EDA HELPER FUNCTION

When performing exploratory data analysis (EDA), we often rely on built-in functions. While these functions can be helpful, they sometimes become time-consuming and make the detailed analysis of each column tedious. To save time and avoid the repetitive tasks commonly associated with EDA, we can create a helper function that automates these processes. Built-in functions may also lead to overlapping results and unnecessary information, making it difficult to focus on more advanced analysis. By developing a helper function, we can ensure a thorough analysis from basic to advanced levels without restricting ourselves to only fundamental insights. Here, we have outlined a sequence for analyzing data, ranging from basic to advanced techniques.

I have divided my functions into three subcategories to address the challenges of conducting detailed analysis and, of course, to save time:

1. **Basic functions**
2. **Intermediate functions**
3. **Advanced functions**

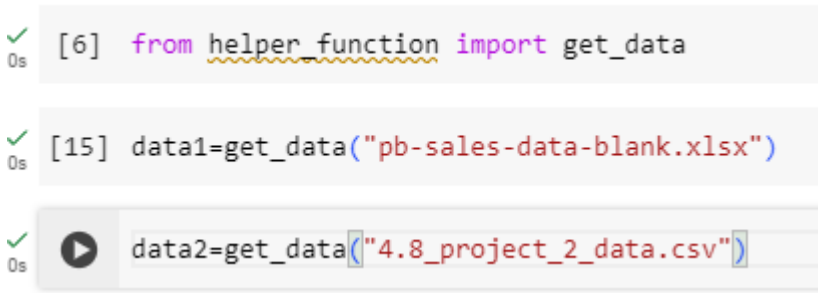
Note: The effort need from your side…….

1. Basic functions:

The major focus of these functions is for task like to avoid axis or title overlapping, loading of datasets , basic guidance for EDA anlysis, igoring the warnings, printing the html contents.

a) Data loading :

```
b) def get_data(name):
c)     if 'csv' in name.lower():
d)         return pd.read_csv(name)
e)     elif 'excel' in name.lower() or 'xls' in name.lower():
f)         return pd.read_excel(name)
g)     else:
h)         raise ValueError("Unsupported file type. Please make sure the
   file name contains 'csv', 'excel', or 'json'.")
```



b) Hide warnings:

```
import helper_function
```

```
suppress_warnings()
```

c) Resolving lable-hidrence:

```
def rotate_xlabels(ax, angle=35):
  ax.set_xticklabels(
        ax.get_xticklabels(),
        rotation=angle,
        ha="right"
    )


def rotate_ylabels(ax, angle=0):
  ax.set_yticklabels(
    ax.get_yticklabels(),
    rotation=angle
  )
```

2. Intermediate functions:

a) pairplot for multivariate plot:

```
[42] import helper_function
```

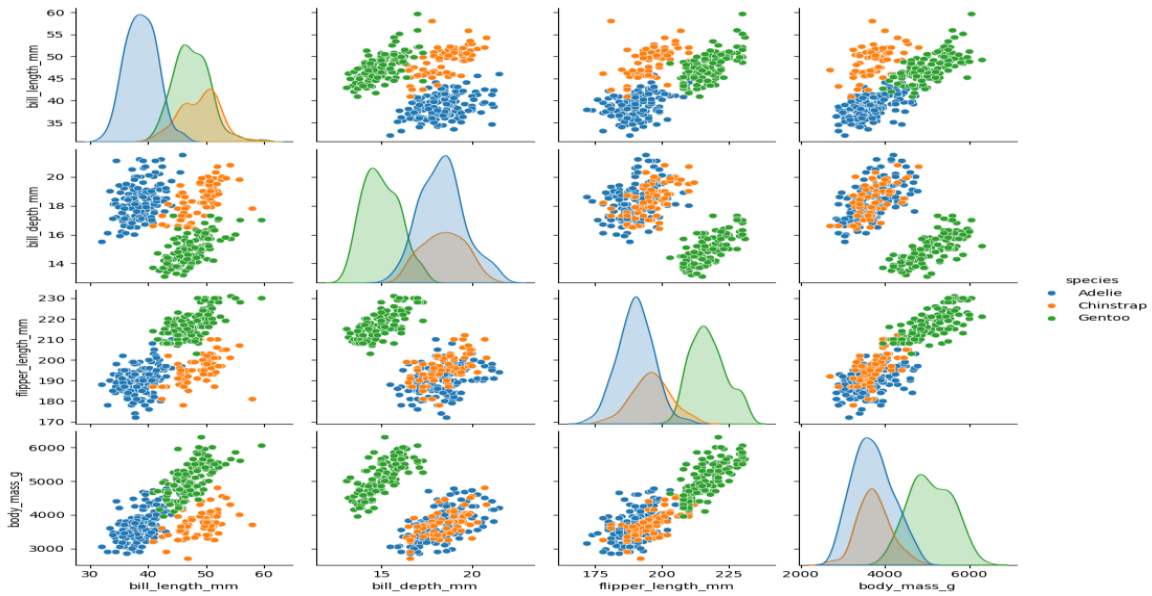
```
[43] import pandas as pd
```

```
[44] penguins = sns.load_dataset("penguins")
     penguins.head()
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female

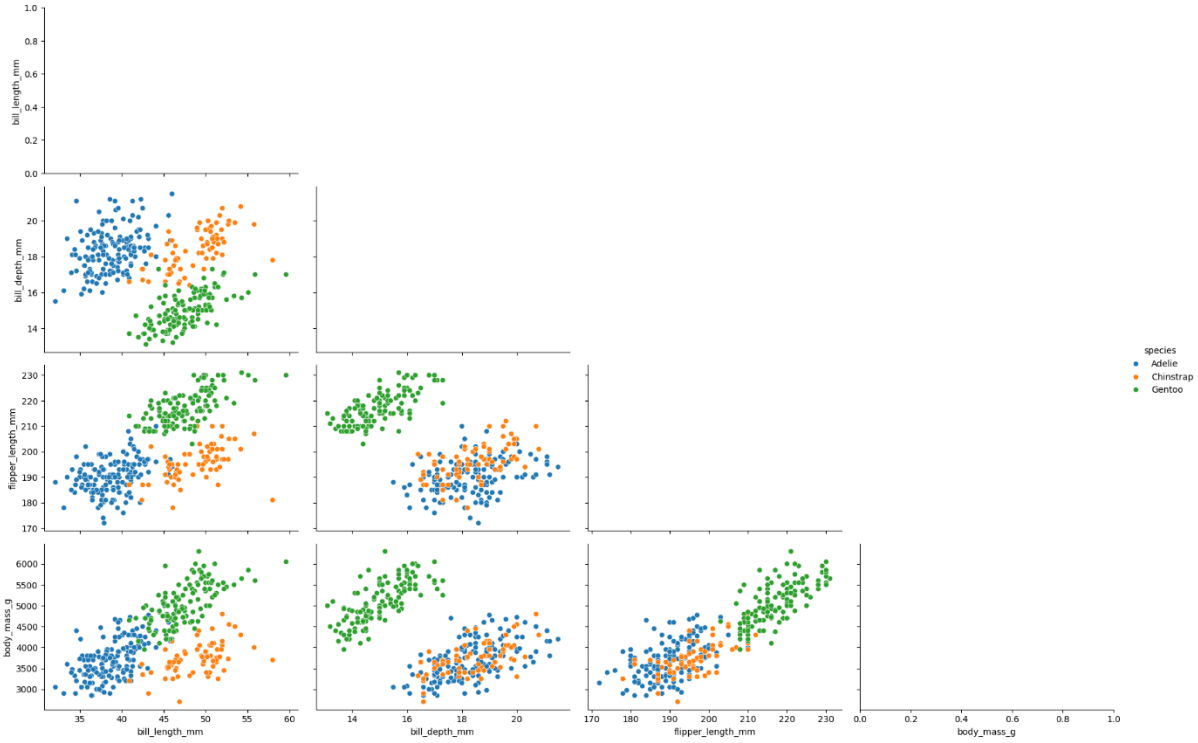
By default with all information

```
sns.pairplot(penguins,hue="species")
```



VS. Modified with only relevant information:

