
Ziptool

Release 0.1.0

Joshua Neronha

Apr 29, 2022

CONTENTS

1	Getting Started	3
1.1	Installation	3
1.2	Obtaining Data	3
1.3	Extracting ZIP-level Data	3
2	Why does this tool exist?	5
3	Modules	7
3.1	query_by_zip module	7
3.2	fetch_data module	8
3.3	geo_conversion module	8
3.4	interface module	9
3.5	utils module	10
4	Examples	11
4.1	Example 1	11
4.2	Example 2	15
	Python Module Index	19
	Index	21

This tool is designed to analyze microdata from the American Community Survey (ACS) on a ZIP-code level. In an effort to anonymize data, the Census Bureau (perhaps surprisingly) publishes microdata only down to the PUMA (Public Use Microdata Area) level, each of which contains 100,000 people.

However, researchers often want community data at the ZIP code level. Using some geographical tricks, Ziptool allows users to obtain approximate data at the ZIP code level by querying the PUMA(s) it is inside.

The primary function of interest is *data_by_zip*, which takes a dataset and ZIP codes of interest and returns either summary statistics or the full data. However, many subfunctions are also documented and can be used standalone.

Table of Contents

GETTING STARTED

1.1 Installation

Ziptool requires Python $\geq 3.8.0$. Install using `pip`:

```
pip install ziptool
```

1.2 Obtaining Data

Ziptool extracts information from ACS data. Data of the following forms can be used:

1. Downloaded, rectangular CSV that contains your variables of interest. You can register at <https://usa.ipums.org/> and download the data you would like. You must be sure to download rectangular data and include PUMA *and* STATEFIP, as variables, as the analysis relies on PUMA and state to convert to ZIP code.
2. Alternatively, you could use `ipumspy`! This very convenient package allows you to generate a `pd.DataFrame` for whatever data you would like and queries IPUMS directly using the API. This is the preferred method for convenience's sake, although both are equally supported.

1.3 Extracting ZIP-level Data

The top-level function used to extract data by ZIP code is `data_by_zip`. Assuming that you have downloaded your data as a CSV, you might use the following code to query data, assuming that you are interested in the education and household income for the zip codes 79901 and 02835.

```
from ziptool import data_by_zip

data_by_zip(['02835', '79901'], path_to_csv,
            {"HHINCOME": {"null": 9999999, "type": 'household'},
             "EDUC": {"null": 0, "type": 'individual'}})
```

You can provide as many ZIP codes as you would like, and as many variables, too. You must provide the null value for a variable and its type (household or individual), both of which are readily available from the IPUMS codebook.

If you performed the following query on the 2019 ACS, it would return the following `DataFrame`. You can easily pull out the statistics you would like.

	HHIN-COME_mean	HHIN-COME_std	HHIN-COME_median	EDUC_mean	EDUC_std	EDUC_median
02835	119943	135844	83000	7.36114	3.01399	7
79901	46493.5	57143.2	30000	5.51116	2.90709	6

The summary statistics for household income make sense! However, education is a categorical variable, so the summary statistics might be less useful. In that case, you might choose to provide no argument for `variables` to simply return the raw `pd.DataFrames` that you could analyze as you wish.

```
from ziptool import data_by_zip

data_by_zip(['02835', '79901'], path_to_csv, None)
```

Because no variables are specified, no analysis is performed – the function simply returns the dataframes for each PUMA within the ZIP code and its ratio (i.e. if some ZIP code 99999 is 75% within PUMA 1 and 25% within PUMA 2, the code would return one `pd.DataFrame` per PUMA along with the ratio.) The above code would return:

```
{'02835':
  YEAR SAMPLE SERIAL CBSERIAL HHWT CLUSTER STATEFIP ...
↪GQ HHINCOME BEDROOMS PERNUM PERWT EDUC EDUCD
  2542312 2019 201901 1125822 2019010002705 77.0 2019011258221 44 ...
↪4 9999999 0 1 77.0 11 115
  2542335 2019 201901 1125845 2019010007698 45.0 2019011258451 44 ...
↪4 9999999 0 1 45.0 7 71
  ... ... ... ... ... ... ...
↪ 2552708 2019 201901 1130856 2019001405444 44.0 2019011308561 44 ...
↪1 78910 5 2 35.0 11 115

'79901':
  YEAR SAMPLE SERIAL CBSERIAL HHWT CLUSTER STATEFIP ...
↪GQ HHINCOME BEDROOMS PERNUM PERWT EDUC EDUCD
  2681259 2019 201901 1189446 2019010001158 39.0 2019011894461 48 ...
↪3 9999999 0 1 39.0 2 23
  2681291 2019 201901 1189478 2019010001549 7.0 2019011894781 48 ...
↪3 9999999 0 1 7.0 6 63
  ... ... ... ... ... ... ...
↪ 2952760 2019 201901 1302906 2019001405840 105.0 2019013029061 48 ...
↪1 11000 4 1 105.0 6 63
```

02835 and 79901 both exist exclusively inside one PUMA, so no ratios are provided (since they are one). The `pd.DataFrame` for each ZIP code can easily be queried from the return dictionary and used as any other `pd.DataFrame` for analysis of your choice.

