```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 15 14:31:19 2025

@author: Martin Omasta
"""


#Section 1: import key packages & libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import shapiro, t
#import types
#from matplotlib.table import Table
#END 1############################################################################



#Section 2: read .csv of salary ranges
infile = "MartinOmasta.module05RProject.csv"
salary_ranges_df = pd.read_csv(infile)
#print(salary_ranges_df.head())
#END 2############################################################################



#Section 3:Thoughts on Summary & Analysis for Prof Lowhorn. Code Starts around Line 85
"""
The thing that stands out most to me is the type of employment that these data
scientists seem to prefer - 588 of the 607 are Full-Time employees - suggesting
that whatever offer we make - it will likely be most well received if it is a
full time position. This full time desire seems to fit with our CEO's desire to
attract a person who can drive data science in the whole company and lead a team
in the future.

In the same vein - 409 of these folk work at a Small-to-Medium sized company,
and given our companies current size and growth trajectory - we should have no
problem in selling them on the idea that this is a company they can see upward
mobility & career growth in.

This dataset mostly contains employees from the US (332 vs 275 non-US). This is
fine enough for relative sample sizes, but not sure how indicitive it is of the actual
talent landscape. (Question for self - what kind of distribution does this
                    follow? Can CLT be applied? Doesn't seem gaussian)

There are a good chunck of the talent pool who range from Middle to Senior levels
of experience. This is likely indicative of the wider talent pool, which is good
for our near-term goals of finding someone to take on the companies data science
needs. I would suggest having me as part of the appliation review/interview
process to be able to give a technical opinion of their capability. However, the
data science industry writ-large is facing bottle-necks of new-entry level positions
(https://nathanrosidi.medium.com/how-we-oversaturated-the-data-science-job-market-52dc88ffe50c)
which seem to be crushing the pipeline for development into Middle to Senior level
experience. If we want to attract & retain young talent as part of a future team -
we will want to select someone as our team lead who we feel can nurture and develop
```

```
fresh entries.

To that end of team development - these computer driven individuals also have a
somewhat high proclivity toward remote work. Over 60% work remote nearly all
of the time, and nearly 80% work remote at least half the time. This is a double
edged sword for our purposes. Finding either OCONUS workers or US CONUS workers
could be irrelevant - and if we want to focus on US based workforce - in our final
decision process we could also take into consideration the cost of living in the
various area people live to drive part of our salary negotions. If we are hoping
to do any in-person development with any sort of regularity, however, it will become
necessary to find talent who live/work remotely within a reasonablly close
geographic region. This sort of work, can of course also be done remotely, but
the benefits of in person interaction have been laid bare in recent years.
(potential examination: How many employee_residence live in the same country as
 their company_location?) Over 91% of these data scientists work in their home
country.


Works Cited:
1) Google, Virtual Assistant (VA). Prompt used to generate response (Various prompts
"Explain and build Pythonic code to for data examination structures (charts, grahps, etc)"). Retrie
VAMMA (Virtual Assistant with Manual Modification and Arrangement, Gemini, Google)

2) Seaborn boxplot. (2012-2024). API explanation. Retrieved 16 Nov. 2025.
https://seaborn.pydata.org/generated/seaborn.boxplot.html

"""
#END 3#########################################################################



#Section 4: convert relevant columns to Categorical
salary_ranges_df['work_year'] = pd.Categorical(salary_ranges_df.work_year)

salary_ranges_df['experience_level'] = pd.Categorical(salary_ranges_df.experience_level)
#also order experience level
skill_order = ['EN', 'MI', 'SE', 'EX']
level_type = pd.CategoricalDtype(categories = skill_order, ordered = True)
salary_ranges_df['experience_level'] = salary_ranges_df['experience_level'].astype(level_type)

salary_ranges_df['employment_type'] = pd.Categorical(salary_ranges_df.employment_type)
salary_ranges_df['job_title'] = pd.Categorical(salary_ranges_df.job_title)
salary_ranges_df['salary_currency'] = pd.Categorical(salary_ranges_df.salary_currency)
#salary_ranges_df['employee_residence'] = pd.Categorical(salary_ranges_df.employee_residence)
salary_ranges_df['remote_ratio'] = pd.Categorical(salary_ranges_df.remote_ratio)
#salary_ranges_df['company_location'] = pd.Categorical(salary_ranges_df.company_location)
salary_ranges_df['company_size'] = pd.Categorical(salary_ranges_df.company_size)
#print(salary_ranges_df.dtypes)
#END 4#########################################################################



#Section 5: Data Summaries & Analysis (VA & Manually Modified Arranged) VAMMA (Virtual Assistant wi
#A) inspect some key salary statistics including mean, std, and the quartiles (VAMMA))
print("--- 1. Salary Statistics (.describe()) ---")
salary_columns = ['salary', 'salary_in_usd']
print(salary_ranges_df[salary_columns].describe().
