**Data Science:** Data Science or Data Analytics is a process of analyzing large sets of data points to get answers on questions related to that data set.

**Python Pandas:**

* Pandas created in 2008 by Wes McKinney
* 100 different contributors

A panda is an open source. BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Pandas is a python module that makes data science easy and effective.

**Installation of Python Pandas:**

**For Windows:**

* **pip install pandas**
* **pip install –user numpy scipy matplotlib ipython pandas sympy nose**

**For Linux:**

* **sudo apt-get install python3-pandas**

Python with *pandas* is in use in a wide variety of **academic and commercial** domains, including Finance, Neuroscience, Economics, Statistics, Advertising, Web Analytics, and more.

>>>import pandas as pd

>>>df=pd.read\_csv (‘C:\Users\Personal\Desktop\data.csv’)

>>>df

O/P:

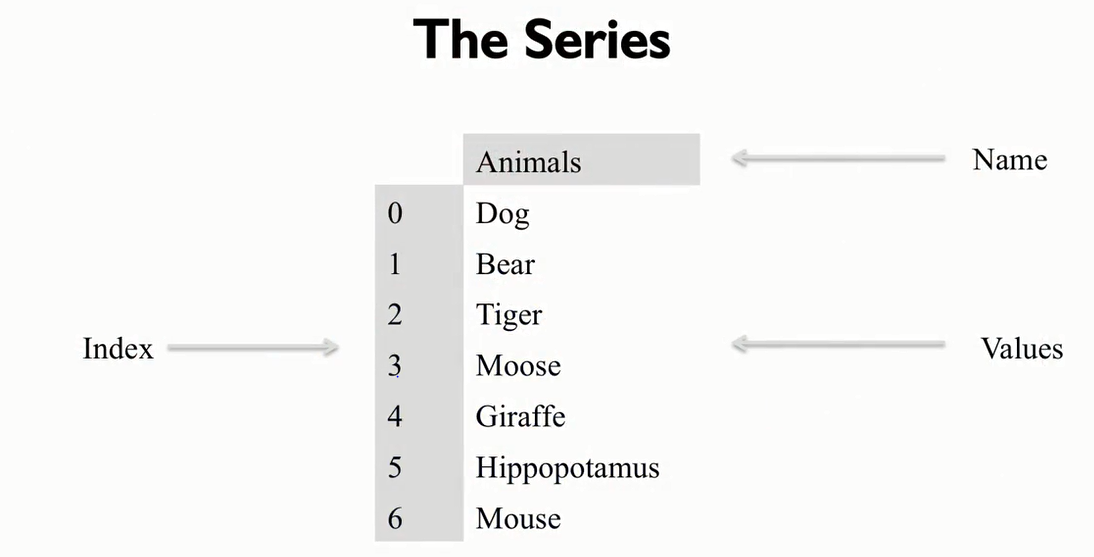
It prints whole data in csv file

|  |  |  |
| --- | --- | --- |
| **S.No** | **Name** | **Last Name** |
| 1 | Sai Sree | Ponnam |
| 2 | Siri | Potnuri |
| 3 | Mallika | Arekapudi |
| 4 | Geeta | Kandula |
| 5 | Krishna | Arekapudi |

* The process of cleaning messy data is called data munging or data wrangling

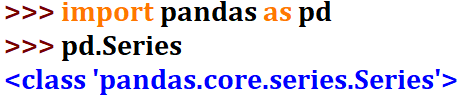
**Python Series:**

The series is one of the core data structures in pandas. it a cross between a list and a dictionary. The items are all stored in an order and there's labels with which you can retrieve them. An easy way to visualize this is two columns of data. The first is the special index, a lot like the dictionary keys. While the second is your actual data. It's important to note that the data column has a label of its own and can be retrieved using the .name attribute. This is different than with dictionaries and is useful when it comes to merging multiple columns of data.

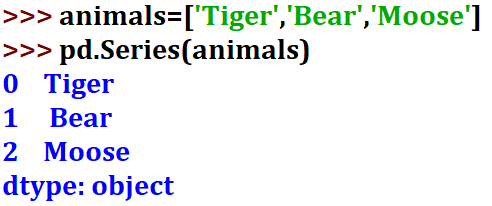


you can create a series by passing in a list of values. When you do this, Pandas automatically assigns an index starting with zero and sets the name of the series to None.

Let's see an example of this. First, I'll start off by importing the pandas library as pd, then let's take a look at the series object.

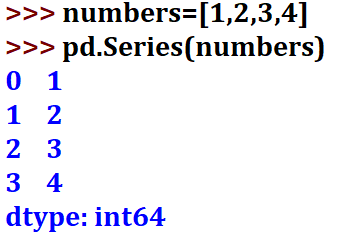


Here you could see the documentation indicates that you can pass in some data, an index and a name.

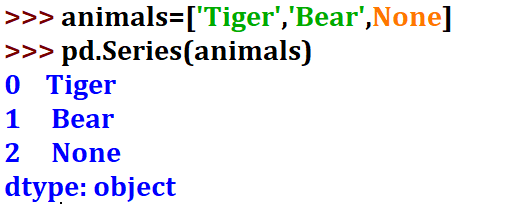


The data can be anything, that's array-like, like a list. So let's give that a try. We'll just make a list of the three of my favorite animals, a tiger, a bear and a moose.

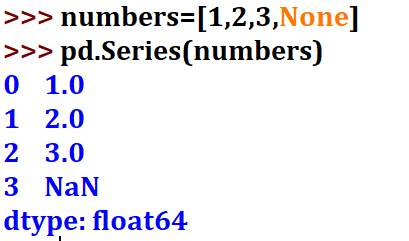
 We'll see here that the pandas has automatically identified the type of the data being held in the list, in this case we passed in a list of strings and panda set the type to object.



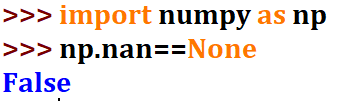
If we passed in a list of whole numbers, for instance, we could see that panda sets the type to n 64. Underneath panda stores series values in a typed array using the Numpy library. This offers significant speed-up when processing data versus traditional python lists.

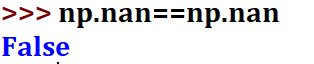


The most important is how Numpy and thus pandas handle missing data. In Python, we have the none type to indicate a lack of data. Underneath, pandas does some type conversion. If we create a list of strings and we have one element, a None type, pandas inserts it as a None and uses the type object for the underlying array.



