Transforming Post-1790 CD Debt Data

Author: Chris Liao ([first name] [last name] (at) [uchicago] [dot] [edu])

Objective

Turn raw post-1790 continental debt (CD) security data into an organized table indexed by individuals.

Process

1. Adding Each Individual's Geography

Code:

- clean_1_geo.ipynb combines the raw CD debt data from all states into one dataset and processes the given geography colum

Inputs:

- Raw Data
    1. Post-1790 Continental Debt Files: csv files with suffix CD
    2. Examples
        1. Connecticut: CT_post1790_CD_ledger.xlsx
        2. Georgia: T694_GA_Loan_Office_CD.xlsx
- cd_raw.csv: arguments for importing state CD files
- zip_code_database.xls: geograhical database matching towns to counties
    - Downloaded from https://www.unitedstateszipcodes.org/zip_code_database.xls?download_auth=7b5b7133a55eef6807fc6da56f62bf27
- town_fix.csv: database of changes to the gegographical classification

Outputs (for future use):

- aggregated_CD_post1790.csv: continental debt files with final geographical classification

Outputs (to check validity of cleaning process):

- change_df_CD: crosswalk mapping aggregation of multiple towns/occupations (raw data) to one town/occupation (aggregated_CD_post1790.csv)

Steps:

1. Using the arguments in cd_raw.csv, the raw CD data for each state is imported and aggregated into one table
2. Our raw data (except for NY) contains a town and state column denoting the place of residence for each debtholder
    1. When an entry for the state column is missing, we impute the state loan office that the debtholder redeemed debt from
    2. When there are multiple town or occupation values for one debtholder entry, we select the value with longest string length (since it likely contains the most information). The results of this selection are in change_df_CD.
    3. When one debtholder entry has multiple names, we group them and note that the entry has multiple names. CT_10 has the value `Joseph Woodruff | Joseph Woodruffe`
3. The town column in our raw data is extremely messy. Here are some of its problems
    1. The same location can be spelled multiple ways
        1. GA_24 and GA_33 have the values `Charleston South Carolina` and `Charleston` in their respective town column values
    2. The listed "town" might be a town, state or column
        1. PA_115 and PA_655 have the values `Cumberland` and `Cumb County Pennsylvania` in their respective town column values
4. Using fuzzy string matching with zip_code_database.xls, we identify whether a "town" value is a town, county or state, and reformat it
    1. For towns, we also find the corresponding county name
5. There are cases where we cannot use fuzzy string matching to clean our geographies (or less commonly, zip_code_database.xls makes a mistake). In this case, we use town_fix.csv to make the required changes
6. Our final results are in aggregated_CD_post1790.csv

`town`, `occupation` and `state` are given columns from the raw data but post step 2

`new_town`, `new_county`, `new_state`, `country`, `name_type` are columns created post-cleaning and represent the location of an individual

	town	state	occupation	new_town	county	new_state	country	name_type
0	Hartford	CT	Merchant	Hartford	Hartford County	CT	US	town
2	Rhode Island	RI	Farmer	nan	nan	RI	US	state
390	City of New York	NY	Merchant	New York City	New York County	NY	US	town
2001	Bucks County Pennsylvania	PA	nan	nan	Bucks County	PA	US	county

```
# generate above - run at end of notebook
print(CD_all[['town', 'state', 'occupation', 'new_town', 'county', 'new_state', 'country',
'name_type']].loc[[0,2,390, 2001]].to_markdown())
```

2. Cleaning Names

Code:

- clean_2_names.ipynb cleans all the names in the CD debt file

Inputs:

- aggregated_CD_post1790.csv: continental debt files with final geographical classification
- company_names_fix.csv: database of name changes for data cleaning purposes

Outputs (for future use):

- aggregated_CD_post1790_names.csv: aggregated_CD_post1790.csv with cleaned names
- name_list.csv: List of all identities in our raw debt data with cleaned names and geographies
    1. Identities have not been aggregated (two slightly mispelled names representing the same identity are denoted as separate identities)

Outputs (for future research)

- company_research.csv: list of companies we want to map to owners/identities

Steps:

1. First, we import aggregated_CD_post1790.csv
2. The names in aggregated_CD_post1790.csv can be quite messy for various reasons
    1. One "name" value can be multiple names: CT_19 has the value `John and James Davenport`
    2. One "name" value can have extraneous information: RI_318 has the value `John Parker as Gaurdian`
    3. One "name" value can be an institution, not a name: RI_597 has the value `Clark and Nightingale Transferred from Register | Clark and Nightingale transferred`. `Clark and Nightingale` is a company owned by Joseph Innes Clark and Joseph Nightingale. In this case, we can match a company to the owner but we may not always be able to do this
        1. company_research.csv contains the list of companies we want to find identities for
    4. One debtholder entry can contain multiple names: CT_10 has the value `Joseph Woodruff | Joseph Woodruffe`. Different names are separated by `|`
3. In section **Known Cleaning Process**, we clean names where we know how to fix the structure
4. In section **Import Name Fixes**, we clean names that have to be manually fixed (looked at each messed up entry one by one, then added the fixed name to the spreadsheet) using company_names_fix.csv
    1. This process was tedious, even with GitHub copilot. we hope that this summer we can automate at least parts of this process
5. In section **Manual Name Fixes**, we make some final name changes
6. Finally, we create a dataset of all unique identities (name + geography combinations) to feed to our scraper, outputted as name_list.csv
    1. Name values that are not actually names (and for who we cannot match to a set of actual names) are excludede from this dataset
    2. NH_22's name is `The Trustees of Phillips Academy`
    3. GA_64's name is `Jackson and Nightingale`
7. we also create aggregated_CD_post1790_names.csv, which contains debt data, the original name and the new (cleaned) name

