PharmaSUG 2025 – Paper HT-353

What's black and white and sheds all over? The Python Pandas DataFrame, the Open-Source Data Structure Supplanting the SAS® Data Set

Troy Martin Hughes

ABSTRACT

Python is a general-purpose, object-oriented programming (OOP) language, consistently rated among the most popular and widely utilized languages, owing to powerful processing, user-friendly syntax, and an abundant open-source community of developers. The Pandas library has become Python's predominant analytic toolkit, in large part due to the flexibility and success of its principal built-in data structure, the *Pandas DataFrame*. Akin to the SAS® data set, the DataFrame stores tabular data in columns and rows (i.e., variables and observations, in SAS parlance). And just as built-in SAS procedures, functions, subroutines, and statements manipulate and transform SAS data sets to deliver analytic insight and business value, so, too, do built-in Pandas methods, functions, and statements deliver similar functionality. However, where the similarities end, the DataFrame inarguably outshines and outpaces the SAS data set. A data set supports only character data, numeric data, and the hash object, whereas a DataFrame additionally can contain built-in lists, sets, tuples, and dictionaries—complex data structures unsupported by SAS. Moreover, whereas the sorting, transformation, and analysis of a SAS data set might require a PROC SORT, then a DATA step, and then another procedure, Python can deliver this functionality in a single line of readable code through *method chaining*—basic OOP syntactical design unsupported by the procedural Base SAS language. What's more, Python does all this for free!!! This text is intended for SAS practitioners interested in exploring Python data transformation and analysis, and leverages US Census data and the Centers for Disease Control and Prevention (CDC) United States Diabetes Surveillance System (USDSS) dashboard to ingest, clean, transform, investigate, and analyze diabetes and obesity data. All examples are first demonstrated in SAS 9.4, after which the functionally equivalent Python 3.13.2 and Pandas 2.2.3 syntax is explored in detail. Base SAS remains an incredibly formidable language and should not be disparaged, but where salient limitations exist, even the most dedicated SAS practitioner deserves to understand the open-source alternatives that can overcome these frustrating gaps.

INTRODUCTION

To run the examples in this scenario, a primary folder should be established, in which SAS and Python programs will reside, and in which subfolders will be created. Within this text, the following folder is the base folder in which files and subfolders will be maintained:

```
c:\shedder\
```

Create these subordinate folders for Census and CDC data:

```
c:\shedder\Census\
c:\shedder\CDC\
c:\shedder\tables\
```

SAS macro variables should be initialized to represent the base folder, Census folder, and CDC folder:

```
%let path_base=c:\shedder\;
%let path_census=&path_base.census\;
%let path_cdc=&path_base.cdc\;
```

Switching to Python, two libraries are imported—**`os`**, which contains operating system (OS) information and functionality, and **`pandas`**, the data analytic toolkit. By convention, Pandas is imported using the alias **`pd`**:

```
import os
import pandas as pd
import csv
```

At this point, the Python program should be saved to the primary folder:

```
c:\shedder\my_little_python.py
```

Global Python variables are initialized to the following folder locations, and are equivalent to the preceding SAS global macro variables:

```
path_base='c:\shedder'
path_census=os.path.join(path_base,'census')
path_cdc=os.path.join(path_base,'CDC')
```

The first thing to note is that Python is a case-sensitive language, unlike Base SAS, so **`import os`** is not the same as **`IMPORT OS`**. It is for this reason that Python language elements (e.g., variable names, methods, functions, statements) are printed using **`Courier New`** typeface and bold font, whereas SAS publications more commonly print built-in SAS language elements in uppercase.

Also note that Python statements are not terminated with those pesky semicolons. Finally, comments in Python are prefaced by octothorps (#) rather than asterisks (*).

The following files should be downloaded:

- From the US Census (https://www2.census.gov/programs-surveys/popest/datasets/2020-2021/counties/totals/), download the CSV file (co-est2021-alldata.csv) to the Census folder.
- From the US Census (https://www2.census.gov/programs-surveys/popest/datasets/2020-2021/state/totals/), download CSV file (NST-EST2021-alldata.csv) to the Census folder.
- From the US Census (https://www2.census.gov/programs-surveys/popest/geographies/2017/all-geocodes-v2017.xlsx), download the XLSX workbook to the Census folder; subsequently save this workbook as a CSV file: all-geocodes-v2017.csv.
- CDC obesity data are downloaded in a subsequent section.

With this brief setup, extract-transform-load (ETL) and data analysis can commence!

INGESTING NATIONAL, STATE, AND COUNTY POPULATION DATA

National- and state-level population estimates for 2021 are maintained in NST-EST2021-alldata.csv, as demonstrated in Table 1.

	A	B	C	D	E	F	G	H
1	SUMLEV	REGION	DIVISION	STATE	NAME	ESTIMATESBASE2020	POPESTIMATE2020	POPESTIMATE2021
2	10	0	0	0	United States	331449281	331501080	331893745
3	20	1	0	0	Northeast Region	57609148	57525633	57159838
4	20	2	0	0	Midwest Region	68985454	68935174	68841444
5	20	3	0	0	South Region	126266107	126409007	127225329
6	20	4	0	0	West Region	78588572	78631266	78667134
7	40	3	6	1	Alabama	5024279	5024803	5039877
8	40	4	9	2	Alaska	733391	732441	732673

Table 1. National- and State-Level Population Estimates for 2021

Similarly, county-level population estimates for 2021 are maintained in co-est2021-alldata.csv, as demonstrated in Table 2.

	A	B	C	D	E	F	G	H	I	J
1	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	ESTIMATESBASE2020	POPESTIMATE2020	POPESTIMATE2021
2	40	3	6	1	0	Alabama	Alabama	5024279	5024803	5039877
3	50	3	6	1	1	Alabama	Autauga County	58805	58877	59095
4	50	3	6	1	3	Alabama	Baldwin County	231767	233140	239294
5	50	3	6	1	5	Alabama	Barbour County	25223	25180	24964
6	50	3	6	1	7	Alabama	Bibb County	22293	22223	22477

Table 2. County-Level Population Estimates for 2021

Three data sets will be created, representing national-, state-, and county-level population data:

- df_pop_us
- df_pop_states
- df_pop_counties

Note that regional-level data are not utilized in these analyses, so these observations will be deleted.

