```
import pandas as pd
import tkinter as tk
from tkinter import filedialog
import datetime
import re
from ast import literal_eval
from difflib import get_close_matches
import os
desktop = os.path.join(os.path.expanduser("~"), "Desktop")



def clean_address(address):
    # Return immediately if NaN
    if pd.isna(address):
        return address

    # Convert to string if not already
    address = str(address)

    # Skip empty strings
    if not address.strip():
        return address

    # Preserve unit numbers (e.g., 4/26)
    if re.match(r'^\s*\d+/\d+\s', address):
```

```
    # If it's a unit number format, only clean the rest of the address
    parts = address.split(' ', 1)
    unit_number = parts[0].strip()  # Keep the unit number as is
    rest_of_address = parts[1] if len(parts) > 1 else ''

    # Clean only the rest of the address
    if rest_of_address:
        rest_of_address = clean_address(rest_of_address)
    return f"{unit_number} {rest_of_address}".strip()

# Handle commas - remove if not followed by space
if ", " not in address:
    address = address.replace(",", "")

# Fix names with "Mc" prefix
mc_pattern = r'Mc([a-z])'
address = re.sub(mc_pattern, lambda m: f"Mc{m.group(1).upper()}", address)

# PO Box standardization
po_box_pattern = r'\bP(\.|\s)?O(\.|\s)?(\.|\s)?Box\b'
address = re.sub(po_box_pattern, 'PO Box', address, flags=re.IGNORECASE)

# Check if it's a rural delivery address first
if re.search(r'\bRD\s+\d+\b', address, re.IGNORECASE):
    return address
```

```
# Rest of the existing replacements...
replacements = {
    r'\bCres\b': 'Crescent',
    r'\bBlvd\b': 'Boulevard',
    r'\bCt\b': 'Court',
    r'\bLn\b': 'Lane',
    r'\bAve\b': 'Avenue',
    r'\bDr\b': 'Drive',
    r'\bst\b$': 'Street',  # Only at end of string
    r'\broad\b': 'Road',
    r'\bcres\b': 'Crescent',
    r'\bblvd\b': 'Boulevard',
    r'\bdrive\b': 'Drive',
    r'\bPl\b': 'Place',
    r'\bplace\b': 'Place',
    r'\bTce\b': 'Terrace',
    r'\bterrace\b': 'Terrace',
    r'\bRd\b(?!\s+\d)': 'Road'  # Only replace Rd when not followed by a number
}

for pattern, replacement in replacements.items():
    address = re.sub(pattern, replacement, address, flags=re.IGNORECASE)

# Capitalize letters after numbers (e.g., 123b to 123B)
address = re.sub(r'(\d)([a-z])', lambda m: m.group(1) + m.group(2).upper(), address)
```

```
    # Add space between house number and street name only if not a unit number format
    if '/' not in address:
        address = re.sub(r'(\d+)([A-Za-z]{2,})', r'\1 \2', address)

    return address.strip()

def extract_postal_code(address):
    if pd.isna(address):
        return '', ''

    address = str(address)

    # Define postal code patterns and their corresponding countries
    postal_patterns = {
        r'\b[A-Z]\d[A-Z]\s*\d[A-Z]\d\b': 'Canada',  # A1A 1A1
        r'\b[A-Z]{1,2}\d{1,2}\s*\d[A-Z]{2}\b': 'United Kingdom',  # AA1 1AA or A1 1AA
        r'\b[A-Z]{3}\s*\d{4}\b': 'Malta',  # AAA 1234
        r'\bVG\d{4}\b': 'British Virgin Islands',  # VG1110
        r'\b\d{4}\s*[A-Z]{2}\b': 'Netherlands',  # 1234 AB
        r'\b[A-Z]\d{2}\s*[A-Z0-9]{4}\b': 'Ireland',  # A65 F4E2
        r'\b[A-Z]{2}\d{4}\b': 'Brunei Darussalam',  # AA1234
        r'\bKY1-\d{4}\b': 'Cayman Islands',  # KY1-1234
        r'\b[A-Z]\d{4}[A-Z]{3}\b': 'Argentina',  # A1234ABC
        r'\b97150\b': 'Saint Martin'  # French Saint Martin
    }
```

```
    # First check for specific country postal codes
    for pattern, country in postal_patterns.items():
        match = re.search(pattern, address, re.IGNORECASE)
        if match:
            postal_code = match.group(0)
            # Remove the postal code from the address to get the remaining text
            remaining = address.replace(postal_code, '').strip()
            return postal_code, country if not remaining else remaining

    # If no specific country pattern matches, try generic postal/zip code pattern
    postal_pattern = r'\s+(\d{4,}|\d{2,}-\d{3,})$'
    match = re.search(postal_pattern, address)

    if match:
        postal_code = match.group(1)
        country = address.replace(postal_code, '').strip()
        return postal_code, country
    return '', address

def is_unit_number(address_part):
    """
    Determine if a number is a unit number or a street number