```

```
      apply(lambda s: s.apply('{:,.2f}'.format))    )#show salary to 2 decimal places, non scientif

#B) Some more inspection as requested by project worksheet
print("\n")
print("--- 2. Data Structure and Types (.info()) ---")
salary_ranges_df.info()

print("\n")
original_max_cols = pd.get_option('display.max_columns') #to reset IPython display later
pd.set_option('display.max_columns', None) #gives all columns, so not truncated by ellipse
print("--- 3. Descriptive Statistics (.describe()) ---")
print(salary_ranges_df.describe(include='all'))
pd.set_option('display.max_columns', original_max_cols) #reset display to values few lines above


#C) Some more inspection as requested by project worksheet (after Categorization)
def df_value_counts(user_df, num_cols_to_display):
    """Function to display value counts for all columns of any datafile passed"""
    # Get a list of all column names
    for column in user_df.columns:
        print(f"\nValue Counts for: {column}")

        # Use value_counts() to show the frequency of the top 5 values
        # You can customize this based on the column's Dtype
        if user_df[column].dtype in ['object', 'category']:
            print(user_df[column].value_counts().head(num_cols_to_display))
        else:
            # For numeric columns, print the mean and standard deviation again for quick check
            print(f"Mean: {user_df[column].mean():.2f}")
            print(f"Std Dev: {user_df[column].std():.2f}")

print("\n")
print("--- 4. Alternative Summary: Top Value Counts ---")
df_value_counts(salary_ranges_df, 17)
#END 5#############################################################################



#Section 6: Boxplot of what the salary ranges are by experience level (VAMMA)
plt.figure(figsize=(10, 8))
new_experience_labels = ['Entry', 'Intermediate', 'Senior', 'Executive']

boxplot_salary_range_by_experience = sns.boxplot(salary_ranges_df,
    x = 'experience_level', y = "salary_in_usd",
    hue = 'experience_level', orient = "v")

boxplot_salary_range_by_experience.set(
    title = "USD $ Salary by Experience Level",
    xlabel = "Experience Level",
    ylabel = "Salary (USD $)")#.set_xticklabels(new_labels)


# This ensures a tick mark exists for every label, & creates the label
num_exp_categories = len(new_experience_labels)
boxplot_salary_range_by_experience.set_xticks(range(num_exp_categories))
boxplot_salary_range_by_experience.set_xticklabels(new_experience_labels)
```

```
plt.tight_layout()
plt.show()
#END 6#########################################################################



#Section 7: Violin plot of what the salary ranges are by experience level (just switched up Sec 6)
plt.figure(figsize=(10, 8))

violinplot_salary_range_by_experience = sns.violinplot(salary_ranges_df,
    x = 'experience_level', y = "salary_in_usd",
    hue = 'experience_level', orient = "v")

violinplot_salary_range_by_experience.set(
    title = "USD $ Salary by Experience Level",
    xlabel = "Experience Level",
    ylabel = "Salary (USD $)")#.set_xticklabels(new_labels)


# This ensures a tick mark exists for every label, & creates the label
num_exp_categories = len(new_experience_labels)
violinplot_salary_range_by_experience.set_xticks(range(num_exp_categories))
violinplot_salary_range_by_experience.set_xticklabels(new_experience_labels)

plt.tight_layout()
plt.show()
#END 7#########################################################################



#Section 8: View the numerical information of the salary_range_by_experience boxplot (VAMMA)
quantiles = [0.25, 0.5, 0.75]

usd_salary_quartile_data = salary_ranges_df.groupby('experience_level',
    observed = True)['salary_in_usd'].quantile(quantiles)#This creates a panda Series

#This moves the quantile levels (0.25, 0.5, 0.75) from the index to columns.
usd_salary_quartile_table = usd_salary_quartile_data.unstack()
usd_salary_quartile_table.columns = ['Q1 (25th)', 'Median (50th)', 'Q3 (75th)']

#Change the output of the table for experience_level column to be more readable
original_abbreviations = ['EN', 'MI', 'SE', 'EX']
label_map = dict(zip(original_abbreviations, new_experience_labels))
usd_salary_quartile_table = usd_salary_quartile_table.rename(index = label_map)

print("\n")
print("--- Salary Quartile Values by Experience Level Worldwide---")
print(usd_salary_quartile_table)
#END 8#########################################################################



#Section 9: Convert the DataFrame index (Experience Level) and columns to lists for the table visua
data_values_world = usd_salary_quartile_table.values.round(0).astype(int) # Round and convert to ir
data_values_world = [[f'${x:,.0f}' for x in row]
                     for row in usd_salary_quartile_table.values]#convert to list of currency forma
```

```
row_labels_world = usd_salary_quartile_table.index.tolist()
col_labels_world = usd_salary_quartile_table.columns.tolist()

# --- Create the Matplotlib Figure and Table ---
plt.figure(figsize = (8, 3)) # Adjust size for the table
ax_world = plt.gca() # Get the current axes object

# 1. Hide the axes (the plot area itself) so only the table is visible
ax_world.axis('off')
ax_world.axis('tight')

# 2. Draw the table onto the axes
# 'cellText' is the data. 'rowLabels' and 'colLabels' define the headers.
table_world = ax_world.table(
    cellText = data_values_world,
    rowLabels = row_labels_world,
    colLabels = col_labels_world,
    loc = 'center',
    cellLoc = 'center', # Center the text within cells
    rowLoc = 'center'
)

#Style and Title
table_world.auto_set_font_size(False)
table_world.set_fontsize(12)
table_world.scale(0.8, 1.5) # Scale width and height of cells

plt.title('Salary Quartile Values by Experience Level Worldwide', fontsize=14, pad=0)

# Display the figure (sends it to the Plots pane in Spyder)
plt.show()
#END 9#############################################################################



#Section 10: Boxplot to examine effect of U.S. Residency on salary ranges
#Create a new column to categorize residence as 'US' or 'Non-US' (VAMMA)
salary_ranges_df['residence_group'] = np.where(
    salary_ranges_df['employee_residence'] == 'US',
    'US',
    'Non-US'
)
salary_ranges_df['residence_group'] = pd.Categorical(salary_ranges_df['residence_group'])

#Boxplot with US & Non-US side by side
plt.figure(figsize=(12, 8))

# Use Residence_Group for the primary axis (x) and experience_level for color (hue)
boxplot_salary_by_exp_and_residency = sns.boxplot(data=salary_ranges_df,
                 x = 'residence_group',
                 y = "salary_in_usd",
                 hue = 'experience_level',
                 orient = "v")

"""
# Get the current y-axis limits
```

```
#y_min, y_max = boxplot_salary_by_exp_and_residency.get_ylim()
# Create an array of tick locations with a step of 50,000
new_yticks = np.arange(0, 600000 + 50000, 50000)
# Set the new y-axis ticks
boxplot_salary_by_exp_and_residency.set_yticks(new_yticks)
#This was TOO Cluttered"""

# Increase font size for the axis ticks & add a horizontal grid
boxplot_salary_by_exp_and_residency.tick_params(axis='both',
                                                which='major',
                                                labelsize=16)
boxplot_salary_by_exp_and_residency.yaxis.grid(True)

# boxplot_salary_by_exp_and_residency.set(title = "Salary Distribution by US/Non-US and Experience
#        xlabel = "Employee Residence",
#        ylabel = "Salary (USD $)")

boxplot_salary_by_exp_and_residency.set_title("Salary Distribution by US/Non-US and Experience Leve
                                              fontsize = 24)
boxplot_salary_by_exp_and_residency.set_xlabel("Employee Residence",
                                               fontsize = 20)
boxplot_salary_by_exp_and_residency.set_ylabel("Salary (USD $)",
                                               fontsize = 20)

#Add a legend label title with modified Experience Level labels
handles, current_labels = boxplot_salary_by_exp_and_residency.get_legend_handles_labels()
boxplot_salary_by_exp_and_residency.legend(handles, new_experience_labels,
                                           title = 'Experience Level',
                                           fontsize = 20,
                                           title_fontsize = 20,
                                           markerscale = 1.5)

plt.tight_layout()
plt.show()
#END 10#########################################################################



#Section 11: Violin plot to examine effect of U.S. Residency on salary ranges
#Violin plot with US & Non-US side by side (just switched up Sec 10)
plt.figure(figsize=(12, 8))

# Use Residence_Group for the primary axis (x) and experience_level for color (hue)
violinplot_salary_by_exp_and_residency = sns.violinplot(data=salary_ranges_df,
                 x = 'residence_group',
                 y = "salary_in_usd",
                 hue = 'experience_level',
                 orient = "v")

violinplot_salary_by_exp_and_residency.set(title = "Salary Distribution by US/Non-US and Experience
       xlabel = "Employee Residence",
       ylabel = "Salary (USD $)")

#Add a legend label title with modified Experience Level labels
handles, current_labels = violinplot_salary_by_exp_and_residency.get_legend_handles_labels()
violinplot_salary_by_exp_and_residency.legend(handles, new_experience_labels,
                                           title = 'Experience Level')
```

```
plt.tight_layout()
plt.show()
#END 11#############################################################################



#Section 12: View the numerical information of the effect of U.S. Residency on salary
#ranges salary_by_exp_and_residency boxplot. (just switched up Sec 8)
quantiles = [0.25, 0.5, 0.75]

usd_salary_residence_quartile_data = salary_ranges_df.groupby(['residence_group', 'experience_level
    observed = True)['salary_in_usd'].quantile(quantiles)

#This moves the quantile levels (0.25, 0.5, 0.75) from the index to columns.