If we create a list of numbers, integers or floats, and put in the None type, pandas automatically converts this to a special floating point value designated as NAN, which stands for not a number.

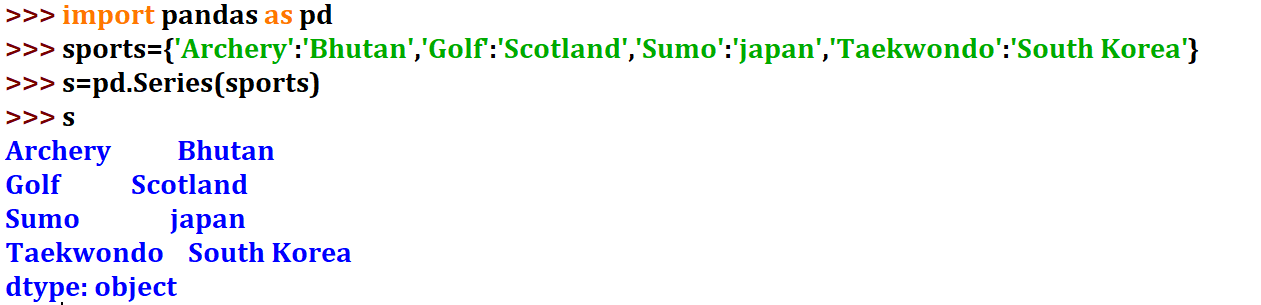




It turns out that you actually can't do an equality test of NAN to itself. When you do, the answer is always false. You need to use special functions to test for the presence of not a number, such as the Numpy library is NAN.



 When you see NAN, it's meaning is similar to none, but it's a numeric value and it's treated differently for efficiency reasons. Let's talk more about how pandas' series can be created. While my list of animals might be a common way to create some play data, often you have label data that you want to manipulate. A series can be created from dictionary data. If you do this, the index is automatically assigned to the keys of the dictionary that you provided and not just incrementing integers.



Here's an example using some data on official national sports.

When we create the series, we see that, since it was string data, panda set the data type of the series to object. We set the list of the countries as the value of the series and that the index values can be set to the keys from our dictionary.



We can get the index object using the index attribute. You could also separate your index creation from the data by passing in the index as a list explicitly to the series.



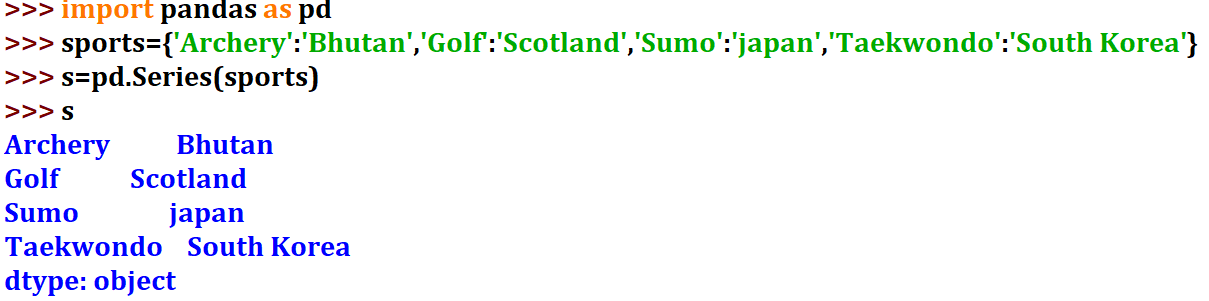
So what happens if your list of values in the index object are not aligned with the keys in your dictionary for creating the series? Well, pandas overrides the automatic creation to favor only and all of the indices values that you provided.

So it will ignore it from your dictionary, all keys, which are not in your index, and

pandas will add non type or NAN values for any index value you provide, which is not in your dictionary key list. In this example, we pass in a dictionary of four items but only two are preserved in the series object because of the index list. We see that hockey has been added but since it's also in the index list, it has no value associated with it. So that's how we create a series. In the next class, we'll dig deeper and look at how we get data out of the series.

**Querying a Series:**

A pandas Series can be queried, either by the index position or the index label. As we saw, if you don't give an index to the series, the position and the label are effectively the same values. To query by numeric location, starting at zero, use the iloc attribute. To query by the index label, you can use the loc attribute.



Here's an example using national sporting event data from Wikipedia. Let's say we want to have a list of all of the sports in our index and a list of countries as values.

If you wanted to see the fourth country on this, we would use the iloc attribute with the parameter 3.





If you wanted to see which country has golf as its national sport, we would use the loc attribute with parameter golf. Keep in mind that iloc and loc are not methods, they are attributes. So you don't use parentheses to query them, but square brackets instead, which we'll call the indexing operator. Though in Python,

this calls get and set an item methods depending on the context of its use. This might seem a bit confusing if you're used to languages where encapsulation of attributes, variables, and properties is common, such as in Java. Pandas tries to make our code a bit more readable and provides a sort of smart syntax using the indexing operator directly on the series itself.

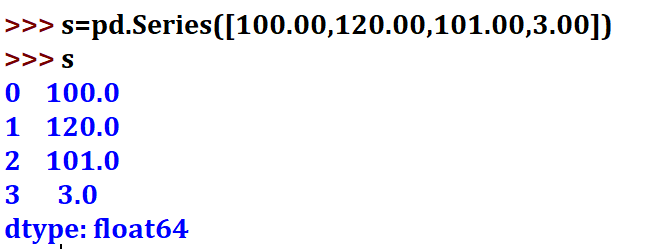
For instance, if you pass in an integer parameter, the operator will behave as if you want it to query via the iloc attribute.



If you pass in an object, it will query as if you wanted to use the label based loc attribute.

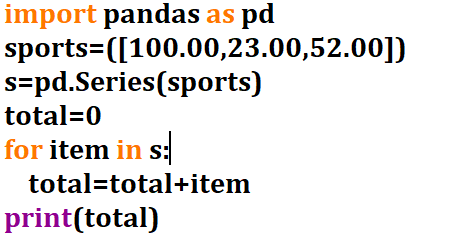


So what happens if your index is a list of integers? This is a bit complicated, and Pandas can't determine automatically whether you're intending to query by index position or index label. So you need to be careful when using the indexing operator on the series itself. And the safer option is to be more explicit and use the iloc or loc attributes directly.