Also note that "DF" is utilized to denote "DataFrame," and this optional convention ensures that all examples—whether coded in SAS or Python—have identically named data structures.

INGESTING POPULATION USING SAS

PROC IMPORT fails to ingest the data because it incorrectly interprets FIPS values (many of which contain leading zeros) as numeric data, so DATA steps must explicitly specify that FIPS codes are character data. This is not uncommon in SAS, and is similarly required when importing FIPS into Python.

DF_pop_us_temp is a temporary data set that is used as an intermediate step to create both DF_pop_us and DF_pop_states:

```
data df_pop_us_temp (rename=(state=FIPS_state popestimate2021=population));
   infile "&path_census.NST-EST2021-alldata.csv" delimiter=',' dsd firstobs=2;
   length sumlev $2 region $1 division $1 state $2 name $50 estimatebase2020 8
      popestimate2020 8 popestimate2021 8;
   input sumlev $ region $ division $ state $ name $ estimatebase2020
      popestimate2020 popestimate2021;
run;
```

Thereafter, three successive DATA steps create DF_pop_us, DF_pop_states, and DF_pop_counties:

```
data df_pop_us (keep=FIPS_state population pop_mil);
   set df_pop_us_temp;
   where name='United States';
   length pop_mil 8;
   pop_mil = pop / 1000000;
run;

data df_pop_states (keep=FIPS_state population pop_mil);
   set df_pop_us_temp;
   where FIPS_state^='00';
   length pop_mil 8;
   pop_mil = population / 1000000;
run;

data df_pop_counties (rename=(state=FIPS_state county=FIPS_county
      popestimate2021=population) keep=state county popestimate2021 pop_mil);
   infile "&path_census.co-est2021-alldata.csv" delimiter=',' dsd firstobs=2;
   length sumlev $2 region $1 division $1 state $2 county $5 stname $50 ctyname $50
      estimatebase2020 8 popestimate2020 8 popestimate2021 8;
   input sumlev $ region $ division $ state $ county $ stname $ ctyname $
      estimatebase2020 popestimate2020 popestimate2021;
   if county^='000';
   length pop_mil 8;
   pop_mil = popestimate2021 / 1000000;
   county=state || county;
run;
```

Presto! The data sets have been created.

INGESTING POPULATION USING PYTHON

The equivalent Python steps follow, in which three DataFrames are created:

```
# create dataframe with national population
```

```
fil = r'NST-EST2021-alldata.csv'
df_pop_us = pd.read_csv(
    os.path.join(path_census, fil), header = 0, sep = ',', quotechar = '"',
    index_col = False, dtype = {'STATE': 'string', 'POPESTIMATE2021': 'float',
'NAME': 'string'},
    usecols = ['STATE','POPESTIMATE2021', 'NAME'])
df_pop_us.rename(columns = {'STATE': 'fips_state', 'POPESTIMATE2021':
'population'}, inplace=True)
df_pop_us['pop_mil'] = df_pop_us['population']/1000000
df_pop_us = df_pop_us.loc[df_pop_us['NAME'] == 'United States'] # remove FIPS 00
which is the entire US
df_pop_us.drop(columns=['NAME'], inplace=True)


# create dataframe with state populations
fil = r'NST-EST2021-alldata.csv'
df_pop_states = pd.read_csv(
    os.path.join(path_census, fil), header = 0, sep = ',', quotechar = '"',
    index_col = False, dtype = {'STATE': 'string', 'POPESTIMATE2021': 'float'},
    usecols = ['STATE','POPESTIMATE2021'])
df_pop_states.rename(columns = {'STATE': 'fips_state', 'POPESTIMATE2021':
'population'}, inplace=True)
df_pop_states['pop_mil'] = df_pop_states['population']/1000000
df_pop_states = df_pop_states.loc[df_pop_states['fips_state'] != '00'] # remove
FIPS 00 which is the entire US



# create dataframe with county populations
fil = r'co-est2021-alldata.csv'
df_pop_counties = pd.read_csv(
    os.path.join(path_census, fil), header = 0, sep = ',', quotechar = '"',
    encoding = 'utf-8', encoding_errors = 'ignore',
    index_col = False, dtype = {'STATE': 'string', 'COUNTY': 'string',
'POPESTIMATE2021': float},
    usecols = ['STATE', 'COUNTY', 'POPESTIMATE2021'])
df_pop_counties.rename(columns = {'STATE': 'fips_state', 'COUNTY':
'fips_county_temp', 'POPESTIMATE2021': 'population'}, inplace=True)
df_pop_counties['pop_mil'] = df_pop_counties['population']/1000000
df_pop_counties['fips_county'] = df_pop_counties['fips_state'] +
df_pop_counties['fips_county_temp']
df_pop_counties = df_pop_counties.loc[df_pop_counties['fips_county_temp'] != '000']
# remove FIPS 000 which is the entire state
df_pop_counties.drop(columns=['fips_county_temp'], inplace=True)
```

The national population DataFrame (**df_pop_us**), state population DataFrame (**df_pop_states**), and county state population DataFrame (**df_pop_counties**) are created next; the steps to create the final DataFrame are explored in more detail.

The **pd.read_csv** method instantiates **df_pop_counties** as a DataFrame, with multiple arguments specifying the manner in which the CSV file should be ingested. For example, the **dtype** argument specifies the data type for each variable, similar to the LENGTH statement in SAS:

```
dtype = {'STATE': str, 'COUNTY': str, 'POPESTIMATE2021': float}
```

The **usecols** argument specifies the variables to retain from the CSV file, similar to the KEEP option in SAS:

```
usecols = ['STATE', 'COUNTY', 'POPESTIMATE2021']
```

The **rename** method renames CSV variables, and **inplace=True** designates that the changes should be made in place (i.e., in **df_pop_counties**) as opposed to in a copy of the DataFrame:

```
df_pop_counties.rename(columns = {'STATE': 'fips_state', 'COUNTY':
'fips_county_temp', 'POPESTIMATE2021': 'population'}, inplace=True)
```

A new column (**fips_county**) in the DataFrame is created by overloading the **+** operator; that is, two columns within the DataFrame (holding string data) are concatenated to form a third column:

```
df_pop_counties['fips_county'] = df_pop_counties['fips_state'] +
df_pop_counties['fips_county_temp']
```

Because a county FIPS value of 000 represents a state-level FIPS value, all rows for which **fips_county_temp** is 000 are removed (because this DataFrame should include only county-level data). The **loc** function "slices" the DataFrame to remove the 000 rows:

```
df_pop_counties = df_pop_counties.loc[df_pop_counties['fips_county_temp'] != '000']
```

Finally, the **drop** method drops the **fips_county_temp** column, the original three-character county FIPS code, now that the **fips_county** column has been created that denotes not only the county FIPS code but also the state FIPS code:

```
df_pop_counties.drop(columns=['fips_county_temp'], inplace=True)
```

The three DataFrames that are created have identical content to the three SAS data sets created in the prior section.

