Indexing Dataframes

As we've seen, both series and DataFrames can have indices applied to them.
The index is essentially a row level label, and we know that rows correspond to axis zero.
In our Olympics data, we indexed the data frame by the name of the country.
Indices can either be inferred, such as when we create a new series without an index, in which case we get numeric values, or they can be set explicitly, like when we use the dictionary object to create the series, or when we loaded data from the CSV file and specified the header.
Another option for setting an index is to use the set_index function.
This function takes a list of columns and promotes those columns to an index.
Set index is a destructive process; it doesn't keep the current index.
If you want to keep the current index, you need to manually create a new column and copy into it values from the index attribute.

Let's go back to our Olympics DataFrame.
Let's say that we don't want to index the DataFrame by countries, but instead want to index by the number of gold medals that were won at summer games.
First we need to preserve the country information into a new column.
We can do this using the indexing operator or the string that has the column label.
Then we can use the set_index to set index of the column to summer gold medal wins.

```
df.head()
```

	# Summer	Gold	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total
Afghanistan (AFG)	13	0	0	2	2	0	0	0	0	0	13	0	0	2	2
Algeria (ALG)	12	5	2	8	15	3	0	0	0	0	15	5	2	8	15
Argentina (ARG)	23	18	24	28	70	18	0	0	0	0	41	18	24	28	70
Armenia (ARM)	5	1	2	9	12	6	0	0	0	0	11	1	2	9	12
Australasia (ANZ) [ANZ]	2	3	4	5	12	0	0	0	0	0	2	3	4	5	12

```
df['country'] = df.index
df = df.set_index('Gold')
df.head()
```

	# Summer	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total	country
Gold															
0	13	0	2	2	0	0	0	0	0	13	0	0	2	2	Afghanistan (AFG)
5	12	2	8	15	3	0	0	0	0	15	5	2	8	15	Algeria (ALG)
18	23	24	28	70	18	0	0	0	0	41	18	24	28	70	Argentina (ARG)
1	5	2	9	12	6	0	0	0	0	11	1	2	9	12	Armenia (ARM)
3	2	4	5	12	0	0	0	0	0	2	3	4	5	12	Australasia (ANZ) [ANZ]

You'll see that when we create a new index from an existing column it appears that a new first row has been added with empty values.
This isn't quite what's happening.
And we know this in part because an empty value is actually rendered either as a none or an NaN if the data type of the column is numeric.
What's actually happened is that the index has a name.
Whatever the column name was in the Jupiter notebook has just provided this in the output.
We can get rid of the index completely by calling the function reset_index.
This promotes the index into a column and creates a default numbered index.

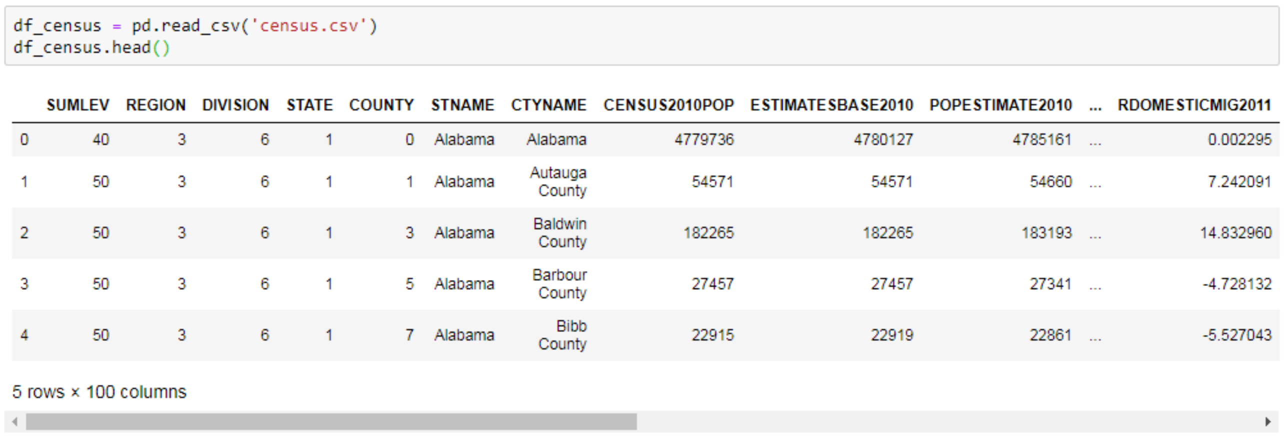
```
df = df.reset_index()
df.head()
```

	Gold	# Summer	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total	country
0	0	13	0	2	2	0	0	0	0	0	13	0	0	2	2	Afghanistan (AFG)
1	5	12	2	8	15	3	0	0	0	0	15	5	2	8	15	Algeria (ALG)
2	18	23	24	28	70	18	0	0	0	0	41	18	24	28	70	Argentina (ARG)
3	1	5	2	9	12	6	0	0	0	0	11	1	2	9	12	Armenia (ARM)
4	3	2	4	5	12	0	0	0	0	0	2	3	4	5	12	Australasia (ANZ) [ANZ]

One nice feature of pandas is that it has the option to do multi-level indexing.
This is similar to composite keys in relational database systems.
To create a multi-level index, we simply call set index and give it a list of columns that we're interested in promoting to an index.