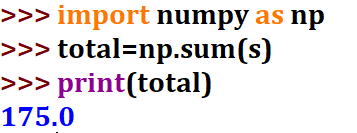


Here's an example using some new sports data, where countries are indexed by integer. If we try and calls subzero, we get a key error, because there's no item in the sports list with an index of zero. Instead we have to call iloc explicitly if we want the first item. Okay, so now we know how to get data out of the series. Let's talk about working with the data. A common task is to want to consider all of the values inside of a series and want to do some sort of operation. This could be trying to find a certain number, summarizing data or transforming the data in some way. A typical programmatic approach to this would be to iterate over all the items in the series, and invoke the operation one is interested in.

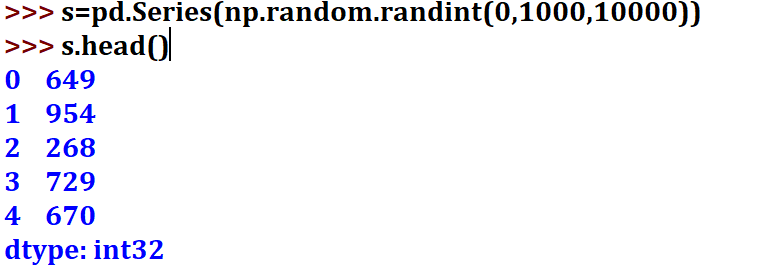
For instance, we could create a data frame of floating point values. Let's think of these as prices for different products. We could write a little routine which iterates over all of the items in the series and adds them together to get a total.



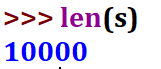
This works, but it's slow. Modern computers can do many tasks simultaneously, especially, but not only, tasks involving mathematics. Pandas and the underlying NumPy libraries support a method of computation called vectorization. Vectorization works with most of the functions in the NumPy library, including the sum function. Here's how we would really write the code using the NumPy sum method.



First we need to import the numpy module, and then we just call np.sum and pass in an iterable item. In this case, our panda series. Now both of these methods create the same value, but is one actually faster? The Jupyter Notebook has a magic function which can help. First, let's create a big series of random numbers.



You'll see this used a lot when demonstrating techniques with Pandas. Note that I've just used the head method, which reduces the amount of data printed out by the series to the first five elements. We can actually verify that length of the series is correct using the len function.



Magic functions begin with a percentage sign. If we type this sign and then hit the Tab key, We can see a list of the available magic functions. You could write your own magic functions too, but that's a little bit outside of the scope of this course. We're actually going to use what's called a cellular magic function. These start with two percentage signs and  modify a raptor code in the current Jupyter cell. The function we're going to use is called timeit.



And as you may have guessed from the name, this function will run our code a few times to determine, on average, how long it takes. Let's run timeit with our original iterative code. You can give timeit the number of loops that you would like to run. By default, we'll use 1,000 loops. I'll ask timeit here to use 100 runs because we're recording this. Timeit ran this code and it doesn't seem like it takes very long at all.

Now let's try with vectorization.



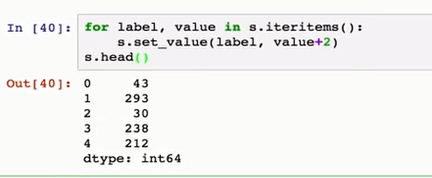
Wow! This is a pretty shocking difference in the speed and demonstrates why data scientists need to be aware of parallel computing features and start thinking in functional programming terms. Related feature in Pandas and NumPy is called broadcasting. With broadcasting, you can apply an operation to every value in the series, changing the series.

For instance, if we wanted to increase every random variable by 2, we could do so quickly using the += operator directly on the series object.



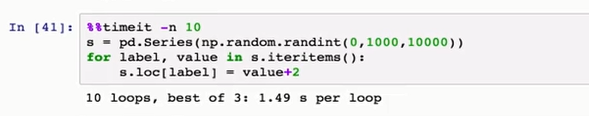
Here I'll just use the head operator to just print out the top five rows in the series.

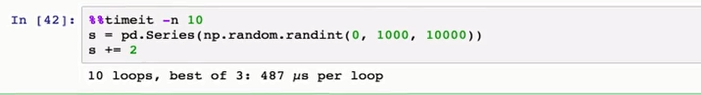
The procedural way of doing this would be to iterate through all of the items in the series and increase the values directly.



Pandas does support iterating through a series much like a dictionary, allowing you to unpack values easily. But if you find yourself iterating through a series, you should question whether you're doing things in the best possible way. Here's how we would do this using the series set value method.

Let's try and time the two approaches.





Amazing. Not only is it significantly faster, but it's more concise and maybe even easier to read too. The typical mathematical operations you would expect are vectorized, and the NumPy documentation outlines what it takes to create vectorized functions of your own. One last note on using the indexing operators to access series data. The .loc attribute lets you not only modify data in place, but

also add new data as well. If the value you pass in as the index doesn't exist, then a new entry is added. And keep in mind, indices can have mixed types. While it's important to be aware of the typing going on underneath, Pandas will automatically change the underlying NumPy types as appropriate.

We could add some new value, maybe an animal, as you know, I like bears.

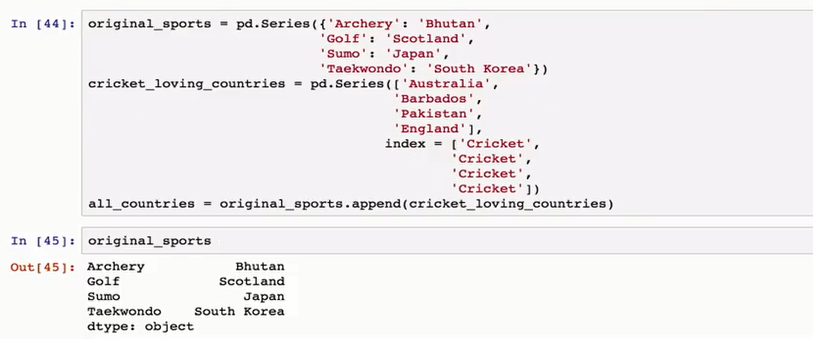
 Just by calling the .loc indexing operator.



We see that mixed types for data values or index labels are no problem for Pandas. Up until now I've shown only examples of a series where the index values were unique. Let’s an example where index values are not unique, and this makes data frames different, conceptually, that a relational database might be.