INGESTING STATE AND COUNTY FIPS CODES

State FIPS codes are two-digit numbers that uniquely identify states (and US territories), and by convention, leading zeros are always retained; thus, California is always shown as "06" and never "6." County FIPS codes are three-digit numbers that uniquely identify a county (within a state), but because county codes are repeated across states, the identification of counties at the national level requires concatenating the state FIPS and county FIPS codes to yield a five-digit FIPS code that is unique (by county) across all states.

FIPS codes maintained in all-geocodes-v2017.csv are demonstrated in Table 3; note the multiple header rows that must be handled when importing these data.

	A	B	C	D	E	F	G
1	**Estimates Geography File: Vintage 2017**						
2	Source: U.S. Census Bureau, Population Division						
3	Internet Release Date: May 2018						
4							
5	**Summary Level**	**State Code (FIPS)**	**County Code (FIPS)**	**County Subdivision Code (FIPS)**	**Place Code (FIPS)**	**Consolidtated City Code (FIPS)**	**Area Name (including legal/statistical area description)**
6	010	00	000	00000	00000	00000	United States
7	040	01	000	00000	00000	00000	Alabama
8	050	01	001	00000	00000	00000	Autauga County
9	050	01	003	00000	00000	00000	Baldwin County
10	050	01	005	00000	00000	00000	Barbour County
11	050	01	007	00000	00000	00000	Bibb County

Table 3. FIPS Codes

FIPS codes are powerful because they overcome the occasional spelling variations or errors that can occur in state or county names—even within federal databases—as demonstrated subsequently. This FIPS data integrity also yields tables that can be more reliably joined by FIPS code than, for example, state name or county name.

INGESTING FIPS CODES USING SAS

The following SAS code ingests the FIPS CSV file and creates the DF_fips data set:

```
%let fil=all-geocodes-v2017.csv;
```

```
data df_fips (drop=summary_level FIPS_state_temp FIPS_county_temp subdiv_code
      place_code city_code name);
   length FIPS_state $2 state $50 county $50 FIPS_county $5;
   infile "&path_census&fil" dsd delimiter=',' firstobs=6 end=eof;
   length summary_level $3 FIPS_state_temp $2 FIPS_county_temp $5 subdiv_code $5
      place_code $5 city_code $5 name $50;
   input summary_level $ FIPS_state_temp $ FIPS_county_temp $ subdiv_code $
      place_code $ city_code $ name $;
   retain FIPS_state state;
   /* 040 is the FIPS summary code for state-level region */
   if summary_level='040' then do;
      FIPS_state=FIPS_state_temp;
      state=name;
      end;
   /* 010 is the FIPS summary code for the entire US */
   else if summary_level^='010' and subdiv_code='00000' and place_code ='00000'
         and city_code='00000' and FIPS_state^= '72' then do;
      FIPS_county=FIPS_state || FIPS_county_temp;
      county=name;
      output;
      end;
run;
```

A user-defined SAS format is both a straightforward and efficient method to map values, and the following code creates the COUNTY_FIPS_DICT format that maps five-digit FIPS county codes to their associated county names:

```
data county_fips_dict_temp;
   set df_fips (rename=(FIPS_county=start county=label));
   length fmtname $20 type $1;
   retain fmtname 'county_fips_dict' type 'c';
run;

proc format cntlin=county_fips_dict_temp;
run;
```

Similarly, the following code creates the STATE_FIPS_DICT user-defined format that maps the two-digit FIPS state codes to their associated state (or territory) names:

```
proc sort data=df_fips (keep=fips_state state) out=df_fips_temp nodupkey;
   by fips_state;
run;

data state_fips_dict_temp;
   set df_fips_temp (rename=(FIPS_state=start state=label));
   length fmtname $20 type $1;
   retain fmtname 'state_fips_dict' type 'c';
run;

proc format cntlin=state_fips_dict_temp;
run;
```

User-defined formats reside in memory, facilitate faster data transformation, and eliminate the need to join the DF_fips data set to other data sets via DATA step MERGE statements or SQL procedure JOIN statements.

INGESTING FIPS CODES USING PYTHON

Python similarly leverages FIPS data to create equivalent memory-resident dictionaries that map FIPS codes to both county and state names—but first, a refined CSV file (FIPS_table_csv) is created, from which the dictionaries can be built:

```
fil = r'all-geocodes-v2017.csv'
df = pd.read_csv(
```

```
    os.path.join(path_census, fil),
    header=4, sep=',', quotechar='"', index_col=False, encoding='latin1',
    dtype = {'Summary Level': 'string', 'State Code (FIPS)': 'string', 'County Code
(FIPS)': 'string',
             'County Subdivision Code (FIPS)': 'string',
             'Place Code (FIPS)': str, 'Consolidtated City Code (FIPS)': str,
             'Area Name (including legal/statistical area description)': str},
    usecols = ['Summary Level', 'State Code (FIPS)', 'County Code (FIPS)', 'County
Subdivision Code (FIPS)',
               'Place Code (FIPS)', 'Consolidtated City Code (FIPS)',
               'Area Name (including legal/statistical area description)'])
df.rename(columns={'State Code (FIPS)': 'fips_state', 'County Code (FIPS)':
'county_code',
                   'County Subdivision Code (FIPS)': 'subdiv', 'Place Code (FIPS)':
'place',
                   'Consolidtated City Code (FIPS)': 'consolidated',
                   'Area Name (including legal/statistical area description)':
'name'}, inplace=True)
df_states = df.loc[df['Summary Level'] == '040']
df_states = df_states.rename(columns={'name': 'state'})
df_states.drop(columns=['Summary Level','county_code','subdiv','place'],
inplace=True)
df_counties = df.loc[(df['county_code'] != '000') & (df['subdiv'] == '00000') &
(df['place'] == '00000') & (df['consolidated'] == '00000'),
['fips_state','county_code','name']]
df_counties.rename(columns={'name': 'county'}, inplace=True)
df_fips = pd.merge(df_states, df_counties, on='fips_state', how='left')
df_fips['fips_county'] = df_fips['fips_state'] + df_fips['county_code']
df_fips.drop(columns=['county_code'], inplace=True)

# Puerto Rico municipalities are removed
df_fips = df_fips.loc[df_fips['fips_state'] != '72']

df_fips.to_csv(os.path.join(path_tables,'FIPS_table.csv'), index=False)
```