    """
    try:
        # Guard against None or empty strings
        if not address_part or pd.isna(address_part):
```

```
            return False

        # If it contains a slash, it's probably a unit number
        if '/' in address_part:
            parts = address_part.split('/')
            # Only consider it a unit number if:
            # 1. It has exactly 2 parts
            # 2. Both parts are numbers
            # 3. First number (unit) is smaller than second (street)
            if (len(parts) == 2 and
                all(part.strip().isdigit() for part in parts) and
                int(parts[0]) < int(parts[1])):
                return True
        return False
    except (ValueError, IndexError, AttributeError):
        # If any error occurs, it's not a valid unit number
        return False


def clean_nz_addresses(df, lookup_df):
    # Add debug prints
    print(f"Total rows: {len(df)}")
    print(f"NZ addresses found: {len(df[df['Country'].str.upper().isin(['NEW ZEALAND',
'NZ'])])}")
    print(f"Addresses with Line 3 data: {len(df[~df['Perm Address line 3'].isna()])}")


    # Ensure required columns exist
```

```
    if 'Country' not in df.columns:
        df['Country'] = ''

    # First, identify and handle unit numbers in Line 2
    unit_number_mask = df['Perm Address line 2'].str.match(r'^\s*\d+/\d+\s', na=False)
    if unit_number_mask.any():
        # For rows with unit numbers in Line 2, preserve them
        df.loc[unit_number_mask, 'Perm Address line 1'] = df.loc[unit_number_mask, 'Perm Address line 2']
        df.loc[unit_number_mask, 'Perm Address line 2'] = ''

    # Clean addresses in both line 1 and line 2
    for col in ['Perm Address line 1', 'Perm Address line 2']:
        if col in df.columns:
            df[col] = df[col].apply(clean_address)

    # If Line 2 starts with a number (but not a unit number), move it to Line 1
    starts_with_number_mask = (df['Perm Address line 2'].str.match(r'^\d+[A-Za-z]?\s', na=False) &
                               ~df['Perm Address line 2'].str.match(r'^\d+/\d+\s', na=False))
    df.loc[starts_with_number_mask, 'Perm Address line 1'] = df.loc[starts_with_number_mask, 'Perm Address line 2']
    df.loc[starts_with_number_mask, 'Perm Address line 2'] = ''

    # Check for number-letter combinations in Line 1 and merge with Line 2
    num_letter_mask = df['Perm Address line 1'].str.match(r'^\s*\d+\s*[a-zA-Z]\s*$', na=False)
```

```
    df.loc[num_letter_mask, 'Perm Address line 1'] = df.loc[num_letter_mask, 'Perm Address line 1'].str.strip() + ' ' + df.loc[num_letter_mask, 'Perm Address line 2']

    df.loc[num_letter_mask, 'Perm Address line 2'] = ''


    # If Line 1 only has numbers and Line 2 doesn't have a unit number, concatenate with Line 2

    numeric_mask = (df['Perm Address line 1'].str.replace(r'\s+', '').str.match(r'^\d+$', na=False) &

                    ~df['Perm Address line 2'].str.contains(r'^\d+/\d+', na=False, regex=True))

    df.loc[numeric_mask, 'Perm Address line 1'] = df.loc[numeric_mask, 'Perm Address line 1'] + ' ' + df.loc[numeric_mask, 'Perm Address line 2']

    df.loc[numeric_mask, 'Perm Address line 2'] = ''


    # Clear PO Box addresses in Line 2

    po_box_pattern = r'\bP(\.|\s)?O(\.|\s)?(\.|\s)?Box\b'

    po_box_mask = df['Perm Address line 2'].str.contains(po_box_pattern, case=False, na=False, regex=True)

    df.loc[po_box_mask, 'Perm Address line 2'] = ''


    # Create Suburb column and move remaining Line 2 data only for NZ addresses

    if 'Suburb' not in df.columns:

        line2_pos = df.columns.get_loc('Perm Address line 2')

        df.insert(line2_pos + 1, 'Suburb', '')


    # Move Line 2 data to Suburb for NZ addresses (handle case-insensitive matching)

    nz_mask = df['Country'].str.upper().isin(['NEW ZEALAND', 'NZ'])

    if nz_mask.any():  # Only proceed if there are NZ addresses
```

```
        df.loc[nz_mask, 'Suburb'] = df.loc[nz_mask, 'Perm Address line 2']
        df.loc[nz_mask, 'Perm Address line 2'] = ''

    # Handle numbers in both Line 1 and Line 2
    for col in ['Perm Address line 1', 'Perm Address line 2']:
        if col in df.columns:
            # Split the address into parts with error handling
            df[col] = df[col].apply(lambda x: x if pd.isna(x) else