Revisiting the issue of countries and their national sports, it turns out that many countries seem to like this game cricket. We go back to our original series on sports. It's possible to create a new series object with multiple entries for cricket, and then use append to bring these together. There are a couple of important considerations when using append. First, Pandas is going to take your series and

try to infer the best data types to use. In this example, everything is a string, so there's no problems here.



 Second, the append method doesn't actually change the underlying series.

It instead returns a new series which is made up of the two appended together.

We can see this by going back and printing the original series of values and seeing that they haven't changed. This is actually a significant issue for new Pandas users who are used to objects being changed in place. So watch out for it, not just with append but with other Pandas functions as well. 

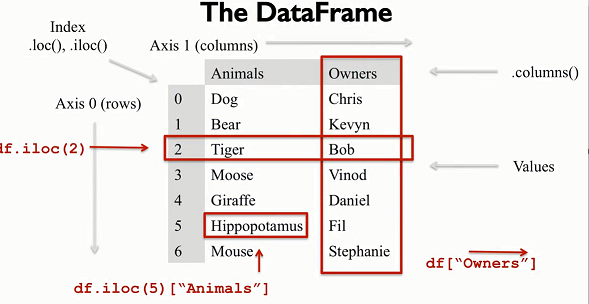
Finally, we see that when we query the appended series for those who have cricket as their national sport, we don't get a single value, but a series itself. This is actually very common, and if you have a relational database background, this is very similar to every table query resulting in a return set which itself is a table.

In this , we focused on one of the primary data types of the Pandas library, the series. There are many more methods associated with this series object that we haven't talked about. But with these basics down, we'll move on to talking about

the Panda's two-dimensional data structure, the data frame. The data frame is very similar to the series object, but includes multiple columns of data, and is the structure that you'll spend the majority of your time working with when cleaning and aggregating data.

**The Data Frame Data Structure:**

The Data Frame data structure is the heart of the Panda's library. It's a primary object that you'll be working with in data analysis and cleaning tasks. The Data Frame is conceptually a two-dimensional series object, where there's an index and multiple columns of content, with each column having a label.



In fact, the distinction between a column and a row is really only a conceptual distinction. And you can think of the Data Frame itself as simply a two-axes labeled array.

 You can create a Data Frame in many different ways, some of which you might expect. For instance, you can use a group of series, where each series represents a row of data. Or you could use a group of dictionaries, where each dictionary represents a row of data.

Let's take a look at an example.



I'm going to create three purchase order records as series objects for a sort of fictional store. Each series has a name of a customer, the string which describes the item being purchased, and the cost of the items. I like dogs, so I'll purchase a bunch of dog food. Kevin Cullens Thompson, the instructor for the third course in the series, he seems more like a cat man to me, so I'll have him purchase some kitty litter. And I think Vinod, who teaches the fourth course in this series, is more of a bird man, so I'll add some bird seed there. Then we'll feed this into the Data Frame as the first argument and set index values indicating which store where each purchase was done. You'll see that when we print out a Data Frame, the Jupiter notebook's trying to pretty it up a bit, and print it out as a table, which is nice.

Similar to the series, we can extract data using the iLock and Lock attributes. Because the Data Frame is two-dimensional, passing a single value to the lock

 indexing operator will return series if there's only one row to return.

 In this example, if we wanted to select data associated with Store 2,we would just query the lock attribute with one parameter. 

You'll note that the name of the series is returned as the row index value, while the column name is included in the output as well.

We can check the data type of the return using the python type function.



It's important to remember that the indices and column names along either axes, horizontal or vertical, could be non-unique. For instance, in this example, we see two purchase records for Store 1 as different rows. If we use a single value with the Data Frame lock attribute, multiple rows of the Data Frame will return, not as a new series, but as a new Data Frame.

For instance, if we query for Store 1 records,



We see that Chris and Kevin both shop at the same pets supply store. One of the powers of the Panda's Data Frame is that you can quickly select data based on multiple axes.

For instance, if you wanted to just list the costs for Store 1, you would supply

two parameters to .log, one being the row index and the other being the column name.

If we're only interested in Store 1 costs, we could write this as df.lock('Store 1',

'Cost').



 What if we just wanted to do column selection and just get a list of all of the costs? Well, there's a couple of options.

First, you can get a transpose of the DataFrame, using the capital T attribute, which swaps all of the columns and rows.



This essentially turns your column names into indices.

And we can then use the .lock method. This works, but it's pretty ugly.





Since iloc and loc are used for row selection, the Panda's developers reserved indexing operator directly on the DataFrame for column selection. In a Panda's DataFrame, columns always have a name. So this selection is always label based, not as confusing as it was when using the square bracket operator on the series objects. For those familiar with relational databases, this operator is analogous to column projection.

Finally, since the result of using the indexing operators, the DataFrame or series, you can chain operations together. For instance, we could have rewritten the query for all Store 1 costs as df.loc('Store 1', 'Cost').



This looks pretty reasonable and gets us the result we wanted.But chaining can come with some costs and is best avoided if you can use another approach.In particular, chaining tends to cause Pandas to return a copy of the DataFrame

instead of a view on the DataFrame.

For selecting a data, this is not a big deal, though it might be slower than necessary. If you are changing data though, this is an important distinction and

 can be a source of error.

Here's another method. As we saw, .loc does row selection, and it can take two parameters, the row index and the list of column names. .loc also supports slicing.

If we wanted to select all rows, we can use a column to indicate a full slice from beginning to end. And then add the column name as the second parameter as a string. In fact, if we wanted to include multiply columns, we could do so in a list.

And Pandas will bring back only the columns we have asked for.

Here's an example, where we ask for all of the name and cost values for all stores using the .loc operator. So that's selecting and projecting data from a DataFrame based on row and column labels. The key concepts to remember are that the rows and columns are really just for our benefit. Underneath this is just a two axes labeled array, and transposing the columns is easy.



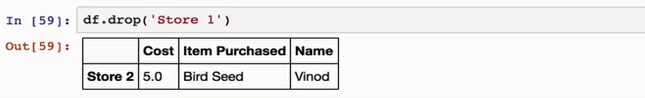
Also, consider the issue of chaining carefully, and try to avoid it, it can cause unpredictable results. Where your intent was to obtain a view of the data, but

instead Pandas returns to you a copy. In the Panda's world, friends don't let friends chain calls.

So if you see it, point it out, and share a less ambiguous solution.

let's talk about dropping data.