```
helper_function.pair_plots(penguins,hue="species",legend=True)
```



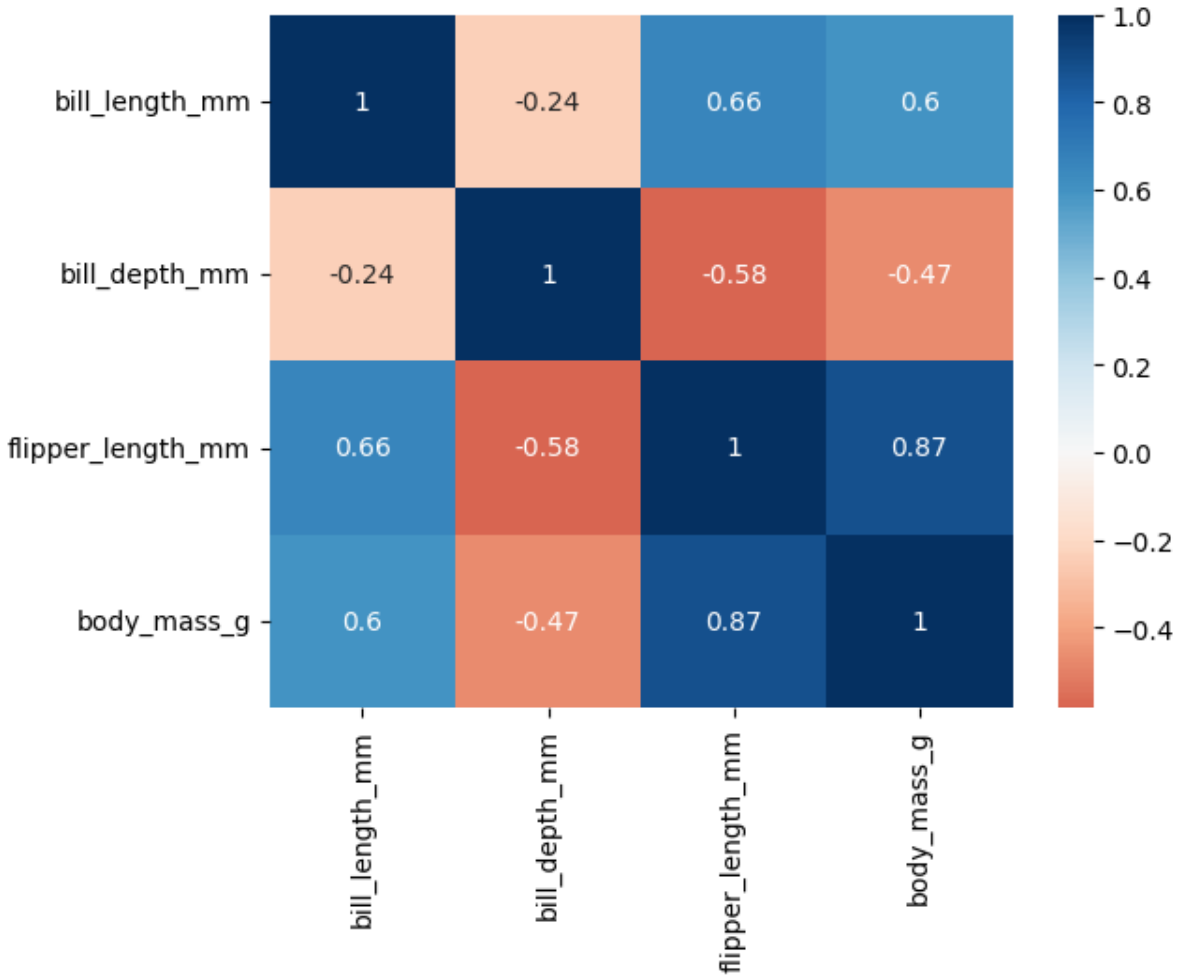
b) plotting-correlation-heat-map for multivariate analysis:

```
def correlation_heatmap(data,
                        figsize=(12, 6),#specifying the size of the figure (default is 12 inches by 6 inches).
                        method="spearman", #The method used to compute the correlation. Options are "pearson", "kendall", and "spearman" (default is "spearman").
```

```
                        cmap="RdBu"): #The colormap used for the heatmap
(default is "RdBu")
  cm = data.corr(method=method, numeric_only=True) #The correlation matrix
computed using the specified method. The numeric_only=True argument ensures
that only numeric columns are included in the correlation calculation.

  mask = np.zeros_like(cm, dtype=bool) #A boolean array of the same shape as
the correlation matrix, initialized to False.
  mask[np.triu_indices_from(mask)] = True #Sets the upper triangle (including
the diagonal) of the mask to True. This mask is used to hide the upper
triangle of the heatmap to avoid redundancy.

  fig, ax = plt.subplots(figsize=figsize) #Creates a Matplotlib figure and
axes with the specified figure size.
  hm = sns.heatmap(
      cm,#The correlation matrix to be plotted.
      vmin=-1,
      vmax=1, #vmin=-1, vmax=1: Sets the limits for the colormap.
      cmap=cmap, #Specifies the colormap.
      center=0, #Centers the colormap at 0.
      annot=True,#Annotates each cell with the correlation coefficient.
      fmt=".2f",#Formats the annotations to two decimal places.
      linewidths=1.5, #Sets the width of the lines between cells.
      square=True, #Makes each cell square-shaped.
      mask=mask, #Applies the mask to hide the upper triangle of the heatmap.
      ax=ax # Plots the heatmap on the specified axes.
  )
  rotate_xlabels(ax)
  rotate_ylabels(ax)
  ax.set(title=f"{method.title()} Correlation Matrix Heatmap")

  #----------------------------------------detailed-summary-of-numerical-
features-------------------------------------

def num_summary(data, var): #var: The column name (string) of the numerical
variable to be summarized.
  import warnings
  warnings.filterwarnings("ignore")  #suppress warnings to keep the output
clean.

                              # ------------------------title----------------
--------
  col = data.loc[:, var].copy() #copy of the specified column from the
DataFrame.
  display_html(size=2, content=var)
```

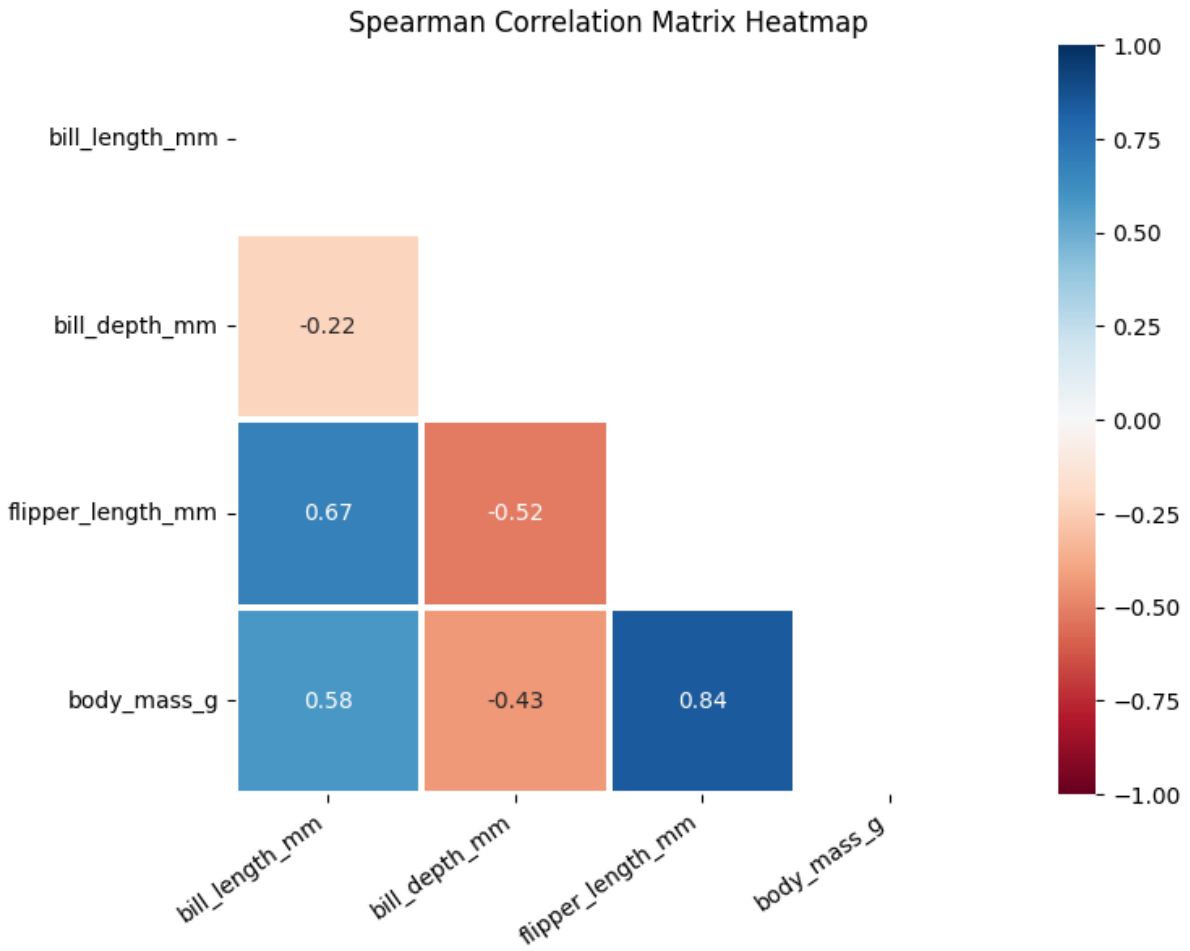
By default with all information

```
numeric_data = penguins.select_dtypes(include='number')

# Create a heatmap using the correlation matrix
sns.heatmap(numeric_data.corr(), annot=True, cmap='RdBu', center=0)
```



VS. Modified with only relevant information:

```
helper_function.correlation_heatmap(penguins)
```



c) univariate plots for numeric variables:

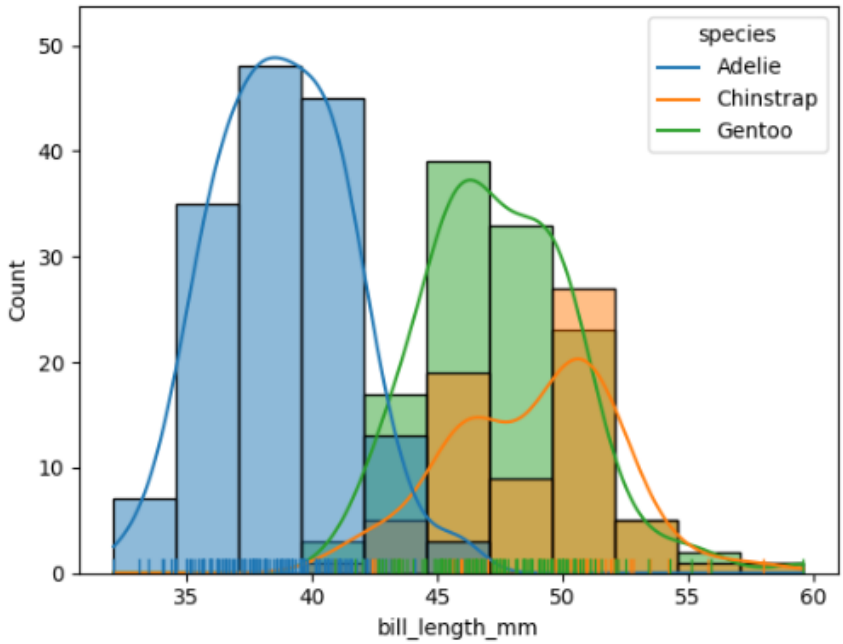
```
def num_univar_plots(data, var, bins=10, figsize=(15, 7)): #bins: Number of
bins for the histogram (default is 10).
  display_html(2, f"Univariate Analysis of {var}") #Displays an HTML header
with the variable name.
  display_html(content="") # Adds a blank line for better readability.
  col = data.loc[:, var].copy() #Creates a copy of the specified column from
the dataset.

  fig, axes = plt.subplots(2, 3, figsize=figsize) #Creates a figure with 2
rows and 3 columns of subplots.
  axes = axes.ravel() #Flattens the array of axes for easier indexing.

