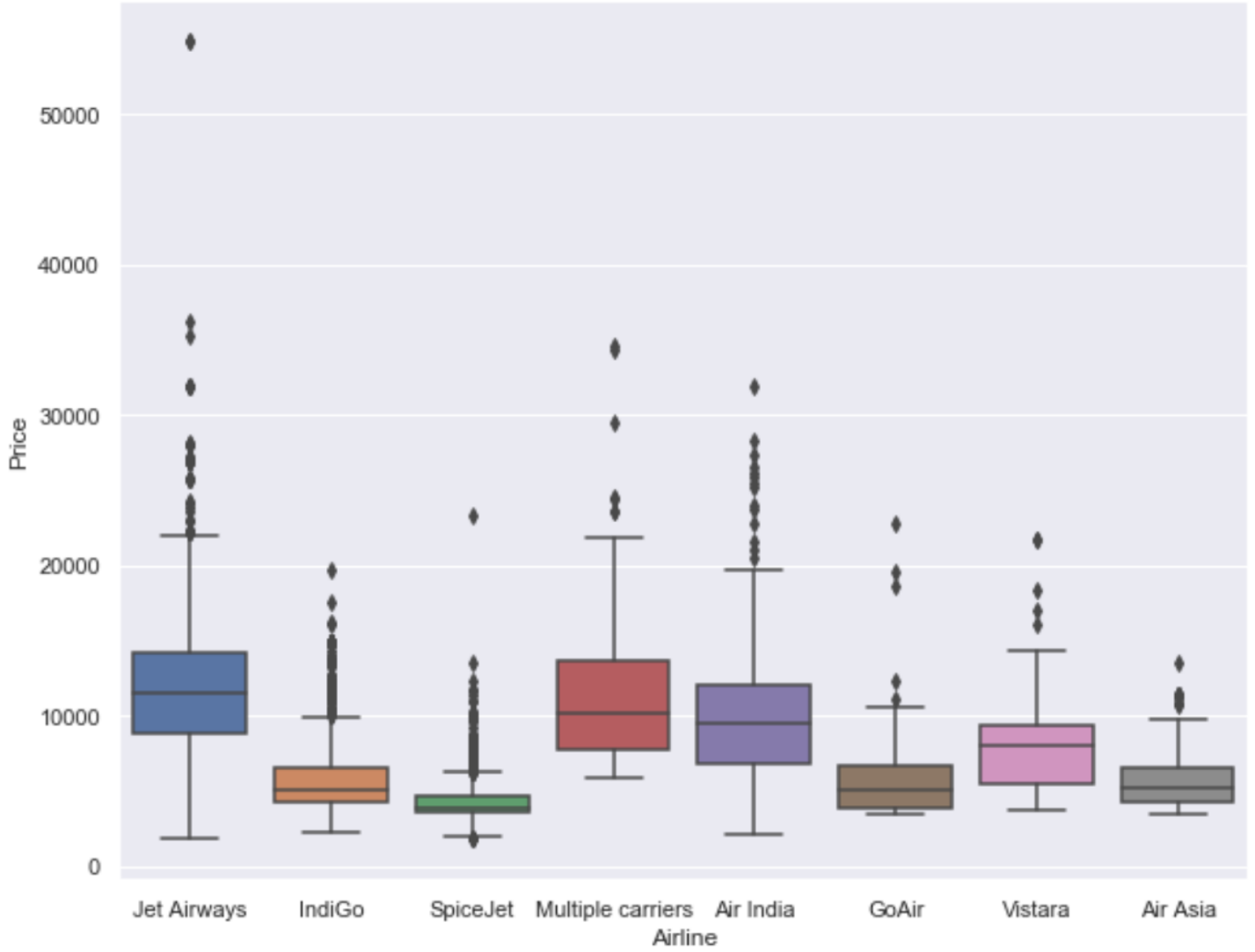
Now, you need to decide what to do with the "Additional_Info" and "Price" columns, which are missing 300 and 368 values respectively.

You'll first take a look at what "Additional_Info" contains, then visualize the price of plane tickets by different airlines.

Run the following code to create boxplot for Price by Airline, and to observe mean value.

```
# Check the values of the Additional_Info column
print(planes["Additional_Info"].value_counts())

# Create a box plot of Price by Airline
sns.boxplot(data=planes, x='Airline', y='Price')
sns.set(rc={"figure.figsize":(8, 6)}) #width=8, #height=6
plt.show()
```



Question

How should you deal with the missing values in "Additional_Info" and "Price"?

