🧠 Overview of This Notebook

Welcome! This notebook walks through the key steps involved in our property tax data analysis. Here's what you'll see:

- **Loading and exploring** the raw property tax data
- **Classifying** property types and subcategories (e.g., Residential, Commercial, Apartment, etc.)
- **Engineering new data features**, such as changes in assessed value and tax amounts
- **Merging multiple data sources** using property IDs (PINs)
- **Preparing the dataset** for visualization or deeper statistical analysis

The goal is to create a structured dataset that can be used to uncover patterns, highlight anomalies, and support decision-making.

📦 Import Required Libraries

In this section, we are importing two essential Python libraries that help us work with data:

- **Pandas** (`pd`): Useful for working with tables and structured data (like spreadsheets)
- **NumPy** (`np`): Provides tools for numerical operations and working with arrays

```
import pandas as pd
import numpy as np
```

> 💡 **First time using Pandas or NumPy?**
> You'll need to install them before importing. Just run the following in a new code cell:

```
!pip install pandas
!pip install numpy
```

In [3]:
```
# Import necessary libraries
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

🗂️ Load Property Tax Data Files

In this section, we are loading three datasets that will be used in our analysis:

- `current_property_tax_2023.csv` : This contains the most recent year's property tax information.
- `previous_property_tax_2022.csv` : This file holds the prior year's data, used for comparison.
- `tax_code.csv` : A reference file to help decode tax classifications and codes.

Each file is loaded into a separate Pandas DataFrame, and we display the top few rows to get a quick sense of the structure.

> 📖 **Make sure the CSV files are either:**
> ✔️ Located in your **Python working directory**, or
> ✔️ Uploaded to your **cloud environment** (e.g., Google Colab)

In [5]:
```
# Load the current year property tax data
current_tax_df = pd.read_csv('current_property_tax_2023.csv')
print("2023 Property Tax Data Sample:")
display(current_tax_df.head())

# Load the previous year property tax data
previous_tax_df = pd.read_csv('previous_property_tax_2022.csv')
print("2022 Property Tax Data Sample:")
display(previous_tax_df.head())

# Load the current tax code information
current_tax_code_df = pd.read_csv('tax_code.csv')
print("Current Tax Code Data Sample:")
display(current_tax_code_df.head())

# Load the previous tax code information
previous_tax_code_df = pd.read_csv('prev_tax_code.csv')
print("Tax Code Data Sample:")
display(previous_tax_code_df.head())
```

```
2023 Property Tax Data Sample:
```

	year	pin	class	tax_code_num	tax_bill_total	av_mailed	av_certified	av_boa
0	2023	23011000050000	663	30022	95622.20	535442	535442	2904
1	2023	23011000070000	663	30022	227815.83	906455	906455	6919
2	2023	23011000100000	670	30022	3912.95	21283	21283	118
3	2023	23011010030000	211	30017	17662.14	57999	57999	579
4	2023	23011010040000	211	30017	13094.61	43000	43000	430

2022 Property Tax Data Sample:

	year	pin	class	tax_code_num	tax_bill_total	av_mailed	av_certified	av_boa
0	2022	23011000050000	663	30022	104036.14	257909	257909	2579
1	2022	23011000070000	663	30022	256644.34	636230	636230	6362
2	2022	23011000100000	670	30022	5030.25	12470	12470	124
3	2022	23011010030000	211	30017	18724.19	49188	49188	491
4	2022	23011010040000	211	30017	13505.65	35479	35479	354

Current Tax Code Data Sample:

	year	agency_num	agency_rate	tax_code_num	tax_code_rate
0	2023	10010000	0.386	30001	9.085
1	2023	10010000	0.386	30002	8.879
2	2023	10010000	0.386	30006	9.934
3	2023	10010000	0.386	30007	10.930
4	2023	10010000	0.386	30008	10.576

Tax Code Data Sample:

	year	agency_num	agency_rate	tax_code_num	tax_code_rate
0	2022	10010000	0.431	30001	11.775
1	2022	10010000	0.431	30002	11.377
2	2022	10010000	0.431	30006	12.929
3	2022	10010000	0.431	30007	13.842
4	2022	10010000	0.431	30008	13.394

📊 Filtering Data for Commercial-to-Residential Analysis

This section focuses on narrowing down the dataset to support analysis of shifts from **commercial to residential** classifications. We apply a series of filters to remove records that could obscure meaningful trends:

- **Exempt properties** (`class = 0`) are removed because they are not taxed and do not participate in market-driven reclassification.
- **Senior freeze properties** (`exe_freeze > 0`) are excluded because their assessments are artificially frozen and won't reflect market shifts.
- We **compare class codes** to the previous year and exclude records where the class changed. This helps us ensure we're comparing similar property types and not capturing unrelated changes (e.g., vacant land becoming residential, or commercial converting to industrial). Our goal is to focus on properties that stayed within the same category and were **reassessed**, not reclassified.
- We **exclude a known farm property** by PIN because it is a unique outlier (one property is classified as a farm).

These steps help us isolate properties where a genuine commercial-to-residential transition might be taking place.

In [7]:
```
# Display starting count
print(f"Starting count: {len(current_tax_df):,} records")