                                    x if not str(x).strip() else  # Handle empty strings
                                    x if is_unit_number(str(x).split()[0] if str(x).split() else '') else
                                    x.replace('/', ''))

    # Create a set of NZ suburbs for lookup
    nz_suburbs = set(lookup_df[lookup_df['Country'] == 'NEW ZEALAND']['NZ
Suburb'].dropna().str.upper().unique())

    # Process suburbs for NZ addresses
    nz_mask = df['Country'].str.upper().isin(['NEW ZEALAND', 'NZ'])
    if nz_mask.any():  # Only proceed if there are NZ addresses
        def match_suburb(row):
            if pd.isna(row['Perm Address line 3']):
                return row

            line3_upper = str(row['Perm Address line 3']).upper()

            # First try exact match
```

```
    if line3_upper in nz_suburbs:
        row['Suburb'] = row['Perm Address line 3']
        row['Perm Address line 3'] = ''
        if pd.isna(row['Edits']) or row['Edits'] == '':
            row['Edits'] = 'Moved suburb from Line 3'
        else:
            row['Edits'] += ' | Moved suburb from Line 3'
        return row

    # If no exact match, try fuzzy matching
    matches = get_close_matches(line3_upper, nz_suburbs, n=1, cutoff=0.85)
    if matches:
        matched_suburb = matches[0]
        # Get the original case from lookup_df
        original_case = lookup_df[lookup_df['NZ Suburb'].str.upper() ==
matched_suburb]['NZ Suburb'].iloc[0]
        row['Suburb'] = original_case
        row['Perm Address line 3'] = ''
        edit_msg = f'Moved suburb from Line 3 (fuzzy matched "{row["Perm Address line
3"]}" to "{original_case}")'
        if pd.isna(row['Edits']) or row['Edits'] == '':
            row['Edits'] = edit_msg
        else:
            row['Edits'] += f' | {edit_msg}'

    return row
```

```
        # Apply the suburb matching only to NZ addresses
        df.loc[nz_mask] = df.loc[nz_mask].apply(match_suburb, axis=1)

    df = clean_rd_addresses(df)
    return df

def clean_rd_addresses(df):
    """
    Clean and standardize Rural Delivery (RD) addresses in a DataFrame.
    Implements functionality from the original VBA macro for RD address handling.

    Parameters:
    df (pandas.DataFrame): DataFrame containing address data

    Returns:
    pandas.DataFrame: Processed DataFrame with cleaned RD addresses
    """
    import re
    import pandas as pd

    # Track if we need to create new columns
    if 'Edits' not in df.columns:
        df['Edits'] = ''
    if 'Original Data' not in df.columns:
        df['Original Data'] = ''
```

```
# Compile regex patterns for RD corrections
rd_patterns = {
    'number_rd': re.compile(r'(\d+)\s*(r\s*d|rd|r\s*D|R\s*D)', re.IGNORECASE),
    'rd_number': re.compile(r'(R\s*D|r\s*d|rd|R\s*D)(\d+)', re.IGNORECASE),
    'r_d_number': re.compile(r'(R)\s*(D)\s*(\d+)', re.IGNORECASE),
    'standalone_rd': re.compile(r'^(RD|R\s*D)$', re.IGNORECASE),
    'text_rd': re.compile(r'([A-Za-z]+.*)\s*(R\s*D\s*\d+)', re.IGNORECASE)
}

def process_rd_value(value, row_idx, col_name):
    """Process a single value for RD corrections"""
    if pd.isna(value):
        return value, ''

    original_value = str(value)
    new_value = original_value
    edits = []
    needs_review = False

    # Remove commas if not followed by space
    if ", " not in new_value:
        new_value = new_value.replace(",", "")

    # Apply RD corrections
    if rd_patterns['number_rd'].search(new_value):
        new_value = rd_patterns['number_rd'].sub(r'RD \1', new_value)
```

```
            edits.append(f"Standardized RD format")

        elif rd_patterns['rd_number'].search(new_value):
            new_value = rd_patterns['rd_number'].sub(r'RD \2', new_value)
            edits.append(f"Standardized RD format")

        elif rd_patterns['r_d_number'].search(new_value):
            new_value = rd_patterns['r_d_number'].sub(r'RD \3', new_value)
            edits.append(f"Standardized RD format")

        # Check for standalone RD
        if rd_patterns['standalone_rd'].match(new_value):
            needs_review = True
            edits.append("Highlight - Please Review (standalone RD)")

        # Check for RD in Address Line 1
        if col_name == 'Perm Address line 1' and re.match(r'^RD\s+\d+$', new_value,
re.IGNORECASE):
            needs_review = True