1. Remove the “Additional_Info” column and impute the mean for missing values of “Price”
2. Remove “No info” values from “Additional_Info” and impute the median for missing values of “Price”
3. Remove the “Additional_Info” column and impute the mean by “Airline” for missing values of “Price”
4. Remove the “Additional_Info” column and impute the median by “Airline” for missing values of “Price”

Let’s drop the “Additional_Info” column, with **planes = planes.drop(columns = ['Additional_Info'])**

Group `planes` by airline and calculate the median price.

- **Convert the grouped median prices to a dictionary.**
- **Conditionally impute missing values for `"Price"` by mapping values in the `"Airline column"` based on `prices_dict`.**
- **Check for remaining missing values.**

Working with Categorial Data.

Now let's explore how to create and analyze categorical data. Recall that we can use the select_dtypes method to filter any non-numeric data. Chaining .head() allows us to preview these columns in our salaries DataFrame, showing columns such as Designation, Experience, Employment_Status, and Company_Size.

Previewing the data

```
print(salaries.select_dtypes("object").head())
```

```
  Designation                Experience    Employment_Status    Employee_Location    Company_Size
0 Machine Learning Scientist Senior        FT                   JP                   S
1 Big Data Engineer          Senior        FT                   GB                   M
2 Product Data Analyst       Mid           FT                   HN                   S
3 Machine Learning Engineer  Senior        FT                   US                   L
4 Data Analyst               Entry         FT                   US                   L
```

Let's examine frequency of values in the Designation column. The output is truncated by pandas automatically since there are so many different job titles!

Job titles

```
print(salaries["Designation"].value_counts())
```

```
Data Scientist                    124
Data Engineer                     114
Data Analyst                       79
Machine Learning Engineer          34
Research Scientist                 16
Data Architect                     10
Data Science Manager                8
Big Data Engineer                   8
Data Science Consultant             7
...
```

We can count how many unique job titles there are using pandas .nunique() method. There are 50 in total!

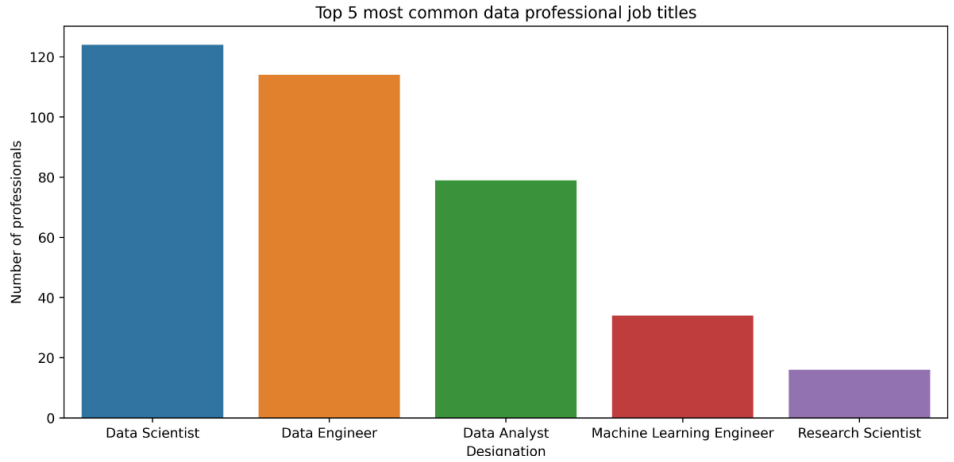
Job titles

```
print(salaries["Designation"].nunique())
```

```
50
```

If we plot the graph using a bar chart, the fifth most popular job title, Research Scientist, appears less than 20 times.

Job titles




```
salaries_count = salaries['Designation'].value_counts().iloc[0:5]
print(salaries_count.index)salaries_count.plot(kind='bar')
```

The current format of the data limits our ability to generate insights. We can use the pandas.Series.string.contains() method, which allows us to search a column for a specific string or multiple strings.

Say we want to know which job titles have Scientist in them. We use the str.contains() method on the Designation column, passing the word Scientist. This returns True or False values depending on whether the row contains this word.

