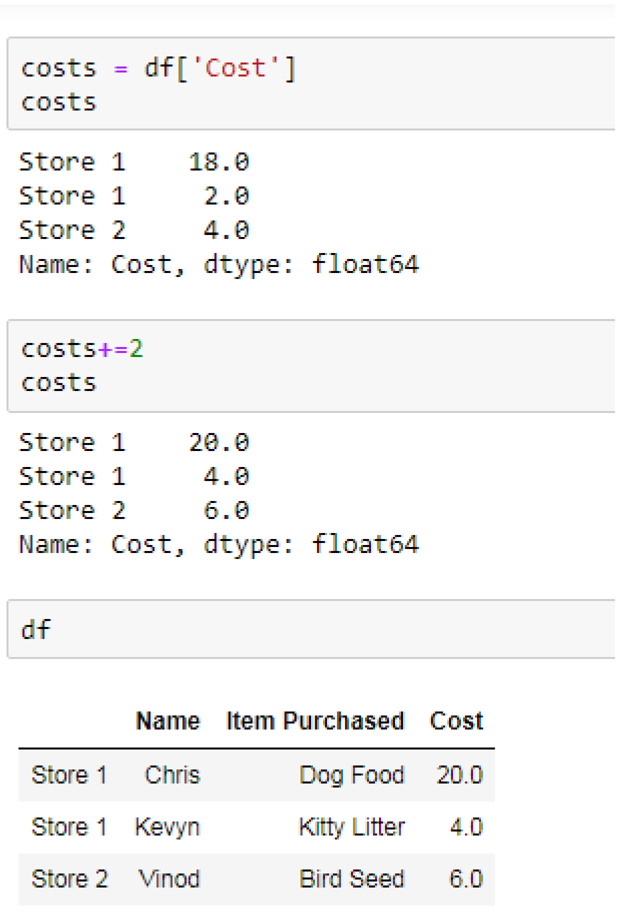
DataFrame Indexing and Loading

The common work flow is to read your data into a DataFrame then reduce this DataFrame to the particular columns or rows that you're interested in working with.
As you've seen, the Panda's toolkit tries to give you views on a DataFrame.
This is much faster than copying data and much more memory efficient too.

But it does mean that if you're manipulating the data you have to be aware that any changes to the DataFrame you're working on may have an impact on the base data frame you used originally.

Here's an example using our same purchasing DataFrame from earlier.
We can create a series based on just the cost category using the square brackets.
Then we can increase the cost in this series using broadcasting.
Now if we look at our original DataFrame, we see those costs have risen as well.
This is an important consideration to watch out for.
If you want to explicitly use a copy, then you should consider calling the copy method on the DataFrame for it first.



```
costs = df['Cost']
costs
```

```
Store 1    18.0
Store 1     2.0
Store 2     4.0
Name: Cost, dtype: float64
```

```
costs+=2
costs
```

```
Store 1    20.0
Store 1     4.0
Store 2     6.0
Name: Cost, dtype: float64
```

```
df
```

	Name	Item Purchased	Cost
Store 1	Chris	Dog Food	20.0
Store 1	Kevyn	Kitty Litter	4.0
Store 2	Vinod	Bird Seed	6.0

In this course, we'll be largely using smaller, moderate-sized datasets.
As I mentioned, a common workflow is to read the dataset in, usually from some external file.
We saw previously how you can do this using Python, and lists, and dictionaries.
You can imagine how you might use those dictionaries to create a Pandas DataFrame.