Here are some examples of the original name and the cleaned name

	original	new
0	Clark and Nightingale	Joseph Innes Clark | Joseph Nightingale
1	Jon and Jacob Starr | Jonathan and Jared Starr	Jacob Starr | Jonathan Starr | Jared Starr
38	Nicholas And Hannah Cooke | Nicholas And Hannah Coske | Robert Crooke	Hannah Cooke | Hannah Coske | Nicholas Cooke | Nicholas Coske | Robert Crooke

```
# generate code
print(df_comp.loc[[0,1,38]].to_markdown())
```

3. Scraping Census Data

Code:

- clean_3_scrape.ipynb cleans all the names in the CD debt file

Inputs:

- name_list.csv: List of all identities in our raw debt data with cleaned names and geographies

Outputs (for future use):

- name_list_scraped.csv: List of scrapable names in our dataset, with scraping results
- scrape_ids.csv: cleaned preliminary dataset of identities from Ancestry.com census scraper
- scrape_results.csv: cleaned preliminary demographic data for identities from scraper

Outputs (preliminary - not relevant outside of serving as checkpoints)

- scrape_ids_prelims.csv: preliminary dataset of identities from scraper
- scrape_results_prelim.csv: preliminary demographic data for identities from scraper

Steps:

1. First, we import name_list.csv
2. The code for the scraper looks complicated, but here's the gist of how it works - describes the **Helper Functions** and **Run Scraper** sections
    1. Navigate to the ancestry.com library landing page and login to my UChicago student portal
    2. Go to the 1790 census collection search page
    3. Loop through my list of individuals, enter the corresponding first name, last name and location, and search
        1. Each location has a special "location code" embedded in the search url - if we have not seen a location before, we find its location code and store it
        2. For each search, if we do not find a result, we reduce the strictness of the search (how exact the name match is, how exact the geography match is) up until we either **a)** find a result or **b)** reach my strictness threshold
    4. Store the results - the below files are saved after each search so no data is lost if the scraper crashes
        1. scrape_ids_prelims.csv contains information on which matches on Ancestry.com an individual matched to and the number of matches (between 0-4)
        2. scrape_results_prelim.csv contains information on the demographic data for each Ancestry.com individual
3. There are some manual changes we have to make to the results of our scraper and we do this in the **Clean Scraped Data** section
4. Finally, we reset the index of our dataframes by removing duplicate entries
    1. This turns out to be a bit more complicated than just using the `.reset_index()` command becuase our two datasets are relational
5. We export our final ancestry.com scraped data as scrape_ids.csv and scrape_results.csv
6. We link our scraped census data to the names from the debt file in name_list_scraped.csv

Example of scraped results matched to debtholder names

	Fn_Fix	Ln_Fix	new_town	county	new_state	country	name_type	Match Index	Match Status
1	Benjamin	Trumbull	Bolton	Tolland County	CT	US	town	0	Complete Match
9	Joseph	Woodruff	Farmington	Hartford County	CT	US	town	8| 9	2 Potential Matches Found
21	Allen	Olcott	Farmington	Hartford County	CT	US	town	nan	No Match

	Name	Home in 1790 (City, County, State)	Free White Persons - Males - 16 and over	Free White Persons - Females	Number of Household Members	Free White Persons - Males - Under 16	Number of Slaves	Number of All Other Free Persons	Match Type
0	Benjn Trumbull | Benja Trunabull	Bolton, Tolland, Connecticut	1	1	2	nan	nan	nan	town
8	Joseph Woodruff	Farmington, Hartford, Connecticut	2	3	6	1	nan	nan	town
9	Joseph Woodruff	Farmington, Hartford, Connecticut	2	4	11	5	nan	nan	town

```python
# generate code for first and second tables
print(pd.read_csv('scrape_tools/name_list_scraped.csv', index_col = 0).loc[[1, 9, 21]][['Fn_Fix', 'Ln_Fix',
'new_town', 'county', 'new_state', 'country','name_type', 'Match Index', 'Match Status']].to_markdown())
print(pd.read_csv('scrape_tools/scrape_results.csv', index_col = 0).loc[[0, 8, 9]][['Name', 'Home in 1790 (City,
County, State)','Free White Persons - Males - 16 and over','Free White Persons - Females', 'Number of Household
Members','Free White Persons - Males - Under 16','Number of Slaves','Number of All Other Free Persons','Match
Type']].to_markdown())
```

4. Creating Final Dataset

Code:

- clean_4_final.ipynb creates our final dataset

Inputs:

- aggregated_CD_post1790.csv: continental debt files with final geographical classification
- name_list_scraped.csv: List of scrapable names in our dataset, with scraping results
- scrape_results.csv: cleaned preliminary dataset of data for matched identities from Ancestry.com census scraper
- name_list.csv: List of all identities in our raw debt data with cleaned names and geographies
    1. Identities have not been aggregated (two slightly mispelled names representing the same identity are denoted as separate identities)
- name_agg.csv: database of names spelled differently that correspond to the same identity
- group_name_state.csv: database of names with locations in multiple states that correspond to the same identity
- occ_correction.csv: database of occupation name changes for data cleaning purposes

Outputs (for future use):

- final_data_CD.csv: final table, indexed by individual, of aggregate CD debt holdings
- match_data_CD.csv: database of ancestry.com data for final_data_CD

Steps:

1. First, we import aggregated_CD_post1790.csv, name_list_scraped.csv, scrape_results.csv and name_list.csv
2. In **Adding Missing Occupations**, there are cases where an occupation is part of a name in aggregated_CD_post1790.csv, but the occupation is not listed in the occupation column. we amend this by adding the occupation
    1. Example: In CT_333, there is no occupation listed even though the debtholder's name is `Wheeler Coit Treasurer and Co` , so we added the `treasurer` occupation to the occupation column
3. There are two steps in **Merge Data** to create the aggregate dataset we will work with
    1. In aggregated_CD_post1790.csv (variable `CD_clean`), one debtholder entry name (`Name` column) sometimes corresponds to multiple names. we merge it with name_list.csv (variable `name_df`), which maps one debtholder entry name (`Name`) to however many actual names are there are (values listed in `Name_Fix`). Note that now, the same debtholder entry may be linked to multiple "names". Here is an example:
        1. `CD_clean`

	Name	state_data	state_data_index	new_town	county	new_state	country	name_type
8	Francis Brown	CT	9	New Haven	New Haven County	CT	US	town
9	Joseph Woodruff | Joseph Woodruffe	CT	10	Farmington	Hartford County	CT	US	town

        2. `name_df`

	Name	Fn_Fix	Ln_Fix	county	new_state	country	name_type
8	Francis Brown	Francis	Brown	New Haven County	CT	US	town
9	Joseph Woodruff | Joseph Woodruffe	Joseph	Woodruff	Hartford County	CT	US	town
9	Joseph Woodruff | Joseph Woodruffe	Joseph	Woodruffe	Hartford County	CT	US	town

        3. merged table (variable `CD_merged`)

	Name	state_data	state_data_index	Name_Fix	Fn_Fix	Ln_Fix	county	new_state	country	name_type
8	Francis Brown	CT	9	Francis Brown	Francis	Brown	New Haven County	CT	US	town
9	Joseph Woodruff | Joseph Woodruffe	CT	10	Joseph Woodruff | Joseph Woodruffe	Joseph	Woodruff	Hartford County	CT	US	town
10	Joseph Woodruff | Joseph Woodruffe	CT	10	Joseph Woodruff | Joseph Woodruffe	Joseph	Woodruffe	Hartford County	CT	US	town

        4. Code:

```
# generate code for all tables
print(CD_clean.loc[[8, 9]][['Name', 'state_data', 'state_data_index', 'new_town', 'county',
'new_state', 'country', 'name_type',]].to_markdown())
print(name_df.loc[[8, 9]][['Name', 'Fn_Fix', 'Ln_Fix', 'county', 'new_state', 'country',
'name_type',]].to_markdown())
print(CD_merged.loc[[8,9,10]][['Name', 'state_data', 'state_data_index', 'Name_Fix', 'Fn_Fix',
'Ln_Fix', 'county', 'new_state', 'country', 'name_type',]].to_markdown())
```

2. Next, we merge our merged table with `name_list_scraped.csv` (variable `scraped_names`) so that our table includes information on the census match results from Ancestry.com. Here is an example
    1. `CD_merged`

	Name	state_data	state_data_index	Name_Fix	Fn_Fix	Ln_Fix	county	new_state	country	name_type
8	Francis Brown	CT	9	Francis Brown	Francis	Brown	New Haven County	CT	US	town

    2. `scraped_names`

	Fn_Fix	Ln_Fix	county	new_state	country	name_type	Match Index	Match Status
8	Francis	Brown	New Haven County	CT	US	town	7	Complete Match

    3. merged table (variable `CD_merged_mind`)
    4.

	Name	state_data	state_data_index	Name_Fix	Fn_Fix	Ln_Fix	Full Search Name	county	new_state	country	name_type	Match Index	Match Status
8	Francis Brown	CT	9	Francis Brown	Francis	Brown	Francis Brown	New Haven County	CT	US	town	7	Complete Match

    5. Code:

```
# generate code for all tables
print(CD_merged.loc[[8]][['Name', 'state_data', 'state_data_index', 'Name_Fix', 'Fn_Fix', 'Ln_Fix',
'county', 'new_state', 'country', 'name_type',]].to_markdown())
print(scraped_names.loc[[8]][['Fn_Fix', 'Ln_Fix','county', 'new_state', 'country', 'name_type','Match
Index', 'Match Status']].to_markdown())
print(CD_merged_mind.loc[[8]][['Name', 'state_data', 'state_data_index', 'Name_Fix', 'Fn_Fix',
'Ln_Fix', 'Full Search Name', 'county', 'new_state', 'country', 'name_type','Match Index', 'Match
Status']].to_markdown())
```

4. Next, we begin the process of grouping names that represent the same identity together in the **Group Names - Using Ancestry.com Matches** section. First, we take advantage of the fact that sometimes, Ancestry.com matches two individuals with slightly different names to the same identity (variable `Match Index`)
    1. We pick the name that is longest (measured by string length) to be the "representative name" for the identity
    2. There are some manual corrections we have to make, when we don't want to the longest name to be the representative name
        1. We create new columns with the prefix `Group` to describe characteristics (town, county, state, name) for the identity
        2. This is important because sometimes, multiple names have different values for the aforementioned characteristics (for example, one name might have location at `name_type` county, while another has location at `name_type` town - see tables below)
        3. We pick values for whatever characteristic is most detailed (in the above example, we pick `name_type` town)
        4. Sometimes the "representative name" also has different characteristics and in these cases, we apply the same procedure and set the characteristics to be the most specific
    3. Here's an example for a case where we don't have to change the location (`Richard Woottan`) and a case where we do (`Gassaway Watkins`)
        1.