  #histogram
  sns.histplot(
      data,
      x=var, #Specifies the variable to be plotted on the x-axis.
      bins=bins, #Specifies the number of bins for the histogram. This
determines the granularity of the histogram.
      kde=True, # If True, adds a Kernel Density Estimate (KDE) curve over the
histogram. KDE is a way to estimate the probability density function of a
random variable.
```

```
      color="#1973bd", #blue color
      ax=axes[0], #axes[0] refers to the first subplot in a grid of subplots.
  )
  sns.rugplot(
      data,
      x=var,
      color="black",
      height=0.035, #Specifies the height of the rug plot ticks. A smaller
value makes shorter ticks.
      ax=axes[0] #axes[0] refers to the first subplot in a grid of subplots.
  )
  axes[0].set(title="Histogram") #axes[0].set(title="Histogram"): Sets the
title of the first subplot to "Histogram".

  # cdf
  sns.ecdfplot(  #empirical cumulative distribution function (ECDF) plot.ECDF
is a step function that shows the proportion or count of observations falling
below each unique value in a dataset.
      data,
      x=var,
      ax=axes[1],
      color="red"
  )
  axes[1].set(title="CDF")

  # power transform
  data = data.assign(**{ #assign method of a pandas DataFrame is used to
create a new column or update an existing column in the DataFrame.
      f"{var}_pwt": (   #new column is being created with a name based on the
variable name (var) with a suffix _pwt indicating it's been power transformed.
          PowerTransformer() #This creates an instance of the PowerTransformer
class from sklearn.preprocessing, which applies a power transformation (such
as the Box-Cox or Yeo-Johnson transform) to make data more normally
distributed.
          .fit_transform(data.loc[:, [var]]) # fits the power transformer to
the data (calculating the parameters for the transformation) and then applies
the transformation to the data.
          .ravel() #This flattens the transformed data array to a 1D array.
      )
  })
  sns.kdeplot( #plot represents the distribution of a continuous variable.
      data,
      x=f"{var}_pwt",
      fill=True, #This fills the area under the KDE curve.
      color="#f2b02c",
      ax=axes[2] #third subplot in a grid of subplots.
  )
  sns.rugplot(
```

```
      data,
      x=f"{var}_pwt",
      color="black",
      height=0.035, #This sets the height of the rug lines.
      ax=axes[2]
  )
  axes[2].set(title="Power Transformed")

  # box plot
  sns.boxplot(
      data,
      x=var,
      color="#4cd138",
      ax=axes[3]
  )
  axes[3].set(title="Box Plot")

  # violin plot
  sns.violinplot(
      data,
      x=var,
      color="#ed68b4",
      ax=axes[4]
  )
  axes[4].set(title="Violin Plot")

  # qq plot
  sm.qqplot(
      col.dropna(),#This is important because missing values can interfere
with the calculation of quantiles and fitting of the line.
      line="45", #In a QQ plot, if the data points lie along the 45-degree
line, it suggests that the sample data follows the theoretical distribution.
      fit=True, #This specifies that the data should be standardized before
plotting
      ax=axes[5]
  )
  axes[5].set(title="QQ Plot")

  plt.tight_layout()
  plt.show()
```

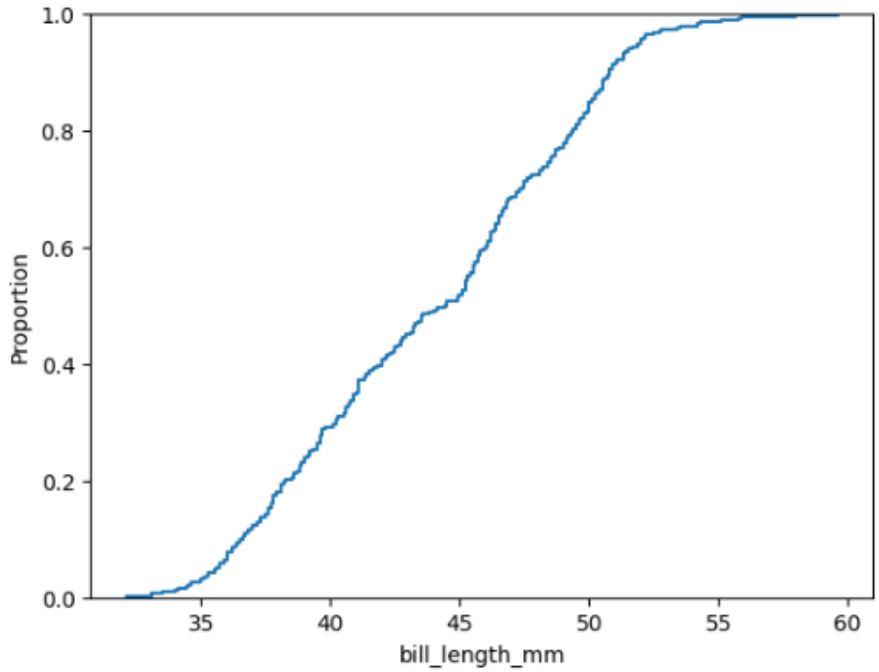
```
sns.histplot(data=penguins, x="bill_length_mm", hue="species",kde=True)
sns.rugplot(data=penguins, x="bill_length_mm", hue="species")
```

```
<Axes: xlabel='bill_length_mm', ylabel='Count'>
```



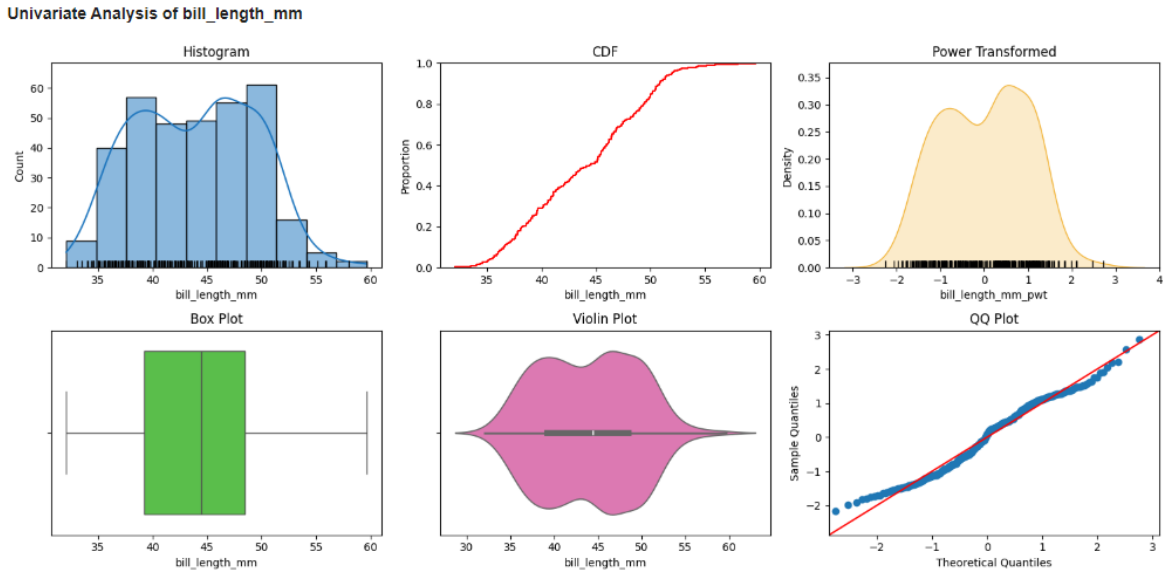

```
sns.ecdfplot(data=penguins, x="bill_length_mm")
```

```
<Axes: xlabel='bill_length_mm', ylabel='Proportion'>
```



Similarly for other plots we have to write individual code, below one simple line of code gives you all univariate plots.

```
helper_function.num_univar_plots(penguins, "bill_length_mm")
```



d) cramers-v corrrelation heatmap:

It is for multivariate analysis to measure the strength of the association between two categorical variables.

```
def cramers_v(data, var1, var2): #to calculates Cramér's V statistic.
  ct = pd.crosstab(
      data.loc[:, var1],
      data.loc[:, var2]  #extract the columns corresponding to the variables
var1 and var2.
  )
  r, c = ct.shape #r is the number of rows, and c is the number of columns.
  n = ct.sum().sum() #calculates the total number of observations by summing
all values in the contingency table.
  chi2 = stats.chi2_contingency(ct).statistic #computes the Chi-Square
statistic for the contingency table, which measures the association between
the two categorical variables.
  phi2 = chi2 / n #normalized Chi-Square statistic, calculated by dividing the
Chi-Square statistic by the total number of observations.