Pandas will search through these in order, finding the distinct data and forming composite indices.
A good example of this is often found when dealing with geographical data which is sorted by regions or demographics.

Let's change data sets and look at some census data for a better example.
This data is stored in the file census.csv and comes from the United States Census Bureau.
In particular, this is a breakdown of the population level data at the US county level.
It's a great example of how different kinds of data sets might be formatted when you're trying to clean them.
For instance, in this data set there are two summarized levels, one that contains summary data for the whole country. And one that contains summary data for each state, and one that contains summary data for each county.



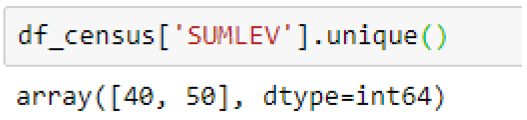
```
df_census = pd.read_csv('census.csv')
df_census.head()
```

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOMESTICMIG2011
0	40	3	6	1	0	Alabama	Alabama	4779736	4780127	4785161	...	0.002295
1	50	3	6	1	1	Alabama	Autauga County	54571	54571	54660	...	7.242091
2	50	3	6	1	3	Alabama	Baldwin County	182265	182265	183193	...	14.832960
3	50	3	6	1	5	Alabama	Barbour County	27457	27457	27341	...	-4.728132
4	50	3	6	1	7	Alabama	Bibb County	22915	22919	22861	...	-5.527043

5 rows × 100 columns

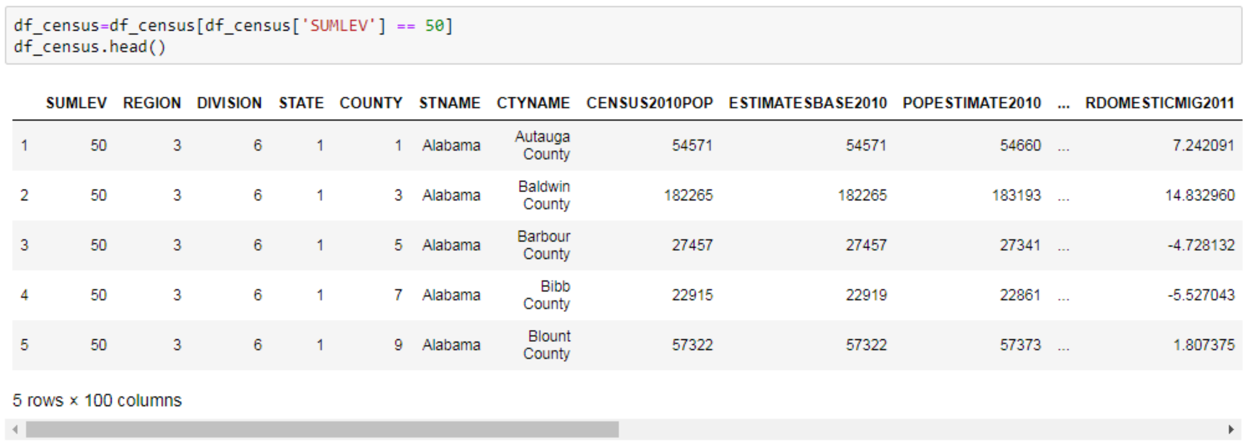
I often find that I want to see a list of all the unique values in a given column. In this DataFrame, we see that the possible values for the sum level are using the unique function on the DataFrame. This is similar to the SQL distinct operator.

