Example: Manipulating DataFrames

In this lecture I'm going to walk through a basic data cleaning process with you and introduce you to a few more pandas API functions.

In [8]:

```
# Let's start by bringing in pandas
import pandas as pd
# And load our dataset. We're going to be cleaning the list of presidents in the
US from wikipedia
df=pd.read_csv("datasets/presidents.csv")
# And lets just take a look at some of the data
df.head()
```

Out[8]:

	#	President	Born	Age atstart of presidency	Age atend of presidency	Post-presidencytimespan	Died	Age
0	1	George Washington	Feb 22, 1732[a]	57 years, 67 daysApr 30, 1789	65 years, 10 daysMar 4, 1797	2 years, 285 days	Dec 14, 1799	67 years, 295 days
1	2	John Adams	Oct 30, 1735[a]	61 years, 125 daysMar 4, 1797	65 years, 125 daysMar 4, 1801	25 years, 122 days	Jul 4, 1826	90 years, 247 days
2	3	Thomas Jefferson	Apr 13, 1743[a]	57 years, 325 daysMar 4, 1801	65 years, 325 daysMar 4, 1809	17 years, 122 days	Jul 4, 1826	83 years, 82 days
3	4	James Madison	Mar 16, 1751[a]	57 years, 353 daysMar 4, 1809	65 years, 353 daysMar 4, 1817	19 years, 116 days	Jun 28, 1836	85 years, 104 days
4	5	James Monroe	Apr 28, 1758	58 years, 310 daysMar 4, 1817	66 years, 310 daysMar 4, 1825	6 years, 122 days	Jul 4, 1831	73 years, 67 days

In [9]:

```
# Ok, we have some presidents, some dates, I see a bunch of footnotes in the "Bo
rn" column which might cause
# issues. Let's start with cleaning up that name into firstname and lastname.
 I'm going to tackle this with
# a regex. So I want to create two new columns and apply a regex to the projecti
on of the "President" column.

# Here's one solution, we could make a copy of the President column
df["First"]=df['President']
# Then we can call replace() and just have a pattern that matches the last name
 and set it to an empty string
df["First"]=df["First"].replace("[ ].*", "", regex=True)
# Now let's take a look
df.head()
```

Out[9]:

	#	President	Born	Age atstart of presidency	Age atend of presidency	Post-presidencytimespan	Died	Age	F
0	1	George Washington	Feb 22, 1732[a]	57 years, 67 daysApr 30, 1789	65 years, 10 daysMar 4, 1797	2 years, 285 days	Dec 14, 1799	67 years, 295 days	Ge
1	2	John Adams	Oct 30, 1735[a]	61 years, 125 daysMar 4, 1797	65 years, 125 daysMar 4, 1801	25 years, 122 days	Jul 4, 1826	90 years, 247 days	J
2	3	Thomas Jefferson	Apr 13, 1743[a]	57 years, 325 daysMar 4, 1801	65 years, 325 daysMar 4, 1809	17 years, 122 days	Jul 4, 1826	83 years, 82 days	Tho
3	4	James Madison	Mar 16, 1751[a]	57 years, 353 daysMar 4, 1809	65 years, 353 daysMar 4, 1817	19 years, 116 days	Jun 28, 1836	85 years, 104 days	Ja
4	5	James Monroe	Apr 28, 1758	58 years, 310 daysMar 4, 1817	66 years, 310 daysMar 4, 1825	6 years, 122 days	Jul 4, 1831	73 years, 67 days	Ja

In [11]:

```
# That works, but it's kind of gross. And it's slow, since we had to make a full
copy of a column then go
# through and update strings. There are a few other ways we can deal with this.
 Let me show you the most
# general one first, and that's called the apply() function. Let's drop the colu
mn we made first
del(df["First"])

# The apply() function on a dataframe will take some arbitrary function you have
written and apply it to
# either a Series (a single column) or DataFrame across all rows or columns. Let
s write a function which
# just splits a string into two pieces using a single row of data
def splitname(row):
    # The row is a single Series object which is a single row indexed by column
 values
    # Let's extract the firstname and create a new entry in the series
    row['First']=row['President'].split(" ")[0]
    # Let's do the same with the last word in the string
    row['Last']=row['President'].split(" ")[-1] # don't have to worry about midd
le names and all
    # Now we just return the row and the pandas .apply() will take of merging th
em back into a DataFrame
    return row

# Now if we apply this to the dataframe indicating we want to apply it across co
lumns
df=df.apply(splitname, axis='columns')
df.head()
```

Out[11]:

	#	President	Born	Age atstart of presidency	Age atend of presidency	Post-presidencytimespan	Died	Age	
0	1	George Washington	Feb 22, 1732[a]	57 years, 67 daysApr 30, 1789	65 years, 10 daysMar 4, 1797	2 years, 285 days	Dec 14, 1799	67 years, 295 days	Was
1	2	John Adams	Oct 30, 1735[a]	61 years, 125 daysMar 4, 1797	65 years, 125 daysMar 4, 1801	25 years, 122 days	Jul 4, 1826	90 years, 247 days	
2	3	Thomas Jefferson	Apr 13, 1743[a]	57 years, 325 daysMar 4, 1801	65 years, 325 daysMar 4, 1809	17 years, 122 days	Jul 4, 1826	83 years, 82 days	J
3	4	James Madison	Mar 16, 1751[a]	57 years, 353 daysMar 4, 1809	65 years, 353 daysMar 4, 1817	19 years, 116 days	Jun 28, 1836	85 years, 104 days	M
4	5	James Monroe	Apr 28, 1758	58 years, 310 daysMar 4, 1817	66 years, 310 daysMar 4, 1825	6 years, 122 days	Jul 4, 1831	73 years, 67 days	