It's easy to delete data in series and DataFrames, and we can use the drop function to do so. This function takes a single parameter, which is the index or roll label, to drop. This is another tricky place for new users to pad this.



The drop function doesn't change the DataFrame by default. And instead, returns to you a copy of the DataFrame with the given rows removed. We can see that our original DataFrame is still intact.



Let's make a copy with the copy method and do a drop on it instead.



This is a very typical pattern in Pandas, where in place changes to a DataFrame are only done if need be, usually on changes involving indices. So it's important to be aware of.

Drop has two interesting optional parameters. The first is called in place, and if it's set to true, the DataFrame will be updated in place, instead of a copy being returned. The second parameter is the axes, which should be dropped.

By default, this value is 0, indicating the row axes.But you could change it to 1 if you want to drop a column. There is a second way to drop a column, however And that's directly through the use of the indexing operator, using the del keyword.



This way of dropping data, however, takes immediate effect on the DataFrame and does not return a view. Finally, adding a new column to the DataFrame is

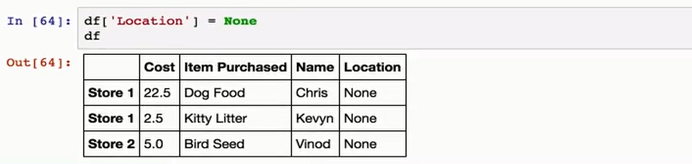
as easy as assigning it to some value.

For instance,

if we wanted to add a new location as a column with default value of none,

we could do so by using the assignment operator after the square brackets.

This broadcasts the default value to the new column immediately.



**Data Frame 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.

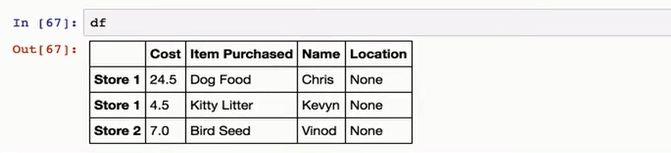
Here's an example using our same purchasing DataFrame from earlier.



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.

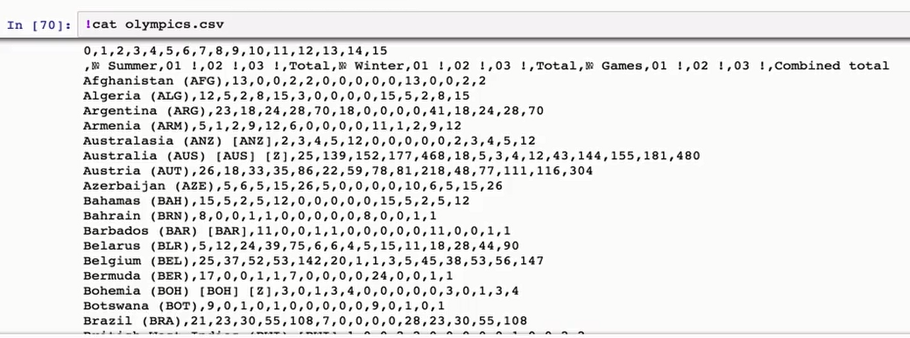
In this course, we'll be largely using smaller, moderate-sized datasets. As I mentioned, a common workflow is to read the dataset in, usually from some external file. We saw previously how you can do this using Python, and lists, and dictionaries. You can imagine how you might use those dictionaries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

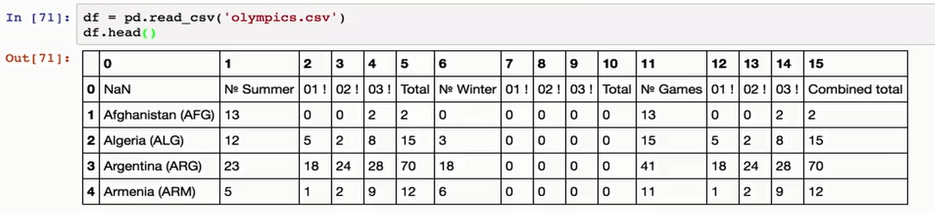
evaluation. So cat works on Linux and Macs, as well as in the Coursera platform, but might not work on Windows. I just wanted to show how a Jupyter notebook can integrate with the operating system to provide you with even more tools to look at your data.



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

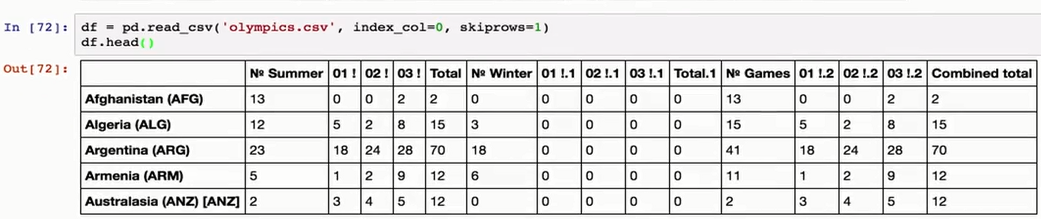
We can read this into a DataFrame by calling the read\_csv function of the module.

 When we look at the DataFrame we see that the first cell has an NaN in it since it's an empty value, and the rows have been automatically indexed for us. 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.



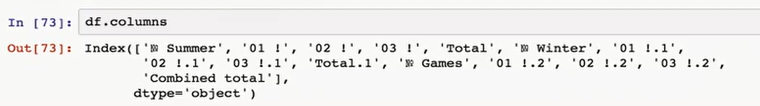
For instance, we can use the index call to indicate which column should be the index and we can also use the header parameter to indicate which row from the data file should be used as the header. Let's re-import that data and center index value to be 0 which is the first column and let set a column 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.



Now this data came from the all time Olympic games medal table. 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.

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.



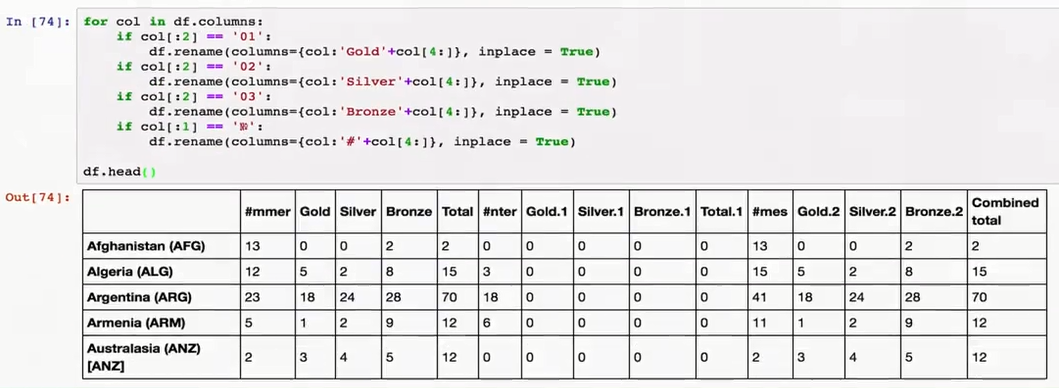
We can change the values of the column names by iterating over this list and

calling the rename method of the data frame.