Here we can run unique on the sum level of our current DataFrame and see that there are only two different values, 40 and 50.



```
df_census['SUMLEV'].unique()
```

```
array([40, 50], dtype=int64)
```



```
df_census=df_census[df_census['SUMLEV'] == 50]
df_census.head()
```

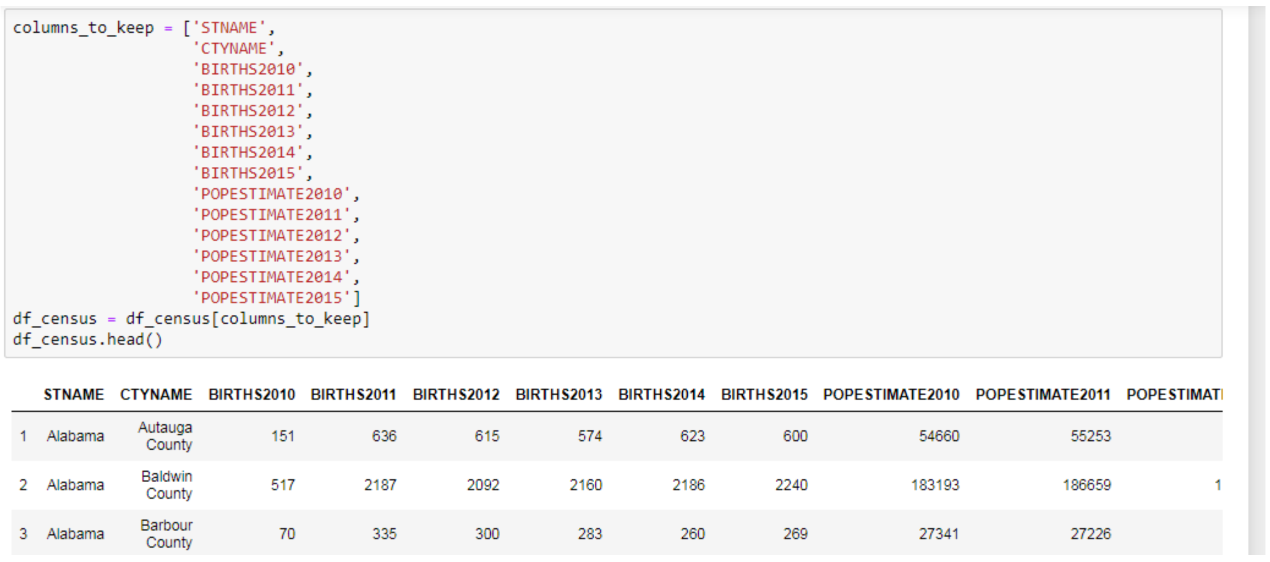
	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOMESTICMIG2011
1	50	3	6	1	1	Alabama	Autauga County	54571	54571	54660	...	7.242091
2	50	3	6	1	3	Alabama	Baldwin County	182265	182265	183193	...	14.832960
3	50	3	6	1	5	Alabama	Barbour County	27457	27457	27341	...	-4.728132
4	50	3	6	1	7	Alabama	Bibb County	22915	22919	22861	...	-5.527043
5	50	3	6	1	9	Alabama	Blount County	57322	57322	57373	...	1.807375

5 rows × 100 columns

Let's get rid of all of the rows that are summaries at the state level and just keep the county data.

Also while this data set is interesting for a number of different reasons, let's reduce the data that we're going to look at to just the total population estimates and the total number of births.

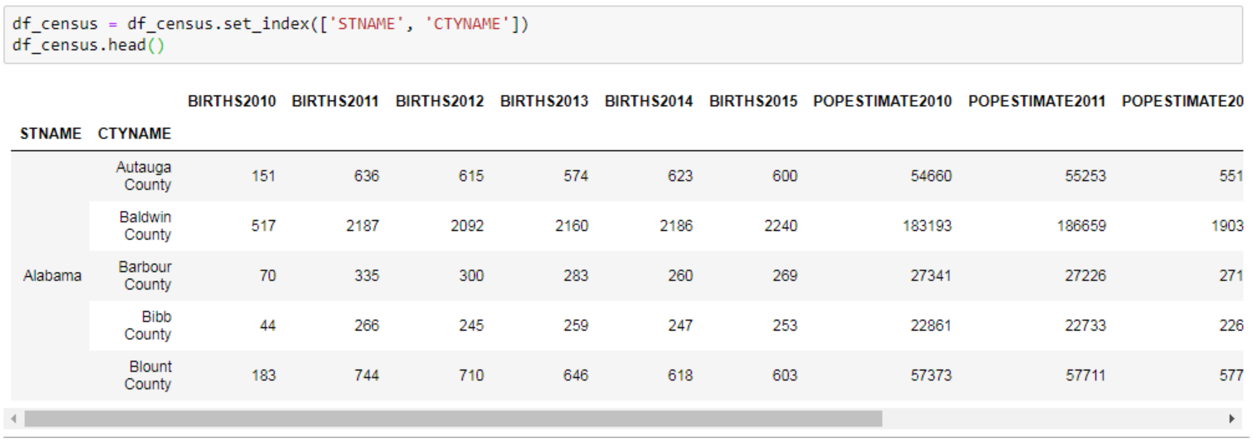
We can do this by creating a list of column names that we want to keep then project those and assign the resulting DataFrame to our df_census variable.



```
columns_to_keep = ['STNAME',
                   'CTYNAME',
                   'BIRTHS2010',
                   'BIRTHS2011',
                   'BIRTHS2012',
                   'BIRTHS2013',
                   'BIRTHS2014',
                   'BIRTHS2015',
                   'POPESTIMATE2010',
                   'POPESTIMATE2011',
                   'POPESTIMATE2012',
                   'POPESTIMATE2013',
                   'POPESTIMATE2014',
                   'POPESTIMATE2015']
df_census = df_census[columns_to_keep]
df_census.head()
```