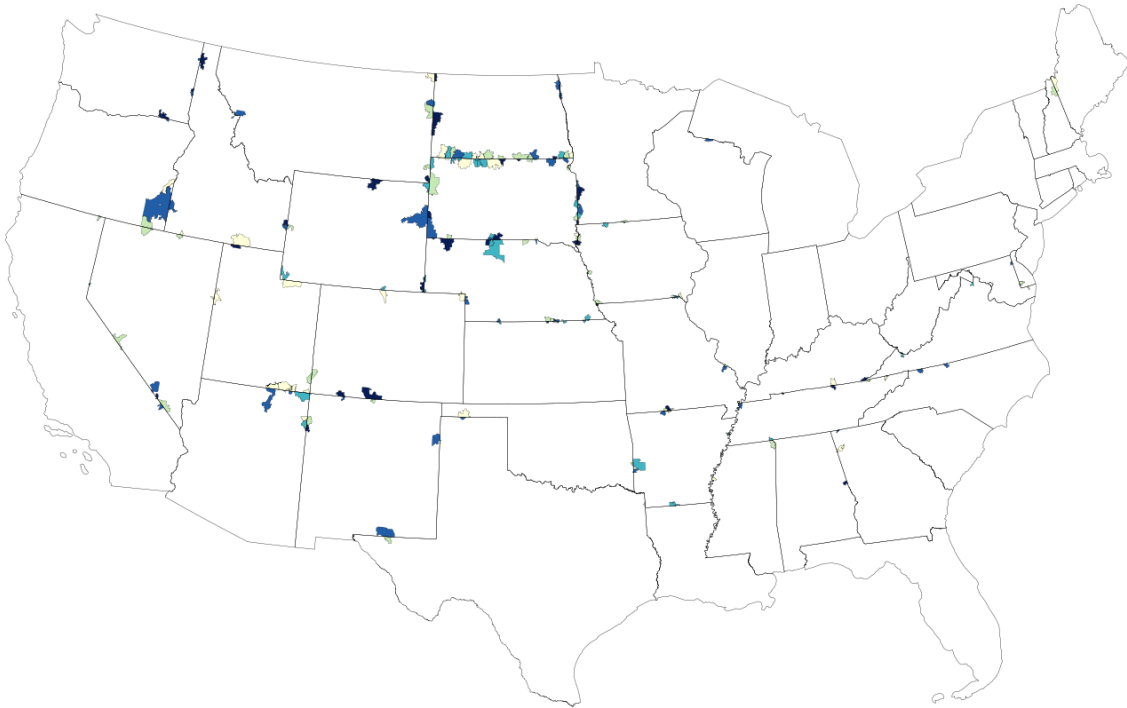
WHY DOES THIS TOOL EXIST?

ZIP codes are ubiquitous in daily lives – mailing packages, verifying your ZIP code when using your credit card, using Google Maps, and so on. So you would think that the Census Bureau (which conducts, of course, the Census, but also supplemental surveys like the American Community Survey, or ACS) would report its data on a ZIP code basis. And to some degree, it does. You can [look up data](#) for any ZIP code you would like right on the Census Bureau’s website.

However, this published data is all aggregated. Sometimes, as researchers, we want the microdata, which is the raw data collected from the ACS. However, by federal law, the Census Bureau must [take precautions](#) to avoid publishing potentially identifying information for 72 years after data collection. So while they do helpfully publish microdata from the ACS, they do not identify the ZIP code of the respondent. This is because some ZIP codes are extremely small – 05141, for example, represents Cambridgeport, Vermont, a town of only 112 people! And since the American Community Survey literally collects every piece of demographic information you can imagine (ancestry, how many cars you have, what kind of broadband you have, etc.) it wouldn’t be too hard to figure out exactly the identity of a respondent.

The other reason the Census Bureau doesn’t publish data on a ZIP code basis is that ZIP codes actually are not geographical areas – they represent mail routes. They’ve sort of been retooled and misused for geographic areas, the same way Social Security numbers have ended up being used as federal identifiers despite that not at all being the intention. Because of this, ZIP codes aren’t quite as geographically clean as we think they are. For example, some ZIP codes refer not to locations but mail routes – 02912, for example, simply represents Brown University’s central mail room from which mail is distributed across the university by Brown itself and not the postal service. And some ZIP codes even span multiple states, as shown below!

Multi-State ZIP Codes



For both of the above reasons, the Census Bureau publishes microdata not by ZIP code, but by Public Use Microdata Areas, commonly known as PUMAs. These are actual geographic regions that are designed to have 100,000 residents each, which the Census Bureau deems large enough to maintain anonymity. That means that we can obtain approximate microdata data for a ZIP code by querying its PUMA – should be easy, right?

