Manipulating Tabular Data

Lesson 5 with *Benoit Parmentier*

Lesson Objectives

- Review what makes a dataset tidy.
- Meet a complete set of functions for most table manipulations.
- Learn to transform datasets with split-apply-combine procedures.
- Understand the basic join operation.

Specific Achievements

- Reshape data frames with pandas
- Summarize data by groups with pandas
- Combine multiple data frame operations with method chaining (piping with pandas ".")
- Combine multiple data frames with "joins" (merge)

Data frames occupy a central place in Python data analysis pipelines. The pandas package provides the objects and most necessary tools to subset, reformat and transform data frames. The key functions in the package have close counterparts in SQL (Structured Query Language), which provides the added bonus of facilitating translation between python and relational databases.

Tidy Concept

Most time is spent on cleaning and wrangling data rather than analysis. In 2014, Hadley Wickam (R developer at RStudio) published a paper that defines the concepts underlying tidy datasets. Hadley Wickam defined tidy datasets as those where:

- each variable forms a column (also called field)
- each observation forms a row
- each type of observational unit forms a table

These guidelines may be familiar to some of you—they closely map to best practices for "normalization" in database design.It correspond to the 3rd normal form's described by Codd 1990 but uses the language of statical analysis rather than relationtional database.

Consider a data set where the outcome of an experiment has been recorded in a perfectly appropriate way:

bloc	drug	control	placebo
1	0.22	0.58	0.31
2	0.12	0.98	0.47
3	0.42	0.19	0.40

The response data are present in a compact matrix, as you might record it on a spreadsheet. The form does not match how we think about a statistical model, such as:

$$response \sim block + treatment$$

In a tidy format, each row is a complete observation: it includes the response value and all the predictor values. In this data, some of those predictor values are column headers, so the table needs to be reshaped. The pandas package provides functions to help re-organize tables.

The third principle of tidy data, one table per category of observed entities, becomes especially important in synthesis research. Following this principle requires holding tidy data in multiple tables, with associations between them formalized in metadata, as in a relational database.

Datasets split across multiple tables are unavoidable in synthesis research, and commonly used in the following two ways (often in combination):

- two tables are "un-tidied" by joins, or merging them into one table
- statistical models conform to the data model through a hierarchical structure or employing "random effects"

The pandas package includes several functions that all perform variations on table joins needed to "un-tidy" your tables, but there are only two basic types of table relationships to recognize:

- **One-to-one** relationships allow tables to be combined based on the same unique identifier (or "primary key") in both tables.
- **Many-to-one** relationships require non-unique "foreign keys" in the first table to match the primary key of the second.

Top of Section

Wide to Long

The pandas package's melt function reshapes "wide" data frames into "long" ones.

worksheet-5.ipynb

```
import pandas as pd
import numpy as np
trial_df = pd.DataFrame({"block": [1,2,3],
              "drug": [0.22,0.12,0.42],
              "control": [0.58,0.98,0.19],
              "placebo": [0.31,0.47,0.40]})
trial_df.head()
```

```
   block  control  drug  placebo
0      1     0.58  0.22     0.31
1      2     0.98  0.12     0.47
2      3     0.19  0.42     0.40
```

worksheet-5.ipynb

```
tidy_trial_df = pd.melt(trial_df,
                  id_vars=['block'],
                  var_name='treatment',
                  value_name='response')
tidy_trial_df.head()
```

```
   block treatment  response
0      1   control      0.58
1      2   control      0.98
2      3   control      0.19
3      1      drug      0.22
4      2      drug      0.12
```

All columns, accept for "block", are stacked in two columns: a "key" and a "value". The key column gets the name treatment and the value column receives the name response. For each row in the result, the key is taken from the name of the column and the value from the data in the column.

Long to Wide

Data can also fail to be tidy when a table is too long. The Entity-Attribute-Value (EAV) structure common in large databases distributes multiple attributes of a single entity/observation into separate rows.