Examining the code more closely, the **`dtype`** and **`usecols`** parameters declare the column data types and columns to keep, respectively:

```
dtype = {'Summary Level': str, 'State Code (FIPS)': str, 'County Code (FIPS)': str,
                 'County Subdivision Code (FIPS)': str,
                 'Place Code (FIPS)': str, 'Consolidtated City Code (FIPS)': str,
                 'Area Name (including legal/statistical area description)': str},
        usecols = ['Summary Level', 'State Code (FIPS)', 'County Code (FIPS)',
'County Subdivision Code (FIPS)',
                   'Place Code (FIPS)', 'Consolidtated City Code (FIPS)',
                   'Area Name (including legal/statistical area description)']
```

The **`rename`** method renames columns; although Pandas supports column names with spaces, they are removed here to facilitate readability, as well as to enable “dot notation” (discussed later):

```
df.rename(columns={'State Code (FIPS)': 'fips_state', 'County Code (FIPS)':
'county_code', 'County Subdivision Code (FIPS)': 'subdiv', 'Place Code (FIPS)':
'place', 'Consolidtated City Code (FIPS)': 'consolidated',
'Area Name (including legal/statistical area description)': 'name'}, inplace=True)
```

A temporary DataFrame **`df_states`** is created, which includes only state-level data:

```
df_states = df.loc[df['Summary Level'] == '040']
df_states = df_states.rename(columns={'name': 'state'})
df_states.drop(columns=['Summary Level','county_code','subdiv','place'],
inplace=True)
```

Similarly, a DataFrame **`df_counties`** is created, which includes only county-level data:

```
df_counties = df.loc[(df['county_code'] != '000') & (df['subdiv'] == '00000') &
                         (df['place'] == '00000') & (df['consolidated'] ==
'00000'), ['fips_state','county_code','name']]
df_counties.rename(columns={'name': 'county'}, inplace=True)
```

The **merge** method performs a left join between **df_states** and **df_counties**; this effectively creates the **df_fips** DataFrame by appending the state-level columns (including state abbreviation and state name) to **df_counties**:

```
df_fips = pd.merge(df_states, df_counties, on='fips_state', how='left')
```

The **fips_county** column is created in the **df_fips** DataFrame, and unnecessary columns are dropped using the **drop** method:

```
df_fips['fips_county'] = df_fips['fips_state'] + df_fips['county_code']
df_fips.drop(columns=['county_code', 'consolidated'], inplace=True)
```

Finally, Puerto Rico municipality values are removed, and the **df_fips** DataFrame is saved to FIPS_table.csv using to **to_csv** method:

```
df_fips = df_fips.loc[df_fips['fips_state'] != '72']
df_fips.to_csv(os.path.join(path_tables,'FIPS_table.csv'), index=False)
```

At this point, FIPS_table.csv has been created, and this can be used for subsequent data transformations. For example, two dictionaries are created by reading the CSV file into memory:

```
# table columns are 0) fips_state, 1) state name, 2) county, 3) fips_county
with open(os.path.join(path_tables,'FIPS_table.csv'), mode='r') as infile:
    reader = csv.reader(infile)
    next(reader, None)
    county_fips_dict = {rows[3]:[rows[2], rows[0], rows[1]] for rows in reader}

with open(os.path.join(path_tables,'FIPS_table.csv'), mode='r') as infile:
    reader = csv.reader(infile)
    next(reader, None)
    state_fips_dict = {rows[0]:[rows[1]] for rows in reader}
```

The **county_fips_dict** dictionary maps the five-digit county FIPS codes to the associated county name, state FIPS code, and state name.

Similarly, **state_fips_dict** maps the two-digit state FIPS codes to the associate state name. Python dictionaries, by definition, cannot maintain duplicate keys, so only the first row for each state is read into this second dictionary, which yields a total of 51 key-value pairs—the 50 states and the District of Columbia.

INGESTING CDC OBESITY DATA

The United States Diabetes Surveillance System (USDSS) is managed by the Centers for Disease Control and Prevention (CDC), and collects and longitudinally monitors diabetes incidence across the nation, including diabetes-related contributing factors such as obesity and physical inactivity. The USDSS interactive “Social Determinants of Health” dashboard (https://gis.cdc.gov/grasp/diabetes/diabetesatlas-sdoh.html) demonstrates county-level diabetes incidence, shown for 2018 in Figure 1, with darker regions denoting higher incidence.

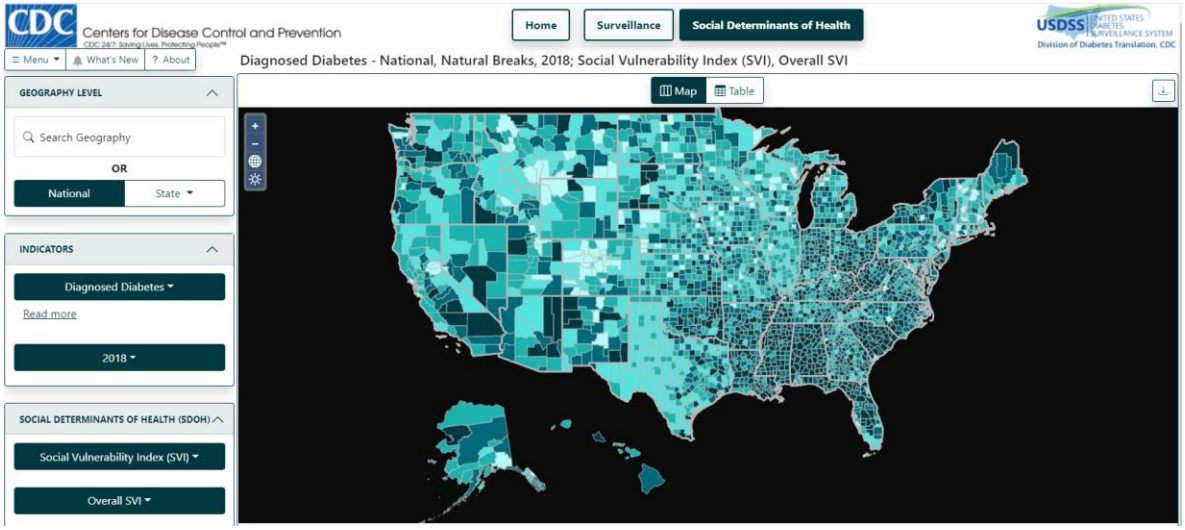

Figure 1. County-Level Diabetes Incidence for 2018 (USDSS)