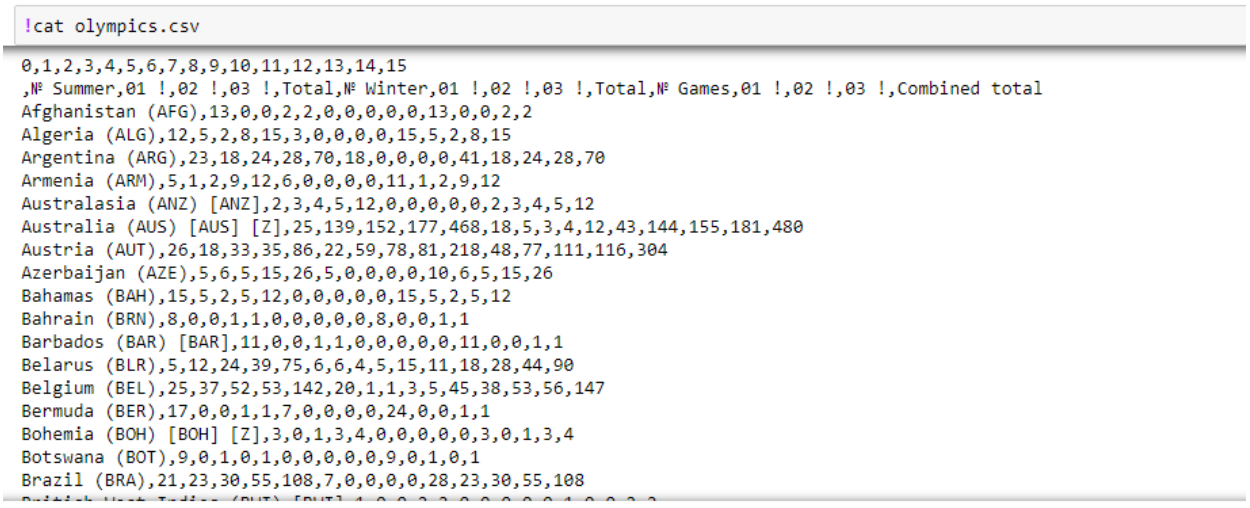
Thankfully, Pandas has built-in support for delimited files such as CSV files as well as a variety of other data formats including relational databases, Excel, and HTML tables.

I've saved a CSV file called olympics.csv, which has data from Wikipedia that contains a summary list of the medal various countries have won at the Olympics.

We can take a look at this file using the shell command *cat*. Which we can invoke directly using the exclamation point.

What happens here is that when the Jupyter notebook sees a line beginning with an exclamation mark, it sends the rest of the line to the operating system shell for evaluation.

So cat works on Linux and Macs, but might not work on Windows.

