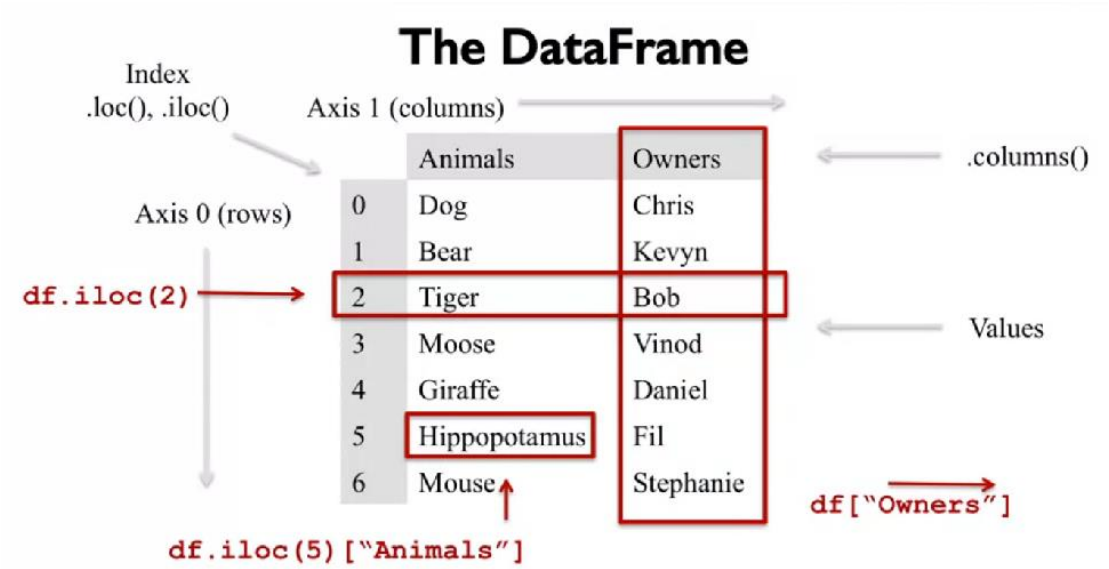
The DataFrame Data Structure

The DataFrame 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 DataFrame 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 DataFrame itself as simply a two-axes labeled array.



You can create a DataFrame 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'll purchase a bunch of dog food.
Kevin Cullens Thompson, I'll have him purchase some kitty litter.
And I think Vinod is more of a bird man, so I'll add some bird seed there.

Then we'll feed this into the DataFrame 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 DataFrame, the Jupiter notebook's trying to pretty it up a bit, and print it out as a table, which is nice.

```
import pandas as pd
purchase_1 = pd.Series({'Name': 'Chris',
                        'Item Purchased': 'Dog Food',
                        'Cost': 22.50})
purchase_2 = pd.Series({'Name': 'Kevyn',
                        'Item Purchased': 'Kitty Litter',
                        'Cost': 2.50})
purchase_3 = pd.Series({'Name': 'Vinod',
                        'Item Purchased': 'Bird Seed',
                        'Cost': 5.00})
df = pd.DataFrame([purchase_1, purchase_2, purchase_3], index=['Store 1', 'Store 1', 'Store 2'])
df.head()
```

	Name	Item Purchased	Cost
Store 1	Chris	Dog Food	22.5
Store 1	Kevyn	Kitty Litter	2.5
Store 2	Vinod	Bird Seed	5.0

Similar to the series, we can extract data using the *iLoc* and *Loc* attributes. Because the DataFrame is two-dimensional, passing a single value to the loc 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 *loc* 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.

```
df.loc['Store 2']
```

```
Name                   Vinod
Item Purchased    Bird Seed
Cost                      5
Name: Store 2, dtype: object
```

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

```
type(df.loc['Store 2'])
```

```
pandas.core.series.Series
```

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 DataFrame loc attribute, multiple rows of the DataFrame will return, not as a new series, but as a new DataFrame.

```
df.loc['Store 1']
```

	Name	Item Purchased	Cost
Store 1	Chris	Dog Food	22.5
Store 1	Kevyn	Kitty Litter	2.5

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 DataFrame 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 .loc, 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.loc['Store 1', 'Cost'].

```
df.loc['Store 1', 'Cost']
```

```
Store 1    22.5
Store 1     2.5
Name: Cost, dtype: float64
```

What if we just wanted to do column selection and just get a list of all of the costs?
Well, there are 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 .loc method. This works, but it's pretty ugly.

```
df.T
```

	Store 1	Store 1	Store 2
Name	Chris	Kevyn	Vinod
Item Purchased	Dog Food	Kitty Litter	Bird Seed
Cost	22.5	2.5	5

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.

```
df.T.loc['Cost']
```

```
Store 1    22.5
Store 1     2.5
Store 2       5
Name: Cost, dtype: object
```

```
df['Cost']
```

```
Store 1    22.5
Store 1     2.5
Store 2     5.0
Name: Cost, dtype: float64
```

```
df.loc['Store 1']['Cost']
```

```
Store 1    22.5
Store 1     2.5
Name: Cost, dtype: float64
```

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.

```
df.loc[:,['Name', 'Cost']]
```

	Name	Cost
Store 1	Chris	22.5
Store 1	Kevyn	2.5
Store 2	Vinod	5.0

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, points it out, and shares a less ambiguous solution.

Now, before we leave the discussion of accessing data in DataFrames, 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.

```
df.drop('Store 1')
```

	Name	Item Purchased	Cost
Store 2	Vinod	Bird Seed	5.0

```
df
```

	Name	Item Purchased	Cost
Store 1	Chris	Dog Food	22.5
Store 1	Kevyn	Kitty Litter	2.5
Store 2	Vinod	Bird Seed	5.0

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.

```
copy_df = df.copy()
copy_df = copy_df.drop('Store 1')
copy_df
```

	Name	Item Purchased	Cost
Store 2	Vinod	Bird Seed	5.0

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.

```
del copy_df['Name']
copy_df
```

	Item Purchased	Cost
Store 2	Bird Seed	5.0

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.

```
df['Location'] = None
df
```

	Name	Item Purchased	Cost	Location
Store 1	Chris	Dog Food	22.5	None
Store 1	Kevyn	Kitty Litter	2.5	None
Store 2	Vinod	Bird Seed	5.0	None