Remember that the exact state of "tidy" may depend on the analysis: the key is knowing what counts as a complete observation. For example, the community ecology package vegan requires a matrix of species counts, where rows correspond to species and columns to sites. This may seem like too "wide" a format, but in the packages several multi-variate analyses, the abundance of a species across multiple sites is considered a complete observation.

Consider survey data on participant's age and income stored in a EAV structure.

worksheet-5.ipynb

```

df2 = tidy_trial_df.pivot(index='block',
                          columns='treatment',
                          values='response')
```

```
df2 = df2.reset_index()
df2.columns
```

```
Index(['block', 'control', 'drug', 'placebo'], dtype='object', name='treatment')
```

worksheet-5.ipynb

```
df2.reset_index()
```

```
treatment  index  block  control  drug  placebo
0              0      1     0.58  0.22     0.31
1              1      2     0.98  0.12     0.47
2              2      3     0.19  0.42     0.40
```

worksheet-5.ipynb

```
df2
```

```
treatment  block  control  drug  placebo
0              1     0.58  0.22     0.31
1              2     0.98  0.12     0.47
2              3     0.19  0.42     0.40
```

Consider survey data on participant's age and income *stored* in a EAV structure.

worksheet-5.ipynb

```

from io import StringIO, BytesIO

text_string = StringIO("""
participant,attr,val
1,age,24
2,age,57
3,age,13
1,income,30
2,income,60
""")

survey_df = pd.read_csv(text_string, sep=",")
survey_df
```

```
   participant    attr  val
0            1     age   24
1            2     age   57
2            3     age   13
3            1  income   30
4            2  income   60
```

Transform the data with the `pivot` function, which "reverses" a `melt`. These are equivalent to `spread` and `gather` in the dplyr r package.

worksheet-5.ipynb

```
tidy_survey = survey_df.pivot(index='participant',
                              columns='attr',
                              values='val')
print(tidy_survey.head())
```

```
attr          age  income
participant
1            24.0    30.0
2            57.0    60.0
3            13.0     NaN
```

worksheet-5.ipynb

```
tidy_survey = tidy_survey.reset_index()
tidy_survey.columns
```

```
Index(['participant', 'age', 'income'], dtype='object', name='attr')
```

worksheet-5.ipynb

```
tidy_survey.reset_index()
```

attr	index	participant	age	income
0	0	1	24.0	30.0
1	1	2	57.0	60.0
2	2	3	13.0	NaN

worksheet-5.ipynb

```
tidy_survey
```

attr	participant	age	income
0	1	24.0	30.0
1	2	57.0	60.0
2	3	13.0	NaN

Top of Section

Sample Data

Credit: US Census Bureau

To learn about data transformation with pandas, we need more data. The Census Bureau collects subnational economic data for the U.S., releasing annual County Business Patterns (CBP) datasets including the number of establishments, employment, and payroll by industry. They also conduct the American Community Survey (ACS) and publish, among other demographic and economic variables, estimates of median income for individuals working in different industries.

- County Business Patterns (CBP)
- American Community Survey (ACS)

worksheet-5.ipynb

```
import pandas as pd
cbp = pd.read_csv('data/cbp15co.csv')
cbp.describe()
```

worksheet-5.ipynb

```
print(cbp.dtypes)
```

```
FIPSTATE     int64
FIPSCTY      int64
```

```
NAICS        object
EMPFLAG      object
EMP_NF       object
EMP           int64
QP1_NF       object
QP1           int64
AP_NF        object
AP            int64
EST           int64
N1_4          int64
```

See the CBP dataset documentation for an explanation of the variables we don't discuss in this lesson.

