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**.

In [1]:

```
# Lets start by importing our pandas library
import pandas as pd
```

In [2]:

```
# I'm going to jump in with an example. Lets create three school records for stu
dents and their
# class grades. I'll create each as a series which has a student name, the class
name, and the score.
record1 = pd.Series({'Name': 'Alice',
                        'Class': 'Physics',
                        'Score': 85})
record2 = pd.Series({'Name': 'Jack',
                        'Class': 'Chemistry',
                        'Score': 82})
record3 = pd.Series({'Name': 'Helen',
                        'Class': 'Biology',
                        'Score': 90})
```

Creating A DataFrame

Using Multiple Series

In [3]:

```
# Like a Series, the DataFrame object is index. Here I'll use a group of series, where each series
# represents a row of data. Just like the Series function, we can pass in our individual items
# in an array, and we can pass in our index values as a second arguments
df = pd.DataFrame([record1, record2, record3],
                  index=['school1', 'school2', 'school1'])

# And just like the Series we can use the head() function to see the first several rows of the
# dataframe, including indices from both axes, and we can use this to verify the columns and the rows
df.head()
```

Out[3]:

	Name	Class	Score
school1	Alice	Physics	85
school2	Jack	Chemistry	82
school1	Helen	Biology	90

List of Dictionaries

You'll notice here that Jupyter creates a nice bit of HTML to render the results of the dataframe. So we have the index, which is the leftmost column and is the school name, and then we have the rows of data, where each row has a column header which was given in our initial record dictionaries

In [4]:

```
# An alternative method is that you could use a list of dictionaries, where each dictionary
# represents a row of data.

students = [{'Name': 'Alice',
              'Class': 'Physics',
              'Score': 85},
            {'Name': 'Jack',
             'Class': 'Chemistry',
             'Score': 82},
            {'Name': 'Helen',
             'Class': 'Biology',
             'Score': 90}]

# Then we pass this list of dictionaries into the DataFrame function
df = pd.DataFrame(students, index=['school1', 'school2', 'school1'])## Rmbr the indexes need not be unique!!
# And lets print the head again
df.head()
```

Out[4]:

	Name	Class	Score
school1	Alice	Physics	85
school2	Jack	Chemistry	82
school1	Helen	Biology	90

In [5]:

```
# 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
# the series if there's only one row to return.

# For instance, if we wanted to select data associated with school2, we would just query the
# .loc attribute with one parameter.
df.loc['school2']
```

Out[5]:

```
Name          Jack
Class    Chemistry
Score           82
Name: school2, dtype: object
```

In [6]:

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

# We can check the data type of the return using the python type function.
type(df.loc['school2'])
```

Out[6]:

```
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. In this example, we see two records for school1 as different rows. If we use a single value with the DataFrame lock attribute, multiple rows of the DataFrame will return, not as a new series, but as a new DataFrame.

In [8]:

```
# Lets query for school1 records
df.loc['school1']
```

Out[8]:

	Name	Class	Score
school1	Alice	Physics	85
school1	Helen	Biology	90

In [9]:

```
# And we can see the the type of this is different too
type(df.loc['school1'])
```

Out[9]:

```
pandas.core.frame.DataFrame
```

1st Column For 'Row', 2nd column for 'Column'

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 student names for school1, you would supply two parameters to .loc, one being the row index and the other being the column name. Remember, just like the Series, the pandas developers have implemented this using the indexing operator and not as parameters to a function.

In [10]:

```
# For instance, if we are only interested in school1's student names
df.loc['school1', 'Name']
```

Out[10]:

```
school1    Alice
school1    Helen
Name: Name, dtype: object
```

Transpose The Matrix If You Want To Call A Column With "One attribute"

In [13]:

```
# What would we do if we just wanted to select a single column though? Well, the
re are a few
# mechanisms. Firstly, we could transpose the matrix. This pivots all of the row
s into columns
# and all of the columns into rows, and is done with the T attribute
df.T
```

Out[13]:

	school1	school2	school1
Name	Alice	Jack	Helen
Class	Physics	Chemistry	Biology
Score	85	82	90

In [14]:

```
# Then we can call .loc on the transpose to get the student names only
df.T.loc['Name']
```

Out[14]:

```
school1    Alice
school2     Jack
school1    Helen
Name: Name, dtype: object
```

However, since iloc and loc are used for row selection, Panda **reserves the 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, and is 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.