# Filter out exempt class (class = 0)
filtered_df = current_tax_df[current_tax_df['class'] != 0]
print(f"After filtering exempt {len(filtered_df):,} records")

# Merge with previous year to compare class
compare_class_df = filtered_df.merge(previous_tax_df[['pin', 'class']], on='pin', h

# Exclude records where class changed from previous year
exclude_class_change_df = compare_class_df[compare_class_df['class'] == compare_cla
print(f"After excluding properties where class changed: {len(exclude_class_change_d

# Exclude farm
exclude_farm_pin_df = exclude_class_change_df[exclude_class_change_df['pin'] != 231
print(f"After excluding farm properties : {len(exclude_farm_pin_df):,} records")

# Select only the PIN and class column to start the new DataFrame
df = exclude_farm_pin_df[['pin']].copy()

# Display the resulting DataFrame structure
df.head()
```

```
Starting count: 10,107 records
After filtering exempt 9,847 records
After excluding properties where class changed: 9,530 records
After excluding farm properties : 9,529 records
```

Out[7]:

	pin
0	23011000050000
1	23011000070000
2	23011000100000
3	23011010030000
4	23011010040000

🔹 Classifying Property Types and Subtypes

This section defines two functions to help us categorize properties into broad types and more specific residential subtypes.

1. `classify_property` :
   - Groups parcels into major categories like **Exempt**, **Residential**, **Commercial**, and **Land** based on the `class` code.
   - Treats codes starting with 3–7 as **Commercial** (including industrial), and 241 as **Land**.
   - Code 397 is classified as **Mobile Homes**.
2. `classify_residential_subtype` :
   - Further refines the **Residential** category into **House**, **Condo**, **Townhouse**, **Apartment**, or **Other**, based on specific class codes.
   - Returns `None` for any properties that are not residential.

These functions will help in analyzing trends within property types and enable more targeted comparisons.

In [9]:

```python
def classify_property(row):
    class_code = str(row['class'])
    if class_code == '0':
        return 'Exempt'
    elif class_code == '241' or class_code.startswith('1'):
        return 'Land'
    elif class_code.startswith('2'):
        return 'Residential'
    elif class_code == '397':
        return 'Mobile Homes'
    elif class_code.startswith(('3', '4', '5','6','7')):
        return 'Commercial'  # includes industrial
    else:
        return 'Other'
```

```
def classify_residential_subtype(row):
    if row['property_type'] != 'Residential':
        return None
    class_code = str(row['class'])
    if class_code in ['200', '201', '202', '203', '204', '205', '206', '207', '208'
        return 'House'
    elif class_code in ['278', '290', '299']:
        return 'Condo'
    elif class_code in ['295']:
        return 'Townhouse'
    elif class_code in ['234']:
        return 'Apartment'
    else:
        return 'Other'

def senior_freeze_flag(row):
    senior_freeze = row['exe_freeze']
    if senior_freeze > 0:
        return True
    else:
        return False
```

🔍 Add and Summarize Property Type Classifications

In this step, we add the **property type** and **residential sub-type** classifications to our filtered dataset using the previously defined functions:

- We start by creating a temporary DataFrame with `pin` and `class` from `current_tax_df`.
- We then apply `classify_property` and `classify_residential_subtype` to generate two new fields: `property_type` and `property_sub_type`.
- These classification features are merged back into our working DataFrame (`df`) using the `pin` column.

Finally, we display the updated structure and print summary counts to understand how many records fall into each category. We also show a sample of PINs with an `Other` classification so we can further investigate unknown or edge case codes.

In [11]:
```
# Create a temporary DataFrame with classification info from current_tax_df
temp_classification_df = current_tax_df[['pin', 'class','exe_freeze']].copy()
temp_classification_df['property_type'] = temp_classification_df.apply(classify_pro
temp_classification_df['property_sub_type'] = temp_classification_df.apply(classify
temp_classification_df['senior_freeze_flag'] = temp_classification_df.apply(senior_

# Merge the new features into df based on PIN
df = df.merge(temp_classification_df[['pin', 'property_type', 'property_sub_type',

# Display the updated DataFrame structure
df.head()
```

```
# Count summary by property_type and property_sub_type
print("Property Type Counts:")
print(df['property_type'].value_counts(dropna=False))
```

```
Property Type Counts:
property_type
Residential     8680
Commercial       568
Land             264
Mobile Homes      17
Name: count, dtype: int64
```

💰 Add Assessed Value to Working Dataset

In this step, we merge the **current assessed value** into our working DataFrame (`df`) using the `pin` as the join key:

- We select the `pin` and `av_clerk` fields from `current_tax_df` .
- The `av_clerk` column is renamed to `assessed_value` for clarity.
- A left join ensures we retain all records in our filtered dataset while bringing in matching assessed values.

This allows us to reference the official assessed value assigned by the clerk for each property in subsequent analysis steps.

In [13]:
```
# Merge assessed value from current_tax_df into df based on pin
df = df.merge(current_tax_df[['pin', 'av_clerk']].rename(columns={'av_clerk': 'curr

df = df.merge(previous_tax_df[['pin', 'av_clerk']].rename(columns={'av_clerk': 'pre

