

Indexing DataFrame_ed

July 21, 2021

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.

```
[1]: # 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.

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

```
[1]:
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	\
Serial No.							
1	337	118	4	4.5	4.5	9.65	
2	324	107	4	4.0	4.5	8.87	
3	316	104	3	3.0	3.5	8.00	
4	322	110	3	3.5	2.5	8.67	
5	314	103	2	2.0	3.0	8.21	

	Research	Chance of Admit
Serial No.		
1	1	0.92
2	1	0.76
3	1	0.72
4	1	0.80
5	0	0.65

```
[2]: # Let's say that we don't want to index the DataFrame by serial numbers, but
      # instead 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
→indexing 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
df['Serial Number'] = df.index
# Then we set the index to another column
df = df.set_index('Chance of Admit ')
df.head()

```

[2]:

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

	Research	Serial Number
Chance of Admit		
0.92	1	1
0.76	1	2
0.72	1	3
0.80	1	4
0.65	0	5

[3]:

```

# You'll see that when we create a new index from an existing column the index
→has 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()

```

[3]:

	Chance of Admit	GRE Score	TOEFL Score	University Rating	SOP	LOR	\
0	0.92	337	118	4	4.5	4.5	
1	0.76	324	107	4	4.0	4.5	
2	0.72	316	104	3	3.0	3.5	
3	0.80	322	110	3	3.5	2.5	
4	0.65	314	103	2	2.0	3.0	

	CGPA	Research	Serial Number
--	------	----------	---------------

0	9.65	1	1
1	8.87	1	2
2	8.00	1	3
3	8.67	1	4
4	8.21	0	5

```
[4]: # 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.

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

```
[4]:
```

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	\
0	40	3	6	1	0	Alabama	Alabama	
1	50	3	6	1	1	Alabama	Autauga County	
2	50	3	6	1	3	Alabama	Baldwin County	
3	50	3	6	1	5	Alabama	Barbour County	
4	50	3	6	1	7	Alabama	Bibb County	

	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOMESTICMIG2011	\
0	4779736		4780127	4785161	...	0.002295
1	54571		54571	54660	...	7.242091
2	182265		182265	183193	...	14.832960
3	27457		27457	27341	...	-4.728132
4	22915		22919	22861	...	-5.527043

	RDOMESTICMIG2012	RDOMESTICMIG2013	RDOMESTICMIG2014	RDOMESTICMIG2015	\
0	-0.193196	0.381066	0.582002	-0.467369	

1	-2.915927	-3.012349	2.265971	-2.530799
2	17.647293	21.845705	19.243287	17.197872
3	-2.500690	-7.056824	-3.904217	-10.543299
4	-5.068871	-6.201001	-0.177537	0.177258

	RNETMIG2011	RNETMIG2012	RNETMIG2013	RNETMIG2014	RNETMIG2015
0	1.030015	0.826644	1.383282	1.724718	0.712594
1	7.606016	-2.626146	-2.722002	2.592270	-2.187333
2	15.844176	18.559627	22.727626	20.317142	18.293499
3	-4.874741	-2.758113	-7.167664	-3.978583	-10.543299
4	-5.088389	-4.363636	-5.403729	0.754533	1.107861

[5 rows x 100 columns]

```
[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()
```

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

```
[6]: # We see that there are only two different values, 40 and 50
```

```
[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()
```

[7]:	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME \
1	50	3	6	1	1	Alabama	Autauga County
2	50	3	6	1	3	Alabama	Baldwin County
3	50	3	6	1	5	Alabama	Barbour County
4	50	3	6	1	7	Alabama	Bibb County
5	50	3	6	1	9	Alabama	Blount County

	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOMESTICMIG2011 \
1	54571	54571	54660	...	7.242091
2	182265	182265	183193	...	14.832960
3	27457	27457	27341	...	-4.728132
4	22915	22919	22861	...	-5.527043
5	57322	57322	57373	...	1.807375

	RDOMESTICMIG2012	RDOMESTICMIG2013	RDOMESTICMIG2014	RDOMESTICMIG2015 \
1	-2.915927	-3.012349	2.265971	-2.530799

2	17.647293	21.845705	19.243287	17.197872
3	-2.500690	-7.056824	-3.904217	-10.543299
4	-5.068871	-6.201001	-0.177537	0.177258
5	-1.177622	-1.748766	-2.062535	-1.369970

	RNETMIG2011	RNETMIG2012	RNETMIG2013	RNETMIG2014	RNETMIG2015
1	7.606016	-2.626146	-2.722002	2.592270	-2.187333
2	15.844176	18.559627	22.727626	20.317142	18.293499
3	-4.874741	-2.758113	-7.167664	-3.978583	-10.543299
4	-5.088389	-4.363636	-5.403729	0.754533	1.107861
5	1.859511	-0.848580	-1.402476	-1.577232	-0.884411

[5 rows x 100 columns]