            edits.append("Highlight - RD in Address Line 1")

        # Only remove trailing slashes, preserve unit number slashes
        if '/' in new_value and not re.match(r'^\d+/\d+', new_value.strip()):
            new_value = new_value.rstrip('/')
            edits.append("Removed trailing slash")
```

```
    # Remove trailing commas
    new_value = re.sub(r',\s*$', '', new_value)
    new_value = new_value.strip()

    return new_value, ' | '.join(edits), needs_review

def move_rd_to_suburb(row):
    """Process a row to move RD values to suburb column"""
    for col in ['Perm Address line 1', 'Perm Address line 2']:
        if pd.isna(row[col]):
            continue

        match = rd_patterns['text_rd'].search(str(row[col]))
        if match:
            text_part = match.group(1).strip()
            rd_part = match.group(2).strip()

            # Update the original address column
            row[col] = text_part

            # Move RD part to suburb
            if pd.isna(row['Suburb']):
                row['Suburb'] = rd_part
            else:
                row['Suburb'] = f"{row['Suburb']} {rd_part}"
```

```
            # Update edits
            if pd.isna(row['Edits']):
                row['Edits'] = f"Moved '{rd_part}' to Suburb"
            else:
                row['Edits'] = f"{row['Edits']} | Moved '{rd_part}' to Suburb"

    return row

# Process each address field
for col in ['Perm Address line 1', 'Perm Address line 2']:
    if col in df.columns:
        # Store original values if not already stored
        mask = df['Original Data'].isna()
        df.loc[mask, 'Original Data'] = df.loc[mask].apply(
            lambda row: str({col: row[col]}), axis=1
        )

        # Process each value
        for idx, value in df[col].items():
            if pd.isna(value):
                continue

            new_value, edits, needs_review = process_rd_value(value, idx, col)

            if new_value != str(value):
                df.loc[idx, col] = new_value
```

```
            if edits:
                current_edits = df.loc[idx, 'Edits']
                df.loc[idx, 'Edits'] = f"{current_edits} | {edits}" if current_edits else edits

    # Move RD values to suburb column
    df = df.apply(move_rd_to_suburb, axis=1)

    return df

def proper_case(text):
    """
    Convert text to proper case while handling special cases like 'RD', 'PO Box', and 'Mc' names.
    """
    if pd.isna(text):
        return text

    text = str(text).strip()
    if not text:
        return text

    # Special cases to preserve
    special_cases = {
        'RD': r'\bRD\b',
        'PO Box': r'\bPO Box\b',
        'NZ': r'\bNZ\b'
```

```
    }

    # First, convert everything to title case
    text = text.title()

    # Handle Mc names (e.g., McDonald)
    text = re.sub(r'Mc([a-z])', lambda m: f"Mc{m.group(1).upper()}", text)

    # Restore special cases
    for replacement, pattern in special_cases.items():
        text = re.sub(pattern, replacement, text, flags=re.IGNORECASE)

    return text

def has_meaningful_edit(row):
    """
    Check if meaningful edits were made by comparing original and final values.
    Only returns True if changes were detected.
    """
    try:
        original = literal_eval(row['Original Data'])
        # Compare original address with current address
        if original['Address'] is not None:
            if str(original['Address']).strip() != str(row['Address Line 1']).strip():
                return True
```

```
            # Compare original Line 2 if it exists
            if 'Line 2' in original and original['Line 2'] is not None:
                if str(original['Line 2']).strip() != str(row['Address Line 2']).strip():
                    return True
        except (ValueError, SyntaxError, TypeError):
            pass
    return False

def main():
    # Create a root window but hide it
    root = tk.Tk()
    root.withdraw()

    # First load the address lookup data
    print("Please select the address lookup CSV file (addresslookup.csv)...")
    lookup_file = filedialog.askopenfilename(
        title='Select Address Lookup CSV File',
        filetypes=[('CSV files', '*.csv'), ('All files', '*.*')]
    )

    if not lookup_file:
        print("No lookup file selected. Exiting...")