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.

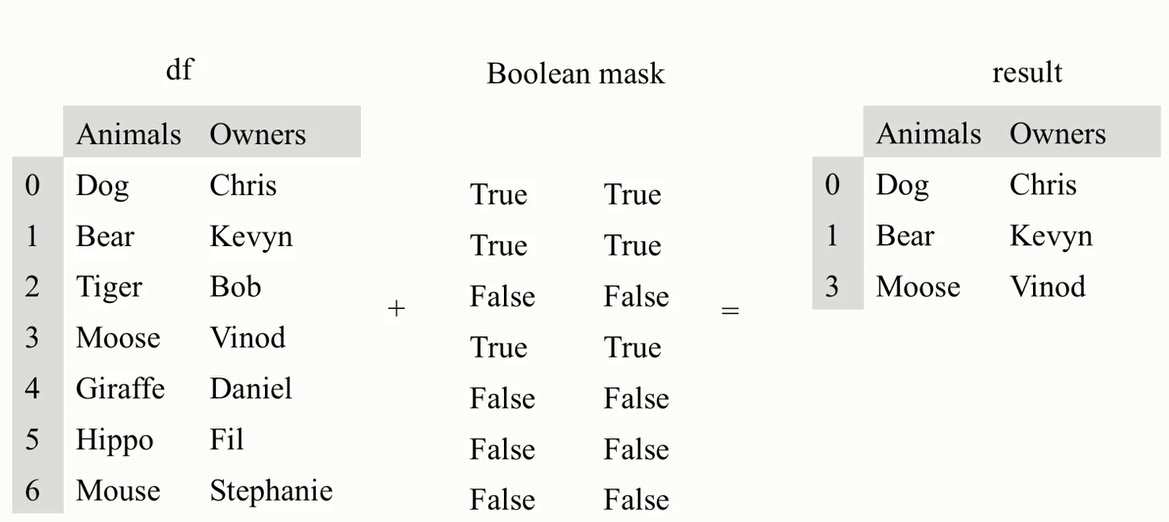
 We'll also set the ever-important in place parameter to true so Pandas knows to update this data frame directly.



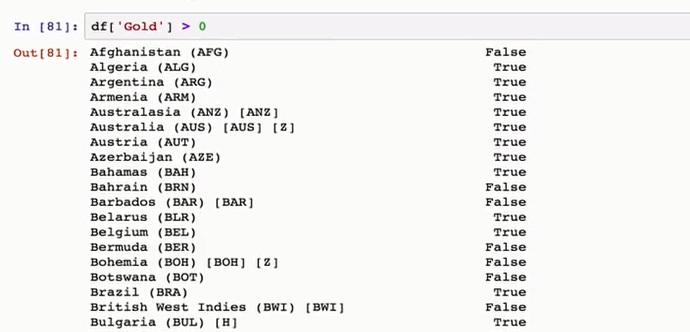
**Querying A Data Frame:**

Boolean masking is the heart of fast and efficient querying in NumPy. It's analogous a bit to masking used in other computational areas.

 A Boolean mask is an array which can be of one dimension like a series, or two dimensions like a data frame, where each of the values in the array are either true or false. This array is essentially overlaid on top of the data structure that we're querying. And any cell aligned with the true value will be admitted into our final result, and any sign aligned with a false value will not. Boolean masking is powerful conceptually and is the cornerstone of efficient NumPy and pandas querying. This technique is well used in other areas of computer science, for instance, in graphics. But it doesn't really have an analogue in other traditional relational databases, so I think it's worth pointing out here. Boolean masks are related by applying operators directly to the pandas series or DataFrame objects.



For instance, in our Olympics data set, you might be interested in seeing only those countries who have achieved a gold medal at the summer Olympics.To build a Boolean mask for this query, we project the gold column using the indexing operator and apply the greater than operator with a comparison value of zero. This is essentially broadcasting a comparison operator, greater than, with the results being returned as a Boolean series.The resultant series is indexed where the value of each cell is either true or false depending on whether a country has won at least one gold medal, and the index is the country name.



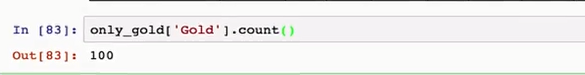
So this builds us the Boolean mask, which is half the battle. What we want to do next is overlay that mask on the data frame. We can do this using the where function. The where function takes a Boolean mask as a condition, applies it to the data frame or series, and returns a new data frame or series of the same shape. Let's apply this Boolean mask to our Olympics data and create a data frame of only those countries who have won a gold at a summer games.



We see that the resulting data frame keeps the original indexed values, and

only data from countries that met the condition are retained. All of the countries which did not meet the condition have NaN data instead. Most statistical functions built into the data frame object ignore values of NaN.

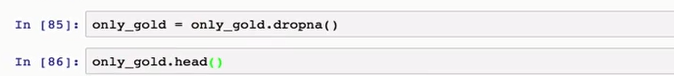
For instance, if we call the df.count on the only gold data frame, we see that there are 100 countries which have had gold medals awarded at the summer games,



while if we call count on the original data frame, we see that there are 147 countries total.



Often we want to drop those rows which have no data. To do this, we can use the dropna() function. You can optionally provide dropnathe axes it should be considering. Remember that the axes is just an indicator for the columns or rows and that the default is zero, which means rows.