  # bias correction
  phi2_ = max(0, phi2 - ((r - 1) * (c - 1) / (n - 1)))   #applies a bias
correction to account for small sample sizes or sparse contingency tables.
  r_ = r - (((r - 1) ** 2) / (n - 1))
  c_ = c - (((c - 1) ** 2) / (n - 1)) #r_ and c_ are the corrected dimensions
of the table, adjusting for sample size and bias.

  return np.sqrt(phi2_ / min(r_ - 1, c_ - 1)) #final value of Cramér's V
                  #---------------------------need to be call---------------
---------------------------
```

```python
def cramersV_heatmap(data, figsize=(12, 6), cmap="Blues"): #heatmap to visualize Cramér's V statistic
  cols = data.select_dtypes(include="O").columns.to_list() #selects all columns with object (categorical) data types and converts the selected column names to a list.

  matrix = (
      pd
      .DataFrame(data=np.ones((len(cols), len(cols)))) #Creates a DataFrame matrix initialized with ones, representing the initial Cramér's V values.
      .set_axis(cols, axis=0)
      .set_axis(cols, axis=1) #label the rows and columns with the categorical variable names.
  )

  for col1 in cols:  #Iterates over pairs of categorical columns (col1, col2).
    for col2 in cols:
      if col1 != col2:
        matrix.loc[col1, col2] = cramers_v(data, col1, col2) #For each pair, if they are different (col1 != col2), it computes Cramér's V

  mask = np.zeros_like(matrix, dtype=bool) #creates a boolean matrix with the same shape as matrix, initialized to False.
  mask[np.triu_indices_from(mask)] = True #provides indices for the upper triangle of the matrix. and sets the upper triangle of the mask to True, which will be used to hide these values in the heatmap.

  fig, ax = plt.subplots(figsize=figsize)
  hm = sns.heatmap(
      matrix,
      vmin=0,
      vmax=1,
      cmap=cmap,
      annot=True,
      fmt=".2f",
      square=True,
      linewidths=1.5, #sets the width of the cell borders.
      mask=mask,
      ax=ax
  )
  ax.set(title="Cramer's V Correlation Matrix Heatmap")
  rotate_xlabels(ax)
  rotate_ylabels(ax)
```

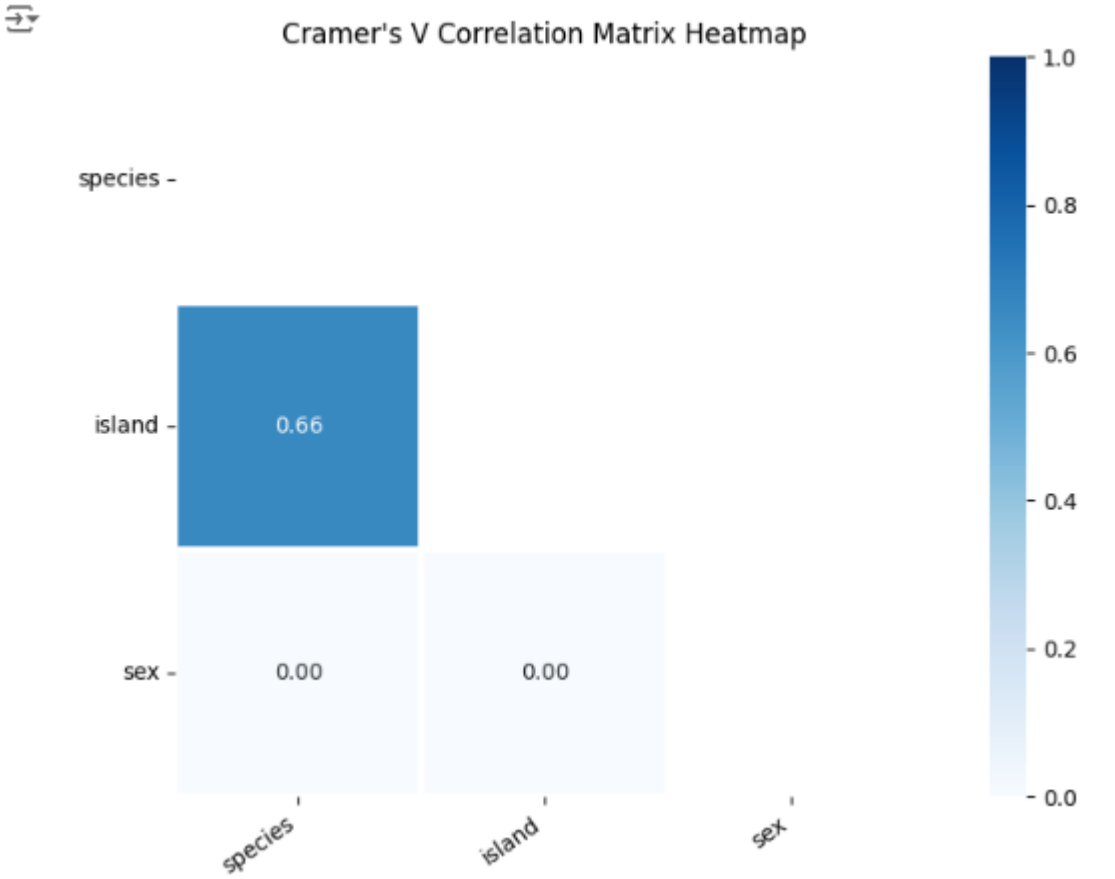
```python
helper_function.cramers_v(penguins, "species","island")
```

```
0.6573290190537864
```

cramersV heatmap:

```
def cramersV_heatmap(data, figsize=(12, 6), cmap="Blues"): #heatmap to
visualize Cramér's V statistic
  cols = data.select_dtypes(include="O").columns.to_list() #selects all
columns with object (categorical) data types and converts the selected column
names to a list.

  matrix = (
      pd
      .DataFrame(data=np.ones((len(cols), len(cols)))) #Creates a DataFrame
matrix initialized with ones, representing the initial Cramér's V values.
      .set_axis(cols, axis=0)
      .set_axis(cols, axis=1) #label the rows and columns with the categorical
variable names.
  )

  for col1 in cols:  #Iterates over pairs of categorical columns (col1, col2).
    for col2 in cols:
      if col1 != col2:
        matrix.loc[col1, col2] = cramers_v(data, col1, col2) #For each pair,
if they are different (col1 != col2), it computes Cramér's V

  mask = np.zeros_like(matrix, dtype=bool) #creates a boolean matrix with the
same shape as matrix, initialized to False.
  mask[np.triu_indices_from(mask)] = True #provides indices for the upper
triangle of the matrix. and sets the upper triangle of the mask to True, which
will be used to hide these values in the heatmap.

  fig, ax = plt.subplots(figsize=figsize)
  hm = sns.heatmap(
      matrix,
      vmin=0,
      vmax=1,
      cmap=cmap,
      annot=True,
      fmt=".2f",
      square=True,
      linewidths=1.5, #sets the width of the cell borders.
      mask=mask,
      ax=ax
  )
  ax.set(title="Cramer's V Correlation Matrix Heatmap")
  rotate_xlabels(ax)
  rotate_ylabels(ax)
```

```
helper_function.cramersV_heatmap(penguins)
```



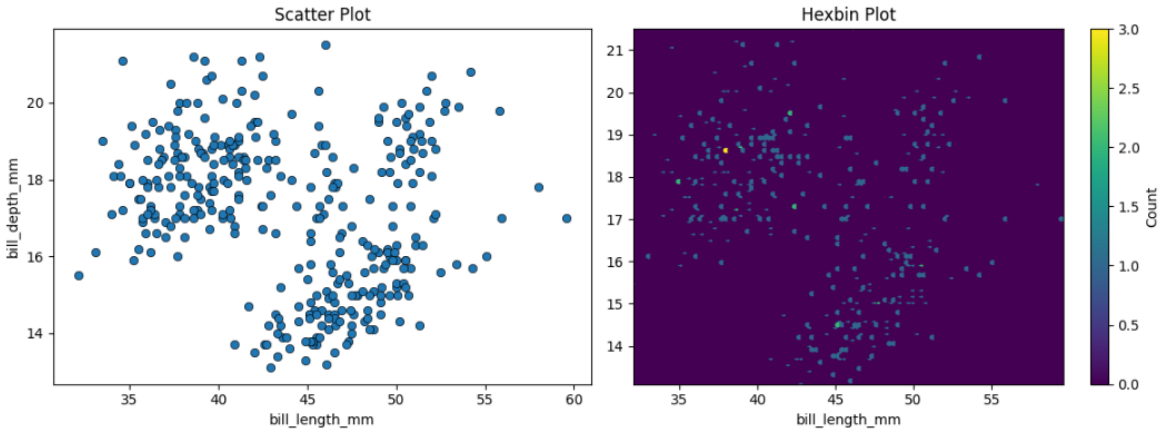
e) bivariate plots between 2 numeric variables:

```
def num_bivar_plots(data, var_x, var_y, figsize=(12, 4.5),
scatter_kwargs=dict(), hexbin_kwargs=dict()): #scatter_kwargs: Additional
keyword arguments for customizing the scatter plot. and hexbin_kwargs:
Additional keyword arguments for customizing the hexbin plot.
  display_html(2, f"Bi-variate Analysis between {var_x} and {var_y}")
  display_html(content="")

  fig, axes = plt.subplots(1, 2, figsize=figsize) #creates a figure with two
subplots arranged in one row and two columns.