usd_salary_residence_quartile_table = usd_salary_residence_quartile_data.unstack()
#This renames the columns to be more meaningful for viewing assessment
usd_salary_residence_quartile_table.columns = ['Q1 (25th)', 'Median (50th)', 'Q3 (75th)']

#Change the output of the table for experience_level column to be more readable
original_abbreviations = ['EN', 'MI', 'SE', 'EX']
label_map = dict(zip(original_abbreviations, new_experience_labels))
usd_salary_residence_quartile_table = usd_salary_residence_quartile_table.rename(index = label_map)

print("\n")
print("--- Salary Quartile Values by Experience Level and US/Non-US ---")
print(usd_salary_residence_quartile_table)
#END 12#############################################################################



#Section 13: Convert the DataFrame index (Experience Level) and columns to lists for the table visu
# ***Not ending up the primary one because the bold styling of the line is too wonky
# with single vertical columns on just one row (6th) being bolded
data_values_residency = usd_salary_residence_quartile_table.values.round(0).astype(int) # Round and
data_values_residency = [[f'${x:,.0f}' for x in row]
                     for row in usd_salary_residence_quartile_table.values]#convert to list of curr


row_labels_residency = usd_salary_residence_quartile_table.index.tolist()
col_labels_residency = usd_salary_residence_quartile_table.columns.tolist()

# --- Create the Matplotlib Figure and Table ---
plt.figure(figsize = (8, 3)) # Adjust size for the table
ax_residency = plt.gca() # Get the current axes object

# 1. Hide the axes (the plot area itself) so only the table is visible
ax_residency.axis('off')
ax_residency.axis('tight')

# 2. Draw the table onto the axes
# 'cellText' is the data. 'rowLabels' and 'colLabels' define the headers.
table_residency = ax_residency.table(
    cellText = data_values_residency,
    rowLabels = row_labels_residency,
    colLabels = col_labels_residency,
    loc = 'center',
```

```
    cellLoc = 'center', # Center the text within cells
    rowLoc = 'center'
)

#Style and Title
table_world.auto_set_font_size(False)
table_world.set_fontsize(12)
table_world.scale(0.4, 1.5) # Scale width and height of cells

plt.title('Salary Quartile Values by Residency & Experience Level', fontsize=14, pad=0)

# Display the figure (sends it to the Plots pane in Spyder)
plt.show()


#Establish the line on which the bolding should occur
NUM_ROWS = len(usd_salary_residence_quartile_table) # Should be 8 (4 Non-US, 4 US)
NON_US_ROWS = NUM_ROWS // 2 # Should be 4
US_ROWS = NUM_ROWS // 2 # Should be 4

# --- MODIFIED DATA PREPARATION ---

data_values_residency = [[f'${x:,.0f}' for x in row]
                         for row in usd_salary_residence_quartile_table.values]