When you find yourself talking about pandas and saying phrases like,

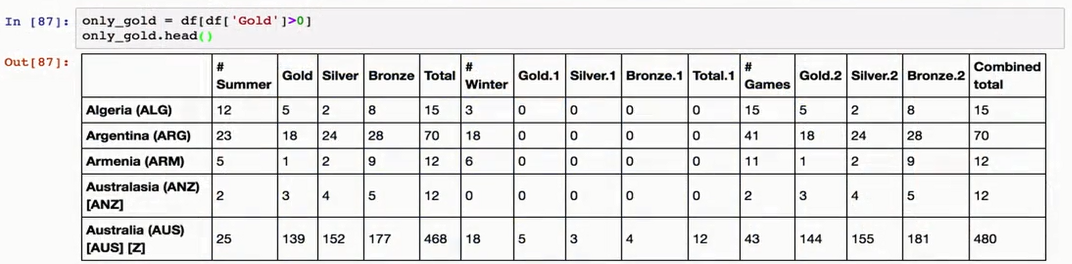
often I want to, it's quite likely the developers have included a shortcut for

this common operation. For instance, in the above example, we don't actually have to use the where function explicitly. The pandas developers allow the indexing operator to take a Boolean mask as a value instead of just a list of column names. The syntax might look a little messy, especially if you're not used

 to programming languages with overloaded operators, but the result is that you're able to filter and reduce data frames relatively quickly.

Here's a more concise example of how we could query this data frame.

You'll notice that there are no NaNs when you query the data frame in this manner. pandas automatically filters out the rows with now values.



One more thing to keep in mind if you're not used to Boolean or bit masking for

data reduction. The output of two Boolean masks being compared with logical operators is another Boolean mask. This means that you can chain together a bunch of and/or statements in order to create more complex queries and the result is a single Boolean mask.

For instance, we could create a mask for all of those countries who have received a gold in the summer Olympics and logically order that with all of those countries who have received a gold in the winter Olympics. If we apply this to the data frame and use the length function to see how many rows there are, we see that there are 101 countries which have won a gold medal at some time.



Another example,

Have there been any countries who have only won gold in the winter Olympics and never in the summer Olympics?

 Here's one way to answer that.



Poor Liechtenstein

I know who I'll be cheering for in 2020 to win their first summer gold

Extremely important, and often an issue for new users, is to remember that each Boolean mask needs to be encased in parenthesis because of the order of operations.

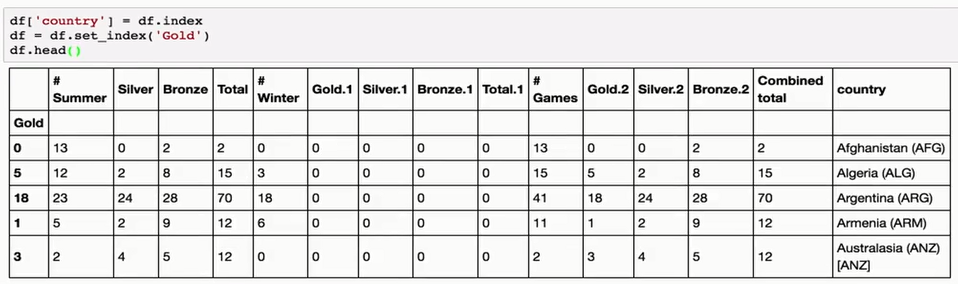
**Indexing Data Frames:**

As we've seen, both series and Data Frames 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.

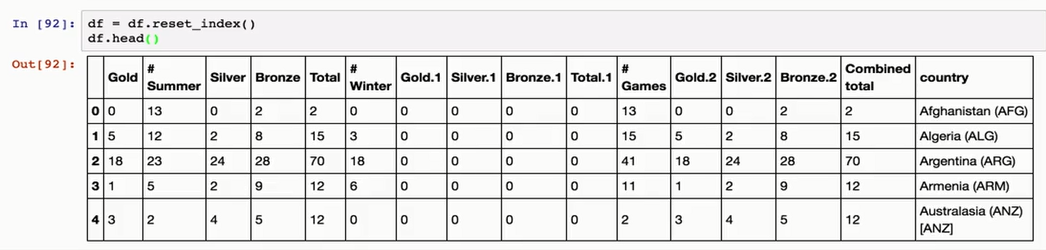
Let's go back to our Olympics Data Frame. Let's say that we don't want to index the Data Frame by countries, but instead want to index by the number of gold medals that were won at summer games. First we need to preserve the country information into a new column. We can do this using the indexing operator 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.

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.

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.



  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.



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.

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.

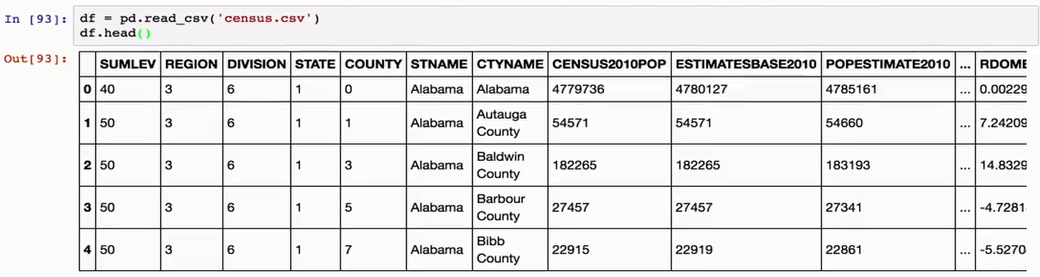
This data is stored in the file census.csv and comes from the United States Census Bureau.

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.



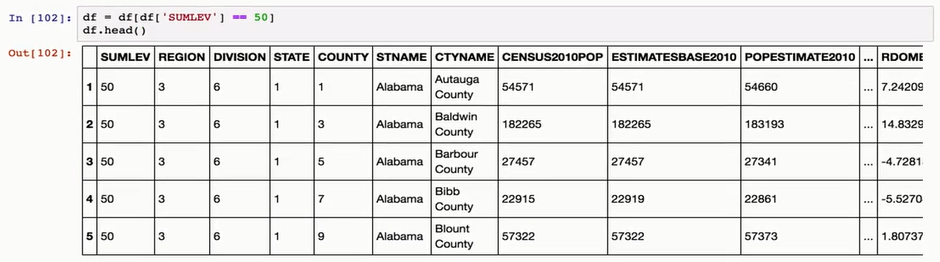
I often find that I want to see a list of all the unique values in a given column. In this Data Frame, we see that the possible values for the sum level are using the unique function on the Data Frame.

This is similar to the SQL distinct operator.

Here we can run unique on the sum level of our current Data Frame and see that there are only two different values, 40 and 50.