# Optional: preview results
display(df.head())
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

💰 Add Total Property Tax data to Working Dataset

In this step, we merge the **tax_bill_total** into our working DataFrame (`df`) using the `pin` as the join key:

- We select the `pin` and `tax_bill_total` fields from `current_tax_df` .
- The `tax_bill_total` column is renamed to `total_tax_bill` for clarity.
- A left join ensures we retain all records in our filtered dataset while bringing in matching assessed values.

This allows us to reference the official assessed value assigned by the clerk for each property in subsequent analysis steps.

In [15]:
```
# Merge assessed value from current_tax_df into df based on pin
df = df.merge(current_tax_df[['pin', 'tax_bill_total']].rename(columns={'tax_bill_t

df = df.merge(previous_tax_df[['pin', 'tax_bill_total']].rename(columns={'tax_bill_

# Optional: preview results
display(df.head())
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

💸 Calculate North Palos District (NPD) Property Tax Amounts

This section calculates the **portion of property tax attributed to the North Palos District (NPD)**, using agency number `10010000` .

The process follows five steps:

1. **Filter and extract NPD agency rates**
   - Select rows from `tax_code_df` where `agency_num` is `10010000`
   - Keep distinct combinations of `tax_code_num` and `agency_rate`
2. **Join NPD rates to the main property tax dataset** using `tax_code_num`
3. **Calculate NPD property tax amount**

- Multiply each property's `tax_bill_total` by its matched `agency_rate`

4. **Select final fields**

- Keep just the `pin` and newly calculated `npd_property_tax_amount`

5. **Merge with working dataset (`df`)**

- Join by `pin` , round values to whole dollars, and display results

This gives us a clean field that isolates the estimated tax amount paid to the North Palos District for each property.

In [17]:
```
# Step 1: Filter tax_code_df for agency_num 10010000 and get distinct tax_code_num
npd_rate_df = current_tax_code_df[current_tax_code_df['agency_num'] == 10010000]
npd_rate_df = npd_rate_df[['tax_code_num', 'agency_rate']].drop_duplicates()

# Step 2: Join with current_tax_df on tax_code_num
full_tax_df = current_tax_df.merge(npd_rate_df, on='tax_code_num', how='left')

# Step 3: Multiply tax_bill_total by agency rate to get North Palos amount
full_tax_df['npd_current_property_tax_amount'] = full_tax_df['tax_bill_total'] * fu

# Step 4: Select final result with pin and npd_property_tax_amount
npd_result_df = full_tax_df[['pin', 'npd_current_property_tax_amount']]

# Step 5: Merge into filtered df
df = df.merge(npd_result_df, on='pin', how='left')

# Round npd_property_tax_amount to the nearest whole number
if 'npd_current_property_tax_amount' in df.columns:
    df['npd_current_property_tax_amount'] = df['npd_current_property_tax_amount'].r

display(df.head())
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

💸 Calculate North Palos District (NPD) Property Tax Amounts

This section calculates the **portion of property tax attributed to the North Palos District (NPD)**, using agency number `10010000`.

The process follows five steps:

1. **Filter and extract NPD agency rates**
   - Select rows from `tax_code_df` where `agency_num` is `10010000`
   - Keep distinct combinations of `tax_code_num` and `agency_rate`
2. **Join NPD rates to the main property tax dataset** using `tax_code_num`
3. **Calculate NPD property tax amount**
   - Multiply each property's `tax_bill_total` by its matched `agency_rate`
4. **Select final fields**
   - Keep just the `pin` and newly calculated `npd_property_tax_amount`
5. **Merge with working dataset (`df`)**
   - Join by `pin`, round values to whole dollars, and display results

This gives us a clean field that isolates the estimated tax amount paid to the North Palos District for each property.