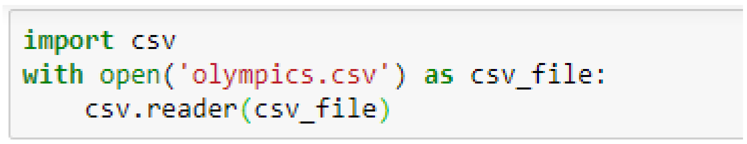


```
!cat olympics.csv
```

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
,№ Summer,01 !,02 !,03 !,Total,№ Winter,01 !,02 !,03 !,Total,№ Games,01 !,02 !,03 !,Combined total
Afghanistan (AFG),13,0,0,2,2,0,0,0,0,0,13,0,0,2,2
Algeria (ALG),12,5,2,8,15,3,0,0,0,0,15,5,2,8,15
Argentina (ARG),23,18,24,28,70,18,0,0,0,0,41,18,24,28,70
Armenia (ARM),5,1,2,9,12,6,0,0,0,0,11,1,2,9,12
Australasia (ANZ) [ANZ],2,3,4,5,12,0,0,0,0,0,2,3,4,5,12
Australia (AUS) [AUS] [Z],25,139,152,177,468,18,5,3,4,12,43,144,155,181,480
Austria (AUT),26,18,33,35,86,22,59,78,81,218,48,77,111,116,304
Azerbaijan (AZE),5,6,5,15,26,5,0,0,0,0,10,6,5,15,26
Bahamas (BAH),15,5,2,5,12,0,0,0,0,0,15,5,2,5,12
Bahrain (BRN),8,0,0,1,1,0,0,0,0,0,8,0,0,1,1
Barbados (BAR) [BAR],11,0,0,1,1,0,0,0,0,0,11,0,0,1,1
Belarus (BLR),5,12,24,39,75,6,6,4,5,15,11,18,28,44,90
Belgium (BEL),25,37,52,53,142,20,1,1,3,5,45,38,53,56,147
Bermuda (BER),17,0,0,1,1,7,0,0,0,0,24,0,0,1,1
Bohemia (BOH) [BOH] [Z],3,0,1,3,4,0,0,0,0,0,3,0,1,3,4
Botswana (BOT),9,0,1,0,1,0,0,0,0,0,9,0,1,0,1
Brazil (BRA),21,23,30,55,108,7,0,0,0,0,28,23,30,55,108
```

You don't need to worry too much about this.
I just wanted to show how a Jupyter notebook can integrate with the operating system to provide you with even more tools to look at your data.

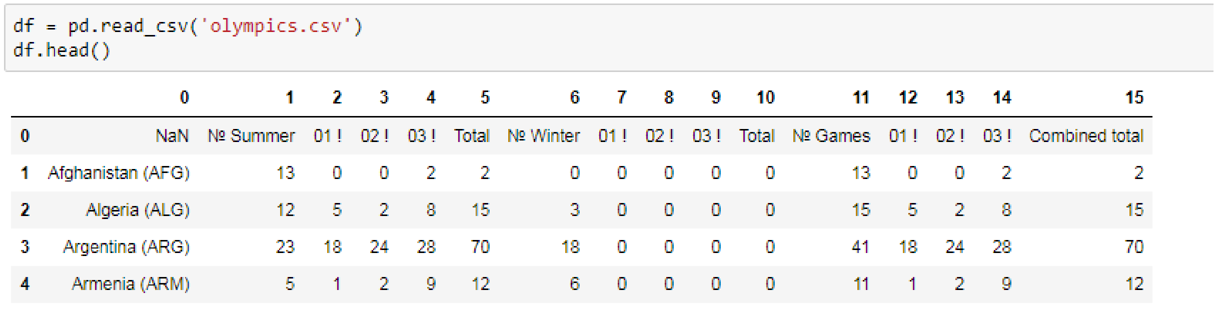
We see from the cat output that there seems to be a numeric list of columns followed by a bunch of column identifiers.
Here is the same output from windows:



```python
import csv
with open('olympics.csv') as csv_file:
    csv.reader(csv_file)
```

The column identifiers have some odd looking characters in them.
This is the unicode numero sign №, which means number of. Then we have rows of data, all columns separated.

We can read this into a DataFrame by calling the read_csv function of the module.
When we look at the DataFrame we see that the first cell has a NaN in it since it's an empty value, and the rows have been automatically indexed for us.



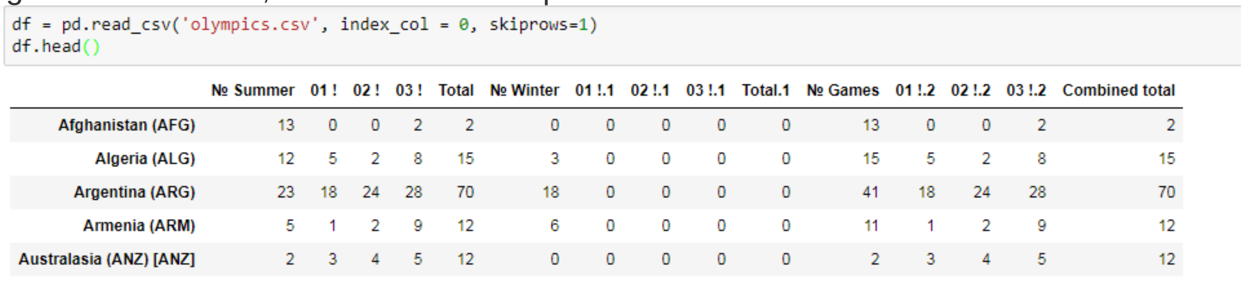
```python
df = pd.read_csv('olympics.csv')
df.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NaN	№ Summer	01 !	02 !	03 !	Total	№ Winter	01 !	02 !	03 !	Total	№ Games	01 !	02 !	03 !	Combined total
1	Afghanistan (AFG)	13	0	0	2	2	0	0	0	0	0	13	0	0	2	2
2	Algeria (ALG)	12	5	2	8	15	3	0	0	0	0	15	5	2	8	15
3	Argentina (ARG)	23	18	24	28	70	18	0	0	0	0	41	18	24	28	70
4	Armenia (ARM)	5	1	2	9	12	6	0	0	0	0	11	1	2	9	12

It seems pretty clear that the first row of data in the DataFrame is what we really want to see as the column names.
It also seems like the first column in the data is the country name, which we would like to make an index.

