Exploratory Data Analysis
by datacamp

head()
info() for descriptive statistics
df.value_counts('col') for a closer look at categorical columns
initial visualization of numerical data with histplot()
.dtypes() to see data types
.astype() to change data type
** common types are str, int, float, dict, list, bool

Validating categorical data
by comparing values in a column to a list of expected values
with .isin()
can run on a series or a dataframe
result is the shape of the original series with either true and false to represent the request of .isin()
example
books['genre'].isin(['Fiction', 'Non Fiction'])
** can use the tilde operator (ie the approx sign) at the beginning of the code block to inver the true/false values

Validating numerical data
df.select_dtypes('number').head()
df['col'].min() or max()
sns.boxplot for quartiles

Exploring groups of data
.groupby() groups data by category
aggregating function indicates how to summarize grouped data
example
books.groupby('genre').mean()
examples of aggregating functions are .sum, .count, .min, .max, .var, .std

Aggregating ungrouped data
.agg() applies aggregating functions across a DataFrame
by default aggregates data across all rows in a given column and is typically used when we want to apply more than one function
can apply more than one at a time
books.agg(['mean', 'std'])
applies only to numeric columns

returns a dataframe only of numerical columns
** can use a dictionary to specify which aggregation functions to apply to which columns
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
example
books.agg({'rating': ['mean', 'std'], 'year': ['median']})

Applying agg to grouped data
Named summary columns
example
books.groupby('genre').agg(mean_rating=('rating', 'mean'), std_rating=('rating', 'std'), median_year=('year', 'median'))

Visualizing categorical summaries
barplots can be good for this
sns.barplot(data, x, y)
plt.show()

Addressing missing data
Why is missing data a problem?
Can affect distributions
Less representative of the population
Can make certain groups disproportianately represented
example - miss the heights of seniors in our assessment of height of high school students, older students tend to be taller, so this could give us a sample mean that is not representative of the population mean

Checking for missing values
example
print(salaries.isna().sum())

Strategies for addressing missing data
Drop missing values if 5% or less of total values are missing
If >5% of total values are missing an option to fill them could be to use a summary statistic like mean, median, mode
 - depends on distribution and context
 - this is known as 'imputation'
imputation is to assign a value to something by inference from the value of the products or processes to which it contributes
** can also impute by sub-group
for example with this example median salary varies by experience, so we could impute different salaries depending on experience

How to calculate if missing values threshold is less than 5%
threshold = len(salaries) * 0.05
print(threshold)

Dropping missing values
Using boolean indexing to filter for columns with missing values less than or equal to the threshold
cols_to_drop = salaries.columns[salaries.isna().sum() <= threshold]
to then drop these columns
salaries.dropna(subset=cols_to_drop, inplace=True)
**setting inplace to True updates the Dataframe

Imputing a summary statistic
then filter for the remaining columns with missing values
cols_with_missing_values = salaries.columns[salaries.isna().sum() > 0]
print(cols_with_missing_values)
this example has four columns left with missing values
these four columns have missing values that are >5% of the total values
decided to place the mode in to fill the missing values for the first three columns
we do this with a for loop
for col in cols_with_missing_values[:-1]: #indexed for everything but the last row
    salaries[col].fillna(salaries[col].mode()[0])   #passing the respective column's mode and indexing the first item which contains the mode
for the last column (in this example) we will impute median salary by experience level by grouping salaries by experience and calculating the median
salaries_dict = salaries.groupby('Experience')['Salary_USD'].median().to_dict()
print(salaries_dict)
this prints out the median salaries for each experience level
we now impute using the .fillna method and calling the .map method
salaries['Salary_USD'] =
salaries['Salary_USD].fillna(salaries['Experience'].map(salaries_dict)

Converting and analyzing categorical data
previewing the data
print(salaries.select_dtypes('object').head())
for frequency of values within a column
print(salaries['Designation'].value_counts())
to count how many unique titles there are
print(salaries['Designation'].nunique())

***Extracting value from categories
pandas.Series.str.contains()

allows us to search a column for a specific string or multiple strings
example
salaries['Designation'].str.contains('Scientist') #in this example looking for jobs with the word scientist in the title
** returns true or false values

Finding multiple phrases in strings
salaries['Designation'].str.contains('Machine Learning|AI')
** spaces matter, if spaces are added before or after the pipe than the command will only return strings that include a space as well
example
job_categories = ['Data Science', 'Data Analytics', 'Data Engineering', 'Machine Learning', 'Managerial', 'Consultant']
now we need to create variables containing our filters
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'
next step is to create a list with our range of conditions for the str.contains()

```
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))
```

then create a new column by using numpy.select()
salaries['Job_Category'] = np.select(conditions, job_categories, default='Other')
**default argument to 'Other' assigns 'Other' when a value in our conditions is not found
preview
print(salaries[['Designation', 'Job_Category']].head())
visualize frequency
sns.countplot(data=salaries, x='Job_Category')
plt.show()

