Coursera   
Week 2 :   
  
  
  
**Indexing DataFrame**

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.

1:19

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.

1:42

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.

1:54

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.

2:10

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.

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.

3:08

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.

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 variable.

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.

4:04

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.

5:01

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.

**# MISSING VALUES**

We're going to end this week of lecture with a quick discussion of missing values.

We've seen a preview of how Pandas handles missing values using the None type

and NumPy NaN values.

Missing values are pretty common in data cleaning activities. There are couple of caveats and discussion points which we should address.

0:25

First, the built in loading from delimited files provides control for

missing values in a few ways.

The most germane of these, is the na\_values list,

to indicate other strings which could refer to missing values.

Some of my sociologist colleagues for instance, regularly

use the value of 99 in binary categories to indicate that there's no value.

So this comes in handy.

You can also use the na\_filter option to turn off white space filtering,

if white space is an actual value of interest.

But in practice, this is pretty rare.

In addition to rules controlling how missing values might be loaded,

it's sometimes useful to consider missing values as actually having information.

I'll give an example from my own research.

I often deal with logs from online learning systems.

In particular,

I've done a couple of projects looking at video use in lecture capture systems.

In these systems it's common for the player for have a heartbeat functionality where

playback statistics are sent to the server every so often, maybe every 30 seconds.

1:26

These heartbeats can get big as they can carry the whole state of the playback

system, such as where the video play head is at, where the video size is,

which video is being rendered to the screen, how loud the volume is, etc.

1:38

If we load the data file log.txt,

we can see an example of what this might look like.

In this data the first column is a timestamp in the Unix epoch format.

The next column is the user name followed by a web page they're visiting and

the video that they're playing.

1:54

Each row of the DataFrame has a playback position.

And we can see that as the playback position increases by one,

the time stamp increases by about 30 seconds.

2:03

Except for user Bob.

It turns out that Bob has paused his playback so

as time increases the playback position doesn't change.

Note too how difficult it is for us to try and derive this knowledge from the data,

because it's not sorted by time stamp as one might expect.

This is actually not uncommon on systems which have a high degree of parallelism.

2:24

There are a lot of missing values in the paused and volume columns.

It's not efficient to send this information across the network if it

hasn't changed.

So this particular system just inserts null values into the database if

there's no changes.

2:36

One of the handy functions that Pandas has for

working with missing values is the filling function, fillna.

This function takes a number or parameters, for

instance, you could pass in a single value which is called a scalar value

to change all of the missing data to one value.

This isn't really applicable in this case, but it's a pretty common use case.

Next up though is the method parameter.

The two common fill values are ffill and bfill.

ffill is for forward filling and it updates an na value for

a particular cell with the value from the previous row.

It's important to note that your data needs to be sorted in order for

this to have the effect you might want.

Data that comes from traditional database management systems usually has no

order guarantee, just like this data.

So be careful.

3:21

In Pandas we can sort either by index or by values.

Here we'll just promote the time stamp to an index then sort on the index.

3:30

If we look closely at the output though we'll notice that the index

isn't really unique.

Two users seem to be able to use the system at the same time.

Again, a very common case.

3:40

Let's reset the index, and use some multi-level indexing instead, and

promote the user name to a second level of the index to deal with that issue.

3:49

Now that we have the data indexed and

sorted appropriately, we can fill the missing datas using ffill.

It's good to remember when dealing with missing values so

you can deal with individual columns or

sets of columns by projecting them just as we spoke about earlier.

So you don't have to fix all missing values in one command.

4:06

It's sometimes useful to use forward filling, sometimes backwards filling, and

sometimes useful to just use a single number.

More recently, the Pandas team introduced a method of filling missing values with

a series which is the same length as your DataFrame.

This makes it easy to derive values which are missing if you have

the underlying to do so.

For instance, if you're dealing with receipts and you have a column for

final price and a column for discount but are missing information from the original

price column, you can fill this automatically using fillna.

4:35

One last note on missing values.

When you use statistical functions on DataFrames,

these functions typically ignore missing values.

For instance if you try and calculate the mean value of a DataFrame,

the underlying NumPy function will ignore missing values.

This is usually what you want but

you should be aware that values are being excluded.