In [19]:
```
# Step 1: Filter tax_code_df for agency_num 10010000 and get distinct tax_code_num
npd_rate_df = previous_tax_code_df[previous_tax_code_df['agency_num'] == 10010000]
npd_rate_df = npd_rate_df[['tax_code_num', 'agency_rate']].drop_duplicates()

# Step 2: Join with current_tax_df on tax_code_num
prev_full_tax_df = previous_tax_df.merge(npd_rate_df, on='tax_code_num', how='left'

# Step 3: Multiply tax_bill_total by agency rate to get North Palos amount
prev_full_tax_df['npd_previous_property_tax_amount'] = prev_full_tax_df['tax_bill_t

# Step 4: Select final result with pin and npd_property_tax_amount
npd_result_df = prev_full_tax_df[['pin', 'npd_previous_property_tax_amount']]

# Step 5: Merge into filtered df
df = df.merge(npd_result_df, on='pin', how='left')

# Round npd_property_tax_amount to the nearest whole number
if 'npd_previous_property_tax_amount' in df.columns:
    df['npd_previous_property_tax_amount'] = df['npd_previous_property_tax_amount']

display(df.head())
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

🔢 Calculate and Add Appeal Granted Amount

In this step, we calculate the **amount of assessed value reduced through an appeal**, then merge that information into our working dataset.

---

1. **Create a subset DataFrame**
   - Select `pin`, `av_mailed`, and `av_clerk` from `current_tax_df`
2. **Calculate `appeal_granted_amount`**
   - Subtract `av_clerk` from `av_mailed` to get the reduction granted via appeal
3. **Merge into the working DataFrame (`df`)**
   - Join by `pin` to add the `appeal_granted_amount` field

This feature can help analyze the impact of appeals on assessed values across properties.

In [21]:
```
# Step 1: Create a DataFrame with PIN, av_mailed, and av_clerk
appeal_df = current_tax_df[['pin', 'av_mailed', 'av_clerk']].copy()

# Step 2: Calculate appeal_granted_amount
appeal_df['appeal_granted_amount'] = appeal_df['av_mailed'] - appeal_df['av_clerk']

# Step 3: Merge appeal_granted_amount into df
df = df.merge(appeal_df[['pin', 'appeal_granted_amount']], on='pin', how='left')

# Optional: Preview
print(df[['pin', 'appeal_granted_amount']].head())

display(df.head())
```

```
              pin  appeal_granted_amount
0  23011000050000                 245026
1  23011000070000                 214551
2  23011000100000                   9399
3  23011010030000                      0
4  23011010040000                      0
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

📊 Add Appeal Reduction Percentage

In this step, we calculate both the **appeal granted amount** and the **percentage reduction in assessed value** due to appeals. We also ensure that calculations are safely handled even when data is missing or zero.

---

1. **Create a subset DataFrame**
   - Select `pin`, `av_mailed`, and `av_clerk` from `current_tax_df`
2. **Safely calculate `appeal_granted_amount`**
   - Subtract `av_clerk` from `av_mailed` only when both values are present
3. **Safely calculate `appeal_reduction_percentage`**
   - Divide the granted amount by `av_mailed` (as a percent), only when `av_mailed` is not null and not zero
   - Round to two decimal places
4. **Merge results into the working DataFrame (`df`)**
   - Drop existing appeal columns if already present
   - Merge new calculated fields using `pin` as the join key

These values can help quantify how much property owners successfully reduced their assessments through the appeal process.

In [23]:
```
# Step 1: Create a DataFrame with PIN, av_mailed, and av_clerk
appeal_df = current_tax_df[['pin', 'av_mailed', 'av_clerk']].copy()
```

```python
# Step 2: Safely calculate appeal_granted_amount
appeal_df['appeal_granted_amount'] = np.where(
    (appeal_df['av_mailed'].notnull()) & (appeal_df['av_clerk'].notnull()),
    appeal_df['av_mailed'] - appeal_df['av_clerk'],
    np.nan
)

# Step 3: Safely calculate appeal_reduction_percentage
appeal_df['appeal_reduction_percentage'] = np.where(
    (appeal_df['av_mailed'].notnull()) & (appeal_df['av_mailed'] != 0),
    (appeal_df['appeal_granted_amount'] / appeal_df['av_mailed']) * 100,
    np.nan
)

# Round to two decimals
appeal_df['appeal_reduction_percentage'] = appeal_df['appeal_reduction_percentage']

# Step 4: Merge result into df
if 'appeal_granted_amount' in df.columns:
    df = df.drop(columns=['appeal_granted_amount'])
if 'appeal_reduction_percentage' in df.columns:
    df = df.drop(columns=['appeal_reduction_percentage'])

df = df.merge(
    appeal_df[['pin', 'appeal_granted_amount', 'appeal_reduction_percentage']],
    on='pin',
    how='left'
)

# Preview
print(df[['pin', 'appeal_granted_amount', 'appeal_reduction_percentage']].head())

display(df.head())
```

```
             pin  appeal_granted_amount  appeal_reduction_percentage
0  23011000050000               245026.0                        45.76
1  23011000070000               214551.0                        23.67
2  23011000100000                 9399.0                        44.16
3  23011010030000                    0.0                         0.00
4  23011010040000                    0.0                         0.00
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

📊 Compare North Palos District (NPD) Property Tax Year-over-Year

This section calculates the **change in property tax amounts paid to the North Palos District (NPD)** from the previous year to the current year.

---

1. **Filter for NPD agency rates**
   - Select rows from `tax_code_df` where `agency_num` is `10010000`, keeping distinct `tax_code_num` and `agency_rate`
2. **Calculate current year NPD tax amount**
   - Join `npd_rate_df` with `current_tax_df` and compute `tax_bill_total * agency_rate`
3. **Calculate previous year NPD tax amount**
   - Join `npd_rate_df` with `previous_tax_df` and compute `tax_bill_total * agency_rate`
4. **Merge current and previous year amounts**
   - Join on `pin` to align both years in one DataFrame
5. **Round to whole numbers**
   - Round current and previous NPD tax amounts for easier comparison
6. **Calculate change in dollar amount and percent**
   - Compute the absolute change and percent difference from previous year

Finally, we merge the change metrics into our main `df` for analysis and display a preview.

In [25]:
```
# Amount change = current - previous
df["npd_property_tax_change_amount"] = (
    df["npd_current_property_tax_amount"] - df["npd_previous_property_tax_amount"]
)