Obesity data are obtained from the CDC Behavioral Risk Factor Surveillance System (BRFSS), a monthly telephonic survey, with CDC providing the following description of its methodology: (CDC, 2022)

- To have diagnosed diabetes if they responded "yes" to the question, "Has a doctor ever told you that you have diabetes?" Women who indicated that they only had diabetes during pregnancy were not considered to have diagnosed diabetes. People who reported having diagnosed diabetes were then asked at what age they were diagnosed.
- To have been diagnosed with diabetes in the last year if they reported having diagnosed diabetes and the difference between their age at the time of the survey and the age they provided to the question, "How old were you when you were told you have diabetes?" was less than one. If the difference was between one year and two years, the person was weighted as half a newly diagnosed case.
- To be obese if their body mass index was 30 or greater. Body mass index (weight [kg]/height [m]2) was derived from self-report of height and weight.
- To be physically inactive if they answered "no" to the question, "During the past month, other than your regular job, did you participate in any physical activities or exercises such as running, calisthenics, golf, gardening, or walking for exercise?"

Obesity data are demonstrated in Figure 2, and can be downloaded to a CSV file.

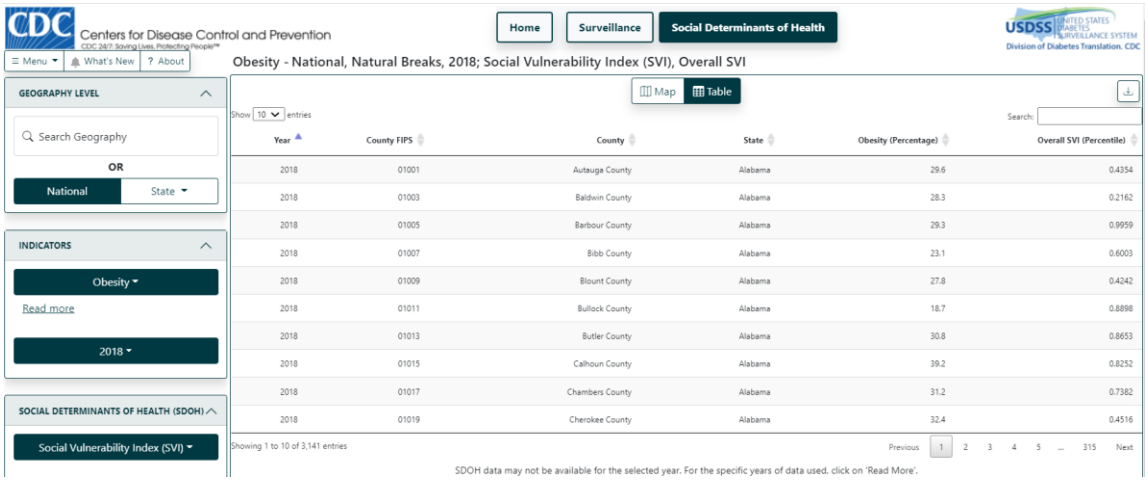

Figure 2. CDC 2018 County-Level Obesity Data

Download this table to the following location:

```
c:\shedder\cdc\DiabetesAtlasData.csv
```

Table 4 demonstrates the obesity CSV file; note the headers that will need to be removed programmatically:

	A	B	C	D	E	F
1	SVI Theme: Overall SVI; Overall SVI (Percentile); NaturalBreaks					
2	Data downloaded on 11-October-2022					
3	Year	County_FIPS	County	State	Obesity Percentage	Overall SVI
4	2018	1001	Autauga County	Alabama	29.6	0.4354
5	2018	1003	Baldwin County	Alabama	28.3	0.2162
6	2018	1005	Barbour County	Alabama	29.3	0.9959
7	2018	1007	Bibb County	Alabama	23.1	0.6003

Table 4. CDC 2018 County-Level Obesity Data (betesAtlasData.csv)

With obesity data downloaded, the next two sections demonstrate how to ingest this CSV file, first using SAS and subsequently using Python.

INGESTING OBESITY DATA USING SAS

The following DATA step ingests the CDC obesity data:

```
%let fil=DiabetesAtlasData.csv;

data df_fat (drop=year svi);
   length year $4 fips_county $5 county $40 state $20 fat_pct 8 svi 8;
   infile "&path_cdc&fil" dsd delimiter=',' firstobs=4 end=eof;
   input year $ fips_county $ county $ state $ fat_pct svi;
run;
```

Note the LOST CARD comment in the log, which results from the final line of the file, which lists reference information for the data, and which thus does not conform to the other tabular cells:

```
US Diabetes Surveillance System; www.cdc.gov/diabetes/data; Division of Diabetes
Translation - Centers for Disease Control and
```

The df_fat data set is created, as demonstrated in Table 5.

	fips_county	county	state	fat_pct
1	01001	Autauga County	Alabama	29.6
2	01003	Baldwin County	Alabama	28.3
3	01005	Barbour County	Alabama	29.3
4	01007	Bibb County	Alabama	23.1
5	01009	Blount County	Alabama	27.8

Table 5. DF_fat Data Set Containing CDC County-Level Obesity Data

Note that the SVI variable, not the focus of this text, has been removed from the data.