- Current format limits our ability generate insights
- `pandas.Series.str.contains()`
  - Search a column for a specific string or multiple strings

```
salaries["Designation"].str.contains("Scientist")
```

```
0       True
1      False
2      False
3      False
       ...
515    False
516    False
517     True
Name: Designation, Length: 518, dtype: bool
```

What if we want to filter for rows containing one or more phrases?

- Words of interest: Machine Learning **or** AI

```
salaries["Designation"].str.contains("Machine Learning|AI")
```

Now we have a sense of how this method works, let's define a list of job titles we want to find. We start by creating a list with the different categories of data roles, which will become the values of a new column in our DataFrame.

Finding multiple phrases in strings

```
job_categories = ["Data Science", "Data Analytics",
                  "Data Engineering", "Machine Learning",
                  "Managerial", "Consultant"]
```

We then need to create variables containing our filters. We will look for Data Scientist or NLP for data science roles. We'll use Analyst or Analytics for data analyst roles. We repeat this for data engineer, machine learning engineer, managerial, and consultant roles.

Finding multiple phrases in strings

```
data_science = "Data Scientist|NLP"
data_analyst = "Analyst|Analytics"
data_engineer = "Data Engineer|ETL|Architect|Infrastructure'
ml_engineer = "Machine Learning|ML|Big Data|AI"
manager = "Manager|Head|Director|Lead|Principal|Staff"
consultant = "Consultant|Freelance"
```

Finding multiple phrases in strings

```
conditions = [
    (salaries["Designation"].str.contains(data_science)),
    (salaries["Designation"].str.contains(data_analyst)),
    (salaries["Designation"].str.contains(data_engineer)),
    (salaries["Designation"].str.contains(ml_engineer)),
    (salaries["Designation"].str.contains(manager)),
    (salaries["Designation"].str.contains(consultant))
]
```

Creating the categorical column

```
salaries["Job_Category"] = np.select(conditions,
                                     job_categories,
                                     default="Other")
```

Previewing job categories

```
print(salaries[["Designation", "Job_Category"]].head())
```

```
     Designation                  Job_Category
0    Machine Learning Scientist   Machine Learning
1    Big Data Engineer            Data Engineering
2    Product Data Analyst         Data Analytics
3    Machine Learning Engineer    Machine Learning
4    Data Analyst                 Data Analytics
```

Visualizing job category frequency

```
sns.countplot(data=salaries, x="Job_Category")
plt.show()
```

Let’s Practice (Load the data planes data set again)

Try (We select columns with object data type)

```
# Filter the DataFrame for object columns
non_numeric = planes.select_dtypes("object")

# Loop through columns
for col in non_numeric.columns:

  # Print the number of unique values
  print(f"Number of unique values in {col} column: ", non_numeric[col].nunique())
```

The number of unique values in the "Duration" column of planes is shown.

Then calling planes["Duration"].head(), we see the following values:

```
0      19h
1    5h 25m
2    4h 45m
3    2h 25m
4   15h 30m
Name: Duration, dtype: object
```

Looks like this won't be simple to convert to numbers. However, you could categorize flights by duration and examine the frequency of different flight lengths!

You'll create a "Duration_Category" column in the planes DataFrame. Before you can do this you'll need to create a list of the values you would like to insert into the DataFrame, followed by the existing values that these should be created from.

Create short_flights, a string to capture values of "0h", "1h", "2h", "3h", or "4h".

Create medium_flights to capture any values between five and nine hours.

Create long_flights to capture any values from 10 hours to 16 hours inclusive.

Adding duration categories

Now that you've set up the categories and values you want to capture, it's time to build a new column to analyze the frequency of flights by duration!

The variables flight_categories, short_flights, medium_flights, and long_flights that you previously created are needed.

Additionally, the following packages have been imported:
pandas as pd
numpy as np
seaborn as sns
matplotlib.pyplot as plt

Create conditions, a list containing subsets of planes["Duration"] based on short_flights, medium_flights, and long_flights.

Create the "Duration_Category" column by calling a function that accepts your conditions list and flight_categories, setting values not found to "Extreme duration".

Create a plot showing the count of each category.

Working with Numerical Data.