	Name	state_data	state_data_index	Name_Fix	Fn_Fix	Ln_Fix	county	new_state	country	name_type	Match Index	Match Status	Group Name	Group Town	Group County	Group State	Group Country	Gr Na Ty
1379	Richard Wootton | Richard Wootton	MD	175	Richard Wootton | Richard Wootton	Richard	Wootton	Montgomery County	MD	US	county	1002	Complete Match	Richard Wootton		Montgomery County	MD	US	co
1380	Richard Wootton | Richard Wootton	MD	175	Richard Wootton | Richard Wootton	Richard	Wootton	Montgomery County	MD	US	county	1002	Complete Match	Richard Wootton		Montgomery County	MD	US	co
1672	Gassaway Watkins | Gassway Watkins	MD	403	Gassaway Watkins | Gassway Watkins	Gassaway	Watkins	Anne Arundel County	MD	US	county	1126	Complete Match	Gassaway Watkins	Annapolis	Anne Arundel County	MD	US	to
1673	Gassaway Watkins | Gassway Watkins	MD	403	Gassaway Watkins | Gassway Watkins	Gassway	Watkins	Anne Arundel County	MD	US	county	1126	Complete Match	Gassaway Watkins	Annapolis	Anne Arundel County	MD	US	to
1877	Gassaway Watkins | Gassway Watkins | William Marbury	MD	589	Gassaway Watkins | Gassway Watkins | William Marbury	Gassaway	Watkins	Anne Arundel County	MD	US	town	1126	Complete Match	Gassaway Watkins	Annapolis	Anne Arundel County	MD	US	to
1878	Gassaway Watkins | Gassway Watkins | William Marbury	MD	589	Gassaway Watkins | Gassway Watkins | William Marbury	Gassway	Watkins	Anne Arundel County	MD	US	town	1126	Complete Match	Gassaway Watkins	Annapolis	Anne Arundel County	MD	US	to
1880	Gassaway Watkins | Tristiam Bowdle | Tristram Bowdle	MD	590	Gassaway Watkins | Tristiam Bowdle | Tristram Bowdle	Gassaway	Watkins	Anne Arundel County	MD	US	county	1126	Complete Match	Gassaway Watkins	Annapolis	Anne Arundel County	MD	US	to

```
# code
print(CD_merged_mind[CD_merged_mind['Group Name'].apply(lambda x: x in ['Richard Woottan', 'Gassaway
Watkins'])][['Name', 'state_data', 'state_data_index', 'Name_Fix', 'Fn_Fix', 'Ln_Fix', 'county',
'new_state', 'country', 'name_type','Match Index', 'Match Status','Group Name', 'Group Town', 'Group
County', 'Group State', 'Group Country', 'Group Name Type','Group Match Index', 'Group Match
Status',]].to_markdown())
```

5. Next, we have a list of names (manually curated) that represent the same identity but are slightly misspelled that we deal with in the **Group Names - Fuzzy Matching**. These are names that the Ancestry.com APwe could not find matches for, or somehow missed. we import name_agg.csv (variable `rep_names`)
    1. Sometimes, a `location` value is needed to identify which individual we are referring to - as there can be multiple individuals with the same name.
    2. Note that we don't need to go through the same process as earlier of filtering different `name_type` values to figure out what location level we want because the `new` name column was manually curated. An algorithm that generates would have to consider this when deciding what name is going to be in the `original` column (which name is being replaced) and what name is going to be in the `new` (the replacement name).
        1. The manual curation process used techniques described in the **Name Matching Algorithms** section of the 2023 Handbook
    3. Here's an example of two entries in `rep_names` and the subsequent result. Notice how John Salter represents the identities of multiple individuals, so we specify a location
        1. `rep_names`

	original	new	location
0	Hannah Hawley	Hannah Howley	nan
12	John Saller	John Salter	Mansfield Center | Tollan County | CT | town

        2. `CD_merged_mind` post changes

	Name	state_data	state_data_index	Name_Fix	Fn_Fix	Ln_Fix	county	new_state	country	name_type	Match Index	Match Status	Group Name	Group Town	Group County	Group State	Group Country	Group Name Type	Gr M In
95	Hannah Hawley | Hannah Howley	CT	90	Hannah Hawley | Hannah Howley	Hannah	Hawley	Fairfield County	CT	US	town		No Match	Hannah Howley	Fairfield	Fairfield County	CT	US	town	
96	Hannah Hawley | Hannah Howley	CT	90	Hannah Hawley | Hannah Howley	Hannah	Howley	Fairfield County	CT	US	town		No Match	Hannah Howley	Fairfield	Fairfield County	CT	US	town	
474	John Saller | John Salter	CT	430	John Saller | John Salter	John	Saller	Tolland County	CT	US	town		No Match	John Salter	Mansfield Center	Tolland County	CT	US	town	41 41
475	John Saller | John Salter	CT	430	John Saller | John Salter	John	Salter	Tolland County	CT	US	town	411 | 412	2 Potential Matches Found	John Salter	Mansfield Center	Tolland County	CT	US	town	41 41
3121	John Salter	PA	709	John Salter	John	Salter	Philadelphia County	PA	US	county	1999	Complete Match	John Salter		Philadelphia County	PA	US	county	19

3.

```
# code
print(rep_names[rep_names['new'].apply(lambda x: x == 'Hannah Howley' or x == 'John
Salter')].to_markdown())
print(CD_merged_mind[CD_merged_mind['Group Name'].apply(lambda x: x == 'Hannah Howley' or x == 'John
Salter')][['Name', 'state_data', 'state_data_index', 'Name_Fix', 'Fn_Fix', 'Ln_Fix', 'county',
'new_state', 'country', 'name_type','Match Index', 'Match Status','Group Name', 'Group Town', 'Group
County', 'Group State', 'Group Country', 'Group Name Type','Group Match Index', 'Group Match
Status',]].to_markdown())
```