# 1. Split the existing row labels to create the two columns:
# Assuming the index is structured as: (Residency, Experience_Level)
# If the index is a flat list of strings like 'Non-US-Entry', you need to split it first.
# We will assume a simple split for now to get the two labels:
exp_labels = []
residency_labels_raw = []

# This structure assumes the index is a Pandas MultiIndex, which is standard after a multi-level gr
if isinstance(usd_salary_residence_quartile_table.index, pd.MultiIndex):
    residency_labels_raw = usd_salary_residence_quartile_table.index.get_level_values(0).tolist()
    exp_labels = usd_salary_residence_quartile_table.index.get_level_values(1).tolist()
else:
    # Fallback if the index is a flat list (less ideal)
    for label in usd_salary_residence_quartile_table.index:
        residency_labels_raw.append(label.split(',')[0].strip('{').strip().replace("'", ""))
        exp_labels.append(label.split(',')[1].strip('}').strip().replace("'", ""))

# 2. Create the Primary Grouping Column (Residency) using a list with repeating labels for the cell
residency_labels = [residency_labels_raw[i] if i % NON_US_ROWS == 0 else '' for i in range(NUM_ROWS

# 3. Combine the experience labels and data values for the final table structure
# The final table will have 5 columns: [Residency, Experience, Q1, Median, Q3]
cell_data = [[residency_labels[i], exp_labels[i]] + data_values_residency[i]
             for i in range(NUM_ROWS)]

# The new column labels now include the Experience Level column
col_labels_final = ['Residency'] + ['Experience'] + col_labels_residency

# --- Create the Matplotlib Figure and Table ---
plt.figure(figsize=(10, 5)) # Increased size to accommodate extra labels
ax_residency = plt.gca()
ax_residency.axis('off')
```

```
ax_residency.axis('tight')

# 4. Draw the table onto the axes
table_residency = ax_residency.table(
    cellText=cell_data,
    colLabels=col_labels_final,
    loc='center',
    cellLoc='center'
)

# --- 5. Custom Styling for Hierarchical Grouping and Bolding ---

# Loop through cells to adjust spans and bold the mid-line
for i in range(NUM_ROWS + 1): # Include header row
    for j in range(len(col_labels_final)):
        cell = table_residency[i, j]
        cell.set_edgecolor('black')

        # BOLDING THE MIDDLE LINE (Row 5 is the first row of the US group)
        if i == NON_US_ROWS + 1: # Row 5 (index 4) is the start of US data
             # Set a thicker top border for all cells in the US group
             cell.set_linewidth(3.0)
        else:
             cell.set_linewidth(1.0)

# Merge (Span) the Residency cells:
# Merge Non-US cells (Rows 1 to 4)

# ============================================================================
# table_residency.get_celld()[(1, 0)].set_rowspan(NON_US_ROWS)
# for i in range(2, NON_US_ROWS + 1):
#     table_residency.get_celld()[(i, 0)].set_visible(False)
#
# # Merge US cells (Rows 5 to 8)
# table_residency.get_celld()[(NON_US_ROWS + 1, 0)].set_rowspan(US_ROWS)
# for i in range(NON_US_ROWS + 2, NUM_ROWS + 1):
#     table_residency.get_celld()[(i, 0)].set_visible(False)
# ============================================================================
#this caused an error, so commented out. Leaving for future reference


# Style and Title
table_residency.auto_set_font_size(False)
table_residency.set_fontsize(12)
table_residency.scale(0.8, 1.5) # Scale to make room for merged cells

plt.title('Salary Quartile Values by Residency & Experience Level', fontsize=14, pad=0)

# Display the figure (sends it to the Plots pane in Spyder)
plt.show()
#END 13####BAD BOLD#############################################################



#Section 14: Same data as Section 11, differnt styling attempt
#BAD Bolding, but does do cell spanning and line removal  (VAMMA)
```

```
NUM_ROWS = len(usd_salary_residence_quartile_table) # Should be 8 (4 Non-US, 4 US)
NON_US_ROWS = NUM_ROWS // 2 # Should be 4
US_ROWS = NUM_ROWS // 2 # Should be 4

# --- MODIFIED DATA PREPARATION ---

data_values_residency = [[f'${x:,.0f}' for x in row]
                         for row in usd_salary_residence_quartile_table.values]

# 1. Split the existing row labels to create the two columns:
# Assuming the index is structured as: (Residency, Experience_Level)
# If the index is a flat list of strings like 'Non-US-Entry', you need to split it first.
# We will assume a simple split for now to get the two labels:
exp_labels = []
residency_labels_raw = []

# This structure assumes the index is a Pandas MultiIndex, which is standard after a multi-level gr
if isinstance(usd_salary_residence_quartile_table.index, pd.MultiIndex):
    residency_labels_raw = usd_salary_residence_quartile_table.index.get_level_values(0).tolist()
    exp_labels = usd_salary_residence_quartile_table.index.get_level_values(1).tolist()
else:
    # Fallback if the index is a flat list (less ideal)
    for label in usd_salary_residence_quartile_table.index:
        residency_labels_raw.append(label.split(',')[0].strip('{').strip().replace("'", ""))
        exp_labels.append(label.split(',')[1].strip('}').strip().replace("'", ""))

# 2. Create the Primary Grouping Column (Residency) using a list with repeating labels for the cell
residency_labels = [residency_labels_raw[i] if i % NON_US_ROWS == 0 else '' for i in range(NUM_ROWS

# 3. Combine the experience labels and data values for the final table structure
# The final table will have 5 columns: [Residency, Experience, Q1, Median, Q3]
cell_data = [[residency_labels[i], exp_labels[i]] + data_values_residency[i]
             for i in range(NUM_ROWS)]

# The new column labels now include the Experience Level column
col_labels_final = ['Residency'] + ['Experience'] + col_labels_residency

# --- Create the Matplotlib Figure and Table ---
plt.figure(figsize=(10, 5)) # Increased size to accommodate extra labels
ax_residency = plt.gca()
ax_residency.axis('off')
ax_residency.axis('tight')

# 4. Draw the table onto the axes
table_residency = ax_residency.table(
    cellText=cell_data,
    colLabels=col_labels_final,
    loc='center',
    cellLoc='center'
)

# --- 5. Custom Styling for Hierarchical Grouping and Bolding ---

# Loop through cells to adjust spans and bold the mid-line
US_GROUP_START_ROW_INDEX = NON_US_ROWS + 1

# Define the thickness for the dividing line
```

```
BOLD_WIDTH = 3.0
DEFAULT_WIDTH = 1.0

for i in range(1, NUM_ROWS + 1): # Start at row 1 (the first data row), end at NUM_ROWS
    for j in range(len(col_labels_final)):

        cell = table_residency.get_celld()[(i, j)]

        if cell is not None:

            # 1. Set the uniform linewidth for the entire cell
            # This ensures the base linewidth property exists for drawing.
            cell.set_linewidth(DEFAULT_WIDTH)
            cell.set_edgecolor('black')

            # 2. Check the row index for bold styling
            if i == US_GROUP_START_ROW_INDEX:
                # This is the first row of the 'US' group. We want a bold TOP border.

                # Check for the internal path artists. In some versions, they are in _edge_artists.
                if hasattr(cell, '_edge_artists') and 'top' in cell._edge_artists:
                    # Found the path artist for the top edge. Style it directly.
                    cell._edge_artists['top'].set_linewidth(BOLD_WIDTH)

                # FALLBACK if _edge_artists doesn't exist (i.e., your current version)
                else:
                    # In older Matplotlib, Cell inherits from Patch, which has the lines
                    # stored differently. We must access the four lines in the order they were draw
                    # This is highly speculative but the last place to check.
                    try:
                        # The lines are often returned as a tuple/list of Line2D objects.
                        # The 'top' border is usually the second or third element (0=bottom, 1=left
                        # We try index 2 for the top border.
                        # **This line is the most speculative but often the solution for older vers
                        cell.get_children()[0].set_linewidth(BOLD_WIDTH) # Try the first child arti

                    except IndexError:
                        # If that fails, we give up on per-edge styling and just set the uniform wi
                        # for the whole cell, which is not what you wanted, but it avoids crashing.
                        cell.set_linewidth(BOLD_WIDTH)

            # 3. Defensive styling for the left border (to support merging if you re-enable it)
            if j == 0 and i > 1 and i != US_GROUP_START_ROW_INDEX:
                 # Try to set the left edge to 0.0 linewidth to hide it.
                 if hasattr(cell, '_edge_artists') and 'left' in cell._edge_artists:
                     cell._edge_artists['left'].set_linewidth(0.0)
                 # Fallback - just set the cell's left edge color to the background color (a hack)
                 cell.set_edgecolor('white')

            # Ensure the right, left, and bottom edges for the US_GROUP_START_ROW are default
            # We don't need to explicitly set bottom/left/right to 1.0 because we already set
            # the uniform cell width to 1.0 at the top of the loop.

# Merge (Span) the Residency cells:
# ... (Keep commented out for now as it caused errors)
"""
table_residency.get_celld()[(1, 0)].set_rowspan(NON_US_ROWS)
```

```
for i in range(2, NON_US_ROWS + 1):
    table_residency.get_celld()[(i, 0)].set_visible(False)

# Merge US cells (Rows 5 to 8)
table_residency.get_celld()[(NON_US_ROWS + 1, 0)].set_rowspan(US_ROWS)
for i in range(NON_US_ROWS + 2, NUM_ROWS + 1):
    table_residency.get_celld()[(i, 0)].set_visible(False)
#this caused an error, so commented out
"""

# Style and Title
table_residency.auto_set_font_size(False)
table_residency.set_fontsize(12)
table_residency.scale(0.8, 1.5) # Scale to make room for merged cells

plt.title('Salary Quartile Values by Residency & Experience Level', fontsize=14, pad=0)

# Display the figure (sends it to the Plots pane in Spyder)
plt.show()

#END 14###BAD BOLD, GOOD CELL SPAN################################################



#Section 15: Same data as Section 11: Final Presentation Usable look, modified
#spanning of rows in the Residency column for clean aesthetic (VAMMA)

NUM_ROWS = len(usd_salary_residence_quartile_table) # Should be 8 (4 Non-US, 4 US)
NON_US_ROWS = NUM_ROWS // 2 # Should be 4
US_ROWS = NUM_ROWS // 2 # Should be 4

# --- MODIFIED DATA PREPARATION ---

data_values_residency = [[f'${x:,.0f}' for x in row]
                         for row in usd_salary_residence_quartile_table.values]

# 1. Split the existing row labels to create the two columns:
# Assuming the index is structured as: (Residency, Experience_Level)
# If the index is a flat list of strings like 'Non-US-Entry', you need to split it first.
# We will assume a simple split for now to get the two labels:
exp_labels = []
residency_labels_raw = []

# This structure assumes the index is a Pandas MultiIndex, which is standard after a multi-level gr
if isinstance(usd_salary_residence_quartile_table.index, pd.MultiIndex):
    residency_labels_raw = usd_salary_residence_quartile_table.index.get_level_values(0).tolist()
    exp_labels = usd_salary_residence_quartile_table.index.get_level_values(1).tolist()
else:
    # Fallback if the index is a flat list (less ideal)
    for label in usd_salary_residence_quartile_table.index:
        residency_labels_raw.append(label.split(',')[0].strip('{').strip().replace("'", ""))
        exp_labels.append(label.split(',')[1].strip('}').strip().replace("'", ""))

# 2. Create the Primary Grouping Column (Residency) using a list with repeating labels for the cell
residency_labels = [residency_labels_raw[i] if i % NON_US_ROWS == 0 else '' for i in range(NUM_ROWS

# 3. Combine the experience labels and data values for the final table structure
```

```
# The final table will have 5 columns: [Residency, Experience, Q1, Median, Q3]
cell_data = [[residency_labels[i], exp_labels[i]] + data_values_residency[i]
             for i in range(NUM_ROWS)]

# The new column labels now include the Experience Level column
col_labels_final = ['Residency'] + ['Experience'] + col_labels_residency

# --- Create the Matplotlib Figure and Table ---
plt.figure(figsize=(10, 5)) # Increased size to accommodate extra labels
ax_residency = plt.gca()
ax_residency.axis('off')
ax_residency.axis('tight')

# 4. Draw the table onto the axes
table_residency = ax_residency.table(
    cellText=cell_data,
    colLabels=col_labels_final,
    loc='center',
    cellLoc='center'
)

# --- 5. Custom Styling for Hierarchical Grouping and Bolding ---

# Loop through cells to adjust spans and bold the mid-line
US_GROUP_START_ROW_INDEX = NON_US_ROWS + 1
BOLD_WIDTH = 3.0
DEFAULT_WIDTH = 1.0

# 1. Manually set the edge properties and manage spanning in one pass
for i in range(1, NUM_ROWS + 1):
    for j in range(len(col_labels_final)):
        cell = table_residency.get_celld()[(i, j)]

        if cell is not None:
            # Get the internal dictionary that holds the edge properties
            if hasattr(cell.get_path(), '_edge_linewidths'):
                edge_linewidths = cell.get_path()._edge_linewidths
                # Order is usually: left, bottom, right, top (LBRT) or (BLRT)

                # Reset all edges to default width
                edge_linewidths[:] = [DEFAULT_WIDTH] * len(edge_linewidths)

                # --- Apply Bolding (Top edge of row 5) ---
                if i == US_GROUP_START_ROW_INDEX:
                    # Top edge is usually index 3 (LBRT order)
                    try:
                        edge_linewidths[3] = BOLD_WIDTH
                    except IndexError:
                        pass

                # --- Apply Merging Style (Hide Left Border for spanned cells) ---
                if j == 0 and i > 1 and i != US_GROUP_START_ROW_INDEX:
                    # Hide the left edge (Index 0 for left edge in LBRT order)
                    try:
                        edge_linewidths[0] = 0.0
                    except IndexError:
                        pass
```

```
            # --- Row Spanning Logic ---
            if j == 0: # Only apply spanning to the Residency column
                if i == 1: # First row of Non-US group (start span)
                    cell.rowspan = NON_US_ROWS # Set the span property
                    cell.set_edgecolor('black')
                elif i > 1 and i < US_GROUP_START_ROW_INDEX: # Intermediate Non-US rows (hide)
                    cell.set_visible(False)
                elif i == US_GROUP_START_ROW_INDEX: # First row of US group (start new span)
                    cell.rowspan = US_ROWS # Set the span property
                    cell.set_edgecolor('black')
                elif i > US_GROUP_START_ROW_INDEX: # Intermediate US rows (hide)
                    cell.set_visible(False)