```
[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', 'BIRTHS2013',
    'BIRTHS2014', 'BIRTHS2015', 'POPESTIMATE2010', 'POPESTIMATE2011',
    'POPESTIMATE2012', 'POPESTIMATE2013', 'POPESTIMATE2014', 'POPESTIMATE2015']
df = df[columns_to_keep]
df.head()
```

```
[8]:
```

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

	BIRTHS2014	BIRTHS2015	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE2012	\
1	623	600	54660	55253	55175	
2	2186	2240	183193	186659	190396	
3	260	269	27341	27226	27159	
4	247	253	22861	22733	22642	
5	618	603	57373	57711	57776	

	POPESTIMATE2013	POPESTIMATE2014	POPESTIMATE2015
1	55038	55290	55347
2	195126	199713	203709
3	26973	26815	26489

4	22512	22549	22583
5	57734	57658	57673

```
[9]: # The US Census data breaks down population estimates 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 second the county name.

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

```
[9]:
```

		BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	\
STNAME	CTYNAME					
Alabama	Autauga County	151	636	615	574	
	Baldwin County	517	2187	2092	2160	
	Barbour County	70	335	300	283	
	Bibb County	44	266	245	259	
	Blount County	183	744	710	646	

		BIRTHS2014	BIRTHS2015	POPESTIMATE2010	\
STNAME	CTYNAME				
Alabama	Autauga County	623	600	54660	
	Baldwin County	2186	2240	183193	
	Barbour County	260	269	27341	
	Bibb County	247	253	22861	
	Blount County	618	603	57373	

		POPESTIMATE2011	POPESTIMATE2012	POPESTIMATE2013	\
STNAME	CTYNAME				
Alabama	Autauga County	55253	55175	55038	
	Baldwin County	186659	190396	195126	
	Barbour County	27226	27159	26973	
	Bibb County	22733	22642	22512	
	Blount County	57711	57776	57734	

		POPESTIMATE2014	POPESTIMATE2015
STNAME	CTYNAME		
Alabama	Autauga County	55290	55347
	Baldwin County	199713	203709
	Barbour County	26815	26489
	Bibb County	22549	22583
	Blount County	57658	57673

```
[10]: # An immediate question which comes up is how we can query this DataFrame. 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.

      # 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
      ↪ Washtenaw County
      df.loc['Michigan', 'Washtenaw County']
```

```
[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
```

```
[11]: # If you are interested in comparing two counties, for example, Washtenaw and
      ↪ Wayne 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
      ↪ the 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')] ]
```

```

[11]:
      BIRTHS2010  BIRTHS2011  BIRTHS2012  BIRTHS2013  \
STNAME  CTYNAME
Michigan Washtenaw County      977      3826      3780      3662
      Wayne County      5918      23819      23270      23377

      BIRTHS2014  BIRTHS2015  POPESTIMATE2010  \
STNAME  CTYNAME
Michigan Washtenaw County      3683      3709      345563
      Wayne County      23607      23586      1815199

      POPESTIMATE2011  POPESTIMATE2012  POPESTIMATE2013  \
STNAME  CTYNAME
Michigan Washtenaw County      349048      351213      354289
      Wayne County      1801273      1792514      1775713

      POPESTIMATE2014  POPESTIMATE2015
STNAME  CTYNAME
Michigan Washtenaw County      357029      358880
      Wayne County      1766008      1759335

```

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 indices 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.