It’s time to switch our focus to working with numeric data.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 594 entries, 0 to 593
Data columns (total 9 columns):
 #   Column               Non-Null Count  Dtype  
--   ------               --------------  -----  
 0   Working_Year         594 non-null    int64  
 1   Designation          567 non-null    object 
 2   Experience           561 non-null    object 
 3   Employment_Status    563 non-null    object 
 4   Salary_In_Rupees     566 non-null    object 
 5   Employee_Location    554 non-null    object 
 6   Company_Location     570 non-null    object 
 7   Company_Size         535 non-null    object 
 8   Remote_Working_Ratio  571 non-null    float64
dtypes: float64(1), int64(1), object(7)
memory usage: 41.9+ KB
None
```

To obtain Salary in USD we'll need to perform a few tasks. First, we need to remove the commas from the values in the Salary_In_Rupees column. Next, we change the data type to float. Lastly, we'll make a new column by converting the currency.

```
print(salaries["Salary_In_Rupees"].head())
```

```
0    20,688,070.00
1     8,674,985.00
2     1,591,390.00
3    11,935,425.00
4     5,729,004.00
Name: Salary_In_Rupees, dtype: object
```

Converting strings to numbers

```
pd.Series.str.replace("characters to remove", "characters to replace them with")
```

```
salaries["Salary_In_Rupees"] = salaries["Salary_In_Rupees"].str.replace(",", "")
print(salary["Salary_In_Rupees"].head())
```

```
1    20688070.00
2     8674985.00
3     1591390.00
4    11935425.00
5     5729004.00
Name: Salary_In_Rupees, dtype: object
```

Converting strings to numbers

```
salaries["Salary_In_Rupees"] = salaries["Salary_In_Rupees"].astype(float)
```

- 1 Indian Rupee = 0.012 US Dollars

```
salaries["Salary_USD"] = salaries["Salary_In_Rupees"] * 0.012
```

Previewing the new column

```
print(salaries[["Salary_In_Rupees", "Salary_USD"]].head())
```

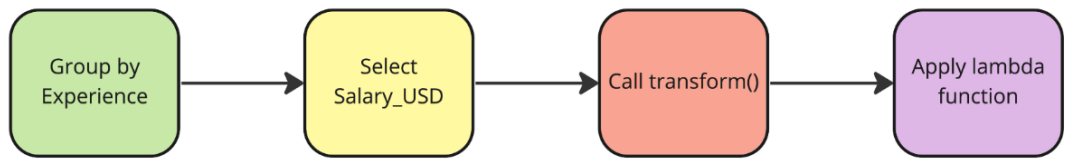
```
   Salary_In_Rupees  Salary_USD
0        20688070.0  248256.840
1         8674985.0  104099.820
2         1591390.0   19096.680
3        11935425.0  143225.100
4         5729004.0   68748.048
```

Adding summary statistics into a DataFrame

```
salaries.groupby("Company_Size")["Salary_USD"].mean()
```

```
Company_Size
L    111934.432174
M    110706.628527
S     69880.980179
Name: Salary_USD, dtype: float64
```

Adding summary statistics into a DataFrame




```
salaries["std_dev"] = salaries.groupby("Experience")["Salary_USD"].transform(lambda x: x.std())
```

Adding summary statistics into a DataFrame

```
print(salaries[["Experience", "std_dev"]].value_counts())
```

```
Experience          std_dev
SE                  52995.385395        257
MI                  63217.397343        197
EN                  43367.256303         83
EX                  86426.611619         24
```

Adding summary statistics into a DataFrame

```
salaries["median_by_comp_size"] = salaries.groupby("Company_Size") \
                                          ["Salary_USD"].transform(lambda x: x.median())
```

```
print(salaries[["Company_Size", "median_by_comp_size"]].head())
```

```
    Company_Size     median_by_comp_size
0    S                60833.424
1    M                105914.964
2    S                60833.424
3    L                95483.400
4    L                95483.400
```

Complete the following tasks.

Print the first five values of the "Duration" column.

Remove "h", "m", " " from the column and make the data format to be hh.mm

Convert the column to float data type.

Plot a histogram of "Duration" values using

```
sns.histplot(data=planes, x="Duration")
```

Adding descriptive statistics

Now "Duration" and "Price" both contain numeric values in planes DataFrame, you would like to calculate summary statistics for them that are conditional on values in other columns.

Add a column to planes containing the standard deviation of "Price" based on "Airline".

Calculate the median for "Duration" by "Airline", storing it as a column called "airline_median_duration".

Find the mean "Price" by "Destination", saving it as a column called "price_destination_mean".

Handling Outliers

What is an outlier?

- An observation far away from other data points
  - Median house price: $400,000
  - Outlier house price: $5,000,000

- Should consider why the value is different:
  - Location, number of bedrooms, overall size etc

Using descriptive statistics