# This is the original Matplotlib way to set spanning. We must try it.
# Merge Non-US cells (Rows 1 to 4)
table_residency.get_celld()[(1, 0)].rowspan = NON_US_ROWS
for i in range(2, NON_US_ROWS + 1):
    table_residency.get_celld()[(i, 0)].set_visible(False)

# Merge US cells (Rows 5 to 8)
table_residency.get_celld()[(NON_US_ROWS + 1, 0)].rowspan = US_ROWS
for i in range(NON_US_ROWS + 2, NUM_ROWS + 1):
    table_residency.get_celld()[(i, 0)].set_visible(False)


# Style and Title
table_residency.auto_set_font_size(False)
table_residency.set_fontsize(12)
table_residency.scale(0.8, 1.5) # Scale to make room for merged cells

plt.title('Salary Quartile Values by Residency & Experience Level',
          fontsize = 16, y = 0.8) #y is the height the title is above the graph

# Display the figure (sends it to the Plots pane in Spyder)
plt.show()
#END 15#########################################################################


"""
#Examine where employee_residence == company_location
plt.figure(figsize=(12, 12))

# Use Seaborn's scatterplot with jitter for clarity
sns.scatterplot(
    data=salary_ranges_df,
    x='employee_residence',
    y='company_location',
    hue='employee_residence',  # Use hue for color distinction (optional)
    palette='viridis',
    s=20, # marker size
    alpha=0.6, # marker transparency
)


# Draw a diagonal line to highlight where the countries match
```

```
max_len = max(len(salary_ranges_df['employee_residence'].unique()), len(salary_ranges_df['company_l
plt.plot(range(max_len), range(max_len), color='red', linestyle='--', linewidth=1, label='Residence

plt.title("Employee Residence vs. Company Location (Jittered Scatter)", fontsize=16)
plt.xlabel("Employee Residence (2-Letter Code)", fontsize=12)
plt.ylabel("Company Location (2-Letter Code)", fontsize=12)
plt.legend(title='Residence Country')
plt.grid(True, linestyle=':', alpha=0.5)
plt.show()
##############################Too much of an eyesore; not very helpful either
"""



#Section 16: Determine how common it is that a DS works in their home country
# Create a Boolean column: True where they match, False where they don't
salary_ranges_df['works_in_home_country'] = (
    salary_ranges_df['employee_residence'] == salary_ranges_df['company_location']
)

print("\n--- Total Match vs. Mismatch Counts ---")
# Use value_counts to get the total number of True (Match) and False (Mismatch) cases
print(salary_ranges_df['works_in_home_country'].value_counts())
print("\n--- Percentage of Match vs. Mismatch ---")
print(salary_ranges_df['works_in_home_country'].value_counts(normalize = True).mul(100).round(2).as
#END 16#########################################################################


"""
#Shows a chart with every pairing of employee residence with company country
print("\n")
print("--- Frequency of ALL Residence/Location Combinations (Crosstab) ---")
# pd.crosstab is excellent for categorical relationship summaries
cross_tab_results = pd.crosstab(
    salary_ranges_df['employee_residence'],
    salary_ranges_df['company_location']
)

# Use set_option to display all columns and rows for the large table
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

print(cross_tab_results)

# Remember to reset the options after viewing
pd.set_option('display.max_columns', 20)
pd.set_option('display.max_rows', 60)
############################## This is a tad excessive
"""



#Section 17: Histogram of salary_in_usd
plt.figure(figsize=(10, 6))
sns.histplot(salary_ranges_df.salary_in_usd, kde = True, bins = 30)
plt.title('Distribution of Salaries in USD')
plt.xlabel('Salary (USD $)')
plt.show()
```

```
#test how gaussian this is
# The Shapiro-Wilk test is good for sample sizes up to 5000
stat, p_value = shapiro(salary_ranges_df.salary_in_usd)

print(f"\nShapiro-Wilk Statistic: {stat:.3f}")
print(f"P-value: {p_value:.5f}")
#essentially 0 p-value - reject null hypotheis (not normal/gaussian distribution)
#END 17#########################################################################



#Section 18: Confidence interval for all salaries with a T-Distribution
confidence_level = 0.95 # For a 95% confidence interval
data = salary_ranges_df.salary_in_usd.values # Use the numpy array data

n = len(data)
mean_salary = np.mean(data)
std_err = np.std(data, ddof=1) / np.sqrt(n) # Standard Error of the Mean

# Calculate the T-score (using degrees of freedom: n-1)
t_score = t.ppf(1 - (1 - confidence_level) / 2, df=n - 1)

# Calculate the Margin of Error (ME)
margin_of_error = t_score * std_err

# Calculate the Confidence Interval (CI)
lower_bound = mean_salary - margin_of_error
upper_bound = mean_salary + margin_of_error

print(f"\n--- 95% Confidence Interval for Mean Salary ---")
print(f"Sample Mean: ${mean_salary:,.2f}")
print(f"Margin of Error (ME): ${margin_of_error:,.2f}")
print(f"Confidence Interval (CI): (${lower_bound:,.2f}, ${upper_bound:,.2f})")
#END 18#########################################################################



#Section 19: Confidence interval for salaries based on experience_level
def calculate_confidence_interval(series, confidence_level=0.95):
    """
    Calculates the mean, standard error, margin of error, and
    confidence interval bounds for a given data series.
