```
import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Example: US States Data

Merge and join operations come up most often when combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at http://github.com/jakevdp/data-USstates:

Let's take a look at the three datasets, using the Pandas `read_csv` function:

```
pop = pd.read_csv('state-population.csv')
areas = pd.read_csv('state-areas.csv')
abbrevs = pd.read_csv('state-abbrevs.csv')

display('pop.head()', 'areas.head()', 'abbrevs.head()')
```

pop.head()

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

Given this information, say we want to compute a relatively straightforward result: rank US states and territories by their 2010 population density. We clearly have the data here to find this result, but we'll have to combine the datasets to do so.

We'll start with a many-to-one merge that will give us the full state names within the population `DataFrame`. We want to merge based on the `state/region` column of `pop` and the `abbreviation` column of `abbrevs`. We'll use `how='outer'` to make sure no data is thrown away due to mismatched labels:

```
merged = pop.merge(abbrevs, how='outer', left_on='state/region', right_on='abbre
merged = merged.drop('abbreviation', axis=1)                                   # dr
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

Let's double-check whether there were any mismatches here, which we can do by looking for rows with nulls:

```
merged.isnull().sum()
```

```
state/region     0
ages             0
year             0
population      20
state           96
dtype: int64
```

Some of the `population` values are null; let's figure out which these are!

```
merged[merged['population'].isnull()].head()
```

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

Double-click (or enter) to edit

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available in the original source.

More importantly, we see that some of the new `state` entries are also null, which means that there was no corresponding entry in the `abbrevs` key! Let's figure out which regions lack this match:

```
merged.loc[merged['state'].isnull(), 'state/region']
```

```
2448     PR
2449     PR
2450     PR
2451     PR
2452     PR
        ...
2539    USA
2540    USA
2541    USA
2542    USA
2543    USA
Name: state/region, Length: 96, dtype: object
```

```
merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

```
array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling in appropriate entries:

```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
```

```
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
```

```
merged.isnull().sum()
```

```
state/region     0
ages             0
year             0
population      20
state            0
dtype: int64
```

No more nulls in the `state` column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining our results, we will want to join on the `state` column in both:

```
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

```
areas.head()
```

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

```
entirepop = merged.merge(areas, on='state', how='left')
entirepop.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Again, let's check for nulls to see if there were any mismatches:

```
entirepop.isnull().sum()
```

```
state/region      0
ages              0
year              0
```

```
population       20
state             0
area (sq. mi)    48
dtype: int64
```

There are nulls in the `area` column; we can take a look to see which regions were ignored here:

```
entirepop.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

```
entirepop[entirepop['area (sq. mi)'].isnull()].head()
```

	state/region	ages	year	population	state	area (sq. mi)
2496	USA	under18	1990	64218512.0	United States	NaN
2497	USA	total	1990	249622814.0	United States	NaN
2498	USA	total	1991	252980942.0	United States	NaN

```
entirepop[entirepop['area (sq. mi)'].isnull()]['state'].unique()
```

```
array(['United States'], dtype=object)
```

```
entirepop.isnull().sum()
```

```
state/region      0
ages              0
year              0
population       20
state             0
area (sq. mi)    48
dtype: int64
```

We see that our `areas DataFrame` does not contain the area of the United States as a whole. We could insert the appropriate value (using the sum of all state areas, for instance), but in this case we'll just drop the null values because the population density of the entire United States is not relevant to our current discussion:

```
entirepop = entirepop.dropna()
entirepop.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

```
entirepop.isnull().sum()
```

```
state/region     0
ages             0
year             0
population       0
state            0
area (sq. mi)    0
dtype: int64
```

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2010, and the total population. We'll use the `query` function to do this quickly (this requires the NumExpr package to be installed; see [High-Performance Pandas: `eval()` and `query()`](#)):

```
filtered = entirepop[(entirepop['year'] == 2010) & (entirepop['ages'] == 'total'
# filtered = entirepop.query("year == 2010 & ages == 'total'")
filtered.head()
```

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570.0	Alabama	52423.0
91	AK	total	2010	713868.0	Alaska	656425.0
101	AZ	total	2010	6408790.0	Arizona	114006.0
189	AR	total	2010	2922280.0	Arkansas	53182.0
197	CA	total	2010	37333601.0	California	163707.0

```
len(filtered)
```

```
52
```

Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result:

```
filtered.set_index('state', inplace=True)
filtered['population_density'] = filtered['population'] / filtered['area (sq. mi
filtered.head()
```

```
<ipython-input-89-d21fe09ca0a2>:2: 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: https://pandas.pydata.org/pandas-docs
  filtered['population_density'] = filtered['population'] / filtered['area
```

	state/region	ages	year	population	area (sq. mi)	population_densit
state						
Alabama	AL	total	2010	4785570.0	52423.0	91.2876(
Alaska	AK	total	2010	713868.0	656425.0	1.0875(
Arizona	AZ	total	2010	6408790.0	114006.0	56.2144(
Arkansas	AR	total	2010	2922280.0	53182.0	54.9486(

```
filtered.sort_values(by='population_density', ascending=False, inplace=True)
filtered[['state/region', 'population_density']].head()
```

```
<ipython-input-97-51692f18306a>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs
  filtered.sort_values(by='population_density', ascending=False, inplace=T
```

	state/region	population_density
state		
District of Columbia	DC	8898.897059
Puerto Rico	PR	1058.665149
New Jersey	NJ	1009.253268
Rhode Island	RI	681.339159
Connecticut	CT	645.600649

The result is a ranking of US states, plus Washington, DC, and Puerto Rico, in order of their 2010 population density, in residents per square mile. We can see that by far the densest region in this dataset is Washington, DC (i.e., the District of Columbia); among states, the densest is New Jersey.

densest is New Jersey.

We can also check the end of the list:

```
filtered[['state/region', 'population_density']].tail()
```

	state/region	population_density
state		
South Dakota	SD	10.583512
North Dakota	ND	9.537565
Montana	MT	6.736171
Wyoming	WY	5.768079
Alaska	AK	1.087509

We see that the least dense state, by far, is Alaska, averaging slightly over one resident per square mile.

This type of data merging is a common task when trying to answer questions using real-world data sources. I hope that this example has given you an idea of some of the ways you can combine the tools we've covered in order to gain insight from your data!

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

```
# Exercise 1
# Today, we will be using the ACS data we used during out first pandas exercise
# To begin, load the ACS Data we used in our first pandas exercise. That data1 c

data = pd.read_stata('/content/US_ACS_2017_10pct_sample.dta')
data.head()
```

	year	datanum	serial	cbserial	numprec	subsamp	hhwt	hhtyp
0	2017	1	177686	2.017001e+12	9	64	55	femal householder no husban presen
1	2017	1	1200045	2.017001e+12	6	79	25	mal householder no wif presen
2	2017	1	70831	2.017000e+12	1 person record	36	57	mal householder living alon
3	2017	1	557128	2.017001e+12	2	10	98	married coupl famil househol
4	2017	1	614890	2.017001e+12	4	96	54	married coupl famil househol

5 rows × 104 columns

```
data.columns
```

```
Index(['year', 'datanum', 'serial', 'cbserial', 'numprec', 'subsamp',
'hhwt',
       'hhtype', 'cluster', 'adjust',
       ...
       'migcounty1', 'migmet131', 'vetdisab', 'diffrem', 'diffphys',
'diffmob',
       'diffcare', 'diffsens', 'diffeye', 'diffhear'],
      dtype='object', length=104)
```

```
data['inctot']
```

```
0         9999999
```

```
1           6000
2           6150
3          14000
4        9999999
          ...
318999     22130
319000   9999999
319001      5000
319002    240000
319003     48000
Name: inctot, Length: 319004, dtype: int32
```

```
# Exercise 2¶
# Let’s begin by calculating the mean US incomes from this data (recall that inc

EX2_AVG_INCOME = np.mean(data['inctot'])
```

```
1723646.2703978634
```

```
# Exercise 3¶
# Hmmm… That doesn’t look right. The average American is definitely not earning
# Let’s look at the values of inctot using value_counts(). Do you see a problem?
# Now use value_counts() with the argument normalize=True to see proportions of
# count of people in each category. What percentage of our sample has an income
# Store that proportion (between 0 and 1) as "EX3_SHARE_MAKING_9999999".
# What percentage has an income of 0? Store that proportion as "EX3_SHARE_MAKING

income_counter = data['inctot'].value_counts(normalize=True)
income_counter
```

```
9999999    0.168967
0          0.105575
30000      0.014978
50000      0.013837
40000      0.013834
             ...
70520      0.000003
76680      0.000003
57760      0.000003
200310     0.000003
505400     0.000003
Name: inctot, Length: 8471, dtype: float64
```

```
income_counter[9999999]
```

```
0.1689665333350052
```

```
income_counter[0]
```

```
0.10557547867738336
```

```
EX3_SHARE_MAKING_9999999 = income_counter[9999999] / len(data)
```

```
EX3_SHARE_MAKING_9999999
```