INGESTING OBESITY DATA USING PYTHON

The following Python code reads the CDC CSV file and creates the **`df_fat`** DataFrame:

```
path_cdc = os.path.join(path_base, r'CDC')

fil = r'DiabetesAtlasData.csv'
df_fat = pd.read_csv(
    os.path.join(path_cdc, fil), header = None, sep = ',', quotechar = '"',
    skiprows = 3, usecols = [1,2,3,4],
    names=['fips_county','county','state','fat_pct'],
```

```
    dtype={'fips_county': str, 'county': str, 'state': str, 'fat_pct': 'float64'})
df_fat.dropna(inplace=True)
```

The solution again relies on the **read_csv** method to ingest the CSV file, with the **names** and **dtype** parameters specifying variables to retain and their data types, respectively. The **dropna** method removes any rows in the **df_fat** DataFrame that are missing at least one column, and this statement is required to remove the final row in the CSV file containing reference information:

```
df_fat.dropna(inplace=True)
```

With obesity data uploaded into both SAS and Python, the next step is to inspect these data, and because they represent county-level statistics, to compare the data with respect to county-level US Census FIPS codes (i.e., master data).

INSPECTING CDC OBESITY DATA

At this point, the savvy analyst may have noticed that the **df_fips** data set (and DataFrame) contains 3,142 observations (rows), whereas the **df_fat** data set (and DataFrame) contains only 3,141 observations (rows). Thus, without even inspecting individual values, one is aware that some discrepancy exists between these two federal data sources.

INSPECTING OBESITY DATA USING SAS

The following SAS code identifies discrepancies between the FIPS county codes listed in the two data sets:

```
proc sort data=df_fat out=df_fat_sorted;
   by fips_county;
run;

proc sort data=df_fips out=df_fips_sorted;
   by fips_county;
run;

data df_fat_merged;
   merge df_fat_sorted (in=a keep=fips_county) df_fips_sorted (in=b);
   by fips_county;
   if a and ^b then put 'Extra FIPS in CDC' fips_county;
   else if b and ^a then put 'Missing FIPS in CDC   ' fips_county;
run;
```

The log demonstrates that 35039 (i.e., Rio Arriba, New Mexico) is missing from the CDC data:

```
Missing FIPS in CDC   35039
NOTE: There were 3141 observations read from the data set WORK.DF_FAT_SORTED.
NOTE: There were 3142 observations read from the data set WORK.DF_FIPS_SORTED.
NOTE: The data set WORK.DF_FAT_MERGED has 3142 observations and 4 variables.
```

A second point to investigate is whether state and county names are spelled identically between Census and CDC data—and were the federal government exercising master data management (MDM) best practices, you would expect this to be true. The following DATA step now applies the STATE_FIPS_DICT and COUNTY_FIPS_DICT user-defined formats, created previously, to compare CDC state and county names to Census equivalents:

```
data df_fat_check_state_county;
   set df_fat;
   length state_census $30 county_census $40;
   state_census=put(substr(fips_county,1,2), $state_fips_dict.);
   county_census=put(fips_county, $county_fips_dict.);
   if state^=state_census then put @1 fips_county @7 state @40 state_census;
   if county^=county_census then put @1 fips_county @7 county @40 county_census;
run;
```

Only the first five exceptions are shown, and they demonstrate a variety of issues—with some variation occurring from capitalization, other variation occurring from word discrepancy, and with some values truncated in the CDC data:

```
01049 Dekalb County                    DeKalb County
02195 Petersburg Census Area           Petersburg Borough
02198 Prince of Wales-Hyder Censu      Prince of Wales-Hyder Census Area
11001 District Of Columbia             District of Columbia
12027 Desoto County                    DeSoto County
```

To remove the capitalization issue, the DATA step can be updated to include UPCASE:

```
data df_fat_check_state_county;
   set df_fat;
   length state_census $30 county_census $40;
   state_census=put(substr(fips_county,1,2), $state_fips_dict.);
   county_census=put(fips_county, $county_fips_dict.);
   if upcase(state)^=upcase(state_census) then put @1 fips_county @7 state
      @40 state_census;
   if upcase(county)^=upcase(county_census) then put @1 fips_county @7 county
      @40 county_census;
run;
```

The following seven county names are different (i.e., wrong) in the CDC data:

```
02195 Petersburg Census Area           Petersburg Borough
02198 Prince of Wales-Hyder Censu      Prince of Wales-Hyder Census Area
19141 O&#39;brien County               O'Brien County
24033 Prince George&#39;s County       Prince George's County
24035 Queen Anne&#39;s County          Queen Anne's County
24037 St. Mary&#39;s County            St. Mary's County
35013 DoÃ±a Ana County                 Do¤a Ana County
```

Opening the raw CDC CSV file in a text editor reveals that values truly are being truncated by CDC:

```
2018,02198,Prince of Wales-Hyder Censu,Alaska,19.9,0.7662
```

Similarly, the text file reveals that escape characters (like #39; representing a single quote) really are maintained within the raw CDC data:

```
2018,19141,O&#39;brien County,Iowa,25.4,0.2322
```

With this confirmation that some CDC counties are incorrectly named, it is best to overwrite the CDC county names and to instead rely on the US Census master data for county names:

```
data df_fat_county_corrected;
   set df_fat (drop=county);
   length county $40;
   county=put(fips_county, $county_fips_dict.);
run;
```

At this point, one county remains missing from the CDC data, but state names have been validated, and county names have been corrected.

INSPECTING OBESITY DATA USING PYTHON

The differences in FIPS county codes can be assessed in Python with a single line of code:

```
print(set(df_fips.fips_county) - set(df_fat.fips_county))
```

The **`set`** function creates a unique series of data, which removes any non-unique values that may exist. Thus, by setting the **`fips_county`** column in these two DataFrames, the difference is demonstrated, which matches the missing county revealed by equivalent SAS code:

```
{'35039'}
```

Also note the use of *dot notation* in Pandas, in which the column **fips_country** was selected in the **df_fips** DataFrame by including a dot between the DataFrame name and column name. Pandas dot notation eliminates the need to enclose the column name in brackets and quotes, although the column name cannot have a space. For example, the following statement is functionally equivalent to the previous:

```
print(set(df_fips['fips_county']) - set(df_fat['fips_county']))
```

The following code identifies the seven CDC county names that do not match Census names, as well as the missing county:

```
df_merge = pd.merge(left=df_fips[['fips_county','county']],
                    right=df_fat[['fips_county','county']], how='left',
on='fips_county')
df_merge_2 = df_merge[df_merge.county_x.str.upper() !=
df_merge.county_y.str.upper()]
print(df_merge_2)
```

Note that the **merge** method by default creates new columns whenever identically named columns occur in the DataFrames being joined; thus, because the **county** column appears in both **df_fips** and **df_fat**, an **_x** and **_y**, respectively, are appended to create **county_x** (representing the master FIPS data) and **county_y** (that includes erroneous CDC county names).