Read csv has a number of parameters that we can use to indicate to Pandas how rows and columns should be labeled.

For instance, we can use the index call to indicate which column should be the index and we can also use the header parameter to indicate which row from the data file should be used as the header.
Let's re-import that data and center index value to be 0 which is the first column and let set a column headers to be read from the second row of data. We can do this by using the skip rows parameters, to tell Pandas to ignore the first row, which was made up of numeric column names.



```python
df = pd.read_csv('olympics.csv', index_col = 0, skiprows=1)
df.head()
```

	№ Summer	01 !	02 !	03 !	Total	№ Winter	01 !.1	02 !.1	03 !.1	Total.1	№ Games	01 !.2	02 !.2	03 !.2	Combined total
Afghanistan (AFG)	13	0	0	2	2	0	0	0	0	0	13	0	0	2	2
Algeria (ALG)	12	5	2	8	15	3	0	0	0	0	15	5	2	8	15
Argentina (ARG)	23	18	24	28	70	18	0	0	0	0	41	18	24	28	70
Armenia (ARM)	5	1	2	9	12	6	0	0	0	0	11	1	2	9	12
Australasia (ANZ) [ANZ]	2	3	4	5	12	0	0	0	0	0	2	3	4	5	12

Now this data came from the all-time Olympic Games medal table on Wikipedia.
If we head to the page we could see that instead of running gold, silver and bronze in the pages, these nice little icons with a one, a two, and a three

in them In our csv file these were represented with the strings 01 !, 02 !, and so on.

We see that the column values are repeated which really isn't good practice. Panda's recognize this in a panda.1 and .2 to make things more unique.

```
df.columns
```

```
Index(['№ Summer', '01 !', '02 !', '03 !', 'Total', '№ Winter', '01 !.1',
       '02 !.1', '03 !.1', 'Total.1', '№ Games', '01 !.2', '02 !.2', '03 !.2',
       'Combined total'],
      dtype='object')
```

But this labeling isn't really as clear as it could be, so we should clean up the data file.
We can of course do this just by going and editing the CSV file directly, but we can also set the column names using the Pandas name property.

Panda stores a list of all of the columns in the .columns attribute.
We can change the values of the column names by iterating over this list and calling the rename method of the data frame.

```
for col in df.columns:
    if col[:2]=='01':
        df.rename(columns={col:'Gold' + col[4:]}, inplace=True)
    if col[:2]=='02':
        df.rename(columns={col:'Silver' + col[4:]}, inplace=True)
    if col[:2]=='03':
        df.rename(columns={col:'Bronze' + col[4:]}, inplace=True)
    if col[:1]=='№':
        df.rename(columns={col:'#' + col[1:]}, inplace=True)

df.head()
```

	# Summer	Gold	Silver	Bronze	Total	# Winter	Gold.1	Silver.1	Bronze.1	Total.1	# Games	Gold.2	Silver.2	Bronze.2	Combined total
Afghanistan (AFG)	13	0	0	2	2	0	0	0	0	0	13	0	0	2	2
Algeria (ALG)	12	5	2	8	15	3	0	0	0	0	15	5	2	8	15
Argentina (ARG)	23	18	24	28	70	18	0	0	0	0	41	18	24	28	70
Armenia (ARM)	5	1	2	9	12	6	0	0	0	0	11	1	2	9	12
Australasia (ANZ) [ANZ]	2	3	4	5	12	0	0	0	0	0	2	3	4	5	12

Here we just iterate through all of the columns looking to see if they start with a 01, 02, 03 or numeric character.
If they do, we can call rename and set the column parameters to a dictionary with the keys being the column we want to replace and the value being the new value we want.

Here we'll slice some of the old values in two, since we don't want to lose the unique appended values. We'll also set the ever-important in place parameter to true so Pandas knows to update this data frame directly.

All right, that's more on the data frame with a focus on the workflow of actually getting data in and into a place where we might want to query it.