In [12]:

```
# Pretty questionable as to whether that is less gross, but it achieves the resu
lt and I find that I use the
# apply() function regularly in my work. The pandas series has a couple of other
nice convenience functions
# though, and the next I would like to touch on is called .extract(). Lets drop
 our firstname and lastname.
del(df['First'])
del(df['Last'])

# Extract takes a regular expression as input and specifically requires you to s
et capture groups that
# correspond to the output columns you are interested in. And, this is a great p
lace for you to pause the
# video and reflect - if you were going to write a regular expression that retur
ned groups and just had the
# firstname and lastname in it, what would that look like?

# Here's my solution, where we match three groups but only return two, the first
and the last name
pattern="(^\w+)(?:.* )(\w*$)"

# Now the extract function is built into the str attribute of the Series object,
so we can call it
# using Series.str.extract(pattern)
df["President"].str.extract(pattern).head()
```

Out[12]:

	0	1
0	George	Washington
1	John	Adams
2	Thomas	Jefferson
3	James	Madison
4	James	Monroe

In [13]:

```
# So that looks pretty nice, other than the column names. But if we name the gro
ups we get named columns out
pattern="(?P<First>^[\w]*)(?:.* )(?P<Last>[\w]*$)"

# Now call extract
names=df["President"].str.extract(pattern).head()
names
```

Out[13]:

	First	Last
0	George	Washington
1	John	Adams
2	Thomas	Jefferson
3	James	Madison
4	James	Monroe

In [14]:

```
# And we can just copy these into our main dataframe if we want to
df["First"]=names["First"]
df["Last"]=names["Last"]
df.head()
```

Out[14]:

	#	President	Born	Age atstart of presidency	Age atend of presidency	Post-presidencytimespan	Died	Age	I
0	1	George Washington	Feb 22, 1732[a]	57 years, 67 daysApr 30, 1789	65 years, 10 daysMar 4, 1797	2 years, 285 days	Dec 14, 1799	67 years, 295 days	Ge
1	2	John Adams	Oct 30, 1735[a]	61 years, 125 daysMar 4, 1797	65 years, 125 daysMar 4, 1801	25 years, 122 days	Jul 4, 1826	90 years, 247 days	J
2	3	Thomas Jefferson	Apr 13, 1743[a]	57 years, 325 daysMar 4, 1801	65 years, 325 daysMar 4, 1809	17 years, 122 days	Jul 4, 1826	83 years, 82 days	Tho
3	4	James Madison	Mar 16, 1751[a]	57 years, 353 daysMar 4, 1809	65 years, 353 daysMar 4, 1817	19 years, 116 days	Jun 28, 1836	85 years, 104 days	Ja
4	5	James Monroe	Apr 28, 1758	58 years, 310 daysMar 4, 1817	66 years, 310 daysMar 4, 1825	6 years, 122 days	Jul 4, 1831	73 years, 67 days	Ja

In [ ]:

```
# It's worth looking at the pandas str module for other functions which have bee
n written specifically
# to clean up strings in DataFrames, and you can find that in the docs in the Wo
rking with Text
# section: https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html
```

In [15]:

```
# Now lets move on to clean up that Born column. First, let's get rid of anythin
g that isn't in the
# pattern of Month Day and Year.
df["Born"]=df["Born"].str.extract("([\w]{3} [\w]{1,2}, [\w]{4})")
df["Born"].head()
```

Out[15]:

```
0    Feb 22, 1732
1    Oct 30, 1735
2    Apr 13, 1743
3    Mar 16, 1751
4    Apr 28, 1758
Name: Born, dtype: object
```

In [16]:

```
# So, that cleans up the date format. But I'm going to foreshadow something else
here - the type of this
# column is object, and we know that's what pandas uses when it is dealing with
 string. But pandas actually
# has really interesting date/time features - in fact, that's one of the reasons
Wes McKinney put his efforts
# into the library, to deal with financial transactions. So if I were building t
his out, I would actually
# update this column to the write data type as well
df["Born"]=pd.to_datetime(df["Born"])
df["Born"].head()
```

Out[16]:

```
0   1732-02-22
1   1735-10-30
2   1743-04-13
3   1751-03-16
4   1758-04-28
Name: Born, dtype: datetime64[ns]
```

In [ ]:

```
# This would make subsequent processing on the dataframe around dates, such as g
etting every President who
# was born in a given time span, much easier.
```

Summary

Now, most of the other columns in this dataset I would clean in a similar fashion. And this would be a good practice activity for you, so I would recommend that you pause the video, open up the notebook for the lecture if you don't already have it opened, and then finish cleaning up this dataframe. In this lecture I introduced you to the str module which has a number of important functions for cleaning pandas dataframes. You don't have to use these - I actually use apply() quite a bit myself, especially if I don't need high performance data cleaning because my dataset is small. But the str functions are incredibly useful and build on your existing knowledge of regular expressions, and because they are vectorized they are efficient to use as well.