The `print` function writes rows containing non-matching county names to the log:

```
     fips_county                        county_x                      county_y
87         02195                Petersburg Borough      Petersburg Census Area
88         02198  Prince of Wales-Hyder Census Area  Prince of Wales-Hyder Censu
859        19141                   O'Brien County           O&#39;brien County
1208       24033            Prince George's County   Prince George&#39;s County
1209       24035               Queen Anne's County      Queen Anne&#39;s County
1210       24037                 St. Mary's County        St. Mary&#39;s County
1802       35013                   Do¤a Ana County              Doña Ana County
1816       35039                 Rio Arriba County                         NaN
```

Instead of applying the **merge** method, the **county_fips_dict** dictionary instead could be applied to the county FIPS code to transform it, and to initialize the new column **county_census**:

```
df_fat_check_state_county = df_fat.copy(deep=True)
df_fat_check_state_county['county_census'] =
df_fat_check_state_county.fips_county.map(county_fips_dict).str[0]
df_fat_check_state_county =
df_fat_check_state_county.loc[df_fat_check_state_county.county.str.upper() !=
df_fat_check_state_county.county_census.str.upper()]
print(df_fat_check_state_county[['fips_county','county','county_census']])
```

Note the use of both *dot notation* and *method chaining* in the second line that initializes the **county_census** column; that is, the **fips_county** column is selected from the Data Frame, after which the **map** method applies the **county_fips_dict** dictionary to the **fips_county** column, after which the **str** method subsequently selects only the first returned value from the matched dictionary key—and all of this functionality is accomplished in a single line of code! To a SAS practitioner unfamiliar with dot notation, methods, and method chaining, the closest comparable Base SAS syntax might be the use of nested functions; however, unlike nested functions, method chaining facilitates readability because the order of operations is read from left to right.

The **copy** method first creates a copy of the **df_fat** DataFrame, thus ensuring that changes made to the copy will not modify the original **df_fat** DataFrame:

```
df_fat_check_state_county = df_fat.copy(deep=True)
```

The **county_census** column is created by applying the **map** method to the five-digit FIPS county code; the **county_fips_dict** dictionary is mapped, and because the dictionary was defined as having a list

of three elements (county name, state FIPS code, and state name), the **str** method selects the first element (county name), as designated by the [0] index:

```
df_fat_check_state_county.fips_county.map(county_fips_dict).str[0]
```

The **loc** function creates a slice of the DataFrame that includes only rows in which the **county** column and **county_census** column values do not match:

```
df_fat_check_state_county =
   df_fat_check_state_county.loc[df_fat_check_state_county.county.str.upper() !=
   df_fat_check_state_county.county_census.str.upper()]
```

When the DataFrame is examined, its results mirror the data quality issues that were noted in the previous **merge** method, with the exception that the one missing county is not listed:

	fips_county	county	county_census
87	02195	Petersburg Census Area	Petersburg Borough
88	02198	Prince of Wales-Hyder Censu	Prince of Wales-Hyder Census Area
859	19141	O'brien County	O'Brien County
1209	24033	Prince George's County	Prince George's County
1210	24035	Queen Anne's County	Queen Anne's County
1212	24037	St. Mary's County	St. Mary's County
1802	35013	Doña Ana County	DoÂ¤a Ana County

Finally, with the Python evaluation demonstrating that the CDC county names cannot be trusted, the **merge** method is utilized again to overwrite the errant county names with trusted Census names:

```
df_fat_county_corrected = pd.merge(left=df_fat[['fips_county','fat_pct']],
                                   right=df_fips, how='left', on='fips_county')
```

Application of dictionaries with the **map** method most closely mirrors the application of SAS formats (including user-defined formats) to variables, whereas the **merge** method most closely mirrors DATA step MERGE statements or SQL JOIN statements. Thus, in both languages, multiple methods exist for performing data lookup operations that cull, standardize, or clean data based on master data values.

ANALYZING CDC OBESITY DATA

With county names now cleaned, some basic data analysis can be performed to answer obesity questions:

- What are the ten counties with the highest obesity rates in the nation?
- What are the ten skinniest counties in the nation?
- What are the ten states with the highest obesity rates in the nation? *Note that as only county-level obesity data have been downloaded, this will require computing a weighted average (by county population size) to estimate state-level obesity rates.*

Note again, in answering these questions, the obesity criteria supplied by CDC that state these metrics are gathered over the telephone, and computed only by measuring reported weight and reported height; thus, numerous biases do exist, none of which are explored in this text.

WHAT ARE THE TEN COUNTIES WITH THE HIGHEST OBESITY RATES?

Obesity rates can be evaluated and ranked in SAS by first sorting the data by fat percentage, and by subsequently selecting the first ten observations in the data set (leveraging the OBS option):

```
proc sort data=df_fat_county_corrected;
   by descending fat_pct;
run;

data df_obese_counties;
   set df_fat_county_corrected (obs=10);
   length county_st $50;
   format fat_pct 8.1;
```

```
   county_st=catx(', ',county,state);
   put @1 county_st @40 fat_pct;
run;
```

The log demonstrates the ten counties with the highest obesity rates:

```
Thurston County, Nebraska              43.8
Cass County, Nebraska                  43.1
Williamsburg County, South Carolina    43.0
Rolette County, North Dakota           41.6
Sunflower County, Mississippi          41.5
Ziebach County, South Dakota           41.5
Lawrence County, Kentucky              41.4
Hidalgo County, Texas                  41.4
Marengo County, Alabama                41.3
Jefferson County, Texas                40.8
```

By comparison, a single line of Python code leverages the **`nlargest`** method to select the ten counties with the highest obesity rates:

```
df_fat_county_corrected.nlargest(10, 'fat_pct')[['county','state','fat_pct']]
```

The tabular output is demonstrated:

```
                   county           state  fat_pct
1740      Thurston County        Nebraska     43.8
1666          Cass County        Nebraska     43.1
2359  Williamsburg County  South Carolina     43.0
2028       Rolette County    North Dakota     41.6
1467     Sunflower County     Mississippi     41.5
2426       Ziebach County    South Dakota     41.5
1056      Lawrence County        Kentucky     41.4
2629       Hidalgo County           Texas     41.4
45         Marengo County         Alabama     41.3
2644     Jefferson County           Texas     40.8
```

WHAT ARE THE TEN LEAST FAT COUNTIES IN THE US?