# 2) Percent change: (change / previous) * 100, guard against zero/NaN previous
df["npd_property_tax_change_pct"] = np.where(
    df["npd_previous_property_tax_amount"].notna() & (df["npd_previous_property_tax
    (df["npd_property_tax_change_amount"] / df["npd_previous_property_tax_amount"])
    np.nan
).round(2)


display(df.head())
```

	pin	property_type	property_sub_type	senior_freeze_flag	current_assessed_val
0	23011000050000	Commercial	None	False	2904
1	23011000070000	Commercial	None	False	6919
2	23011000100000	Commercial	None	False	118
3	23011010030000	Residential	House	False	579
4	23011010040000	Residential	House	False	430

📊 Calculate Change in Assessed Value (Year-over-Year)

This section calculates the **year-over-year change in assessed value** for each property using `av_clerk` from both the current and previous tax years.

---

1. **Load current and previous tax data**
   - Assumes `current_tax_df` and `previous_tax_df` are already loaded
2. **Select relevant fields**
   - Extract `pin` and `av_clerk` from both datasets
   - Merge on `pin` to align current and previous values
3. **Calculate change and percent change**
   - `assessed_change_amount` : difference between current and previous `av_clerk`
   - `assessed_change_percentage` : percent change from previous year, rounded to 2 decimal places
4. **Merge into main dataset (`df`)**
   - Join results back into the working DataFrame using `pin`

These new fields can help identify significant shifts in property assessments that may impact tax bills or indicate reassessments.

In [27]:
```
# If your dataframes accidentally have a comma in this column name, normalize it:
fix_names = {'exe,_longtime_homeowner': 'exe_longtime_homeowner'}
current_tax_df = current_tax_df.rename(columns=fix_names)
previous_tax_df = previous_tax_df.rename(columns=fix_names)

# Columns to evaluate
exemption_cols = [
    'exe_homeowner',
    'exe_senior',
```

```
    'exe_freeze',
    'exe_longtime_homeowner',
    'exe_disabled',
    'exe_vet_returning',
    'exe_vet_dis_lt50',
    'exe_vet_dis_50_69',
    'exe_vet_dis_ge70',
    'exe_abate'
]

# Use only columns present in BOTH frames
use_cols = [c for c in exemption_cols
            if c in current_tax_df.columns and c in previous_tax_df.columns]
if not use_cols:
    raise KeyError("None of the expected exemption columns were found in both dataf

# Convert to numeric, treat NaN as 0, count non-zero across the selected columns
cur_counts = (current_tax_df[use_cols]
              .apply(pd.to_numeric, errors='coerce')
              .fillna(0)
              .ne(0)
              .sum(axis=1))

prev_counts = (previous_tax_df[use_cols]
               .apply(pd.to_numeric, errors='coerce')
               .fillna(0)
               .ne(0)
               .sum(axis=1))

# Join key; adjust if your key is different
join_key = 'pin'
if not ({join_key} <= set(current_tax_df.columns)
        and {join_key} <= set(previous_tax_df.columns)
        and {join_key} <= set(df.columns)):
    raise KeyError(f"Join key '{join_key}' must exist in current_tax_df, previous_t

# Build a comparison table per PIN
cur_tbl = current_tax_df[[join_key]].copy()
cur_tbl['_cur_exempt_cnt'] = cur_counts

prev_tbl = previous_tax_df[[join_key]].copy()
prev_tbl['_prev_exempt_cnt'] = prev_counts

cmp = cur_tbl.merge(prev_tbl, on=join_key, how='outer')

# Map comparison (per your spec):
# previous < current  -> "Less"
# previous > current  -> "More"
# previous == current -> "Same"
cmp['Change_in_Exemptions'] = np.select(
    [
        cmp['_prev_exempt_cnt'] < cmp['_cur_exempt_cnt'],
        cmp['_prev_exempt_cnt'] > cmp['_cur_exempt_cnt'],
    ],
    ['More', 'Less'],
    default='Same'
```

```
)

# Add the new column to your existing df
df = df.merge(cmp[[join_key, 'Change_in_Exemptions']], on=join_key, how='left')
```

In [28]:
```
# Coerce to numeric just in case
v   = pd.to_numeric(df["npd_property_tax_change_pct"], errors="coerce")
amt = pd.to_numeric(df["npd_property_tax_change_amount"], errors="coerce")
sf  = df.get("senior_freeze_flag", False)
sf  = pd.Series(sf, index=df.index).fillna(False).astype(bool)