    5.296690114700919e-07

```
EX3_SHARE_MAKING_ZERO = income_counter[0] / len(data)
EX3_SHARE_MAKING_ZERO
```

    3.3095346352203535e-07

```
# Exercise 4¶
# As we discussed before, the ACS uses a value of 9999999 to denote that income
# The problem with using this kind of "sentinel value" is that pandas doesn't ur
# and so when it averages the variable, it doesn't know to ignore 9999999.
# To help out pandas, use the replace command to replace all values of 9999999 v
data.replace(9999999, np.nan, inplace=True)
data
```

	year	datanum	serial	cbserial	numprec	subsamp	hhwt	
0	2017	1	177686	2.017001e+12	9	64	55	hous no h
1	2017	1	1200045	2.017001e+12	6	79	25	hous
2	2017	1	70831	2.017000e+12	1 person record	36	57	hous livir
3	2017	1	557128	2.017001e+12	2	10	98	r ho
4	2017	1	614890	2.017001e+12	4	96	54	r ho
...	...	...	...	...	...	...	...	
318999	2017	1	734396	2.017001e+12	4	78	100	r ho
319000	2017	1	586263	2.017001e+12	4	57	77	r ho
319001	2017	1	510444	2.017001e+12	2	43	152	hous

319001	2017	1	310444	2.017001e+12	2	43	152	no h
319002	2017	1	1220474	2.017001e+12	4	16	148	r ho
319003	2017	1	219435	2.017000e+12	2	17	47	r ho

319004 rows × 104 columns

```
# Exercise 5¶
# Now that we've properly labeled our missing data as np.nan, let's calculate th
income_counter = data['inctot'].value_counts(normalize=True)
income_counter
```

```
0.0         0.127041
30000.0     0.018023
50000.0     0.016650
40000.0     0.016646
20000.0     0.015341
              ...
246600.0    0.000004
90810.0     0.000004
341380.0    0.000004
15790.0     0.000004
505400.0    0.000004
Name: inctot, Length: 8470, dtype: float64
```

```
# Exercise 6¶
# OK, now we've been able to get a reasonable average income number. As we can s
# But it's not enough to just get rid of the people who had inctot values of 999
# anyone who made more than 100,000 dollars: if we just dropped those people, th

# So let's make sure we understand why data is missing for some people. If you r
# be the case that most of the people who had incomes of 9999999 were children.
# the variable age for people for whom inctot is missing (i.e. subset the data t

# Then do the opposite: look at the distribution of the age variable for people
# Can you determine when 9999999 was being used? Is it ok we're excluding those
# Note: In this data, Python doesn't understand age is a number; it thinks it is
# like "90 (90+ in 1980 and 1990)" and "less than 1 year old". So you can't just
# We'll discuss converting string variables into numbers in a future class.


print("Null ", data['inctot'].isnull().sum())

data[data['inctot'].isnull()]['age'].value_counts()
```

```
Null  53901
```

```
10    3997
9     3977
14    3847
12    3845
13    3800
      ...
39       0
38       0
37       0
36       0
96       0
Name: age, Length: 97, dtype: int64
```

```
# Exercise 7¶
# Great, so now we know why those people had missing data, and we’re ok with exc
# But as we previously noted, there are also a lot of observations of zero incom
# Let’s limit our attention to people who are currently working by subsetting to
data['empstat'].value_counts()
```

```
employed              148758
not in labor force    104676
n/a                    57843
unemployed              7727
Name: empstat, dtype: int64
```

```
employed = data[data['empstat'] == 'employed']
employed
```

	year	datanum	serial	cbserial	numprec	subsamp	hhwt	
1	2017	1	1200045	2.017001e+12	6	79	25	hous
2	2017	1	70831	2.017000e+12	1 person record	36	57	hous livir
5	2017	1	563897	2.017001e+12	3	19	66	hous no h
9	2017	1	856859	2.017001e+12	5	69	12	r ho
10	2017	1	175930	2.017001e+12	9	72	171	r ho
...	...	...	...	...	...	...	...	