Nice example to get started
# 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())

Working with numeric data
example - obtaining a new column 'Salary_USD' from column 'Salary_In_Rupees'
first convert strings to numbers
- remove comma values in 'Salary_In_Rupees'
- then convert the column to a float data type
- then create a new column by converting the currency
to remove commas:
pd.Series.str.replace('character to remove', 'characters to replace them with')
** here we don't want to pass characters back in so we use an empty string
salaries['Salary_In_Rupees'] = salaries['Salary_In_Rupees'].str.replace(",", "")
update to float:
salaries['Salary_In_Rupees'] = salaries['Salary_In_Rupees].astype(float)
now currency exchange (for this example 1 Indian Rupee = 0.012 USD)
salaries['Salary_USD'] = salaries['Salary_In_Rupees'] * 0.012
look at your manipulated data
print(salaries[['Salary_In_Rupees', 'Salary_USD']].head())

Adding summary statistics into a DataFrame
for our example
salaries.groupby('Company_Size')['Salary_USD'].mean()
this creates a summary table which is useful but sometimes we may want to add this info directly into our DataFrame
example - create a new column containing the standard deviation of Salary_USD where values are conditional based on the Experience column
group by 'Experience' > select 'Salary_USD' > call transform() > apply lambda function
salaries['std_dev'] = salaries.groupby('Experience')
['Salary_USD'].transform(lambda x: x.std())
this calculates the standard deviation of salaries based on experience
we can check the frequencies
print(salaries[['Experience', 'std_dev']].value_counts())
** can use the same process for other summary statistics such as mean and median

Handling outliers
a good first place to start is with the .describe()
look at max and min compared to median
view the IQR
IQR again is the difference between the 75th and 25th percentile

a good way to visualize is with the boxplot
sns.boxplot(data=, y='')

Using IQR to find outliers
upper outliers > 75th percentile + (1.5*IQR)
lower outliers < 25th percentile - (1.5*IQR)

Identifying thresholds
calculate percentiles using .quantile()
seventy_fifth = salaries['Salary_USD'].quantile(0.75)
twenty_fifth = salaries['Salary_USD'].quantile(0.25)
salaries_iqr = seventy_fifth - twenty_fifth
print(salaries_iqr)
upper = seventy_fifth + (1.5*salary_iqr)
lower = twenty_fifth -(1.5*salary_iqr)
print(upper, lower)

Finding nonsensical values or values outside of these limits
can do this by subsetting
salaries[(salaries['Salary_USD'] < lower) | (salaries['Salary_USD'] > upper)] \
    [['Experience', 'Employee_Location', 'Salary_USD']]

Why outliers are important
-extreme values that may not accurately represent the data
-they skew mean and standard deviation
-can affect statistical tests and machine learning models that need normally distributed data

What to do with them?
do they represent a subset and therefore should be left in the data?
was there an error in data collection and therefore should we remove the outlier?

Dropping outliers
no_outliers = salaries[(salaries['Salary_USD'] > lower) & (salaries['Salary_USD'] < upper)]
print(no_outliers['Salary_USD'].describe())

Patterns over time
DateTime data needs to be explicitly declared to Pandas
we can do this with the 'parse' argument when we are reading csv files in example
divorce = pd.read_csv('divorce.csv', parse_dates=['marriage_date'])
this turns 'marriage_date' into a date time object

check with
divorce.dtypes
we can also do this after we have imported the data with the pd.to_datetime()
divorce['marriage_date'] = pd.to_datetime(divorce['marriage_date'])
another neat trick with pd.to_datetime
can say take three separate columns for month, day, and year we can combine them into a single DateTime value
divorce['marriage_date'] = pd.to_datetime(divorce[['month', 'day', 'year']])
**key note, these three columns can be passed in any order BUT they have to be labeled exactly 'month', 'day', and 'year'

Creating DateTime data
we can extract parts of a full date from a date time object
divorc['marriage_month'] = divorce['marriage_date'].dt.month

Visualizing patterns over time
line plots are a great way to examine relationships between variables
sns.lineplot(data=divorce, x='marriage_month', y='marriage_duration')
plt.show()
in seaborn line plots aggregate y values at each value of x and show the estimated mean and a confidence interval for that estimate
example - check relationship between the month that a now-divorced couple got married and the length of their marriage

Correlation
describes direction and strength of relationship between two variables
can help us use variables to predict future outcomes
divorce.corr() #quick way to see the pairwise correlation of numeric columns in a DataFrame
**negative correlation coefficient indicates that as one variable increases, the other decreases
**value closer to 0 is indicative of a weaker relationship, closer to 1 or -1 indicative of a stronger relationship
.corr() is the Pearson correlation coefficient and measures the linear relationship between two variables