6. Another group of people we want to group together are those that have the same name, but different places of residence, which we handle in **Group Names - same name, within state**.These are individuals who have the same name and reside in the same state, but have different locations.
    1. In particular, these names, collectively, must have
        1. One unique `town` name and one unique `county` name (excluding cases where no town or county name is listed)
        2. At least one `name_type` (county + town, county + state, or town + state)
        3. No more than 2 different `county` names
    2. These ensure that we don't have individuals with contradicting information (for example, Bob Rob in Anne Arundel County, MD and Baltimore County MD) who clearly correspond to different identities
    3. Note that many of these individuals have overlapping or the same Ancestry.com match data. However, they were not covered in **Group Names - Using Ancestry.com Matches** because a) their names are the same and b) they might only have overlapping, as opposed to the same Ancestry.com match data
    4. Here's an example. Note that `Abner Johnson` has overlapping, as opposed to identical Ancestry.com match data because in the case where we have town information, we can identify his census record more precisely
        1. `dup_state`

	Group Name	Group State	Group County	Group Town	Group Name Type
34	Abigail Robbins	CT	2	2	2
45	Abner Johnson	CT	2	2	2

        2. `CD_merged_mind`

	Name	state_data	state_data_index	Name_Fix	Fn_Fix	Ln_Fix	county	new_state	country	name_type	Match Index	Match Status	Group Name	Group Town	Group County	Group State	Group Country	Group Name Type	Group Match Index
143	Abigail Robbins	CT	128	Abigail Robbins	Abigail	Robbins	Hartford County	CT	US	town	130	Complete Match	Abigail Robbins	Wethersfield	Hartford County	CT	US	town	130
938	Abner Johnson	CT	864	Abner Johnson	Abner	Johnson	New Haven County	CT	US	town	738	Complete Match	Abner Johnson	Waterbury	New Haven County	CT	US	town	738
1053	Abigail Robbins	CT	964	Abigail Robbins	Abigail	Robbins		CT	US	state	130	Complete Match	Abigail Robbins	Wethersfield	Hartford County	CT	US	town	130
1064	Abner Johnson	CT	975	Abner Johnson	Abner	Johnson		CT	US	state	738 | 807	2 Potential Matches Found	Abner Johnson	Waterbury	New Haven County	CT	US	town	738

```
# code to generate tables
print(dup_state[dup_state['Group Name'].apply(lambda x: x in ['Abigail Robbins', 'Abner
Johnson'])].to_markdown())
print(CD_merged_mind[CD_merged_mind.apply(lambda x: x['Group Name'] in ['Abigail Robbins', 'Abner
Johnson'] and x['Group State'] == 'CT', axis=1)][['Name', 'state_data', 'state_data_index', 'Name_Fix',
'Fn_Fix', 'Ln_Fix', 'county', 'new_state', 'country', 'name_type','Match Index', 'Match Status','Group
Name', 'Group Town', 'Group County', 'Group State', 'Group Country', 'Group Name Type','Group Match
Index', 'Group Match Status',]].to_markdown())
```

7. In **Create Group Columns and Group Data**, we group our individuals by identity (name + location + match data)
    1. Column explanations
        1. the `Name_Fix` column contains all "fixed names" associated with an individual
        2. the `Full Search Name` column contains all names used to search for this person on Ancestry.com, separated by `|`
        3. the `assets` column contains all of the assets asociated with this individual. Different debt entries are separated by `|`, each debt entry separates the index and assets by `:`, and 6%/6% deferred/3% stocks are separated by ","
        4. the `occupation` column contains all of the assets asociated with this individual, separated by `|`
    2.

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Name_Fix	Full Search Name	assets	occupation
4	Aaron Caldwell II	CT	Hartford County	Hartford	town		['Aaron Cadwell Ii, Aaron Caldwell Ii']	Aaron Cadwell Iwe | Aaron Caldwell Ii	CT_98 : 9.25, 4.63, 0.0	
8	Aaron Kelsey	CT	Middlesex County	Killingworth	town	423 | 424	['Aaron Kelsey']	Aaron Kelsey	CT_445 : 63.4, 31.7, 75.62	Farmer

```
# code
print(df_final.loc[[4,8]].drop('Group Match Url', axis = 1).to_markdown())
```

8. In **Impute Location - Corporations**, we use a clever trick to obtain more location information for people in our dataset
    1. In the example below, we see that Alexander Mowatt has his own entry in our final table, but holds assets as part of two partnerships: `Alexander Mowatt | John Mowatt, Alexander Mowatt | Michael Mowatt`. We can assume they are business partners (they're likely also related, although that's not relevant for our case)
    2. Moreover, we see that John Mowatt has the location `New York City | New York County | NY` while Alexander Mowatt only has the location `NY`. Our underlying assumption is that Alexander Mowatt is likely also located in NYC, so we impute his location with John Mowatt's location
    3. Here is how we identify these cases:
        1. We look at what partnerships an individual has within their state of residence (who else lives in NY and has a `Name_Fix` column with Alexander Mowatt's name)
        2. If the number of unique counties and names is greater than one (including the absence of a county as a unique value), then there is an opportunity to impute a more specific location
            1. Once again, we can only perform this process if we don't have contradicting information - so we cannot impute locations if two people live in different counties, or if they live in the same county but different towns
            2. We refer to these as "contradicting cases"
        3. We run a loop to do this multiple times until we only have "contradicting cases" left. Note that we cannot do this just once, we have to use a loop. Here's an example why.
            1. Suppose John and Robert are part of a partnership, and they both live in CT. In our first iteration, Robert, who is also in a partnership with William, is imputed with location New Haven, New Haven County, CT because William has that location. Now, it is likely that John, by virtue of being Robert's partner, is likely a resident of New Haven, New Haven County, CT. However, in the first round, John's location was not imputed but now, in the second round of the loop, his location will be imputed.
            2. If you want to see an example of this behavior, check out `Abram Belknap`.
            3. **Something to think about**: Do we also want to impute **occupation?** If John is a merchant, do we think Robert is also a merchant?
        4. Example
            1. Example (before)