    """
    data = series.values
    n = len(data)

    # Check for insufficient data
    if n < 2:
        return pd.Series([np.nan, np.nan, np.nan, np.nan],
                         index=['Mean', 'Margin of Error', 'Lower Bound', 'Upper Bound'])

    # 1. Calculate Core Statistics
    mean_salary = np.mean(data)
    # Calculate Standard Error of the Mean (SEM)
    std_err = np.std(data, ddof=1) / np.sqrt(n)
```

```
    # 2. Calculate T-score (using n-1 degrees of freedom)
    t_score = t.ppf(1 - (1 - confidence_level) / 2, df=n - 1)

    # 3. Calculate Margin of Error (ME) and Bounds
    margin_of_error = t_score * std_err
    lower_bound = mean_salary - margin_of_error
    upper_bound = mean_salary + margin_of_error

    return pd.Series({
        'Mean': mean_salary,
        'Margin of Error': margin_of_error,
        'Lower Bound': lower_bound,
        'Upper Bound': upper_bound
    })


# --- Application ---

# 1. Group the data by 'experience_level'
grouped_salaries = salary_ranges_df.groupby(['experience_level', 'residence_group'], observed=True)

# 2. Apply the custom function to each group
ci_results_df = grouped_salaries.apply(calculate_confidence_interval)

# 3. Print the formatted results
print("\n--- 95% Confidence Intervals by Experience Level ---")
# Use map to format the currency output for clean display
formatted_results = ci_results_df.map(lambda x: f"${x:,.2f}")
print(formatted_results)
#print(ci_results_df)
#END 19#############################################################################




#Section 20: Convert the IPython Output for the Confidence Interval to a matplotlib plot

# --- 1. Robust Data Preparation for Plotting using Unstacking ---

# The ci_results_df index structure is (experience_level, residence_group, metric)
# We need to unstack to move metrics and residence_group into columns.

# First, unstack the last level (the metric: Mean, Lower Bound, etc.)
plot_data_unstacked = ci_results_df.unstack(level=-1)

# Now, unstack the second level (residence_group: US, Non-US)
# This results in a MultiIndex column structure: (Mean, Non-US), (Mean, US), etc.
plot_data_pivot = plot_data_unstacked.unstack(level=-1)


# Flatten the column MultiIndex for easier access (e.g., Mean_Non-US)
plot_data_pivot.columns = [f'{col[0]}_{col[1]}' for col in plot_data_pivot.columns]


# --- 2. Data Extraction with Corrected Column Names ---

# Extract data for plotting
```

```
mean_non_us = plot_data_pivot['Mean_Non-US'].values
ci_non_us_lower = plot_data_pivot['Lower Bound_Non-US'].values
ci_non_us_upper = plot_data_pivot['Upper Bound_Non-US'].values
# Calculate the length of the confidence interval bars (error in both directions)
errors_non_us = [mean_non_us - ci_non_us_lower, ci_non_us_upper - mean_non_us]

mean_us = plot_data_pivot['Mean_US'].values
ci_us_lower = plot_data_pivot['Lower Bound_US'].values
ci_us_upper = plot_data_pivot['Upper Bound_US'].values
errors_us = [mean_us - ci_us_lower, ci_us_upper - mean_us]

# Labels for the Y-axis (Experience Levels)
experience_levels = plot_data_pivot.index.tolist()
# Reverse the order to plot 'Entry' at the bottom and 'Executive' at the top
experience_levels.reverse()
# Reverse the data arrays to match the reversed labels
mean_non_us = np.flip(mean_non_us)
ci_non_us_lower = np.flip(ci_non_us_lower)
ci_non_us_upper = np.flip(ci_non_us_upper)
errors_non_us = [np.flip(errors_non_us[0]), np.flip(errors_non_us[1])]

mean_us = np.flip(mean_us)
ci_us_lower = np.flip(ci_us_lower)
ci_us_upper = np.flip(ci_us_upper)
errors_us = [np.flip(errors_us[0]), np.flip(errors_us[1])]

# --- 3. Matplotlib Plot Setup ---

fig, ax = plt.subplots(figsize=(12, 6)) # Create a figure and an axis object
y_positions = np.arange(len(experience_levels)) # Positions for the bars

# Set the plot title and remove the right/top axis spines
ax.set_title("95% Confidence Intervals for Mean Salary by Experience & Residency", fontsize=16, y=1
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# --- 4. Plotting the Confidence Intervals (Dot-and-Whisker Plot) ---

# Plot Non-US (slightly below the center line)
ax.errorbar(
    mean_non_us,
    y_positions - 0.15,  # Offset downwards
    xerr=errors_non_us,
    fmt='o',  # Format for the mean point (circle)
    capsize=5,
    elinewidth=2, # Thickness of the whisker (CI line)
    markeredgecolor='darkorange',
    markerfacecolor='darkorange',
    color='darkorange',
    markersize=8,
    label='Non-US'
)

# Plot US (slightly above the center line)
ax.errorbar(
    mean_us,
    y_positions + 0.15,  # Offset upwards
```

```
    xerr=errors_us,
    fmt='o',
    capsize=5,
    elinewidth=2,
    markeredgecolor='navy',
    markerfacecolor='navy',
    color='navy',
    markersize=8,
    label='US'
)

# --- 5. Custom Styling and Formatting ---

# Set Y-axis labels and ticks
ax.set_yticks(y_positions)
ax.set_yticklabels(experience_levels, fontsize=12)
ax.set_xlabel("Mean Salary (USD)", fontsize=12)

# Format the X-axis labels to currency
from matplotlib.ticker import FuncFormatter
def currency_formatter(x, pos):
    """Formats the axis tick as currency."""
    return f'${x:,.0f}'
formatter = FuncFormatter(currency_formatter)
ax.xaxis.set_major_formatter(formatter)

# Add a vertical grid line for readability (using a light gray color)
ax.grid(axis='x', linestyle='--', alpha=0.6, color='lightgray')
ax.set_axisbelow(True) # Ensures grid lines are behind the plot elements

# Add a legend
ax.legend(loc='lower right', frameon=False, fontsize=10)

plt.tight_layout()
plt.show()
#END 20#############################################################################



#Section 21: Calculate percentage discount Non-US DS salary is for each experience level
#usd_salary_residence_quartile_data #variable to work with


#print(type(usd_salary_residence_quartile_data))
# print(type(1 / (usd_salary_residence_quartile_data.loc[idx['US', 'MI', 0.50:0.75]].values / usd_s
idx = pd.IndexSlice
non_us_resident_intermediate_q1_q2_salary_array = usd_salary_residence_quartile_data.loc[idx['Non-U
us_resident_intermediate_q1_q2_salary_array = usd_salary_residence_quartile_data.loc[idx['US', 'MI'
#print(non_us_resident_intermediate_q1_q2_salary_array)
#print(us_resident_intermediate_q1_q2_salary_array)

#print(non_us_resident_intermediate_q1_q2_salary_array / us_resident_intermediate_q1_q2_salary_arra

def compare_multiindex_data(series: pd.Series, numerator_level: str, denominator_level: str) -> pd.