318995	2017	1	46231	2.017001e+12	3	36	104	r ho
318999	2017	1	734396	2.017001e+12	4	78	100	r ho
319001	2017	1	510444	2.017001e+12	2	43	152	hous no h
319002	2017	1	1220474	2.017001e+12	4	16	148	r ho
319003	2017	1	219435	2.017000e+12	2	17	47	r ho

148758 rows × 104 columns

```
employed['race'].value_counts()
```

```
white                               116017
black/african american/negro         13175
other asian or pacific islander       6424
other race, nec                       5755
two major races                       3135
chinese                               2149
american indian or alaska native      1290
three or more major races              426
japanese                               387
Name: race, dtype: int64
```

```
employed[employed['race'] == 'white']['inctot'].mean()
```

```
60473.15372747098
```

```
# Exercise 8¶
# Now let’s estimate the racial income gap in the United States. What is the ave
# In percentage terms (between 0 and 100), how much more does the average White
# Note: these values are not quite accurate estimates. As we’ll discuss in later
# Note: This is actually an underestimate of the wage gap. The US Census treats

EX8_AVG_INCOME_WHITE = employed[employed['race'] == 'white']['inctot'].mean()
EX8_AVG_INCOME_BLACK = employed[employed['race'] == 'black/african american/negr

EX8_RACIAL_DIFFERENCE = ((EX8_AVG_INCOME_WHITE - EX8_AVG_INCOME_BLACK) / EX8_AVG
```

```
print(EX8_AVG_INCOME_WHITE, EX8_AVG_INCOME_BLACK, EX8_RACIAL_DIFFERENCE)
```

```
60473.15372747098 41747.949905123336 44.85299006275197
```

```
# Exercise 9¶
# As noted above, these estimates are not actually quite correct because we arer
# (As you can see, when  is constant for all observations, this just simplifies
# In this data, weights are stored in the variable perwt, which is the number of
# Using the formula, re-calculate the weighted average income for both populatic

EX9_AVG_INCOME = (data['inctot'] * data['perwt']).sum() / data['perwt'].sum()
EX9_AVG_INCOME
```

```
32135.705606168296
```

```
employed['hispan']
```

```
1                 other
2          not hispanic
5          not hispanic
9          not hispanic
10              mexican
              ...
318995          mexican
318999     not hispanic
319001     not hispanic
319002     not hispanic
319003     not hispanic
Name: hispan, Length: 148758, dtype: category
Categories (5, object): ['not hispanic' < 'mexican' < 'puerto rican' <
'cuban' < 'other']
```

```
# Exercise 10¶
# While all ethnic distinctions are socially constructed, and so on some level t
# So now calculate the weighted average income gap between non-Hispanic White Am
non_Hispanic_White_Americans = employed[(employed['race'] == 'white') & (employe
non_Hispanic_White_Americans.head()
```

	year	datanum	serial	cbserial	numprec	subsamp	hhwt	hhtyp
2	2017	1	70831	2.017000e+12	1 person record	36	57	mal householder living alon
5	2017	1	563897	2.017001e+12	3	19	66	femal householder no husban presen
								marriec coupl

9	2017	1	856859	2.017001e+12	5	69	12	coupl famil househol
12	2017	1	331527	2.017001e+12	4	3	101	hhtyp could not b determine
16	2017	1	274584	2.017000e+12	2	29	42	marriec coupl famil househol

5 rows × 104 columns

```
EX10_WAGE_GAP = ((non_Hispanic_White_Americans['inctot'].mean() - EX8_AVG_INCOME
EX10_WAGE_GAP
```

```
51.02404906450736
```

```
# Exercise 11¶
# Is that greater or less than the difference you found in Exercise 8? Why do yc
```