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Name_Fix	Full Search Name	assets	occupation
133	Amasa Keyes	CT	Hartford County	Hartford	town	760	['Amasa Keyes | Elnathan Keyes', 'Amasa Keyes']	Amasa Keyes	CT_894 : 16.62, 8.31, 11.95 | CT_927 : 266.67, 0.0, 192.0 | CT_932 : 145.84, 72.92, 164.86	Executor to Stephen Keyes Deceased
911	Elnathan Keyes	CT			state	776	['Amasa Keyes | Elnathan Keyes']	Elnathan Keyes	CT_927 : 266.67, 0.0, 192.0	Executor to Stephen Keyes Deceased

            2. Example (after)

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Name_Fix	Full Search Name	assets	occupation
133	Amasa Keyes	CT	Hartford County	Hartford	town	760	['Amasa Keyes | Elnathan Keyes', 'Amasa Keyes']	Amasa Keyes	CT_894 : 16.62, 8.31, 11.95 | CT_927 : 266.67, 0.0, 192.0 | CT_932 : 145.84, 72.92, 164.86	Executor to Stephen Keyes Deceased
911	Elnathan Keyes	CT	Hartford County	Hartford	state	776	['Amasa Keyes | Elnathan Keyes']	Elnathan Keyes	CT_927 : 266.67, 0.0, 192.0	Executor to Stephen Keyes Deceased

3. Code has to be run before and after our process - see notebook for specific example of implementation

```
# code
print(df_final[df_final['Name_Fix'].apply(lambda x: any(['Elnathan Keyes' in ele for ele in
x]))].drop('Group Match Url', axis = 1).to_markdown())
```

9. Now, in the section **Cleaning Name_Fix**, we unify name spellings. Consider the example below. Notice how in the `Name_Fix` column, we have very similar names that correspond to the same identity (`Ebenezar Denny` and `Ebenezer Denny`). We want to unify these so that one identity is represented by one name (in particular, the name used for `Group Name`). From this, we can tell that Ebenezer Denny owned debt on his own, and with Edward Denny.
   1. Original output

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Name_Fix	Full Search Name	assets	occupation
738	Ebenezer Denny	PA			state		['Ebenezar Denny', 'Ebenezar Denny | Ebenezer Denny | Edward Denny']	Ebenezer Denny | Ebenezer Denny	PA_185 : 1330.73, 665.37, 449.96 | PA_219 : 0.0, 0.0, 1000.0	

   2. In our result, (second table, the one below), we see in the column `Name_Fix_Clean` that Ebenezer Denny belongs to two groups of debtholders: himself (Ebenezer Denny) and with Edward (Ebenezer Denny | Edward Denny). Groups are separated by `:` and individuals within a group are separated by `|`
   3. Moreover, in `Name_Fix_Transfer`, we have a mapping between the original `Name_Fix` value and the corresponding value in `Name_Fix_Clean`. Different mappings are separated by `:` and within a mapping, we have the format `value / key`. For example, `Ebenezer Denny  / Ebenezar Denny` shows that the original name `Ebenezar Denny` is now represented as `Ebenezer Denny`. Moreover, the original name `Ebenezar Denny | Ebenezer Denny | Edward Denny` is now represented as `Ebenezer Denny  | Edward Denny`, as shown by `Ebenezer Denny | Edward Denny  / Ebenezar Denny | Ebenezer Denny | Edward Denny`
   4. Output after cleaning

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Full Search Name	assets	occupation	Name_Fix_Transfer	Name_Fix_Clean
738	Ebenezer Denny	PA			state		Ebenezar Denny | Ebenezer Denny	PA_185 : 1330.73, 665.37, 449.96 | PA_219 : 0.0, 0.0, 1000.0		Ebenezer Denny / Ebenezar Denny : Ebenezer Denny | Edward Denny / Ebenezar Denny | Ebenezer Denny | Edward Denny	Ebenezer Denny : Ebenezer Denny | Edward Denny

   5.

```
# code
print(df_final[df_final['Group Name'] == 'Ebenezer Denny'].drop('Group Match Url', axis = 1).to_markdown())
```

10. Next, in **Manual Adjustments**, for some odd reason ,we have a few cases where the same identity appears twice. We fix these by combining those data points.
11. we do this manually, but it would be nice to code up a function that does it automatically
12. Another group of people we want to group together are those that have the same name, but different places of residence, which we handle in **Group Names - incorrect states** This occurs because when we impute values for the `state` column, sometimes the incorrect value is imputed (example: CT debt file, individual from MA but missing `state` column value but we impute CT as the state). In our example, we know this is an incorrect value because another individual from `MA` has the same name, an ancestry.com match, and (often times) also holds CT debt in the same file.
    1. In this case, we set the group characteristics to be whatever data is most specific (most specific `name_type` for location)
    2. The variable `state_group_names` contains our list of potential duplicates, imported from group_name_state.csv
    3. To identify these cases, we find all names that appear across multiple states, then make sure that there are strictless less counties than states (removing cases where counties don't exist). This ensures that there is at least one pair of name + state that does not contradict each other (if we have 2 different states + 2 different counies, the state value column was not imputed which tells us that those 2 names represent different identities)
       1. Then, we go through the list manually and using techniques from the **Name Matching Algorithms** section of the 2023 Handbook (and examining whether the two individuals in question both own debt from the same state debt file, of which affirmation is a positive sign)
    4. When both names only have location `name_type` at the state value, we input manually which state to aggregate data at (based on manual inspection)