Well, not quite. As we discussed before, ZIP codes are not geographic areas, but mail routes, so it's not like there's a one-to-one conversion between PUMA and ZIP. Thankfully, the U.S. Department of Housing and Urban Development (HUD) publishes the [HUD-USPS ZIP Code Crosswalk](#) which allows us to convert ZIP codes to census tracts. This is good because we now have actual geographical boundaries for a given ZIP code, but census tracts are distinct from PUMAs and again are not 1:1 related. That means the final step is computing the geographical intersection between census tracts and PUMA using shape files provided by the Census Bureau. At this point, we can query approximate data by ZIP code from the American Community Survey.

So, in summary, what does this tool actually do? It works through these messy geographical conversions (ZIP -> census tract -> PUMA) to pull out approximate data from the ACS on a ZIP code level, all without you having to think about this geographical weirdness.

MODULES

3.1 query_by_zip module

This module contains the primary functions of `ziptool`, `query_by_zip`. You don't need to use any other function to get data by ZIP code, but they are documented regardless for special use cases.

`ziptool.query_by_zip.data_by_zip(zips: List[str], acs_data, variables=None, year='2019')`

Extracts data from the ACS pertaining to a particular ZIP code. Can either return the full raw data or summary statistics.

Parameters

- **zips** – a list of zipcodes, represented as strings i.e. ['02906', '72901', ...]
- **acs_data** – a string representing the path of the datafile OR a dataframe containing ACS datafile
- **variables** (*optional*) – To return the raw data, pass `None`. To extract summary statistics, pass a dictionary of the form:

```
{
  variable_of_interest_1: { #the variable name in IPUMS
    "null": null_val, #the value (float or int) of null data
    "type": type #"household" or "individual"
  },
  variable_of_interest_2: {
    "null": null_val,
    "type": type
  }
}
```

- **year** (*optional*) – a string representing the year of shapefiles to use for matching PUMAs to ZIPs. Default is 2019.

Returns

When variables of interest are passed, a `pd.DataFrame` containing the summary statistics for each ZIP code.

When variables of interest are NOT passed, a dictionary of the form:

```
{
  zip_1: [
    [
```

(continues on next page)

(continued from previous page)

```
        puma_1_df,  
        puma_1_ratio  
    ],  
    [  
        puma_2_df,  
        puma_2_ratio  
    ],  
    ...,  
    ],  
    zip_2: ...  
}
```

3.2 fetch_data module

This module contains helper functions used to obtain shape files required to determine the intersections of Census tracts, PUMAs, and ZIP codes. Downloaded data is stored in the computer's temporary cache and should be deleted automatically at termination.

`ziptool.fetch_data.download_file(url: str, output_filename: Union[str, pathlib.Path], session: Optional[requests.sessions.Session] = None)`

Downloads a file from the provided URL and saves it at the desired path.

Parameters

- **url** – a string representing the URL of the file you want to download
- **output_filename** – a FilenameType representing the desired download path

Returns None

`ziptool.fetch_data.get_shape_files(state_fips_code, year)`

For a given state (in particular its FIPS code), downloads its census tracts and PUMA shapefiles from the Census Bureau. The function skips the download if the file already has been fetched!

Parameters

- **state_fips_code** – string representing the state of interest
- **year** – string representing the year

Returns Saves .shp files for both PUMA and census tracts within the data directory.

3.3 geo_conversion module

`ziptool.geo_conversion.get_state_intersections(state_fips_code: str) → geopandas.geodataframe.GeoDataFrame`

For a given state, computes the intersections between Census tracts and PUMAs. Note that you must run `fetch_data.get_shape_files()` before using this function.

Parameters **state_fips_code** – a string representing the FIPS code of the state of interest.

Returns a geopandas dataframe detailing the intersections of tracts and PUMAs for a state

`ziptool.geo_conversion.tracts_to_puma(tracts, state_fips_code: str)`

Takes in a list of tracts and ratios for a given zip code (in a given state) and returns the PUMAs composing the ZIP code with ratios (i.e. 88% in PUMA 00101 and 12% in PUMA 00102).

Parameters

- **tracts** – a 2D list generated by `zip_to_tract` containing census tracts and `weighted_ratios`
- **state_fips_code** – string representing state of interest

Returns series containing ratio of population for each PUMA

`ziptool.geo_conversion.zip_to_tract(zipcode: Union[str, int]) → Tuple[Tuple[List[str], List[float]], str]`

For a given ZIP code, uses HUD Crosswalk data (https://www.huduser.gov/portal/datasets/usps_crosswalk.html) to find the ratio of persons in each census tract for the given ZIP code.

Parameters **zip** – the five-digit ZIP code of interest, written as a string

Returns List containing the same number of entries as census tracts within the ZIP code. Each entry is a list, entry 0 is the census tract and entry 1 is the residential ratio of the census tract within that ZIP.