  # scatter plot
  sns.scatterplot(
      data,
      x=var_x, #Specifies the variable for the x-axis.
      y=var_y, #Specifies the variable for the y-axis.
      ax=axes[0],
      edgecolors="black", #Adds black borders around the scatter plot points.
      **scatter_kwargs  #Additional keyword arguments for further
customization of the scatter plot.
  )
  axes[0].set(title="Scatter Plot")
```

```
  # hexbin plot
  col_x = data.loc[:, var_x]
  col_y = data.loc[:, var_y]
  hexbin = axes[1].hexbin(
      x=col_x,
      y=col_y,
      **hexbin_kwargs #Additional keyword arguments for further customization
of the hexbin plot.
  )
  axes[1].set(
      title="Hexbin Plot",
      xlabel=var_x,
      xlim=(col_x.min(), col_x.max()), #Sets the limits for the x-axis from
the minimum to the maximum value of col_x.
      ylim=(col_y.min(), col_y.max()) #Sets the limits for the y-axis from the
minimum to the maximum value of col_y.
  )
  cb = plt.colorbar(  #adds a colorbar to the plot
      hexbin, #The hexbin plot object, which is used to create the colorbar.
      label="Count" #Label for the colorbar indicating what the colors
represent (in this case, the count of points in each hexagon).
  )

  plt.tight_layout()
  plt.show()
```

```
helper_function.num_bivar_plots(penguins, "bill_length_mm", "bill_depth_mm")
```

Bi-variate Analysis between bill_length_mm and bill_depth_mm



f) univariate plots for categorical variables:

```
def get_top_k(data, var, k): #Defines a function named get_top_k that takes
three parameters: data (a DataFrame), var (the column name), and k (an integer
indicating the number of top categories to retain).
```

```
  col = data.loc[:, var].copy() #Creates a copy of the specified column (var) from the DataFrame (data).
  cardinality = col.nunique(dropna=True)# Calculates the number of unique values in the column, excluding missing values.
  if k >= cardinality:
    raise ValueError(f"Cardinality of {var} is {cardinality}. K must be less than {cardinality}.")
  else:
    top_categories = (
        col
        .value_counts(dropna=True)
        .index[:k] #Identifies the top k most frequent categories in the column.
    )
    data = data.assign(**{ #Uses a dictionary to specify the new column name and values.
        var: np.where(   #np.where(condition, x, y): Returns x where condition is True and y where condition is False.
            col.isin(top_categories),
            col,
            "Other" #Replaces values not in the top k categories with "Other".
        )
    })
    return data


def pie_chart(counts, colors, ax):
  pie = ax.pie(
        counts.values,
        labels=counts.index,
        autopct="%.2f%%",
        colors=colors,
        wedgeprops=dict(alpha=0.7, edgecolor="black"),  #Sets the transparency of the wedges. and Sets the edge color of the wedges to black.
  )

  ax.set_title("Pie Chart")

  ax.legend( #Adds a legend to the pie chart.
      loc="upper left",
      bbox_to_anchor=(1.02, 1), #Adjusts the position of the legend relative to the axes.
      title="Categories",
      title_fontproperties=dict(weight="bold", size=10)
  )

  plt.setp( #Sets properties on the percentage text labels of the pie chart.
      pie[2],
```

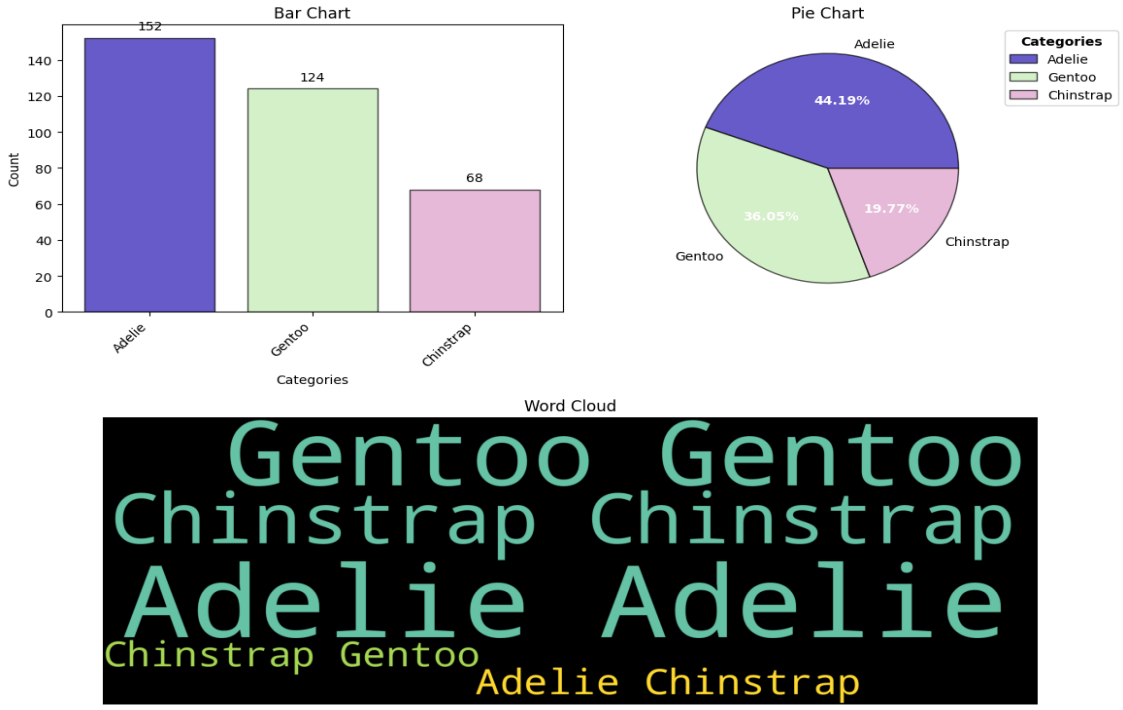
```
      weight="bold",
      color="white"
  )


def bar_chart(counts, colors, ax):
  barplot = ax.bar(
        x=range(len(counts)),
        height=counts.values,
        tick_label=counts.index,
        color=colors,
        edgecolor="black",
        alpha=0.7
  )

  ax.bar_label(
      barplot,
      padding=5,
      color="black"
  )

  ax.set(
      title="Bar Chart",
      xlabel="Categories",
      ylabel="Count"
  )

  ax.set_xticklabels(
      ax.get_xticklabels(),
      rotation=45,
      ha="right"
  )
```

```
#----------------------------------------callable-------------------------------
-------------
def cat_univar_plots(data, #DataFrame containing the data.
                     var, #The categorical variable to analyze.
                     k=None, #Number of top categories to consider (optional).
                     order=None, #Order of categories for the bar chart
(optional).
                     show_wordcloud=True, #Boolean indicating whether to
display a word cloud (default is True).
                     figsize=(12, 8.5)):
  display_html(2, f"Univariate Analysis of {var}")
  display_html(content="")

  fig = plt.figure(figsize=figsize) #Creates a new figure with the specified
size.
```

```
  gs = GridSpec(2, 2, figure=fig) #Defines a grid specification with 2 rows
and 2 columns for subplots.
  ax1 = fig.add_subplot(gs[0, 0]) # bar-chart
  ax2 = fig.add_subplot(gs[0, 1]) # pie-chart
  ax3 = fig.add_subplot(gs[1, :]) # word-cloud

  if k is None: #Checks if k is not provided.
    counts = (
        data
        .loc[:, var]
        .value_counts()
        .reindex(index=order) #Calculates the counts of each category in var
and reorders them according to order if provided.
    )
  else: #If k is provided.
    temp = get_top_k( #Calls the get_top_k function to get the top k
categories.
        data,
        var,
        k=k
    )
    counts = (
        temp
        .loc[:, var]
        .value_counts()
    )

  colors = [tuple(np.random.choice(256, size=3) / 255) for _ in
range(len(counts))] #Generates random colors for each category in counts.

  # bar-chart
  bar_chart(
      counts,
      colors,
      ax1
  )

  # pie_chart
  pie_chart(
      counts,
      colors,
      ax2
  )

  # word-cloud
  if show_wordcloud: #Checks if show_wordcloud is True.
    var_string = " ".join( #Creates a string by joining all values in the var
column with spaces, replacing spaces within each value with underscores.
```

```
        data
        .loc[:, var]
        .dropna()
        .str.replace(" ", "_")
        .to_list()
    )

    word_cloud = WordCloud( #Creates a word cloud from the var_string.
        width=2000,
        height=700,
        random_state=42,
        background_color="black",
        colormap="Set2",
        stopwords=STOPWORDS
    ).generate(var_string)

    ax3.imshow(word_cloud)
    ax3.axis("off")
    ax3.set_title("Word Cloud")
  else:
    ax3.remove()

  plt.tight_layout() #Adjusts the layout of the plot to make sure everything
fits without overlapping.
  plt.show()
```

```
helper_function.cat_univar_plots(penguins, "species")
```



g) bivariate plots between numeric and categorical variable:

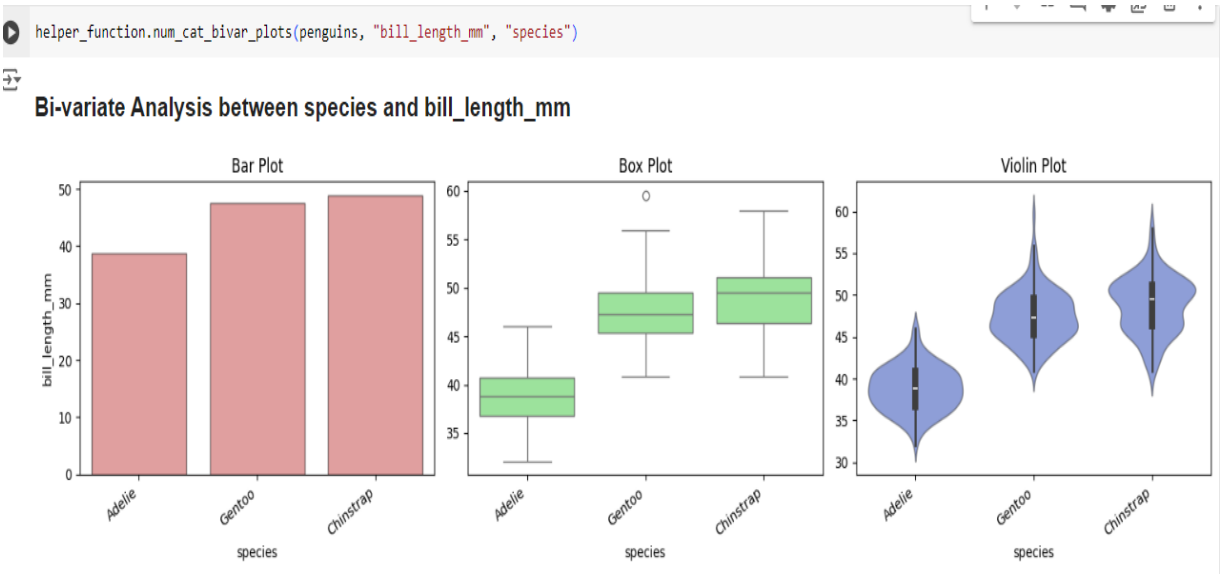
```
def num_cat_bivar_plots(data,
                        num_var,
                        cat_var,
                        k=None,
                        estimator="mean",
                        orient="v", #Orientation of plots, either vertical
("v") or horizontal ("h") by default verticle.
                        order=None,
                        figsize=(15, 4)):

  def get_values(data,
                 num_var,
                 cat_var,
                 estimator,
                 order=None):
    return (
        data
        .groupby(cat_var)
        .agg(estimator, numeric_only=True)
        .loc[:, num_var]
        .dropna()
        .sort_values()
        .reindex(index=order)
    )

  import warnings
  warnings.filterwarnings("ignore")

  display_html(2, f"Bi-variate Analysis between {cat_var} and {num_var}")
  display_html(content="")

  if k is None:
    temp = get_values(
        data,
        num_var,
        cat_var,
        estimator,
        order=order
    )
  else:
    data = get_top_k(
        data,
        cat_var,
        k=k
    )
    temp = get_values(
        data,
        num_var,
```