    1. Otherwise, we pick the most specific `name_type`, as per our usual procedure. See example below
5. Before

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index
79	Adam Gilchrist	NY			state	
80	Adam Gilchrist	SC	Charleston County	Charleston	town	858

6. After

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index
79	Adam Gilchrist	SC	Charleston County	Charleston	town	858

```
# code
print(df_final[df_final['Group Name'] == 'Adam Gilchrist'][['Group Name', 'Group State', 'Group County',
'Group Town', 'Group Name Type', 'Group Match Index']].to_markdown())
```

13. Next, we move onto cleaning the dataset that contains Ancestry.com demographic data in section **Add Villages**. We import the dataset scrape_results.csv (variable `match_df`)
    1. The Ancestry.com Census data does not list a geography that can be classified as a town for most Philadelphia/Charleston/New York City residents; instead, it lists a subdivision of the city, such as `New York City East Ward`
    2. I create a separate category called `Match Village` for the relevant census data, and replace the `Match Town` column value with the corresponding city (Philadelphia, Charleston, New York City)
    3. Example of an entry that is a village

	Name	Home in 1790 (City, County, State)	Free White Persons - Males - 16 and over	Free White Persons - Females	Number of Household Members	Free White Persons - Males - Under 16	Number of Slaves	Number of All Other Free Persons	Match Type	Match Town	Match County	Match State	index_temp	index_new	Match Village
613	Thomas Vail	New York City East Ward, New York, New York	1	4	7	2			village	New York City	New York County	New York	648	613	New York City East Ward

```
print(match_df[match_df['Name'] == 'Thomas Vail'].to_markdown())
```

14. For some individuals in `match_df`, their `Name` column also includes an occupation. In the section **Get Occupations from Ancestry** I extract these occupations from the `Name` column. Most of these names are recognizable by the fact that they have a comma (`,`) or parentheses (`(` or `)`) which contains the occupation in question. You can also identify all possible instances where an occupation is in a name (and unfortunately ome false positives) if you filter for names that are over 3 words (at least 2 words for a name + occupation)
    1. Cleaning this was really tedious as we had to inspect the variety of situations above. However, it does yield us occupations for **400** individuals from `match_df`
    2. Before example

	Name	Home in 1790 (City, County, State)	Free White Persons - Males - 16 and over	Free White Persons - Females	Number of Household Members	Free White Persons - Males - Under 16	Number of Slaves	Number of All Other Free Persons	Match Type	Match Town	Match County	Match State	index_temp	index_new	Match Village
145	Comfort (Wd) Clock | Wd Combert Clock	Norwalk and Stamford, Fairfield, Connecticut	1	1	2				town	Norwalk and Stamford	Fairfield County	Connecticut	155	145	

    3. After example

	Name	Home in 1790 (City, County, State)	Free White Persons - Males - 16 and over	Free White Persons - Females	Number of Household Members	Free White Persons - Males - Under 16	Number of Slaves	Number of All Other Free Persons	Match Type	Match Town	Match County	Match State	index_temp	index_new	Match Village	Occupation
145	Combert Clock | Comfort Clock	Norwalk and Stamford, Fairfield, Connecticut	1	1	2				town	Norwalk and Stamford	Fairfield County	Connecticut	155	145		Wd

    4. Quite annoying...

```
# before and after code
print(match_df[match_df['Name'] == 'Comfort (Wd) Clock | Wd Combert Clock'].to_markdown())
print(match_df.loc[[145]].to_markdown())
```

15. Sometimes, our scraping algorithm assigns multiple matches to an individual because it is unsure about which census record matches our specified individual. However, due to both a) idiosyncracies with the scraping algorithm and b) idiosyncracies with Ancestry.com, sometimes (not often) we will have an individual who matches to multiple individuals, but has a direct name match with one of them. We resolve this in **Improve Scraper Match** by removing matches when we have a case where
    1. one identity matches to multiple census records
    2. one of those multiple census records is a direct name match with the associated identity
16. In **Eliminate Broad Location Matches**, we consider cases where an identity is matched to multiple Ancestry.com census records. Now that we have improved location data from **Impute Location - Corporations**, we can retroactively improve the specificity of the potential census records. Sometimes, due to idosyncracies of the scraping algorithm, we also have cases where we have more matches than we should nad this process can remove those.
    1. Here's an example of an improvement
        1. Original entry in main table

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Full Search Name	assets	occupation	Name_Fix_Transfer	Name_Fix_Clean
1775	John Davis	RI	Providence County	Providence	town	2862 | 2863 | 2864	John Davis	RI_657 : 219.76, 6.23, 35.4	Administrator	James Burrill | John Brown | John Davis | Mehitable Davis / James Burrill | John Brown | John Davis | Mehitable Davis	James Burrill | John Brown | John Davis | Mehitable Davis

        2. Match Data - We can keep just 2862 in this case.

	Name	Home in 1790 (City, County, State)	Free White Persons - Males - 16 and over	Free White Persons - Females	Number of Household Members	Free White Persons - Males - Under 16	Number of Slaves	Number of All Other Free Persons	Match Type	Match Town	Match County	Match State	index_temp	index_new	Match Village	Occupation
2862	John Davis	Little Compton, Kent, Rhode Island	2	4	9	3			town	Little Compton	Kent County	Rhode Island	3245	2862		nan
2863	John Davis	Glocester, Providence, Rhode Island	3	5	9	1			town	Glocester	Providence County	Rhode Island	3246	2863		nan
2864	John Davis	Providence, Providence, Rhode Island	3	4	7				town	Providence	Providence County	Rhode Island	3247	2864		nan