    """
    Compares two hierarchically equivalent subsets of a MultiIndex Series
    by dividing the numerator subset by the denominator subset, element-wise.
```

```
    The function assumes the top-most level of the MultiIndex contains the
    numerator and denominator labels.

    Args:
        series: The input Pandas Series with a MultiIndex.
        numerator_level: The label for the numerator (e.g., 'Non-US').
        denominator_level: The label for the denominator (e.g., 'US').

    Returns:
        A new Pandas Series representing the percentage difference
        ((Numerator / Denominator) - 1), formatted as a string percentage.
    """
    # 1. Unstack the top-most level of the MultiIndex (level 0)
    # This creates a DataFrame where 'Non-US' and 'US' are separate columns.
    df_wide = series.unstack(level=0)

    # 2. Check for required columns before calculation
    if numerator_level not in df_wide.columns or denominator_level not in df_wide.columns:
        raise ValueError(
            f"Levels '{numerator_level}' and/or '{denominator_level}' not found in the top index le
        )

    # 3. Perform the vector-wise calculation using Pandas columns
    # This is highly efficient as it avoids Python loops.
    # Calculation: (Non-US / US) - 1
    difference = abs((df_wide[numerator_level] / df_wide[denominator_level]) - 1)

    # 4. Format the result for presentation
    # The resulting Series index is now (Experience, Quartile)

    # Drop the level name for cleaner printing if it was automatically set
    if difference.index.names[0] is None:
         difference.index.names = [None, None] # Clean up unnamed levels

    formatted_output = difference.apply(lambda x: f"{x:.2%}")

    # Rename the resulting series for clarity
    formatted_output.name = f'({numerator_level} / {denominator_level}) - 1'

    return formatted_output

non_us_percentage_savings_table = compare_multiindex_data(usd_salary_residence_quartile_data, 'Non-

non_us_percentage_savings_table.columns = ['Q1 (25th)', 'Median (50th)', 'Q3 (75th)']

#Change the output of the table for experience_level column to be more readable
original_abbreviations = ['EN', 'MI', 'SE', 'EX']
label_map = dict(zip(original_abbreviations, new_experience_labels))
non_us_percentage_savings_table = non_us_percentage_savings_table.rename(index = label_map)


print("\n--- Percentage Savings by Hiring Non-US Data Scientists ---")
print(non_us_percentage_savings_table.unstack())

#print(compare_multiindex_data(usd_salary_residence_quartile_data, 'US', 'Non-US'))
```

```
#END 21#########################################################################


# ================================================================================
# #Section 22: Convert the DataFrame index (Experience Level) and columns to lists
# # for the table visualization
# data_values_world = non_us_percentage_savings_table.values
# #data_values_world = [[f'${x:,.0f}' for x in row]
# #                     for row in non_us_percentage_savings_table.values]#convert to list of curre
#
#
# row_labels_world = non_us_percentage_savings_table.index.tolist()
# #col_labels_world = non_us_percentage_savings_table.columns.tolist()
#
# # --- Create the Matplotlib Figure and Table ---
# plt.figure(figsize = (8, 3)) # Adjust size for the table
# ax_world = plt.gca() # Get the current axes object
#
# # 1. Hide the axes (the plot area itself) so only the table is visible
# ax_world.axis('off')
# ax_world.axis('tight')
#
# # 2. Draw the table onto the axes
# # 'cellText' is the data. 'rowLabels' and 'colLabels' define the headers.
# table_world = ax_world.table(
#     cellText = data_values_world,
#     rowLabels = row_labels_world,
# #    colLabels = col_labels_world,
#     loc = 'center',
#     cellLoc = 'center', # Center the text within cells
#     rowLoc = 'center'
# )
#
# #Style and Title
# table_world.auto_set_font_size(False)
# table_world.set_fontsize(12)
# table_world.scale(0.8, 1.5) # Scale width and height of cells
#
# plt.title('Percentage Savings by Hiring Non-US Data Scientists', fontsize=14, pad=0)
#
# # Display the figure (sends it to the Plots pane in Spyder)
# plt.show()
# #END 22#########################################################################
# ================================================================================




#Section for SCRATCH LEARNING scrap code below
# ================================================================================
# print("\n")
# print(type(usd_salary_residence_quartile_data.index))
# usd_salary_residence_quartile_data.index.levels)
# print(usd_salary_residence_quartile_data.index.names)
# for salary_index_tuple in usd_salary_residence_quartile_data.index:
```

```
#     print(salary_index_tuple)
# ================================================================================
#[['Non-US', 'US'], ['EN', 'MI', 'SE', 'EX'], [0.25, 0.5, 0.75]]

#print(usd_salary_residence_quartile_data['US']['SE'][0.5])

# for exp_level in usd_salary_residence_quartile_data['Non-US']:
#     non_US_discount = (usd_salary_residence_quartile_data['US'] / exp_level)
#     print("Fart", usd_salary_residence_quartile_data['US'])
#     print(non_US_discount)


# print(type(usd_salary_residence_quartile_data))
# print(usd_salary_residence_quartile_data.unstack())

# print(usd_salary_residence_quartile_data)


# print(type(usd_salary_residence_quartile_data.loc[idx['US', 'MI', 0.50:0.75]]))
# print(usd_salary_residence_quartile_data.loc[idx['US', 'MI', 0.50:0.75]].values)
# print(usd_salary_residence_quartile_data.loc[idx['Non-US', 'MI', 0.50:0.75]])




# Example MultiIndex Series
#index = pd.MultiIndex.from_product([['A', 'B'], ['one', 'two'], ['ghost', 'ship']], names=['level1
#my_series = pd.Series([10, 20, 30, 40, 50, 60, 70, 80], index=index)

# Select a specific value from two different branches
# ================================================================================
# value = my_series.loc[('A', 'two')]
# print(value)
# print(my_series.unstack())
# print(my_series)
# ================================================================================
#END SCRAP CODE for Learnign#############################################################
```