        return

    try:
        # Try different encodings
```

```
encodings = ['utf-8', 'latin1', 'iso-8859-1', 'cp1252']
for encoding in encodings:
    try:
        lookup_df = pd.read_csv(lookup_file, encoding=encoding)
        break  # If successful, break the loop
    except UnicodeDecodeError:
        continue
else:  # If no encoding worked
    print("Error: Unable to read the lookup file with any standard encoding.")
    return

# Open file dialog for input file
print("Please select your input CSV file...")
input_file = filedialog.askopenfilename(
    title='Select Input CSV File',
    filetypes=[('CSV files', '*.csv'), ('All files', '*.*')]
)

if not input_file:
    print("No input file selected. Exiting...")
    return

# Try different encodings with error handling
encodings = ['utf-8', 'latin1', 'iso-8859-1', 'cp1252', 'utf-8-sig']
for encoding in encodings:
    try:
```

```
            df = pd.read_csv(input_file, encoding=encoding)
            print(f"Successfully read file with {encoding} encoding")
            break
        except UnicodeDecodeError:
            continue
    else:  # If no encoding worked
        raise Exception("Unable to read the file with any standard encoding. Please check the file encoding.")

except Exception as e:
    print(f"Error reading the input file: {str(e)}")
    return




# Create tracking columns early
if 'Edits' not in df.columns:
    df['Edits'] = ''
if 'Original Data' not in df.columns:
    df['Original Data'] = ''

# Capture the original CSV data immediately after reading the file
df['Original Data'] = df.apply(lambda row: row.to_dict(), axis=1).apply(str)

# Process Perm Address line 4 first to extract Country
```

```
    if 'Perm Address line 4' in df.columns:
        # Extract postal codes from Perm Address line 4 but keep them in a new column
        df['Postal Code'], df['Country'] = zip(*df['Perm Address line 4'].apply(extract_postal_code))
        # Drop the original line 4 column
        df = df.drop(columns=['Perm Address line 4'])

    # Create a set of NZ cities for faster lookup
    nz_cities = set(lookup_df[lookup_df['Country'] == 'NEW ZEALAND']['NZ City'].dropna().unique())

    # Then handle city detection and Country updates
    city_mask = df['Country'].isin(nz_cities)
    cities_to_move = df.loc[city_mask, 'Country'].copy()
    df.loc[city_mask, 'Perm Address line 2a'] = cities_to_move
    df.loc[city_mask, 'Country'] = 'New Zealand'

    # Now that we have properly identified NZ addresses, clean them
    df = clean_nz_addresses(df, lookup_df)

    # Clean the addresses
    df['Perm Address line 1'] = df['Perm Address line 1'].apply(clean_address)


    # After all other processing, apply proper case to address columns
    address_columns = [
        'Perm Address line 1',
```

```
    'Perm Address line 2',
    'Perm Address line 2a',
    'Perm Address line 2b',
    'Perm Address line 3',
    'Suburb'
]

# Apply proper case only to columns that exist in the DataFrame
existing_columns = [col for col in address_columns if col in df.columns]
for col in existing_columns:
    df[col] = df[col].apply(proper_case)

# After proper case application and before saving, rename columns
column_rename = {
    'Perm Address line 1': 'Address Line 1',
    'Perm Address line 2': 'Address Line 2',
    'Perm Address line 2a': 'City',
    'Perm Address line 2b': 'Region/Province/State',
    'Perm Address line 3': 'Address Line 3'
}
df = df.rename(columns=column_rename)

# Reorder columns
column_order = [
    'Person Id',
    'Address Line 1',
```

```
        'Address Line 2',
        'Address Line 3',
        'Suburb',
        'City',
        'Region/Province/State',
        'Postal Code',
        'Country',
        'Original Data'
    ]
    final_columns = [col for col in column_order if col in df.columns]
    df = df[final_columns]

    # Temporarily disable the Has Edits flag by setting it to True for all rows until more refined coding can be implimented.
    df['Has Edits'] = True



    # Update column order to include new flag
    column_order = [
        'Person Id',
        'Address Line 1',
        'Address Line 2',
        'Address Line 3',
        'Suburb',
        'City',
        'Region/Province/State',
```

```
        'Postal Code',
        'Country',
        'Has Edits',
        'Original Data'
    ]
    final_columns = [col for col in column_order if col in df.columns]
    df = df[final_columns]

    # Create output filename with timestamp
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    output_file = f"processed_addresses_{timestamp}.csv"

    # Save to current directory
    df.to_csv(output_file, index=False)
    print(f"Data saved to {output_file}")

if __name__ == "__main__":
    main()
```