Visualizing
heatmaps is a nice way to see correlation
sns.heatmap(divorce.corr(), annot=True)
remember annot argument places the value within the heatmap squares
**always remember the context of your data - for example, in this example correlation is likely going to be different depending on earlier or later divorce date
**also just because there isn't a strong linear relationship doesn't mean that there

isn’t a strong nonlinear relationship
scatter plots can help us navigate this
sns.scatterplot(data=divorce, x=‘income_man’, y=‘income_woman’)
can compare this to our heatmap
in this example the Pearson correlation and the scatter plot match up

Pairplots
this is the next level
plots all pairwise relationships between numerical variables in one visualization
sns.pairplot(data=divorce)
plt.show()
can cut it down as needed
sns.pairplot(data=divorce, vars=[‘income_man’, ‘income_woman’, ‘marriage_duration’])
plt.show()

Factor relationships and distributions
explore categorical variables
divorce[‘education_man’].value_counts()
categorical variables are harder to summarize numerically so visualizations help
sns.histplot(data=divorce, x=‘marriage_duration’, hue=‘education_man’, binwidth=1)

a KDE plot can make this easier to visualize
sns.kdeplot(data=divorce, x=‘marrige_duration’, hue=‘education_man’)
plt.show()
kde’s are more interpretable with multiple distributions are shown
**with KDE plots you have to make sure that good smoothing parameters are set
we can use argument ‘cut’
tells seaborn how far past the minimum and maximum data values the curve should go when smoothing is applied
setting cut to 0, the curve will be limited to values between the minimum and maximum x values
in this example that will be 0 years and the max marriage duration
KDE plots also allow us to apply the cumulative distribution function
done by adding argument ‘cumulative’ and setting it to True
**for this example this describes the probability that marriage duration is less than or equal to the value on the x-axis for each level of male partner education

Example
Is there a relationship between age at marriage and education level?
divorce[‘man_age_marriage’] = divorce[‘marriage_year’] - divorce[‘dob_man’].dt.year

divorce[‘woman_age_marriage’] = divorce[‘marriage_year’] - divorce[‘dob_woman’].dt.year
then create a scatterplot with these new variables
sns.scatterplot(data=divorce, x=‘woman_age_marriage’, y=‘man_age_marriage’)
plt.show()
can layer on hue for additional analysis

Considerations for categorical data
representation of classes or sometimes called labels
classes = labels
this distinction can help us discover imbalance
example
we want to know people’s attitudes towards marriage
we take a survey of a 1000 people but after defining the classes we realize that 750 are divorced, 200 are single, and only 50 are married
our data with this sample is likely to be skewed
this is important cause this can bias results

Relative class frequency
planes[‘Destination’].value_counts(normalize=True)
‘normalize’ argument set to True will return relative frequencies for each class
this means that instead of giving us raw counts it will return proportions
say our output returns that internal flights to Delhi is 11% but we show from previous studies that it is suppose to be 40%?
we may have a skewed sample and not representative of the population

Cross-tabulation is another method for looking at class frequency
enables us to examine the frequency of combinations of classes
call pd.crosstab()> select column for index> select column (values in this column will become the names of the columns in the table and the values will be the count of combined observations)
pd.crosstab(planes[‘Source’], planes[‘Destination’])
another example
extending cross-tabulation
say we know the median prices, we can now cross-tab our sample and compare our sample’s median price
pd.crosstab(planes[‘Source’], planes[‘Destination’], values=[planes[‘Price’], aggfunc=‘median’)

Generating new features
one technique is grouping numeric data and labeling them as classes
example - create a column for ticket type
labels = [‘Economy’, ‘Premium Economy’, ‘Business Class’, ‘First Class’]

bins = [0, twenty_fifth, median, seventy_fifth, maximum]
then use pd.cut:
call pd.cut()> pass the data> set the labels> provide the bins
planes['Price Category'] = pd.cut(planes['Price'], labels=labels, bins=bins)
ensure mapping as been done proper
print(planes[['Price', 'Price_Category']].head())
then visualize
sns.countplot(data=planes, x='Airline', hue='Price_Category')
plt.show()

Spurious correlation
our example
appeared Total_Stops was correlated to Price but on further examination
Total_Stops correlated more with Duration
this is an example of spurious correlation

What is true?
detecting relationships, differences, and patterns
to do this we use hypothesis testing
hypothesis testing requires, prior to data collection:
 - generating a hypothesis or question
 - a decision on what statistical test to use (which test can we perform in order to reasonably conclude whether the hypothesis was true or false)

Data snooping or p-hacking
acts of excessive exploratory analysis, generation of multiple hypotheses, execution of multiple statistical tests
we want to avoid this

Generating hypotheses
ask a question
bar plotting it is a good option
then design experiment > choose a sample > calculate how many data points we need > decide what statistical test to run