```
        cat_var,
        estimator
    )

if orient == "v":
    fig, axes = plt.subplots(1, 3, figsize=figsize)

    # bar plot
    bar = sns.barplot(
      x=temp.index,
      y=temp.values,
      color="#d92b2b",
      ax=axes[0],
      edgecolor="black",
      alpha=0.5
    )
    axes[0].set(
        title="Bar Plot",
        xlabel=cat_var,
        ylabel=num_var
    )
    rotate_xlabels(axes[0])

    # box plot
    sns.boxplot(
      data,
      x=cat_var,
      y=num_var,
      color="lightgreen",
      order=temp.index,
      ax=axes[1]
    )
    axes[1].set(
        title="Box Plot",
        xlabel=cat_var,
        ylabel=""
    )
    rotate_xlabels(axes[1])

    # violin plot
    sns.violinplot(
      data,
      x=cat_var,
      y=num_var,
      color="#0630c9",
      order=temp.index,
      ax=axes[2],
      alpha=0.5
```

```
    )
    axes[2].set(
        title="Violin Plot",
        xlabel=cat_var,
        ylabel=""
    )
    rotate_xlabels(axes[2])
else:
    fig, axes = plt.subplots(3, 1, figsize=figsize)

    # bar plot
    bar = sns.barplot(
      y=temp.index,
      x=temp.values,
      color="#d92b2b",
      ax=axes[0],
      edgecolor="black",
      alpha=0.5
    )
    axes[0].set(
        title="Bar Plot",
        xlabel="",
        ylabel=cat_var
    )

    # box plot
    sns.boxplot(
      data,
      y=cat_var,
      x=num_var,
      color="lightgreen",
      order=temp.index,
      ax=axes[1]
    )
    axes[1].set(
        title="Box Plot",
        xlabel="",
        ylabel=cat_var
    )

    # violin plot
    sns.violinplot(
      data,
      y=cat_var,
      x=num_var,
      color="#0630c9",
      order=temp.index,
      ax=axes[2],
```

```
      alpha=0.5
    )
    axes[2].set(
        title="Violin Plot",
        xlabel=num_var,
        ylabel=cat_var
    )

  plt.tight_layout()
  plt.show()


# categorical bivariate plots
def cat_heat_map(data, mask=True, **kwargs):
  if mask:
    mask = np.zeros_like(data, dtype=bool)
    mask[np.triu_indices_from(mask)] = True
  else:
    mask = None

  return sns.heatmap(
      data=data,
      mask=mask,
      annot=True,
      linewidths=1.5,
      linecolor="white",
      square=True,
      **kwargs
  )
```

```
helper_function.num_cat_bivar_plots(penguins, "bill_length_mm", "species")
```



h) bivariate analysis between 2 categorical variables:

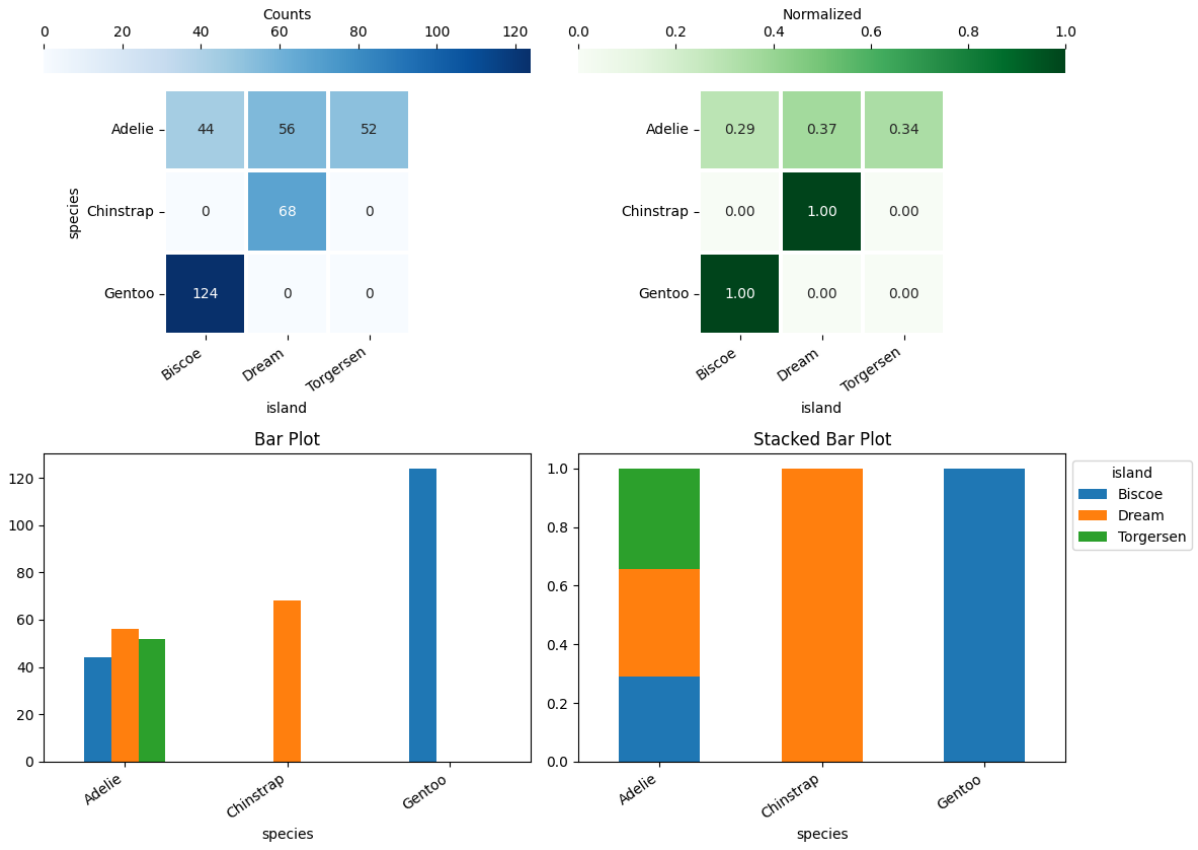
```
def cat_bivar_plots(data,
                    var1,
```

```
                    var2,
                    k1=None,
                    k2=None,
                    order1=None,
                    order2=None,
                    figsize=(12, 8.5)):

import warnings
warnings.filterwarnings("ignore")

display_html(2, f"Bi-variate Analysis between {var1} and {var2}")
display_html(content="")

if k1 is not None:
  data = get_top_k(
      data,
      var1,
      k=k1
  )

if k2 is not None:
  data = get_top_k(
      data,
      var2,
      k=k2
  )

fig, axes = plt.subplots(2, 2, figsize=figsize)
axes = axes.ravel()

# cross-tab heatmap
ct = (
    pd
    .crosstab(
        index=data.loc[:, var1],
        columns=data.loc[:, var2]
    )
    .reindex(
        index=order1,
        columns=order2
    )
)
hm = cat_heat_map(
    ct,
    mask=False,
    vmin=ct.values.min(),
    vmax=ct.values.max(),
    fmt="d",
```

```
    cmap="Blues",
    cbar_kws=dict(location="top", label="Counts"),
    ax=axes[0]
)
rotate_ylabels(axes[0])
rotate_xlabels(axes[0])

# normalized cross-tab heatmap
norm_ct = (
    pd
    .crosstab(
        index=data.loc[:, var1],
        columns=data.loc[:, var2],
        normalize="index"
    )
    .reindex(
        index=order1,
        columns=order2
    )
)
norm_hm = cat_heat_map(
    norm_ct,
    mask=False,
    vmin=0,
    vmax=1,
    fmt=".2f",
    cmap="Greens",
    cbar_kws=dict(location="top", label="Normalized"),
    ax=axes[1]
)
axes[1].set(ylabel="")
rotate_ylabels(axes[1])
rotate_xlabels(axes[1])

# bar plot
(
    ct
    .plot
    .bar(
        ax=axes[2],
        title="Bar Plot",
        legend=False
    )
)
rotate_xlabels(axes[2])

# stacked bar plot
(
```

```
    norm_ct
    .plot
    .bar(
        ax=axes[3],
        title="Stacked Bar Plot",
        stacked=True
    )
)
rotate_xlabels(axes[3])
axes[3].legend(
    loc="upper left",
    bbox_to_anchor=(1, 1),
    title=var2
)

plt.tight_layout()
plt.show()
```

```
helper_function.cat_bivar_plots(penguins, "species", "island")
```

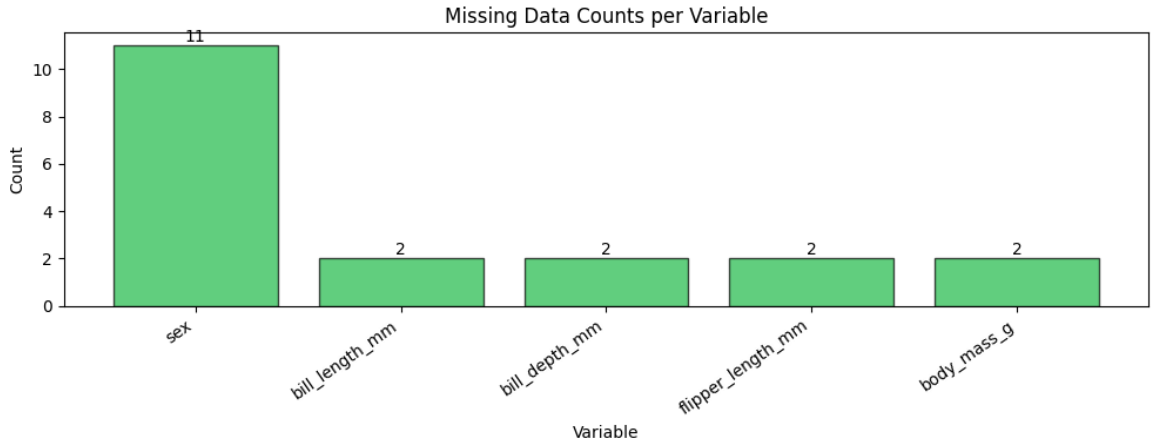


i) Plot missing info:

```
# ----------------------------------------------missing values-----------------
----------------------------------------
```

```python
def missing_info(data):
  na_cols = [col for col in data.columns if data[col].isna().any()]
  na_counts = [data[col].isna().sum() for col in na_cols]
  na_pct = [(data[col].isna().mean() * 100) for col in na_cols]

  return (
      pd
      .DataFrame(data={
          "variable": na_cols,
          "count": na_counts,
          "percentage": na_pct
      })
      .sort_values(by="count", ascending=False)
      .set_index("variable")
  )

# ----------------------------callable---------------------------------
def plot_missing_info(data, bar_label_params=dict(), figsize=(10, 4)):
  na_data = missing_info(data)
  fig, ax = plt.subplots(1, 1, figsize=figsize)
  bar = ax.bar(
      range(len(na_data)),
      height=na_data["count"].values,
      color="#1eba47",
      edgecolor="black",
      tick_label=na_data.index.to_list(),
      alpha=0.7
  )
  ax.bar_label(
      bar,
      **bar_label_params
  )
  ax.set(
      xlabel="Variable",
      ylabel="Count",
      title="Missing Data Counts per Variable"
  )
  rotate_xlabels(ax)
  plt.tight_layout()
  plt.show()
```

```
helper_function.plot_missing_info(penguins)
```



j) iqr outliers:

```
def get_iqr_outliers(data, var, band=1.5):
    q1, q3 = (
      data
      .loc[:, var]
      .quantile([0.25, 0.75])
      .values
    )

    iqr = q3 - q1
    lower_limit = q1 - (band * iqr)
    upper_limit = q3 + (band * iqr)

    display_html(3, f"{var} - IQR Limits:")
    print(f"{'Lower Limit':12}: {lower_limit}")
    print(f"{'Upper Limit':12}: {upper_limit}")

    return (
      data
      .query(f"{var} > @upper_limit | {var} < @lower_limit")
      .sort_values(var)
    )
```

```
helper_function.get_iqr_outliers(penguins, "bill_length_mm")
```

bill_length_mm - IQR Limits:

```
Lower Limit : 25.312500000000004
Upper Limit : 62.412499999999994
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex

1. **Advanced functions:**

a) detailed-summary-of-numerical-features:

```
def num_summary(data, var): #var: The column name (string) of the numerical
variable to be summarized.
  import warnings
  warnings.filterwarnings("ignore")  #suppress warnings to keep the output
clean.

                             # ------------------------title-----------------
--------
  col = data.loc[:, var].copy() #copy of the specified column from the
DataFrame.
  display_html(size=2, content=var)

                            # ----------------------------quick glance--------
----------------------
  display_html(3, "Quick Glance:")
  display(col)   #Displays the contents of the column.

                            # -----------------------------meta-data---------
----------------------
  display_html(3, "Meta-data:")
  print(f"{'Data Type':15}: {col.dtype}") #string 'Data Type' should be left-
aligned in a field of 15 characters. If 'Data Type' is shorter than 15
characters, the remaining space will be filled with spaces.
  print(f"{'Missing Data':15}: {col.isna().sum():,} rows ({col.isna().mean() *
100:.2f} %)") #Prints the number and percentage of missing data.
  print(f"{'Available Data':15}: {col.count():,} / {len(col):,} rows") #Prints
the number of available data points.

  # quantiles
  display_html(3, "Percentiles:")
  display(
      col
      .quantile([0.0, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99, 1.0])
#Computes the specified percentiles of the column.
      .rename(index=lambda val: f"{val * 100:.0f}")#Renames the indices to
percentage values.
      .rename("value")#Converts the Series to a DataFrame for better display.
      .rename_axis(index="percentile")#Displays the DataFrame with
percentiles.
      .to_frame()
  )

                           #-----------------------------------------------
central tendancy-----------------------------------
  display_html(3, "Central Tendancy:")
```

```
  display(
      pd    #Creates a Series with various measures of central tendency
      .Series({"mean": col.mean(), #Mean of the column.
               "trimmed mean (5%)": stats.trim_mean(col.values, 0.05), #5%
trimmed mean.
               "trimmed mean (10%)": stats.trim_mean(col.values, 0.1), #10%
trimmed mean.
               "median": col.median()}) #Median of the column
      .rename("value") #Renames the Series.
      .to_frame() #Converts the Series to a DataFrame for better display.
  )

  # spread
  display_html(3, "Measure of Spread:")
  std = col.std() #Standard deviation of the column.
  iqr = col.quantile(0.75) - col.quantile(0.25) #Interquartile range (75th
percentile - 25th percentile).
  display(
      pd
      .Series({
          "var": col.var(), # Variance of the column.
          "std": std, #Standard deviation of the column.
          "IQR": iqr, #Interquartile range.
          "mad": stats.median_abs_deviation(col.dropna()),#Median absolute
deviation.
          "coef_variance": std / col.mean() #Coefficient of variation (std /
mean).
      })
      .rename("value")
      .to_frame()
  )

                          # ------------------------------skewness and
kurtosis---------------------------------
  display_html(3, "Skewness and Kurtosis:")
  display(
      pd  #Creates a Series with skewness and kurtosis.
      .Series({
          "skewness": col.skew(), #Skewness of the column.
          "kurtosis": col.kurtosis() #Kurtosis of the column.
      })
      .rename("value")# renames the series
      .to_frame() #Converts the Series to a DataFrame for better display.
  )
```

```
helper_function.num_summary(penguins,"bill_length_mm")
```

bill_length_mm

Quick Glance:

	bill_length_mm
0	39.1
1	39.5
2	40.3
3	NaN
4	36.7
...	...
339	NaN
340	46.8
341	50.4
342	45.2
343	49.9

344 rows × 1 columns

Meta-data:

Data Type : float64
Missing Data : 2 rows (0.58 %)
Available Data : 342 / 344 rows

Percentiles:

percentile	value
0	32.100
5	35.700
10	36.600
25	39.225
50	44.450
75	48.500
90	50.800
95	51.995
99	55.513
100	59.600

Central Tendancy:

	value
mean	43.921930
trimmed mean (5%)	43.935161
trimmed mean (10%)	43.956884
median	44.450000

Measure of Spread:

	value
var	29.807054
std	5.459584
IQR	9.275000
mad	4.750000
coef_variance	0.124302

Skewness and Kurtosis:

	value
skewness	0.053118
kurtosis	-0.876027

Hypothesis Testing for Normality:

Shapiro-Wilk Test:

```
Significance Level   : 0.05
Null Hypothesis      : The data is normally distributed
Alternate Hypothesis : The data is not normally distributed
p-value              : 1.1197299438900768e-05
Test Statistic       : 0.9748548096753171
- Since p-value is less than alpha (0.05), we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: We conclude that the data sample is not normally distributed
```

Anderson-Darling Test:

```
Significance Level   : 0.05
Null Hypothesis      : The data is normally distributed
Alternate Hypothesis : The data is not normally distributed
Critical Value       : 0.778
Test Statistic       : 3.0204613418821964
- Since the Test-statistic is greater than Critical Value, we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: We conclude that the data sample is not normally distributed
```

b) hypothesis testing for association between 2 numeric variables:

```python
#if there is a statistically significant relationship between the variables.
def num_num_hyp_testing(data, var1, var2, alpha=0.05): #alpha: The
significance level for the hypothesis tests, defaulting to 0.05.
  display_html(2, f"Hypothesis Test for Association between {var1} and
{var2}")

  temp = (
      data
      .dropna(subset=[var1, var2], how="any") #Drops any rows from the dataset
where either var1 or var2 is missing (NaN).
      .copy() #Creates a copy of the filtered dataset.
  )

  # pearson test
  pearson = stats.pearsonr(temp[var1].values, temp[var2].values) #Calculates
the Pearson correlation coefficient and the associated p-value using
  pvalue = pearson.pvalue #The p-value of the Pearson correlation test.
  statistic = pearson.statistic #The Pearson correlation coefficient.
  display_html(3, "Pearson Test")
  print(f"- {'Significance Level':21}: {alpha * 100}%")
  print(f"- {'Null Hypothesis':21}: The samples are uncorrelated")
  print(f"- {'Alternate Hypothesis':21}: The samples are correlated")
  print(f"- {'Test Statistic':21}: {statistic}")
  print(f"- {'p-value':21}: {pvalue}")
  if pvalue < alpha: #Based on the p-value, determines whether to reject or
fail to reject the null hypothesis.
    print(f"- Since p-value is less than {alpha}, we Reject the Null
Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {var1} and {var2} are correlated")
  else:
    print(f"- Since p-value is greater than {alpha}, we Fail to Reject the
Null Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {var1} and {var2} are uncorrelated")

  # spearman test
  spearman = stats.spearmanr(temp[var1].values, temp[var2].values)
  pvalue = spearman.pvalue
  statistic = spearman.statistic
  display_html(3, "Spearman Test")
  print(f"- {'Significance Level':21}: {alpha * 100}%")
  print(f"- {'Null Hypothesis':21}: The samples are uncorrelated")
  print(f"- {'Alternate Hypothesis':21}: The samples are correlated")
  print(f"- {'Test Statistic':21}: {statistic}")
  print(f"- {'p-value':21}: {pvalue}")
  if pvalue < alpha: #Based on the p-value, determines whether to reject or
fail to reject the null hypothesis.
    print(f"- Since p-value is less than {alpha}, we Reject the Null
Hypothesis at {alpha * 100}% significance level")
```

```
    print(f"- CONCLUSION: The variables {var1} and {var2} are correlated")
  else:
    print(f"- Since p-value is greater than {alpha}, we Fail to Reject the
Null Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {var1} and {var2} are uncorrelated")
```

```
helper_function.num_num_hyp_testing(penguins, "bill_length_mm", "bill_depth_mm", alpha=0.05)
```