Modify the import to clean up this read: consider the data type for FIPS codes along with what string in this CSV file represents NAs, a.k.a. data that is not-available or missing.

worksheet-5.ipynb

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

cbp = pd.read_csv(
  'data/cbp15co.csv',
  na_values = "NULL",
  keep_default_na=False,
  dtype =  {"FIPSTATE": np.str,
  "FIPSCTY": np.str}
  )
```

Question

What changed?

Answer

Using `dtypes()` shows that the character string `""` in the CSV file is no longer read into R as missing data (an `NA`) but as an empty string. The two named "FIPS" columns are now correctly read as strings.

worksheet-5.ipynb

```python
import pandas as pd
import numpy as np
acs =  pd.read_csv(
  'data/ACS/sector_ACS_15_5YR_S2413.csv',
  dtype = {"FIPS": np.str}
  )
```

Now let's display the data types

worksheet-5.ipynb

```python
#acs.dtypes
print(acs.dtypes)
```

```
FIPS              object
County            object
Sector            object
median_income    float64
dtype: object
```

The two datasets both contain economic variables for each U.S. county and specified by different categories of industry. The data could potentially be manipulated into a single table reflecting the follow statistical model.

$$median\_income \sim industry + establishment\_size$$

Top of Section

Key Functions

Function	Returns
`query`	keep rows that satisfy conditions
`assign`	apply a transformation to existing [split] columns
`['col1', 'col2']`	select and keep columns with matching names
`merge`	merge columns from separate tables into one table
`groupby`	split data into groups by an existing factor
`agg`	summarize across rows to use after groupby [and combine split groups]

The table above summarizes the most commonly used functions in pandas, which we will demonstrate in turn on data from the U.S. Census Bureau.

Filter and pattern matching

The `cbp` table includes character `NAICS` column. Of the 2 million observations, lets see how many observations are left when we keep only the 2-digit NAICS codes, representing high-level sectors of the economy.

worksheet-5.ipynb

```
#import pandas as pd
cbp2 = cbp[cbp['NAICS'].str.contains("----")]
cbp2 = cbp2[~cbp2.NAICS.str.contains("-----")]
cbp2.head()
```

```
   FIPSTATE FIPSCTY   NAICS EMPFLAG  ... N1000_3  N1000_4 CENSTATE  CENCTY
1        01     001  11----          ...       0        0       63       1
10       01     001  21----          ...       0        0       63       1
17       01     001  22----          ...       0        0       63       1
27       01     001  23----          ...       0        0       63       1
93       01     001  31----          ...       0        0       63       1

[5 rows x 26 columns]
```

Note that a logical we used the function `contains` from pandas to filter the dataset in two steps. The function contains allows for pattern matching of any character within strings. The `~` is used to remove the rows that contains specific patterns.

Filtering string often uses pattern matching by regular expressions which may be a bit more manageable, and streamlines the operations.

worksheet-5.ipynb

```
cbp3 = cbp[cbp['NAICS'].str.contains('[0-9]{2}----')]
cbp3.head()
```

```
   FIPSTATE FIPSCTY   NAICS EMPFLAG  ... N1000_3  N1000_4 CENSTATE  CENCTY
1        01     001  11----          ...       0        0       63       1
10       01     001  21----          ...       0        0       63       1
17       01     001  22----          ...       0        0       63       1
27       01     001  23----          ...       0        0       63       1
93       01     001  31----          ...       0        0       63       1

[5 rows x 26 columns]
```

Altering, updating and transforming columns

The `assign` function is the pandas answer to updating or altering your columns. It performs arbitrary operations on existing columns and appends the result as a new column of the same length.

Here are two ways to create a new column using `assign` and the `[ ]` operators.

worksheet-5.ipynb

```
cbp3["FIPS"] = cbp3["FIPSTATE"]+cbp3["FIPSCTY"]
```

```
/usr/bin/python3:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexir
```

worksheet-5.ipynb