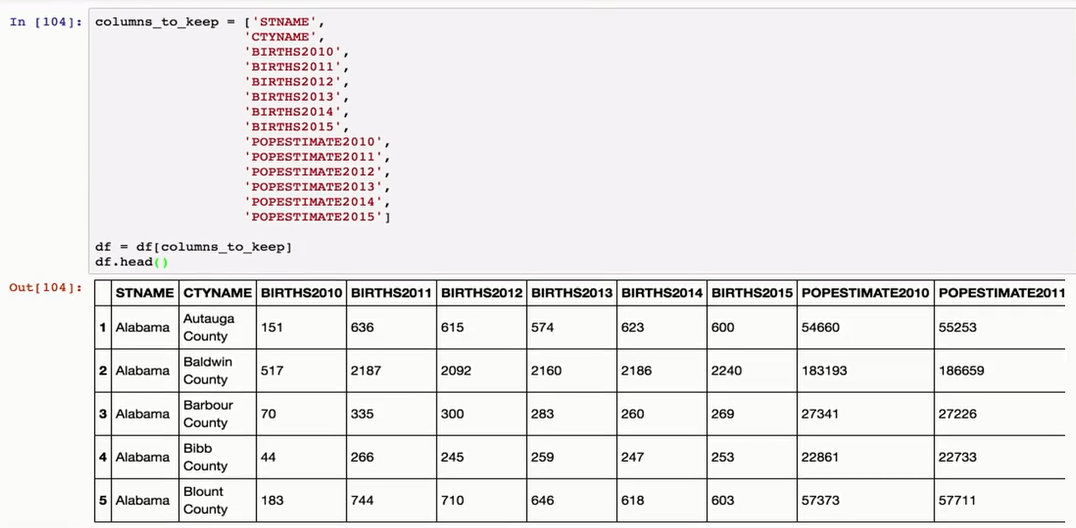
Let's get rid of all of the rows that are summaries at the state level and just keep the county data. Also while this data set is interesting for a number of different reasons, let's reduce the data that we're going to look at to just the total population estimates and the total number of births.



We can do this by creating a list of column names that we want to keep then project those and assign the resulting Data Frame 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 Data Frame.

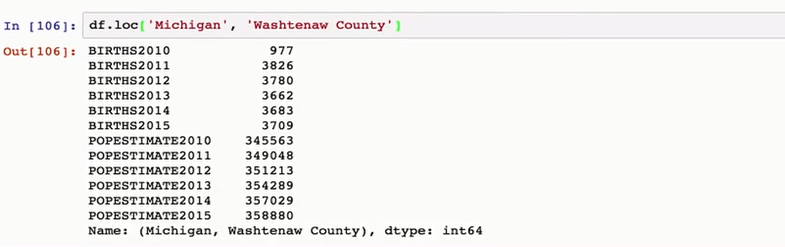


We do this by creating a list of the column identifiers we want to have indexed and then calling set index with this list and assigning the output as appropriate.  We see here that we have a dual index, first the state name and

 then the county name.



 An immediate question which comes up is how we can query this Data Frame. For instance, we saw previously that the loc attribute of the Data Frame can take multiple arguments. And it could query both the row and the columns. When you use a Multi Index, 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 Multi Index of two values, the state and the county, we need to provide two values as each element of our filtering list.

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.

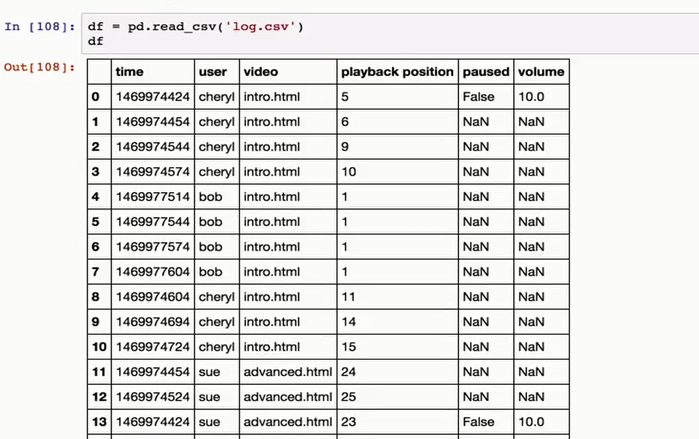
**Missing Values**

We'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 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 earning 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 heart beat 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

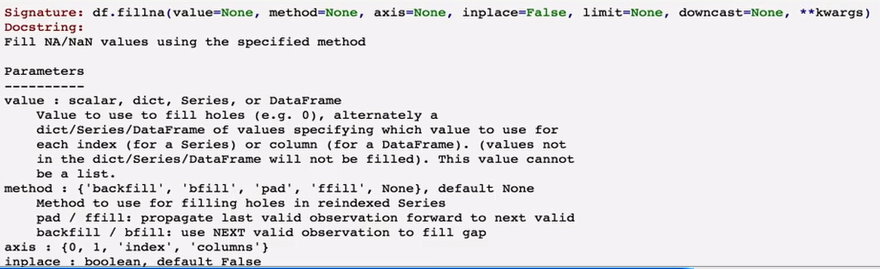
might look like. In this data the first column is a timestamp in the Unix epoch format. The next column is the user name followed by a web page they're visiting and the video that they're playing. Each row of the DataFrame has a playback position. And we can see that as the playback position increases by one, the time stamp increases by about 30 seconds.

  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's no changes.

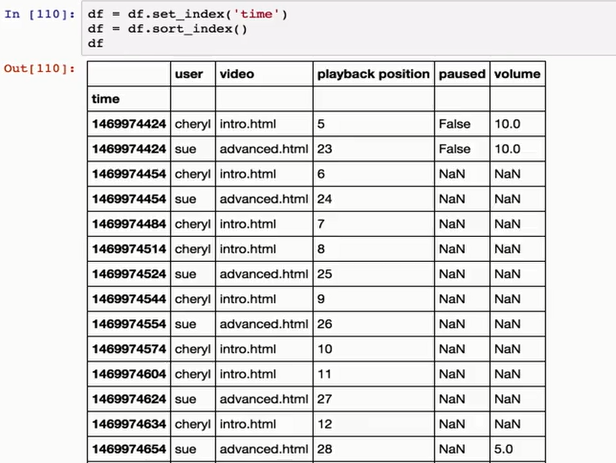
One of the handy functions that Pandas has for working with missing values is the filling function, fillna.





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. 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.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.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.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