3.4 interface module

This module is how we connect the PUMAs and their ratios for a given ZIP code to the ACS dataset. It is called for each ZIP code to return the summary statistics or dataframe for that specific ZIP code.

`ziptool.interface.get_acs_data(file: Union[str, pathlib.Path, pandas.core.frame.DataFrame],
state_fips_code: Union[int, str], pumas: List[str], variables:
Optional[Dict[str, str]] = None)`

Pulls ACS data from a given file and extracts the data pertaining to a particular ZIP code. Can either return the full raw data or summary statistics.

Parameters

- **file** – a path to a datafile OR a dataframe containing ACS datafile
- **variables** – To extract summary statistics, pass a dictionary of the form:

```
{
    variable_of_interest_1: {
        "null": null_val,
        "type": type
    },
    variable_of_interest_2: {
        "null": null_val,
        "type": type
    }...
}
```

`variable_of_interest`: the variable name you wish to summarize `null_val`: the value, as a float or integer, of null values to filter out. `type`: “household” or “individual”, depending on the variable type

To return the raw data, pass `None`.

- **state_fips_code** – an integer (or two-digit representation thereof) representing the state of interest’s FIPS codes

- **pumas** – each PUMA of interest within the state and its ratio (returned by `geo_conversion.tracts_to_puma`)

Returns

When variables of interest are passed, a `pd.DataFrame` containing the summary statistics.

When variables of interest are NOT passed, a dictionary of the form:: {puma_1: [puma1_df, ratio1], puma_2: [puma2_df, ratio2]... }

3.5 utils module

Some simple utilities for handling data interaction.

`ziptool.utils.cast_fips_code(state_fips_code: Union[str, int]) → str`

FIPS codes are technically two-digit strings representing a number between 0 and 99, e.g., “01” and “44”. But it is very common that people pass them as integers. This guarantees they always appear as two-digit strings.

Parameters `state_fips_code` – A FIPS code as either a string or an int

Returns The appropriately styled version of the code

`ziptool.utils.cast_zipcode(zipcode: Union[str, int]) → str`

ZIP codes are five-digit strings, but it is very common that people pass them as integers. This guarantees that they always appear as five-digit strings

Parameters `zipcode` – A ZIP code

Returns The appropriately styled version of the code

`ziptool.utils.get_fips_code_from_abbr(state: str) → str`

Given a state postal abbreviation, e.g., “RI”, return its FIPS code, e.g., “44”

Parameters `state` – The abbreviation of the state

Returns The FIPS code of the state

Raises:

`ziptool.utils.puma_shapefile_name(state_fips_code: Union[str, int]) → str`

Return the expected filename of the PUMA shapefile. Note that this assumes a 2019 filename.

Parameters `code` – The FIPS code for the state of interest

Returns The expected filename

`ziptool.utils.tract_shapefile_name(state_fips_code: Union[str, int]) → str`

Return the expected filename of the tract shapefile. Note that this assumes a 2019 filename.

Parameters `code` – The FIPS code for the state of interest

Returns The expected filename

EXAMPLES

4.1 Example 1

Ziptool can be used with downloaded CSV files of ACS data, but it works best with `ipumspy`, a Python package that uses your IPUMS API key to pull data directly. The example below details a sample use case of `ziptool` for basic demographic research using `ipumspy` and both implementations of `data_by_zip`.

In this example, we want to explore various demographic traits of five coastal New England towns throughout the region: Jamestown, RI (02835); Kennebunkport, ME (04046); New Bedford, MA (02740); Stonington, CT (06355); and Westerly, RI (02804). We are particularly interested in household income and ancestry.

4.1.1 Setup

First, import `pandas`, `numpy`, and some other important dependencies.

```
import pandas as pd
import numpy as np
from pathlib import Path
```

We give two options for pulling data. Using `ipumspy` is recommended as it is much easier to use, but importing CSVs is fully supported as well.

4.1.2 Option 1: Manually Pulling Data

1. Go to <https://usa.ipums.org/usa/> and create a free account.
2. Click the “Select Data” tab.
3. Click “Select Samples” to select the year of ACS data you are interested in

(`ziptool` only supports one year at a time).

4. Under the “Select Harmonized Variables” dropdown, choose the variables you would like. Be sure to add “PUMA” and “STATEFIP” under the Household -> Geographic tab
5. Hit “View Cart” then select “Create Data Extract.” Select `.csv` as the data format and rectangular under structure.
6. Hit submit extract and wait until it is finished so you can download!

Once you have the data downloaded, simply pass the path to the CSV as an argument in `data_by_zip`. `Ziptool` will handle the import for you.

The rest of the tutorial will use the `ipumspy` option because of its ability to import and parse the associated codebook, which we need in this example.