```
cbp3.assign(FIPS2=lambda x: x['FIPSTATE']+x['FIPSCTY'])
```

```
        FIPSTATE FIPSCTY   NAICS  EMPFLAG  ... CENSTATE  CENCTY   FIPS   FIPS2
1             01     001  11----           ...       63       1  01001  01001
10            01     001  21----           ...       63       1  01001  01001
17            01     001  22----           ...       63       1  01001  01001
27            01     001  23----           ...       63       1  01001  01001
93            01     001  31----           ...       63       1  01001  01001
163           01     001  42----           ...       63       1  01001  01001
218           01     001  44----           ...       63       1  01001  01001
351           01     001  48----           ...       63       1  01001  01001
381           01     001  51----           ...       63       1  01001  01001
401           01     001  52----           ...       63       1  01001  01001
429           01     001  53----           ...       63       1  01001  01001
465           01     001  54----           ...       63       1  01001  01001
```

worksheet-5.ipynb

```
cbp3.shape
```

```
(58901, 27)
```

worksheet-5.ipynb

```
cbp3.head()
```

```
   FIPSTATE FIPSCTY   NAICS  EMPFLAG  ... N1000_4  CENSTATE  CENCTY   FIPS
1        01     001  11----           ...       0        63       1  01001
10       01     001  21----           ...       0        63       1  01001
17       01     001  22----           ...       0        63       1  01001
27       01     001  23----           ...       0        63       1  01001
93       01     001  31----           ...       0        63       1  01001

[5 rows x 27 columns]
```

Select

To keep particular columns of a data frame (rather than filtering rows), use the `filter` or `[ ]` functions with arguments that match column names.

worksheet-5.ipynb

```
cbp2.columns
```

```
Index(['FIPSTATE', 'FIPSCTY', 'NAICS', 'EMPFLAG', 'EMP_NF', 'EMP', 'QP1_NF',
       'QP1', 'AP_NF', 'AP', 'EST', 'N1_4', 'N5_9', 'N10_19', 'N20_49',
       'N50_99', 'N100_249', 'N250_499', 'N500_999', 'N1000', 'N1000_1',
       'N1000_2', 'N1000_3', 'N1000_4', 'CENSTATE', 'CENCTY'],
      dtype='object')
```

One way to "match" is by including complete names, each one you want to keep:

worksheet-5.ipynb

```
cbp3 = cbp3[['FIPS','NAICS','N1_4', 'N5_9', 'N10_19']]
cbp3.head()
```

```
     FIPS   NAICS  N1_4  N5_9  N10_19
1   01001  11----     5     1       0
10  01001  21----     0     1       1
17  01001  22----     2     1       2
27  01001  23----    51    13       7
93  01001  31----     9     4       4
```

Alternatively, we can use the `filter` function to select all columns starting with N or matching with 'FIPS' or 'NAICS' pattern. The `filter` command is useful when chaining methods (or piping operations).

worksheet-5.ipynb

```
cbp4= cbp.filter(regex='^N|FIPS|NAICS',axis=1)
cbp4.head()
```

```
  FIPSTATE FIPSCTY   NAICS  N1_4  ...  N1000_1  N1000_2  N1000_3  N1000_4
0       01     001  ------   430  ...        0        0        0        0
1       01     001  11----     5  ...        0        0        0        0
2       01     001  113///     4  ...        0        0        0        0
3       01     001  1133//     4  ...        0        0        0        0
4       01     001  11331/     4  ...        0        0        0        0

[5 rows x 16 columns]
```

Top of Section

Join

The CBP dataset uses FIPS to identify U.S. counties and NAICS codes to identify types of industry. The ACS dataset also uses FIPS but their data may aggregate across multiple NAICS codes representing a single industry sector.

worksheet-5.ipynb

```

sector =  pd.read_csv(
  'data/ACS/sector_naics.csv',
  dtype = {"NAICS": np.int64})
print(sector.dtypes)
```