Hypothesis Test for Association between bill_length_mm and bill_depth_mm

Pearson Test

- Significance Level : 5.0%
- Null Hypothesis : The samples are uncorrelated
- Alternate Hypothesis : The samples are correlated
- Test Statistic : -0.2350528703555327
- p-value : 1.119662196137322e-05
- Since p-value is less than 0.05, we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: The variables bill_length_mm and bill_depth_mm are correlated

Spearman Test

- Significance Level : 5.0%
- Null Hypothesis : The samples are uncorrelated
- Alternate Hypothesis : The samples are correlated
- Test Statistic : -0.22174915179457863
- p-value : 3.511539739648998e-05
- Since p-value is less than 0.05, we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: The variables bill_length_mm and bill_depth_mm are correlated

c) detailed summary of categorical features:

```
def cat_summary(data, var):
  import warnings
  warnings.filterwarnings("ignore")

  # title
  col = data.loc[:, var].copy()
  display_html(2, var)

  # quick glance
  display_html(3, "Quick Glance:")
  display(col)

  # meta-data
  display_html(3, "Meta-data:")
  print(f"{'Data Type':15}: {col.dtype}") #prints the data type of the column
col
  print(f"{'Cardinality':15}: {col.nunique(dropna=True)} categories")
  print(f"{'Missing Data':15}: {col.isna().sum():,} rows ({col.isna().mean() *
100:.2f} %)")
  print(f"{'Available Data':15}: {col.count():,} / {len(col):,} rows")

  # summary
  display_html(3, "Summary:")
```

```
  display(
      col
      .describe() #generates descriptive statistics for the column col. It
provides a summary of the column's numerical data,
      .rename("") #it's setting the index name to an empty string, which
effectively removes any existing index name.
      .to_frame() #This method converts the Series returned by col.describe()
into a DataFrame
  )

  # categories
  display_html(3, "Categories Distribution:")
  with pd.option_context("display.max_rows", None): #This context manager
temporarily sets the display option display.max_rows to None, which means that
all rows in a DataFrame or Series will be displayed without truncation. This
is useful for viewing the full distribution of category counts.
    display(
        col
        .value_counts() #Series containing counts of unique values in the
column col.
        .pipe(lambda ser: pd.concat( #This concatenates two Series along the
columns (axis=1). The first Series is the counts of each category, and the
second Series is the percentage of each category.
            [
                ser,
                col.value_counts(normalize=True)
            ],
            axis=1 #pipe method allows you to apply a function (in this case,
a lambda function) to the result of col.value_counts(). The lambda function
takes this Series (ser) and concatenates it with another Series created by
col.value_counts(normalize=True).
        ))
        .set_axis(["count", "percentage"], axis=1)  #returns the relative
frequencies (proportions) of unique values in the column
        .rename_axis(index="category")
    )
```

```
helper_function.cat_summary(penguins, "species")
```

species

Quick Glance:

	species
0	Adelie
1	Adelie
2	Adelie
3	Adelie
4	Adelie
...	...
339	Gentoo
340	Gentoo
341	Gentoo
342	Gentoo
343	Gentoo

344 rows × 1 columns

dtype: object

Meta-data:

```
Data Type      : object
Cardinality    : 3 categories
Missing Data   : 0 rows (0.00 %)
Available Data : 344 / 344 rows
```

Summary:

count	344
unique	3
top	Adelie
freq	152

Categories Distribution:

category	count	percentage
Adelie	152	0.441860
Gentoo	124	0.360465
Chinstrap	68	0.197674

d) hypothesis testing for association between numeric and categorical variable:

```
def num_cat_hyp_testing(data, num_var, cat_var, alpha=0.05):
  display_html(2, f"Hypothesis Test for Association between {num_var} and
{cat_var}")

  groups_df = (
      data
      .dropna(subset=[num_var])   #num_var column should be checked for
missing values. Rows where num_var is NaN will be dropped from the DataFrame.
      .groupby(cat_var) #Groups the DataFrame by the values in the cat_var
column.
  )
  groups = [group[num_var].values for _, group in groups_df]  #Creates a list
of arrays containing the values of num_var for each group.
  #_ is used to ignore the group keys (which are the unique values of
cat_var).
  # performs the one-way ANOVA test.
  anova = stats.f_oneway(*groups)  #* operator is used to unpack the groups
list into separate arguments. Each element of groups is a NumPy array
containing the num_var values for one group.
  statistic = anova[0] #Extracts the F-statistic from the ANOVA results.
  pvalue = anova[1] #Extracts the p-value from the ANOVA results.
  display_html(3, "ANOVA Test")
```

```
  print(f"- {'Significance Level':21}: {alpha * 100}%")
  print(f"- {'Null Hypothesis':21}: The groups have similar population mean")
  print(f"- {'Alternate Hypothesis':21}: The groups don't have similar
population mean")
  print(f"- {'Test Statistic':21}: {statistic}")
  print(f"- {'p-value':21}: {pvalue}")
  if pvalue < alpha:
    print(f"- Since p-value is less than {alpha}, we Reject the Null
Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {num_var} and {cat_var} are associated
to each other")
  else:
    print(f"- Since p-value is greater than {alpha}, we Fail to Reject the
Null Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {num_var} and {cat_var} are not
associated to each other")

  # kruskal-wallis test
  kruskal = stats.kruskal(*groups)
  statistic = kruskal[0]
  pvalue = kruskal[1]
  display_html(3, "Kruskal-Wallis Test")
  print(f"- {'Significance Level':21}: {alpha * 100}%")
  print(f"- {'Null Hypothesis':21}: The groups have similar population
median")
  print(f"- {'Alternate Hypothesis':21}: The groups don't have similar
population median")
  print(f"- {'Test Statistic':21}: {statistic}")
  print(f"- {'p-value':21}: {pvalue}")
  if pvalue < alpha:
    print(f"- Since p-value is less than {alpha}, we Reject the Null
Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {num_var} and {cat_var} are associated
to each other")
  else:
    print(f"- Since p-value is greater than {alpha}, we Fail to Reject the
Null Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {num_var} and {cat_var} are not
associated to each other")
```

```
helper_function.num_cat_hyp_testing(penguins, "bill_length_mm", "species", alpha=0.05)
```

Hypothesis Test for Association between bill_length_mm and species

ANOVA Test

- Significance Level : 5.0%
- Null Hypothesis : The groups have similar population mean
- Alternate Hypothesis : The groups don't have similar population mean
- Test Statistic : 410.6002550405077
- p-value : 2.6946137388895484e-91
- Since p-value is less than 0.05, we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: The variables bill_length_mm and species are associated to each other

Kruskal-Wallis Test

- Significance Level : 5.0%
- Null Hypothesis : The groups have similar population median
- Alternate Hypothesis : The groups don't have similar population median
- Test Statistic : 244.13671803364164
- p-value : 9.691371997194331e-54
- Since p-value is less than 0.05, we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: The variables bill_length_mm and species are associated to each other

e) hypothesis testing between 2 categorical variables:

```
def hyp_cat_cat(data, var1, var2, alpha=0.05):
  display_html(2, f"Hypothesis Test for Association between {var1} and
{var2}")

  ct = pd.crosstab(
      data.loc[:, var1],
      data.loc[:, var2]
  )

  display_html(3, "Chi-square Test")
  chi2 = stats.chi2_contingency(ct)
  statistic = chi2.statistic
  pvalue = chi2.pvalue
  print(f"- {'Cramers V':21}: {cramers_v(data, var1, var2)}")
  print(f"- {'Significance Level':21}: {alpha * 100}%")
  print(f"- {'Null Hypothesis':21}: The samples are uncorrelated")
  print(f"- {'Alternate Hypothesis':21}: The samples are correlated")
  print(f"- {'Test Statistic':21}: {statistic}")
  print(f"- {'p-value':21}: {pvalue}")
  if pvalue < alpha:
    print(f"- Since p-value is less than {alpha}, we Reject the Null
Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {var1} and {var2} are correlated")
  else:
    print(f"- Since p-value is greater than {alpha}, we Fail to Reject the
Null Hypothesis at {alpha * 100}% significance level")
    print(f"- CONCLUSION: The variables {var1} and {var2} are uncorrelated")

# missing values
def missing_info(data):
```

```
na_cols = [col for col in data.columns if data[col].isna().any()]
na_counts = [data[col].isna().sum() for col in na_cols]
na_pct = [(data[col].isna().mean() * 100) for col in na_cols]

return (
    pd
    .DataFrame(data={
        "variable": na_cols,
        "count": na_counts,
        "percentage": na_pct
    })
    .sort_values(by="count", ascending=False)
    .set_index("variable")
)
```

```
helper_function.hyp_cat_cat(penguins, "species", "island")
```

Hypothesis Test for Association between species and island

Chi-square Test

- Cramers V : 0.6573290190537864
- Significance Level : 5.0%
- Null Hypothesis : The samples are uncorrelated
- Alternate Hypothesis : The samples are correlated
- Test Statistic : 299.55032743148195
- p-value : 1.3545738297192517e-63
- Since p-value is less than 0.05, we Reject the Null Hypothesis at 5.0% significance level
- CONCLUSION: The variables species and island are correlated