4.1.3 Option 2: Pulling Data with ipumspy

Import ipumspy and the modules we need explicitly.

```
import ipumspy
from ipumspy import IpumsApiClient, UsaExtract, readers, ddi
```

Then, using the API key, we request the variable we are interested in ('HHINCOME' and 'ANCESTR1') along with 'PUMA' and 'STATEFIP', both of which are required variables for usage with ziptool. We also would like to get data from the 2019 ACS, which is labeled in ipums as 'us2019a'. The request is then submitted and downloaded (note that this can take quite a while depending on how many variables you request.)

```
IPUMS_API_KEY = your_api_key
DOWNLOAD_DIR = Path(your_download_dir)

ipums = IpumsApiClient(IPUMS_API_KEY)

extract = UsaExtract(
    ["us2019a"],
    ["STATEFIP", "PUMA", "HHINCOME", "ANCESTR1"],
)
ipums.submit_extract(extract)
ipums.wait_for_extract(extract)
ipums.download_extract(extract, download_dir=DOWNLOAD_DIR)
```

4.1.4 Continuous Variables

Now all the data needed for analysis is downloaded, and we can read it in as a `pd.DataFrame` along with the codebook that contains the information associated with each variable so that we can properly conduct our analysis.

```
ddi_file = list(DOWNLOAD_DIR.glob("*.xml"))[0]
ddi = ipumspy.readers.read_ipums_ddi(ddi_file)

ipums_df = ipumspy.readers.read_microdata(ddi,
    DOWNLOAD_DIR / ddi.file_description.filename)
```

In this example, we want to analyze two different traits for these communities: mean household income and reported ancestry. The former is a numerical ratio variable whereas the latter is categorical. That means that we can take advantage of ziptool's built-in analysis functions for HHINCOME but will read in the raw data for the categorical data of 'ANCESTR1'. We import the relevant modules of ziptool, `data_by_zip` (which will calculate the ZIP-level data) and `convert_to_df` (which will convert the returned data into a `pd.DataFrame` for easier analysis). Because we only want to analyze HHINCOME using summary statistics, we pass *only* 'HHINCOME' as a variable of interest. The null value comes from the codebook, as does the type (household vs. individual variable).

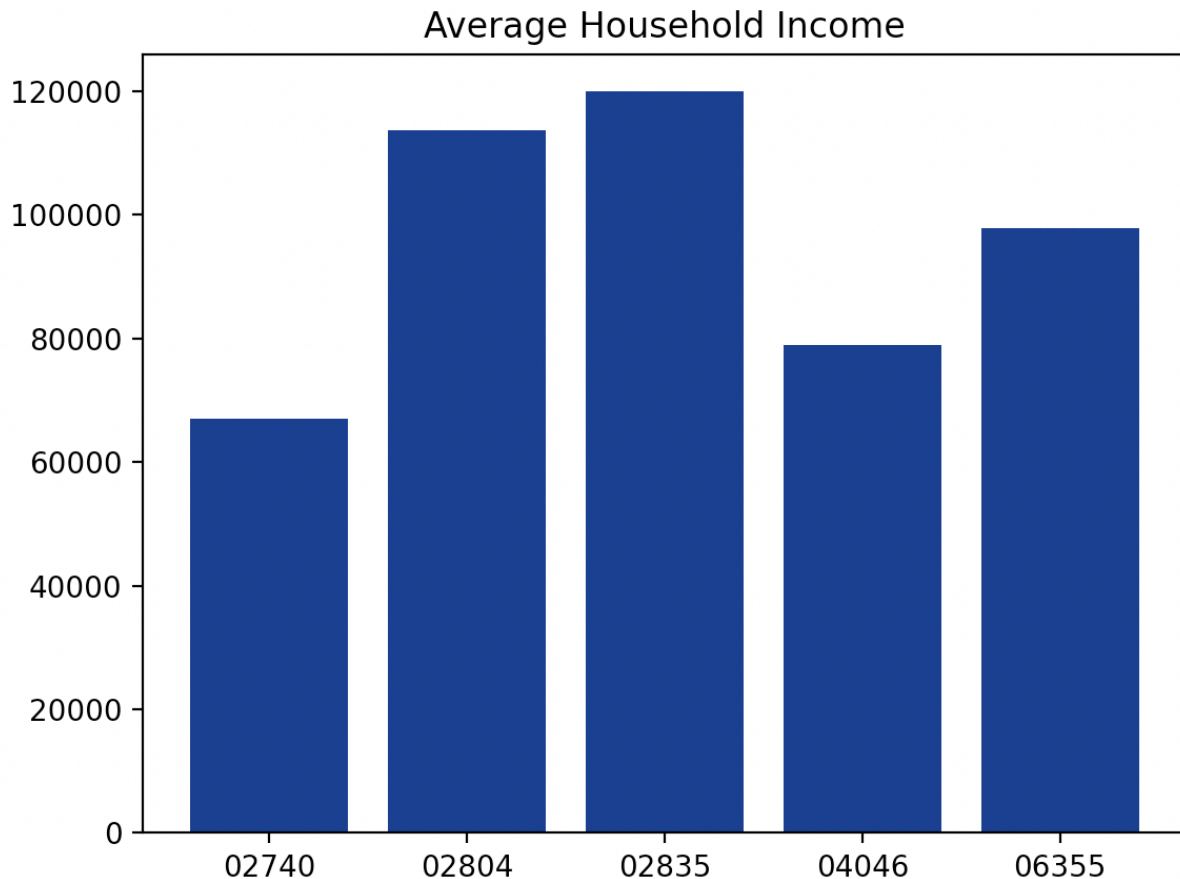
```
from ziptool.query_by_zip import data_by_zip
from ziptool.utils import convert_to_df

income_data = data_by_zip(['02835', '04046', '02740', '06355', '02804'], ipums_df,
    {"HHINCOME": {"null": 9999999, "type": "household"}})
```