```
Sector    object
NAICS      int64
dtype: object
```

worksheet-5.ipynb

```
print(cbp.dtypes)
```

```
FIPSTATE    object
FIPSCTY     object
NAICS       object
EMPFLAG     object
EMP_NF      object
EMP          int64
QP1_NF      object
QP1          int64
AP_NF       object
AP           int64
EST          int64
N1_4         int64
```

worksheet-5.ipynb

```
cbp.head()
```

```
  FIPSTATE FIPSCTY   NAICS EMPFLAG  ... N1000_3  N1000_4 CENSTATE  CENCTY
0       01     001  ------          ...       0        0       63       1
1       01     001  11----          ...       0        0       63       1
2       01     001  113///          ...       0        0       63       1
3       01     001  1133//          ...       0        0       63       1
4       01     001  11331/          ...       0        0       63       1

[5 rows x 26 columns]
```

worksheet-5.ipynb

```
cbp.dtypes
```

```
FIPSTATE    object
FIPSCTY     object
NAICS       object
EMPFLAG     object
EMP_NF      object
EMP          int64
QP1_NF      object
QP1          int64
AP_NF       object
AP           int64
EST          int64
N1_4         int64
N5_9         int64
```

worksheet-5.ipynb

```
cbp.head()
```

```
  FIPSTATE FIPSCTY   NAICS EMPFLAG  ... N1000_3  N1000_4 CENSTATE  CENCTY
0       01     001  ------          ...       0        0       63       1
1       01     001  11----          ...       0        0       63       1
2       01     001  113///          ...       0        0       63       1
3       01     001  1133//          ...       0        0       63       1
4       01     001  11331/          ...       0        0       63       1

[5 rows x 26 columns]
```

worksheet-5.ipynb

```
print(sector.dtypes)
```

```
Sector    object
NAICS      int64
dtype: object
```

worksheet-5.ipynb

```
print(sector.shape) #24 economic sectors
```

```
(24, 2)
```

worksheet-5.ipynb

```
sector.head()
```

```
                                      Sector  NAICS
0     agriculture forestry fishing and hunting     11
1  mining quarrying and oil and gas extraction     21
```

```
2                          utilities     22
3                       construction     23
4                      manufacturing     31
```

Probably the primary challenge in combining secondary datasets for synthesis research is dealing with their different sampling frames. A very common issue is that data are collected at different “scales”, with one dataset being at higher spatial or temporal resolution than another. The differences between the CBP and ACS categories of industry present a similar problem, and require the same solution of re-aggregating data at the “lower resolution”.

Many-to-One

Before performing the join operation, some preprocessing is necessary to extract from the NAICS columns the first two digits matching the sector identifiers.

worksheet-5.ipynb

```python
logical_idx = cbp['NAICS'].str.match('[0-9]{2}----') #boolean index
cbp = cbp.loc[logical_idx]
cbp.head()
```

```
   FIPSTATE FIPSCTY   NAICS  EMPFLAG  ... N1000_3  N1000_4 CENSTATE  CENCTY
1        01     001  11----           ...       0        0       63       1
10       01     001  21----           ...       0        0       63       1
17       01     001  22----           ...       0        0       63       1
27       01     001  23----           ...       0        0       63       1
93       01     001  31----           ...       0        0       63       1

[5 rows x 26 columns]
```

worksheet-5.ipynb

```python
cbp.shape
```

```
(58901, 26)
```

worksheet-5.ipynb

```python
cbp['NAICS']= cbp.NAICS.apply(lambda x: np.int64(x[0:2])) # select first two digits
```

worksheet-5.ipynb