# Base buckets (your ordering)
cats = np.select(
    [v < -80, v < 0, v > 200, v > 0],
    ["ExcludeA", "Reduction", "ExcludeB", "Increase"],
    default="Exclude"
)
df["Category"] = cats

# --- Large Increase override (only rows currently labeled "Increase") ---
mask_large = df["Category"].eq("Increase") & v.gt(5) & amt.gt(300)
df.loc[mask_large, "Category"] = "Large Increase"

# --- Senior Freeze overrides everything, but do NOT reset to 'cats' ---
df.loc[sf, "Category"] = "Senior Freeze"

# (Optional) quick diagnostics
print("Large Increase rows:", int(mask_large.sum()))
print(df["Category"].value_counts(dropna=False))
```

```
Large Increase rows: 4315
Category
Large Increase    4315
Increase          2197
Reduction         2016
Senior Freeze      806
ExcludeB           105
Exclude             78
ExcludeA            12
Name: count, dtype: int64
```

📁 Export Final Dataset to CSV

In this step, we export the cleaned and enriched DataFrame to a CSV file so it can be shared, backed up, or used for further analysis outside the notebook.

- The file is saved as `npd_date_features.csv` in the current working directory.
- We exclude the index column for a cleaner output.

In [30]:
```
# Save the working DataFrame to CSV
df.to_csv('npd_analysis_0913.csv', index=False)
```

```
print(f"Data Saved ")
```

```
Data Saved
```

In [31]:

```
# Define desired ordering
cat_order  = ["Senior Freeze", "Reduction", "Increase", "Large Increase"]
type_order = ["Residential", "Commercial", "Mobile Homes", "Land"]

# Keep just Residential/Commercial rows with a non-null Category
sub = df.loc[
    df["property_type"].isin(type_order) & df["Category"].notna(),
    ["property_type", "Category"]
].copy()

# Normalize labels (safe if they're already clean)
sub["property_type"] = sub["property_type"].astype(str).str.strip().str.title()
sub["Category"] = pd.Categorical(
    sub["Category"].astype(str).str.strip().str.title(),
    categories=cat_order,
    ordered=True
)

# Raw counts by property type x category
counts_by_prop = (
    sub.groupby(["property_type", "Category"])
       .size()
       .unstack("Category")
       .reindex(index=type_order, columns=cat_order)
       .fillna(0)
       .astype(int)
)

# Percent share within each property type
denom = counts_by_prop.sum(axis=1).replace(0, np.nan)
percent_by_prop = (counts_by_prop.div(denom, axis=0) * 100).round(1).fillna(0)

# Display
print("Share of records by Category within each property_type (percent):")
print(percent_by_prop.to_string())

print("\nRaw counts by Category within each property_type:")
print(counts_by_prop.to_string())
```

```
Share of records by Category within each property_type (percent):
Category       Senior Freeze  Reduction  Increase  Large Increase
property_type
Residential              9.5       15.8      25.2            49.5
Commercial               0.0       77.9       7.8            14.3
Mobile Homes             0.0       70.6       0.0            29.4
Land                     0.0       92.6       3.5             3.9

Raw counts by Category within each property_type:
Category       Senior Freeze  Reduction  Increase  Large Increase
property_type
Residential              806       1350      2145            4220
Commercial                 0        441        44              81
Mobile Homes               0         12         0               5
Land                       0        213         8               9
```

In [32]:

```python
import difflib

# --- Config (fix typo + keep consistent casing) ---
cat_order       = ["Senior Freeze", "Reduction", "Increase", "Large Increase"]
sub_type_order  = ["House", "Condo", "Apartment", "Townhouse"]  # fixed "Aparment"

# Normalize desired orders to Title case (same as we apply to the data)
order_norm    = [s.strip().title() for s in sub_type_order]
cat_order_norm= [s.strip().title() for s in cat_order]

# --- Quick column sanity check ---
required_cols = ["property_sub_type", "Category"]
missing = [c for c in required_cols if c not in df.columns]
if missing:
    raise KeyError(f"Missing columns in df: {missing}. Got: {list(df.columns)}")

# --- Filter rows & normalize labels ---
sub = df.loc[
    df["property_sub_type"].notna() & df["Category"].notna(),
    ["property_sub_type", "Category"]
].copy()

sub["property_sub_type_norm"] = sub["property_sub_type"].astype(str).str.strip().st
sub["Category_norm"]          = sub["Category"].astype(str).str.strip().str.title()

# --- Debug: coverage vs. desired orders ---
present_types = sorted(sub["property_sub_type_norm"].unique())
present_cats  = sorted(sub["Category_norm"].unique())

print("\nDesired sub_type_order (normalized):", order_norm)
print("Types present in data (normalized, first 20):", present_types[:20])
missing_types = [t for t in order_norm if t not in present_types]
extra_types   = [t for t in present_types if t not in order_norm]
print("Types missing from data:", missing_types or "None")
if extra_types:
    print("Extra types in data not in desired order:", extra_types)
    for e in extra_types[:10]:
        print(f"  • '{e}' close to {difflib.get_close_matches(e, order_norm, n=1)}"

print("\nDesired categories (normalized):", cat_order_norm)
```

```
print("Categories present in data:", present_cats)
cats_missing = [c for c in cat_order_norm if c not in present_cats]
print("Categories missing from data:", cats_missing or "None")