We now have a `pd.DataFrame`, `income_data`, that contains all of our data! We can easily generate a bar plot to visualize differences by income as an example of the easy analysis that we can now perform.


```
import matplotlib.pyplot as plt
ylgnbu = ['#7fcdbb', '#41b6c4', '#225ea8',
          '#0c2c84', '#f29c33', '#666462']
#defining our colorscale

plt.bar(income_df.index, income_df['HHINCOME_mean'], color = ylgnbu[3])
plt.title('Average Household Income')
plt.show()
```



4.1.5 Categorical Variables

Categorical variables like ANCESTR1 are not usefully summarized by summary statistics, so in this case, we can read in the raw data and perform our own analysis. We do this by simply not specifying any variables:

```
raw_dfs = data_by_zip(['02835', '04046', '02740', '06355', '02804'], ipums_df)
```

We are particularly interested in four ancestral groups that often formed much of the populations of some coastal New England towns in the late 1800s : people of Portuguese, Irish, Italian, and English ancestry. However, countries are encoded as numbers in 'ANCESTR1' from the ACS, so we must access the codebook to pull out the codes corresponding to the ancestries we are interested in.

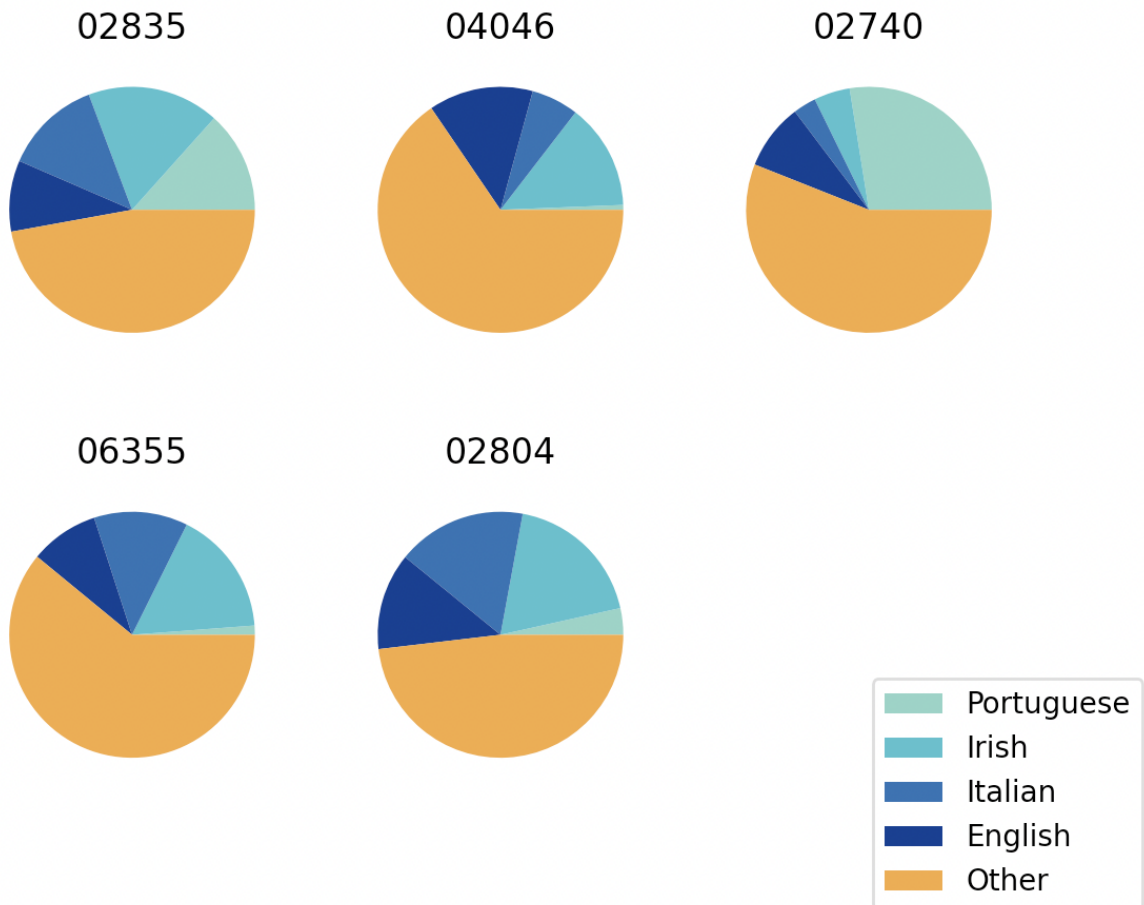
```
ancestry_info = ddi.get_variable_info('ANCESTR1')
ancestry_codes = ancestry_info.codes
top_codes = [ancestry_codes['Portuguese'],
             ancestry_codes['Irish, various subheads'],
             ancestry_codes['Italian'],
             ancestry_codes['English']]
```