```python
#Many to one to join economic sector code to NAICS

cbp_test = cbp.merge(sector, on = "NAICS", how='inner')
cbp_test.head()
```

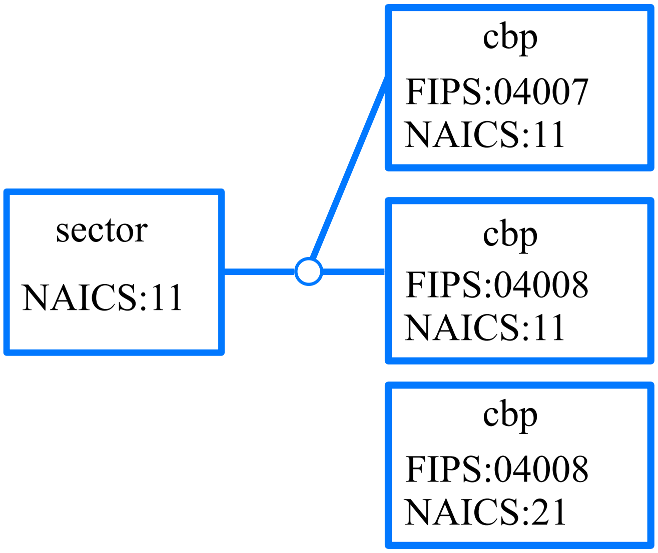
```
  FIPSTATE FIPSCTY  ...  CENCTY                                Sector
0       01     001  ...       1  agriculture forestry fishing and hunting
1       01     003  ...       3  agriculture forestry fishing and hunting
2       01     005  ...       5  agriculture forestry fishing and hunting
3       01     007  ...       7  agriculture forestry fishing and hunting
4       01     009  ...       9  agriculture forestry fishing and hunting

[5 rows x 27 columns]
```

worksheet-5.ipynb

```python
print(cbp_test.shape)
```

```
(56704, 27)
```



The NAICS field in the `cbp` table can have the same value multiple times, it is not a primary key in this table. In the `sector` table, the NAICS field is the primary key uniquely identifying each record. The type of relationship between these tables is therefore “many-to-one”.

Question
: Note that we lost a couple thousand rows through this join. How could `cbp` have fewer rows after a join on NAICS codes?

Answer
: The CBP data contains an NAICS code not mapped to a sector—the “error code” 99 is not present in `sector`. The use of “error codes” that could easilly be mistaken for data is frowned upon.

Group By

A very common data manipulation procedure know as “split-apply-combine” tackles the problem of applying the same transformation to subsets of data while keeping the result all together. We need the total number of establishments in each size class *aggregated within* each county and industry sector.

The pandas function `groupby` begins the process by indicating how the data frame should be split into subsets.

worksheet-5.ipynb

```
cbp["FIPS"] = cbp["FIPSTATE"]+cbp["FIPSCTY"]
cbp = cbp.merge(sector, on = "NAICS")

cbp_grouped = cbp.groupby(['FIPS','Sector'])
cbp_grouped
```

```
&lt;pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f5974d485f8&gt;
```

At this point, nothing has really changed:

worksheet-5.ipynb

```
cbp_grouped.dtypes
```

```
                                                     FIPSTATE  ... CENCTY
FIPS  Sector                                                   ...
01001 accommodation and food services                  object  ...  int64
      administrative and support and waste management...  object  ...  int64
      agriculture forestry fishing and hunting         object  ...  int64
      arts entertainment and recreation                object  ...  int64
      construction                                     object  ...  int64
      educational services                             object  ...  int64
      finance and insurance                            object  ...  int64
      health care and social assistance                object  ...  int64
```