# Keep only desired property sub-types (after normalization)
sub = sub[sub["property_sub_type_norm"].isin(order_norm)]

# Set categoricals for stable ordering
sub["property_sub_type_norm"] = pd.Categorical(sub["property_sub_type_norm"], categ
sub["Category_norm"]          = pd.Categorical(sub["Category_norm"], categories=cat

# --- Build tables ---
counts_by_prop = (
    sub.groupby(["property_sub_type_norm", "Category_norm"])
       .size()
       .unstack("Category_norm")
       .reindex(index=order_norm, columns=cat_order_norm)
       .fillna(0)
       .astype(int)
       .rename_axis(index="property_sub_type", columns="Category")
)

print("\nCounts table shape:", counts_by_prop.shape)
print("Total counted rows:", int(counts_by_prop.values.sum()))
print("\nRaw counts by Category within each property_sub_type:")
print(counts_by_prop.to_string())

# Percent share within each property type
denom = counts_by_prop.sum(axis=1).replace(0, np.nan)
percent_by_prop = (counts_by_prop.div(denom, axis=0) * 100).round(1).fillna(0)

print("\nShare of records by Category within each property_sub_type (percent):")
print(percent_by_prop.to_string())

# If you want these available under your original variable names:
# counts_by_prop  -> counts_by_prop
# percent_by_prop -> percent_by_prop
```

```
Desired sub_type_order (normalized): ['House', 'Condo', 'Apartment', 'Townhouse']
Types present in data (normalized, first 20): ['Apartment', 'Condo', 'House', 'Townh
ouse']
Types missing from data: None

Desired categories (normalized): ['Senior Freeze', 'Reduction', 'Increase', 'Large I
ncrease']
Categories present in data: ['Exclude', 'Excludea', 'Excludeb', 'Increase', 'Large I
ncrease', 'Reduction', 'Senior Freeze']
Categories missing from data: None

Counts table shape: (4, 4)
Total counted rows: 8521

Raw counts by Category within each property_sub_type:
Category           Senior Freeze  Reduction  Increase  Large Increase
property_sub_type
House                        288        667       747            1241
Condo                        252        484       871            1537
Apartment                    245        167       464            1293
Townhouse                     21         32        63             149

Share of records by Category within each property_sub_type (percent):
Category           Senior Freeze  Reduction  Increase  Large Increase
property_sub_type
House                        9.8       22.7      25.4            42.2
Condo                        8.0       15.4      27.7            48.9
Apartment                   11.3        7.7      21.4            59.6
Townhouse                    7.9       12.1      23.8            56.2
```

In [33]:

```python
# Keep only Residential & Commercial rows
type_order = ["Residential", "Commercial", "Mobile Homes", "Land"]
df["property_type"] = df["property_type"].astype(str).str.strip().str.title()
sub = df[df["property_type"].isin(type_order)].copy()

# Success flag: appeal granted amount > 0
sub["appeal_success"] = sub["appeal_granted_amount"].fillna(0) > 0

# Clean/normalize appeal_reduction_percentage to numeric (handles "7.5%" strings)
def _to_float_pct(x):
    if pd.isna(x):
        return np.nan
    try:
        return float(str(x).replace("%", "").strip())
    except Exception:
        return np.nan

sub["appeal_reduction_pct"] = sub["appeal_reduction_percentage"].apply(_to_float_pc

# Grouped metrics
grp = sub.groupby("property_type", sort=False)

# 1) Percent of properties with successful appeals
percent_success = (grp["appeal_success"].mean() * 100).round(1)

# 2) Average reduction % among successful appeals only
```

```
avg_reduction_success = grp.apply(
    lambda g: g.loc[g["appeal_success"], "appeal_reduction_pct"].mean()
).round(2)

# Combine into a tidy summary table
summary = pd.concat(
    [
        percent_success.rename("Percent Successful Appeals"),
        avg_reduction_success.rename("Average Reduction % (among successful)"),
    ],
    axis=1,
).reindex(type_order)

# Display on screen
print("Appeals Summary (Residential vs. Commercial)")
print(summary.to_string())
```

```
Appeals Summary (Residential vs. Commercial)
               Percent Successful Appeals  Average Reduction % (among successful)
property_type
Residential                          24.7                                    8.98
Commercial                           67.6                                   20.05
Mobile Homes                        100.0                                    6.29
Land                                  2.3                                   39.20
```