We can now plot a pie chart of each ZIP code's ancestry demographics:

```
fig, ax = plt.subplots(2,3)

for i,zip in enumerate(['02835','04046','02740','06355','02804']):
    row = int(np.floor(i/3))
    column = int(i % 3)
    data = raw_dfs[zip]
    ancestry_data= data.groupby('ANCESTR1').sum()['PERWT']
    other = pd.Series([ancestry_data.loc[~ancestry_data.index.isin(top_codes + [ancestry_
↪ codes['Not Reported']])] .sum()),index=[0])
    to_plot = ancestry_data[top_codes].append(other)
    ax[row,column].pie(to_plot, colors = ylgnbu)
    ax[row,column].set_title(zip)

ax[1,2].axis('off')
fig.legend(['Portuguese','Irish','Italian','English','Other'], loc = 4)
plt.show()
```



And just like that, we have analyzed our categorical variable! You can manipulate, analyze, and visualize display data like you normally would with the ZIP-level data in a standard `pd.DataFrame`!

4.2 Example 2

Here, we are interested in comparing the correlations between different household variables for different variables. We will use `ipumspy` and `ziptool` to easily extract this data and generate correlation matrices.

4.2.1 Setup

First, import `pandas`, `ziptool`, `ipumspy` and some plotting mechanisms.

```
import sys
from configuration import *
import pandas as pd
from pathlib import Path
import matplotlib.pyplot as plt
import seaborn as sns
import ipumspy
from ipumspy import IpumsApiClient, UsaExtract, readers, ddi
```

(continues on next page)

(continued from previous page)

```
from ziptool.query_by_zip import data_by_zip
```

First, we pull our data using `ipumspy`.

4.2.2 Fetching Data

Using our API key, we request the variables we are interested in (a selection of household dwelling and economic indicators) along with ‘PUMA’ and ‘STATEFIP’, both of which are required variables for usage with `ziptool`. We also would like to get data from the 2019 ACS, which is labeled in `ipums` as ‘us2019a’. The request is then submitted and downloaded (note that this can take quite a while).

```
IPUMS_API_KEY = your_api_key
DOWNLOAD_DIR = Path(your_download_dir)

ipums = IpumsApiClient(IPUMS_API_KEY)

extract = UsaExtract(
    ["us2019a"],
    ["STATEFIP", "PUMA", "HHINCOME", "ROOMS", "BEDROOMS", \
     "CHISPEED", "FUELHEAT", "VEHICLES", "VALUEH"],
)
ipums.submit_extract(extract)
ipums.wait_for_extract(extract)
ipums.download_extract(extract, download_dir=DOWNLOAD_DIR)
```

4.2.3 Analyzing The Data

Now all the data needed for analysis is downloaded, and we can read it in as a `pd.DataFrame` along with the codebook that contains the information associated with each variable so that we can properly conduct our analysis.

```
ddi_file = list(DOWNLOAD_DIR.glob("*.xml"))[0]
ddi = ipumspy.readers.read_ipums_ddi(ddi_file)

ipums_df = ipumspy.readers.read_microdata(ddi,
    DOWNLOAD_DIR / ddi.file_description.filename)
```

First, we define the null values of each of our variables (obtained from the IPUMS online codebook).

```
HHINCOME_null = 9999999
BEDROOMS_null = 0
ROOMS_null = 0
CHISPEED_null = 0
FUELHEAT_null = 0
VEHICLES_null = 0
VALUEH_null = 9999999
```