```
      information                                        object  ...  int64
      management of companies and enterprises            object  ...  int64
```

The `groupby` statement generates a groupby data frame. You can add multiple variables (separated by commas) in `groupby`; each distinct combination of values across these columns defines a different group.

Summarize

The operation to perform on each group is summing: we need to sum the number of establishments in each group. Using pandas functions, the summaries are automically combined into a data frame.

worksheet-5.ipynb

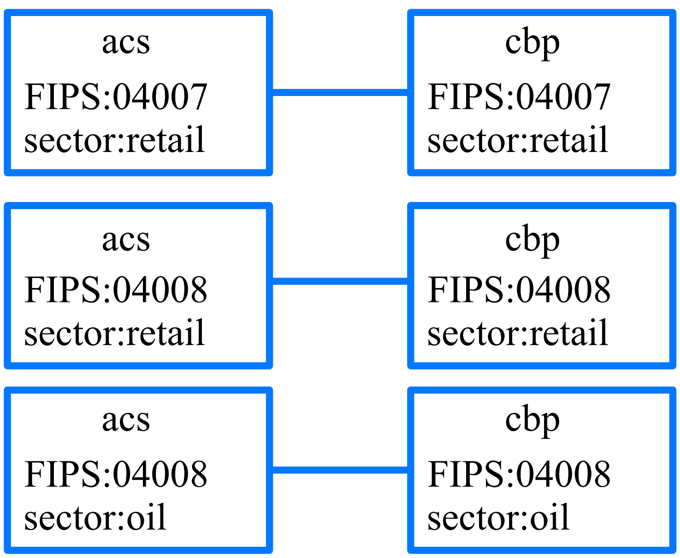
```python
grouped_df = (cbp
.groupby(['FIPS', 'Sector'])
.agg('sum')
.filter(regex='^N')
.drop(columns=['NAICS'])
)

grouped_df.head(5)
```

```
                                                         N1_4  ...  N1000_4
FIPS  Sector                                                   ...
01001 accommodation and food services                      23  ...        0
      administrative and support and waste management...  18  ...        0
      agriculture forestry fishing and hunting              5  ...        0
      arts entertainment and recreation                     5  ...        0
      construction                                         51  ...        0

[5 rows x 13 columns]
```

The “combine” part of “split-apply-combine” occurs automatically, when the attributes introduced by `groupby` are dropped. You can see attributes by running the `dtypes` function on the data frame.



There is now a one-to-one relationship between `cbp` and `acs`, based on the combination of FIPS and Sector as the primary key for both tables.

worksheet-5.ipynb

```python
print(grouped_df.shape)
```

```
(56704, 13)
```

worksheet-5.ipynb

```
print(acs.shape)
```

```
(59698, 4)
```

worksheet-5.ipynb

```
acs_cbp = grouped_df.merge(acs,on='FIPS',)
print(acs_cbp.shape)
```

```
(1061416, 17)
```

Again, however, the one-to-one relationship does not mean all rows are preserved by the join. The specific nature of the `inner_join` is to keep all rows, even duplicating rows if the relationship is many-to-one, where there are matching values in both tables, and discarding the rest.

The `acs_cbp` table now includes the `median_income` variable from the ACS and appropriatey aggregated establishment size information (the number of establishments by employee bins) from the CBP table.

worksheet-5.ipynb

```
acs_cbp.head()
```

```
    FIPS  N1_4  ...                                  Sector  median_income
0  01001    23  ...     agriculture forestry fishing and hunting        27235.0
1  01001    23  ...  mining quarrying and oil and gas extraction       102722.0
2  01001    23  ...                                 construction        31632.0
3  01001    23  ...                                manufacturing        40233.0
4  01001    23  ...                              wholesale trade        41656.0

[5 rows x 17 columns]
```

Top of Section

Additional Resources

The following cheat sheets and tutorials repeat much of this lesson, but also provide information on additional functions for “data wrangling”.

- Data Wrangling Cheat Sheet
- Tidyverse In Pandas
- String and Text With Pandas

The first is a set of cheat sheets created by pydata.org, and provides a handy, visual summary of all the key functions discussed in this lesson. It also lists some of the auxiliary functions that can be used within each type of expression, e.g. aggregation functions for summarize, “moving window” functions for mutate, etc. For those familiar with the tidyverse univers, please consult the second link.

Top of Section

If you need to catch-up before a section of code will work, just squish it's to copy code above it into your clipboard. Then paste into your interpreter's console, run, and you'll be ready to start in on that section. Code copied by both and will also appear below, where you can edit first, and then copy, paste, and run again.

```
# Nothing here yet!
```