Similarly, the SAS data set can be sorted in ascending order by fat percentage, after which the DATA step selects the ten skinniest counties:

```
proc sort data=df_fat_county_corrected;
   by fat_pct;
run;
data df_skinniest;
   set df_fat_county_corrected (obs=10);
   length county_st $50;
   county_st=catx(', ',county,state);
   put @1 county_st @40 fat_pct;
run;
```

The log demonstrates the ten skinniest counties:

```
Teton County, Wyoming                  10.5
Boulder County, Colorado               13.6
Routt County, Colorado                 13.7
Gunnison County, Colorado              13.8
Pitkin County, Colorado                14.2
Summit County, Utah                    14.2
Chaffee County, Colorado               14.4
Taos County, New Mexico                14.6
Summit County, Colorado                15.2
San Francisco County, California       15.4
```

A single line of Python code leverages the **`nsmallest`** method to select the ten skinniest counties:

```
df_fat_county_corrected.nsmallest(10, 'fat_pct') [['county','state','fat_pct']]
```

The tabular output is demonstrated:

```
                    county       state  fat_pct
3137          Teton County     Wyoming     10.5
250         Boulder County    Colorado     13.6
298           Routt County    Colorado     13.7
270        Gunnison County    Colorado     13.8
293           Pitkin County    Colorado     14.2
2797          Summit County        Utah     14.2
252         Chaffee County    Colorado     14.4
1823           Taos County  New Mexico     14.6
303           Summit County    Colorado     15.2
223   San Francisco County  California     15.4
```

WHAT ARE THE TEN STATES WITH THE HIGHEST OBESITY RATES?

Because these CDC obesity statistics are calculated at the county level, a simple mean cannot be utilized to aggregate county-level percentages to calculate state-level obesity; rather, weighted averages (by county population) must be used to approximate state-level obesity. *This is not to assert or imply that state-level obesity data are unpublished, but rather to demonstrate how to calculate weighted averages.*

The SAS SQL procedure first joins county population to the obesity data, after which the MEANS procedure computes the weighted averages utilizing the WEIGHT statement to weight by county population:

```
proc sql;
   create table df_fat_pop as
      select a.*, b.population from df_fat_county_corrected as a
         left join df_pop_counties as b on a.fips_county = b.fips_county;
quit;

proc means data=df_fat_pop sum sumwgt mean;
   class state;
   weight population;
   var fat_pct;
   output out=df_fat_weighted_avg mean=fat_avg;
run;
```

The output, shown in Table 6, demonstrates that Alabama has the highest obesity in the US.

The MEANS Procedure

Analysis Variable : fat_pct				
state	N Obs	Sum	Sum Wgts	Mean
Alabama	67	164758811	5039877.00	32.6910382
Alaska	29	21223043.40	723102.00	29.3499996
Arizona	15	205462794	7276316.00	28.2372005
Arkansas	75	93653940.90	3025891.00	30.9508640
California	58	974923452	39237836.00	24.8465143
Colorado	64	129719737	5812069.00	22.3190291
Connecticut	8	94799282.40	3605597.00	26.2922568
Delaware	3	32854487.20	1003384.00	32.7436826
District of Columbia	1	16483230.00	670050.00	24.6000000
Florida	67	602427824	21781128.00	27.6582473

Table 6. Ten States with the Highest Obesity, Calculated by Weighted Means of County-Level CDC Data

As in SAS, multiple methods also exist to calculate weighted means in Python. For example, the following code creates the **df_grouped** DataFrame, which orders the states by mean obesity percentage:

```
df_fat_pop = pd.merge(left=df_fat,
right=df_pop_counties[['fips_county','population']],
                      how='inner', on='fips_county')
df_fat_pop['fat_pct_x_pop'] = df_fat_pop.fat_pct * df_fat_pop.population
df_grouped = df_fat_pop.groupby('state').sum()
df_grouped['fat_pct_state'] = df_grouped.fat_pct_x_pop / df_grouped.population
df_grouped =
df_grouped[['fat_pct_state','fat_pct','population','fat_pct_x_pop','fat_pct_state']
]
print(df_grouped)
```

The ten states within the **df_grouped** DataFrame follow, and mirror the SAS results:

```
                     fat_pct  population  fat_pct_x_pop  fat_pct_state
state
Alabama               2071.4   5039877.0    164758811.4      32.691038
Alaska                 720.5    723102.0     21223043.4      29.350000
Arizona                448.0   7276316.0    205462794.1      28.237201
Arkansas              2124.8   3025891.0     93653940.9      30.950864
California            1398.6  39237836.0    974923451.7      24.846514
Colorado              1352.9   5812069.0    129719736.9      22.319029
Connecticut            215.5   3605597.0     94799282.4      26.292257
Delaware               102.8   1003384.0     32854487.2      32.743683
District of Columbia    24.6    670050.0     16483230.0      24.600000
Florida               2052.3  21781128.0    602427823.8      27.658247
```

The data that are generated are identical to the previous obesity data generated in SAS. Although these simple analytic examples have only scratched the surface of SAS and Python capabilities, the functional differences, including increased flexibility of Python—both in object-oriented syntax and expanded built-in data structures—are clearly demonstrated.

CONCLUSION

SAS and Python are widely popular industry leaders that occupy discrete corners of the analytics arena—proprietary and open-source. Although programming language selection is often made at the organization, team, customer, or product owner level, many developers and analysts do have the ability to select their *langue de choix,* so long as they can demonstrate that a language can deliver a functional solution. And in these flexible environments, open-source, freely available programming languages (such as Python) should be considered and explored as viable alternatives to pricey, proprietary software. The Pandas library and its DataFrame data structure are especially well-equipped for data analysis, and this text has demonstrated only a handful of its ever-expanding array of tools.

REFERENCES

CDC. (2022). *Behavioral Risk Factor Surveillance System (BFRSS)*. Retrieved from Centers for Disease Control and Prevention (CDC): https://gis.cdc.gov/grasp/diabetes/diabetesatlas-sdoh.html

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com