        3.

```
# code
print(df_final.loc[[1775]].drop('Group Match Url', axis = 1).to_markdown())
print(match_df.loc[[2862,2863,2864]].to_markdown())
```

    2. Here's an example of an idiosyncracy. Benjamin Gallup was originally matched to two records because originally we only had information on the county he lived in (note that Voluntown used to belong to Windham but now belongs to New London County). We can now filter to only include Match Index 638 because we know Benjamin Gallup resides in Voluntown.
        1. Original entry in main table

	Group Name	Group State	Group County	Group Town	Group Name Type	Group Match Index	Full Search Name	assets	occupation	Name_Fix_Transfer	Name_Fix_Clean
310	Benjamin Gallup	CT	New London County	Voluntown	town	638 | 639	Benjamin Gallup	CT_692 : 49.42, 24.72, 31.93	Farmer	Benjamin Gallup / Benjamin Gallup	Benjamin Gallup

        2. Match Data - We can just keep entry 638

	Name	Home in 1790 (City, County, State)	Free White Persons - Males - 16 and over	Free White Persons - Females	Number of Household Members	Free White Persons - Males - Under 16	Number of Slaves	Number of All Other Free Persons	Match Type	Match Town	Match County	Match State	index_temp	index_new	Match Village	Occupation
638	Benjn Gallop	Voluntown, Windham, Connecticut	3	11	15	1			town	Voluntown	Windham County	Connecticut	673	638		nan
639	Benjn Gallop	Plainfield, Windham, Connecticut	1	4	7	2			town	Plainfield	Windham County	Connecticut	674	639		nan

        3.

```
# code
print(df_final[df_final['Group Name'] == 'Benjamin Gallup'].drop('Group Match Url', axis = 1).to_markdown())
print(match_df[match_df['Name'] == 'Benjn Gallop'].to_markdown())
```

17. Next, in **Use Census to Impute Location** we have the opportunity to use the amazing information the Ancestry.com census provides us with.
    1. If an individual only has one match, and the Ancestry.com census location is more specific (ie: provides town geography when our data only has county geography) then we add the Ancestry.com census location to our dataset
    2. If an individual has multiple matches, we find the most detailed level of information for which all of the Ancestry.com census matches are the same. Similarly to earlier, when this aggregated census location is more specific (ie: provides town geography when our data only has county geography) then we add the aggregated census location to our dataset
    3. In cases where we have a location conflict at the state level (census state differs from given state), except for when the state equals `NY`, we remove the match
        1. NY is a special case because no state column was in NY (all location values were imputed + at state level), so we exclude it
    4. In cases where we have a location conflict at the county level (census county differs from given county), or a location conflict at the town level, we mark what level of location conflict we have and **do not** merge the ancestry.com location into our location for our final dataset
    5. In cases where we do not have a location conflict, and the census data was added, we **do** merge the ancestry.com location into our location for our final dataset
18. In **Occupation Column Cleaning**, I clean the `occupation` column by unifying the name format using a mapping. We worked on this last year and explored using automated solutions, but there was still a large (too large) degree of manual input required so I just exported the manually craeted mapping as occ_correction.csv and used that.
    1. We should look into using NLP solutions, which we did not do last year, to automate this process.
    2. Example of mapping table

	Original	Corrected
0	Merchant	Merchant
7	Trader	Merchant
12	Merchants	Merchant
44	Traders	Merchant
60	Marchant	Merchant
65	Charleston Merchants	Merchant
99	Traders In Company	Merchant
204	Merchents	Merchant
274	Mercht	Merchant

```
# code
print(occ_data[occ_data['Corrected'].apply(lambda x: x in ['Merchant'])].head(9)[['Original',
'Corrected']].to_markdown())
```

19. In **Reset Match Data Index**, we reset the index of our match data by removing entries that are no longer in the data, and then reindexing everything so that our match values go from 1... number of match values. This turns out to be a bit more complicated than just using the `.reset_index()` command becuase our two datasets are relational, but the code is essentially the same as in clean_3_scrape.ipynb
20. In **Aggregate Debt Totals**, we want to calculate an individual's total holdings of 6%, 6% deferred and 3% stock. We calculate two measures. The first measure is the sum of all of an individual's holdings. However, this measure does not take into account that multiple individuals can hold the same debt, so our second measure is the sum of an individual's holdings, where debt held by multiple people is divided into equal shares. For example, if Bob and George both hold $200 worth of debt together, then their second measure would have value $100
21. In **Final Data Export**, I export our final table (variable `df_final`) and the match table (variable `match_df`). For any individual, the `|` separated values in Group Match Index indicate their associated Ancestry.com census records, and correspond to the same value in the `index_new` column of `match_df`.

Other Things

The following links to state population data (statepop.csv: https://web.viu.ca/davies/H320/population.colonies.htm) and county population data (countyPopulation.csv: https://www.socialexplorer.com/tables/Census1790/R13347861) that may be helpful in understanding the proportion of individuals in a state who held debt.

To make maps, we use shapefiles. We have shapefiles at the county and state level.

- historicalcounties: IPUMS NHGIS, 1790 census data, 1790 County 2000 Tiger/Line GIS
- historicalstates: https://digital.newberry.org/ahcb/downloads/gis/US_AtlasHCB_StateTerr_Gen001.zip