Then, we define a function called `compare_variables` which takes in a list of zip codes and computes / plots the cross-correlation matrix between each variable of interest. We first use `ziptool`’s `data_by_zip` function to return the raw dataframes for each ZIP code (we do not want any intermediate analysis or summary statistics) and remove the null

values for each zip code; we then pre-process the CIHSPEED variable to transform it from a categorical variable to a binary variable (i.e. does a household have broadband access, not what kind). Then, we generate a heatmap and plot!

```
def compare_variables(zips):

    fig, axes = plt.subplots(1,len(zips), figsize = (12,4))

    df = data_by_zip(zips, ipums_df)

    for index, (zip, value) in enumerate(df.items()):

        mask = [value["BEDROOMS"] > BEDROOMS_null] and [value["ROOMS"] > ROOMS_null] and \
        [value["CIHSPEED"] > CHISPEED_null] and [value["FUELHEAT"] > FUELHEAT_null] and \
        [value["VEHICLES"] > VEHICLES_null] and [value["VALUEH"] != VALUEH_null] and \
        [value["HHINCOME"] != HHINCOME_null ]

        filtered = value[mask[0]]

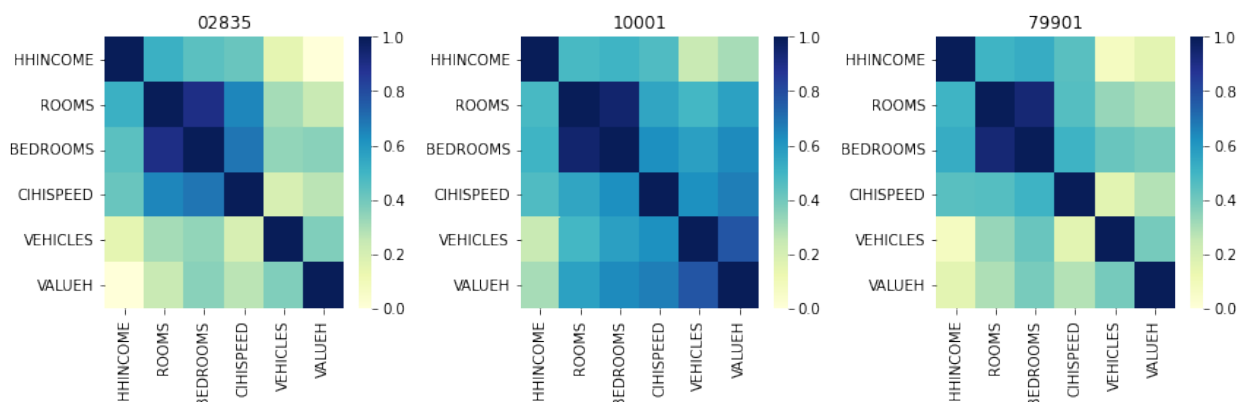
        oneperson = filtered[filtered['PERNUM'] == 1]
        oneperson['CIHSPEED'] = oneperson['CIHSPEED'].replace(20,0)
        oneperson['CIHSPEED'] = oneperson['CIHSPEED'].replace([10,11,12,13,14,15,16,
        17],1)

        sns.heatmap(oneperson[["HHINCOME", "ROOMS", "BEDROOMS", "CIHSPEED", "VEHICLES",
        "VALUEH"]].multiply(oneperson['HHWT'],axis = 'index').corr(), cmap = 'YlGnBu', vmin = 0, vmax = 1, ax = axes[index])
        axes[index].set_title(zip)

    plt.tight_layout()
```

Now that the function is defined, we can simply call it for whichever variables we are interested in.

```
compare_variables(['02835', '10001', '79901'])
```



Boom! We've used ziptool to extract ZIP-level data which we can use to perform advanced geographical analyses for any variables we would like.

Indices and tables

- `genindex`

- modindex
- search

PYTHON MODULE INDEX

Z

- `ziptool.fetch_data`, [8](#)
- `ziptool.geo_conversion`, [8](#)
- `ziptool.interface`, [9](#)
- `ziptool.query_by_zip`, [7](#)
- `ziptool.utils`, [10](#)

INDEX

C

`cast_fips_code()` (in module `ziptool.utils`), 10
`cast_zipcode()` (in module `ziptool.utils`), 10

module, 7
`ziptool.utils`
module, 10

D

`data_by_zip()` (in module `ziptool.query_by_zip`), 7
`download_file()` (in module `ziptool.fetch_data`), 8

G

`get_acs_data()` (in module `ziptool.interface`), 9
`get_fips_code_from_abbrev()` (in module `ziptool.utils`),
10
`get_shape_files()` (in module `ziptool.fetch_data`), 8
`get_state_intersections()` (in module `ziptool.geo_conversion`), 8

M

module
 `ziptool.fetch_data`, 8
 `ziptool.geo_conversion`, 8
 `ziptool.interface`, 9
 `ziptool.query_by_zip`, 7
 `ziptool.utils`, 10

P

`puma_shapefile_name()` (in module `ziptool.utils`), 10

T

`tract_shapefile_name()` (in module `ziptool.utils`), 10
`tracts_to_puma()` (in module `ziptool.geo_conversion`),
8

Z

`zip_to_tract()` (in module `ziptool.geo_conversion`), 9
`ziptool.fetch_data`
 module, 8
`ziptool.geo_conversion`
 module, 8
`ziptool.interface`
 module, 9
`ziptool.query_by_zip`