```
C:\Users\debbi\AppData\Local\Temp\ipykernel_24368\481739438.py:27: DeprecationWarnin
g: DataFrameGroupBy.apply operated on the grouping columns. This behavior is depreca
ted, and in a future version of pandas the grouping columns will be excluded from th
e operation. Either pass `include_groups=False` to exclude the groupings or explicit
ly select the grouping columns after groupby to silence this warning.
  avg_reduction_success = grp.apply(
```

In [34]:

```
# Keep consistent labels and focus on Residential/Commercial
type_order = ["Residential", "Commercial", "Mobile Homes","Land"]
df["property_type"] = df["property_type"].astype(str).str.strip().str.title()
sub = df[df["property_type"].isin(type_order)].copy()

# Helper to coerce amounts like "$1,234.56" -> 1234.56
def _to_amount(series):
    return pd.to_numeric(
        series.astype(str).str.replace(r"[^0-9.\-]", "", regex=True),
        errors="coerce"
    ).fillna(0.0)

# Clean numeric versions of the amounts
sub["npd_current_amt"]  = _to_amount(sub["npd_current_property_tax_amount"])
sub["npd_previous_amt"] = _to_amount(sub["npd_previous_property_tax_amount"])

# Group and summarize
grp = sub.groupby("property_type", sort=False)
summary = pd.DataFrame({
    "Previous Total": grp["npd_previous_amt"].sum(),
    "Current Total":  grp["npd_current_amt"].sum(),
})

summary["Change Amount"] = summary["Current Total"] - summary["Previous Total"]
summary["Change %"] = np.where(
```

```
    summary["Previous Total"] != 0,
    100 * summary["Change Amount"] / summary["Previous Total"],
    np.nan
)

# Order rows and round for display
summary = summary.reindex(type_order).round({
    "Previous Total": 0, "Current Total": 0, "Change Amount": 0, "Change %": 1
})

print("NPD Property Tax Summary (Residential vs. Commercial)")
print(summary.to_string())
```

```
NPD Property Tax Summary (Residential vs. Commercial)
               Previous Total  Current Total  Change Amount  Change %
property_type
Residential          19542912       22113622        2570710      13.2
Commercial           13258171       11971570       -1286601      -9.7
Mobile Homes           590973         553726         -37247      -6.3
Land                   221025         194658         -26367     -11.9
```

In [35]:

```
# Keep exactly these four types
wanted = ["Residential", "Commercial", "Mobile Homes", "Land"]

# Filter to wanted types and non-null amounts
mask = df["property_type"].isin(wanted) & df["npd_current_property_tax_amount"].not

# Sum current NPD by property type
summary = (df.loc[mask]
             .groupby("property_type", as_index=False)["npd_current_property_tax_am
             .sum()
             .rename(columns={"npd_current_property_tax_amount": "current_total"}))

# Ensure all four rows appear (0 if a type is missing)
summary = summary.set_index("property_type").reindex(wanted, fill_value=0).reset_in

# Percent of total
total = summary["current_total"].sum()
summary["percent_of_total"] = np.where(total > 0, (summary["current_total"] / total

# --- SHOW RESULTS ---
# Pretty print with currency and % formatting
to_show = summary.copy()
to_show["current_total"] = to_show["current_total"].map(lambda x: f"${x:,.0f}")
to_show["percent_of_total"] = to_show["percent_of_total"].map(lambda x: f"{x:.2f}%"

print(to_show.to_string(index=False))

# Optional: save to CSV
# summary.to_csv("npd_current_contribution_by_type.csv", index=False)
```

```
property_type current_total percent_of_total
  Residential   $22,113,622           63.48%
   Commercial   $11,971,570           34.37%
 Mobile Homes      $553,726            1.59%
         Land      $194,658            0.56%
```

In [36]:
```python
# Keep exactly these four types
wanted = ["Residential", "Commercial", "Mobile Homes", "Land"]

# Filter to wanted types and non-null amounts
mask = df["property_type"].isin(wanted) & df["npd_previous_property_tax_amount"].no

# Sum current NPD by property type
summary = (df.loc[mask]
             .groupby("property_type", as_index=False)["npd_previous_property_tax_a
             .sum()
             .rename(columns={"npd_previous_property_tax_amount": "previous_total"}

# Ensure all four rows appear (0 if a type is missing)
summary = summary.set_index("property_type").reindex(wanted, fill_value=0).reset_in

# Percent of total
total = summary["previous_total"].sum()
summary["percent_of_total"] = np.where(total > 0, (summary["previous_total"] / tota

# --- SHOW RESULTS ---
# Pretty print with currency and % formatting
to_show = summary.copy()
to_show["previous_total"] = to_show["previous_total"].map(lambda x: f"${x:,.0f}")
to_show["percent_of_total"] = to_show["percent_of_total"].map(lambda x: f"{x:.2f}%"

print(to_show.to_string(index=False))

# Optional: save to CSV
# summary.to_csv("npd_current_contribution_by_type.csv", index=False)
```

```
property_type previous_total percent_of_total
  Residential    $19,542,912           58.14%
   Commercial    $13,258,171           39.44%
 Mobile Homes       $590,973            1.76%
         Land       $221,025            0.66%
```