	STNAME	CTYNAME	BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	BIRTHS2015	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATI
1	Alabama	Autauga County	151	636	615	574	623	600	54660	55253	
2	Alabama	Baldwin County	517	2187	2092	2160	2186	2240	183193	186659	1
3	Alabama	Barbour County	70	335	300	283	260	269	27341	27226	

The US Census data breaks down estimates of population data by state and county.
We can load the data and set the index to be a combination of the state and county values and see how pandas handles it in a DataFrame.
We do this by creating a list of the column identifiers we want to have indexed.
And then calling set index with this list and assigning the output as appropriate.
We see here that we have a dual index, first the state name and then the county name.



```
df_census = df_census.set_index(['STNAME', 'CTYNAME'])
df_census.head()
```

		BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	BIRTHS2015	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE20
STNAME	**CTYNAME**									
Alabama	Autauga County	151	636	615	574	623	600	54660	55253	551
	Baldwin County	517	2187	2092	2160	2186	2240	183193	186659	1903
	Barbour County	70	335	300	283	260	269	27341	27226	271
	Bibb County	44	266	245	259	247	253	22861	22733	226
	Blount County	183	744	710	646	618	603	57373	57711	577

An immediate question which comes up is how we can query this DataFrame.
For instance, we saw previously that the loc attribute of the DataFrame can take multiple arguments.
And it could query both the row and the columns.
When you use a MultiIndex, you must provide the arguments in order by the level you wish to query.
Inside of the index, each column is called a level and the outermost column is level zero.

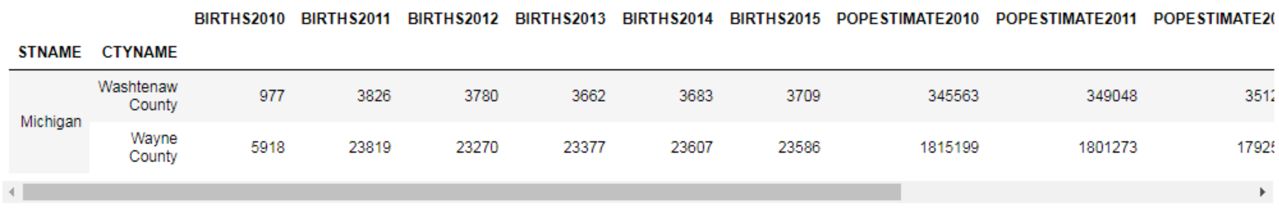
For instance, if we want to see the population results from Washtenaw County, which is where I live, you'd want to the first argument as the state of Michigan.
You might be interested in just comparing two counties.
For instance, Washtenaw where I live and Wayne County which covers Detroit.
To do this, we can pass the loc method, a list of tuples which describe the indices we wish to query.
Since we have a MultiIndex of two values, the state and the county, we need to provide two values as each element of our filtering list.

```
df_census.loc['Michigan', 'Washtenaw County']
```

```
BIRTHS2010              977
BIRTHS2011             3826
BIRTHS2012             3780
BIRTHS2013             3662
BIRTHS2014             3683
BIRTHS2015             3709
POPESTIMATE2010      345563
POPESTIMATE2011      349048
POPESTIMATE2012      351213
POPESTIMATE2013      354289
POPESTIMATE2014      357029
POPESTIMATE2015      358880
Name: (Michigan, Washtenaw County), dtype: int64
```

Okay so that's how hierarchical indices work in a nutshell.
They're a special part of the pandas library which I think can make management and reasoning about data easier.
Of course hierarchical labeling isn't just for rows.
For example, you can transpose this matrix and now have hierarchical column labels.
And projecting a single column which has these labels works exactly the way you would expect it to.

```
df_census.loc[ [('Michigan', 'Washtenaw County'),
         ('Michigan', 'Wayne County')] ]
```



		BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	BIRTHS2015	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE2(
STNAME	CTYNAME									
Michigan	Washtenaw County	977	3826	3780	3662	3683	3709	345563	349048	3512
	Wayne County	5918	23819	23270	23377	23607	23586	1815199	1801273	1792