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.

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 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.

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.

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.

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.

```
df_log = pd.read_csv('log.csv')
df_log
```

	time	user	video	playback position	paused	volume
0	1469974424	cheryl	intro.html	5	False	10.0
1	1469974454	cheryl	intro.html	6	NaN	NaN
2	1469974544	cheryl	intro.html	9	NaN	NaN
3	1469974574	cheryl	intro.html	10	NaN	NaN
4	1469977514	bob	intro.html	1	NaN	NaN
5	1469977544	bob	intro.html	1	NaN	NaN
6	1469977574	bob	intro.html	1	NaN	NaN
7	1469977604	bob	intro.html	1	NaN	NaN
8	1469974604	cheryl	intro.html	11	NaN	NaN
9	1469974694	cheryl	intro.html	14	NaN	NaN
10	1469974724	cheryl	intro.html	15	NaN	NaN
11	1469974454	sue	advanced.html	24	NaN	NaN
12	1469974524	sue	advanced.html	25	NaN	NaN
13	1469974424	sue	advanced.html	23	False	10.0
14	1469974554	sue	advanced.html	26	NaN	NaN

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.

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 are no changes.

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.

```
df_log.fillna?
```

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.

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.

```
df_log = df_log.set_index('time')
df_log = df_log.sort_index()
df_log
```

time	user	video	playback position	paused	volume
1469974424	cheryl	intro.html	5	False	10.0
1469974424	sue	advanced.html	23	False	10.0
1469974454	cheryl	intro.html	6	NaN	NaN
1469974454	sue	advanced.html	24	NaN	NaN
1469974484	cheryl	intro.html	7	NaN	NaN
1469974514	cheryl	intro.html	8	NaN	NaN
1469974524	sue	advanced.html	25	NaN	NaN
1469974544	cheryl	intro.html	9	NaN	NaN
1469974554	sue	advanced.html	26	NaN	NaN
1469974574	cheryl	intro.html	10	NaN	NaN
1469974604	cheryl	intro.html	11	NaN	NaN
1469974624	sue	advanced.html	27	NaN	NaN
1469974634	cheryl	intro.html	12	NaN	NaN
1469974654	sue	advanced.html	28	NaN	5.0
1469974664	cheryl	intro.html	13	NaN	NaN

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.

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.

```
df_log = df_log.reset_index()
df_log = df_log.set_index(['time', 'user'])
df_log
```

time	user	index	video	playback position	paused	volume
1469974424	cheryl	0	intro.html	5	False	10.0
1469974454	cheryl	1	intro.html	6	NaN	NaN
1469974544	cheryl	2	intro.html	9	NaN	NaN
1469974574	cheryl	3	intro.html	10	NaN	NaN
1469977514	bob	4	intro.html	1	NaN	NaN
1469977544	bob	5	intro.html	1	NaN	NaN
1469977574	bob	6	intro.html	1	NaN	NaN
1469977604	bob	7	intro.html	1	NaN	NaN
1469974604	cheryl	8	intro.html	11	NaN	NaN
1469974694	cheryl	9	intro.html	14	NaN	NaN
1469974724	cheryl	10	intro.html	15	NaN	NaN
1469974454	sue	11	advanced.html	24	NaN	NaN
1469974524	sue	12	advanced.html	25	NaN	NaN
1469974424	sue	13	advanced.html	23	False	10.0

Now that we have the data indexed and sorted appropriately, we can fill the missing data 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.

```
df_log = df_log.fillna(method='ffill')
df_log.head()
```

time	user	video	playback position	paused	volume
1469974424	cheryl	intro.html	5	False	10.0
	sue	advanced.html	23	False	10.0
1469974454	cheryl	intro.html	6	False	10.0
	sue	advanced.html	24	False	10.0
1469974484	cheryl	intro.html	7	False	10.0

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*.

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.