```
print(salaries["Salary_USD"].describe())
```

```
count       518.000
mean     104905.826
std       62660.107
min        3819.000
25%       61191.000
50%       95483.000
75%      137496.000
max      429675.000
Name: Salary_USD, dtype: float64
```

We can define an outlier mathematically. First, we need to know the interquartile range, or IQR, which is the difference between the 75th and 25th percentiles

Using the interquartile range

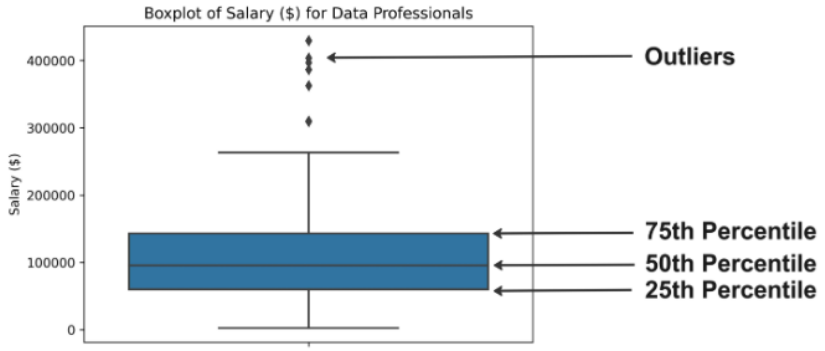
Interquartile range (IQR)

- IQR = 75th - 25th percentile

Recall that these percentiles are included in box plots, like this one showing salaries of data professionals.

IQR in box plots

```
sns.boxplot(data=salaries,
            y="Salary_USD")
plt.show()
```



Once we have the IQR, we can find an upper outlier by looking for values above the sum of the 75th percentile plus 1.5 times the IQR. Lower outliers have values below the sum of the 25th percentile minus 1.5 times the IQR

Using the interquartile range

Interquartile range (IQR)

- IQR = 75th - 25th percentile
- Upper Outliers > 75th percentile + (1.5 * IQR)
- Lower Outliers < 25th percentile - (1.5 * IQR)

Identifying thresholds

```
# 75th percentile
seventy_fifth = salaries["Salary_USD"].quantile(0.75)

# 25th percentile
twenty_fifth = salaries["Salary_USD"].quantile(0.25)

# Interquartile range
salaries_iqr = seventy_fifth - twenty_fifth

print(salaries_iqr)
```

Identifying outliers

```
# Upper threshold
upper = seventy_fifth + (1.5 * salaries_iqr)

# Lower threshold
lower = twenty_fifth - (1.5 * salaries_iqr)

print(upper, lower)
```

Subsetting our data

```
salaries[(salaries["Salary_USD"] < lower) | (salaries["Salary_USD"] > upper)] \
        [["Experience", "Employee_Location", "Salary_USD"]]
```

	Experience	Employee_Location	Salary_USD
0	Mid	DE	76227.0
2	Senior	GB	104100.0
3	Mid	HN	19097.0
4	Senior	US	143225.0
5	Entry	US	68748.0
...	...	...	...
601	Entry	CA	49651.0

Why look for outliers?

- Outliers are extreme values
  - may not accurately represent our data
- Can change the mean and standard deviation
- Statistical tests and machine learning models need normally distributed data

What to do about outliers?

Questions to ask:

- Why do these outliers exist?
  - More senior roles / different countries pay more
  - Consider leaving them in the dataset
- Is the data accurate?
  - Could there have been an error in data collection?
    - If so, remove them

Dropping outliers

```
no_outliers = salaries[(salaries["Salary_USD"] > lower) & (salaries["Salary_USD"] < upper)]
```

```
print(no_outliers["Salary_USD"].describe())
```

Removing outliers

While removing outliers isn't always the way to go, for your analysis, you've decided that you will only include flights where the "Price" is not an outlier.

Therefore, you need to find the upper threshold and then use it to remove values above this from the planes DataFrame.

Find the 75th and 25th percentiles, saving as price_seventy_fifth and price_twenty_fifth, respectively.

Calculate the IQR, storing it as prices_iqr.

Calculate the upper and lower outlier thresholds.

Remove the outliers from planes