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 in pandas the rows correspond to axis zero. Indices can either be either autogenerated, 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 set appropriate parameters. 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. In this lecture we'll explore more about how indexes work in pandas.

Loading Data

In [1]:

```
# Lets import pandas and our admissions dataset
import pandas as pd
df = pd.read_csv("datasets/Admission_Predict.csv", index_col=0)
df.head()
```

Out[1]:

Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
1	337	118	4	4.5	4.5	9.65	1	0.92
2	324	107	4	4.0	4.5	8.87	1	0.76
3	316	104	3	3.0	3.5	8.00	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.80
5	314	103	2	2.0	3.0	8.21	0	0.65

Using `set_index()` function

The `set_index()` function is a destructive process, and 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. 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**. In this lecture we'll explore more about how indexes work in pandas.

In [2]:

```
# Let's say that we don't want to index the DataFrame by serial numbers, but ins
tead by the
# chance of admit. But lets assume we want to keep the serial number for later.
 So, lets
# preserve the serial number into a new column. We can do this using the indexin
g operator
# on the string that has the column label. Then we can use the set_index to set
 index
# of the column to chance of admit

# So we copy the indexed data into its own column (if you don't plan to do this
 you don't need the first line. )
df['Serial Number'] = df.index
# Then we set the index to another column
df = df.set_index('Chance of Admit ')
df.head()
```

Out[2]:

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Serial Number
Chance of Admit								
0.92	337	118	4	4.5	4.5	9.65	1	1
0.76	324	107	4	4.0	4.5	8.87	1	2
0.72	316	104	3	3.0	3.5	8.00	1	3
0.80	322	110	3	3.5	2.5	8.67	1	4
0.65	314	103	2	2.0	3.0	8.21	0	5

Using `reset_index()` Function

In [3]:

```
# You'll see that when we create a new index from an existing column the index h
as a name,
# which is the original name of the column.

# 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()
```

Out[3]:

	Chance of Admit	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Serial Number
0	0.92	337	118	4	4.5	4.5	9.65	1	1
1	0.76	324	107	4	4.0	4.5	8.87	1	2
2	0.72	316	104	3	3.0	3.5	8.00	1	3
3	0.80	322	110	3	3.5	2.5	8.67	1	4
4	0.65	314	103	2	2.0	3.0	8.21	0	5

Multi-Level Indexing

One nice feature of Pandas is 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 form 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.

In [4]:

```
# Let's import and see what the data looks like
df = pd.read_csv('datasets/census.csv')
df.head()
```

Out[4]:

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	E
0	40	3	6	1	0	Alabama	Alabama	4779736	
1	50	3	6	1	1	Alabama	Autauga County	54571	
2	50	3	6	1	3	Alabama	Baldwin County	182265	
3	50	3	6	1	5	Alabama	Barbour County	27457	
4	50	3	6	1	7	Alabama	Bibb County	22915	

5 rows × 100 columns

In [5]:

```
# 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.
# 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
df['SUMLEV'].unique()
```

Out[5]:

```
array([40, 50])
```

In [6]:

```
# We see that there are only two different values, 40 and 50
```

In [7]:

```
# Let's exclude all of the rows that are summaries
# at the state level and just keep the county data.
df=df[df['SUMLEV'] == 50]
df.head()
```

Out[7]:

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	E
1	50	3	6	1	1	Alabama	Autauga County	54571	
2	50	3	6	1	3	Alabama	Baldwin County	182265	
3	50	3	6	1	5	Alabama	Barbour County	27457	
4	50	3	6	1	7	Alabama	Bibb County	22915	
5	50	3	6	1	9	Alabama	Blount County	57322	

5 rows × 100 columns

In [8]:

```
# 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 variable.

columns_to_keep = ['STNAME','CTYNAME','BIRTHS2010','BIRTHS2011','BIRTHS2012','BI
RTHS2013',
                   'BIRTHS2014','BIRTHS2015','POPESTIMATE2010','POPESTIMATE2011'
,
                   'POPESTIMATE2012','POPESTIMATE2013','POPESTIMATE2014','POPEST
IMATE2015']
df = df[columns_to_keep]
df.head()
```

Out[8]:

	STNAME	CTYNAME	BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	E
1	Alabama	Autauga County	151	636	615	574	623	
2	Alabama	Baldwin County	517	2187	2092	2160	2186	
3	Alabama	Barbour County	70	335	300	283	260	
4	Alabama	Bibb County	44	266	245	259	247	
5	Alabama	Blount County	183	744	710	646	618	

In [9]:

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

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

Out[9]:

		BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	BIRT
STNAME	**CTYNAME**						
Alabama	**Autauga County**	151	636	615	574	623	
	Baldwin County	517	2187	2092	2160	2186	
	Barbour County	70	335	300	283	260	
	Bibb County	44	266	245	259	247	
	Blount County	183	744	710	646	618	

Querying The DataFrame using `.loc[]`

In [10]:

```
# An immediate question which comes up is how we can query this DataFrame. We sa
w previously that
# the loc attribute of the DataFrame can take multiple arguments. And it could q
uery 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 an
d the outermost
# column is level zero.

# If we want to see the population results from Washtenaw County in Michigan the
state, which is
# where I live, the first argument would be Michigan and the second would be Was
htenaw County
df.loc['Michigan', 'Washtenaw County']
```

Out[10]:

```
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
```

In [11]:

```
# If you are interested in comparing two counties, for example, Washtenaw and Wa
yne County, we can
# pass a list of tuples describing the indices we wish to query into loc. 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. Each tuple should have two elements, the first element being t
he first index and
# the second element being the second index.

# Therefore, in this case, we will have a list of two tuples, in each tuple, the
first element is
# Michigan, and the second element is either Washtenaw County or Wayne County

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

Out[11]:

		BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	BII
STNAME	**CTYNAME**						
Michigan	**Washtenaw County**	977	3826	3780	3662	3683	
	Wayne County	5918	23819	23270	23377	23607	

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. Now, in reality, I don't tend to use hierarchical indicies very much, and instead just keep everything as columns and manipulate those. But, it's a unique and sophisticated aspect of pandas that is useful to know, especially if viewing your data in a tabular form.