In [16]:

```
df['Name']
```

Out[16]:

```
school1    Alice
school2     Jack
school1    Helen
Name: Name, dtype: object
```

In [17]:

```
# In practice, this works really well since you're often trying to add or drop n
ew columns. However,
# this also means that you get a key error if you try and use .loc with a column
name
try:
    df.loc['Name']
except:
    print ("KeyError")
```

```
KeyError
```

In [18]:

```
# Note too that the result of a single column projection is a Series object
type(df['Name'])
```

Out[18]:

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

Chaining Operations Together

In [19]:

```
# Since the result of using the indexing operator is either a DataFrame or Serie
s, you can chain
# operations together. For instance, we can select all of the rows which related
to school1 using
# .loc, then project the name column from just those rows
df.loc['school1']['Name']
```

Out[19]:

```
school1    Alice
school1    Helen
Name: Name, dtype: object
```

SideTrack: Checking Using Type

In [21]:

```
# If you get confused, use type to check the responses from resulting operations
print(type(df.loc['school1'])) #should be a DataFrame
print(type(df.loc['school1']['Name'])) #should be a Series
```

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.series.Series'>
```

Limitations of Chaining

Chaining, by indexing on the return type of another index, 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 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.

In [24]:

```
# Here's another approach. As we saw, .loc does row selection, and it can take t
wo parameters,
# the row index and the list of column names. The .loc attribute also supports s
licing.

# If we wanted to select all rows, we can use a colon to indicate a full slice f
rom beginning to end.
# This is just like slicing characters in a list in python. Then we can add the
 column name as the
# second parameter as a string. If we wanted to include multiple columns, we cou
ld 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 the names and scores for all schools u
sing the .loc operator.
df.loc[:,['Name', 'Score']]
```

Out[24]:

	Name	Score
school1	Alice	85
school2	Jack	82
school1	Helen	90

In [ ]:

```
# Take a look at that again. The colon means that we want to get all of the row
s, and the list
# in the second argument position is the list of columns we want to get back
```

In [ ]:

```
# 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 ben
efit. Underneath
# this is just a two axes labeled array, and transposing the columns is easy. Al
so, consider the
# issue of chaining carefully, and try to avoid it, as it can cause unpredictabl
e results, where
# your intent was to obtain a view of the data, but instead Pandas returns to yo
u a copy.
```

Deleting Data

Before we leave the discussion of accessing data in DataFrames, lets 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 row label, to drop. This is another tricky place for new users -- **the drop function doesn't change the DataFrame by default!** Instead, the drop function **returns to you a copy** of the DataFrame with the given rows removed.

In [26]:

```
df.drop('school1')
```

Out[26]:

	Name	Class	Score
school2	Jack	Chemistry	82

In [27]:

```
# But if we look at our original DataFrame we see the data is still intact.
df
```

Out[27]:

	Name	Class	Score
school1	Alice	Physics	85
school2	Jack	Chemistry	82
school1	Helen	Biology	90

Two Optional Parameters

Drop has two interesting optional parameters:

- `inplace` , and if it's set to `true` , the DataFrame will be updated in place, instead of a copy being returned.
- `axis` which is **by default** `0` which indicates the row to be dropped, which can be changed to `1` if you want to drop a column.

In [32]:

```
# For example, lets make a copy of a DataFrame using .copy()
copy_df = df.copy()
# Now lets drop the name column in this copy
copy_df.drop("Name", inplace=True, axis=1)
copy_df
```

Out[32]:

	Class	Score	ClassRanking
school1	Physics	85	None
school2	Chemistry	82	None
school1	Biology	90	None

Using The del Keyword

There is a second way to drop a column, 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.

In [33]:

```
del copy_df['Class']
copy_df
```

Out[33]:

	Score	ClassRanking
school1	85	None
school2	82	None
school1	90	None

Adding A New Column

Adding a new column to the DataFrame is as easy as assigning it to some value using the indexing operator.

For instance, if we wanted to add a `class ranking` 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.

In [30]:

```python
df['ClassRanking'] = None
df
```

Out[30]:

	Name	Class	Score	ClassRanking
school1	Alice	Physics	85	None
school2	Jack	Chemistry	82	None
school1	Helen	Biology	90	None

Summary

In this lecture you've learned about the data structure you'll use the most in pandas, the DataFrame. The dataframe is indexed both by row and column, and you can easily select individual rows and project the columns you're interested in using the familiar indexing methods from the Series class. You'll be gaining a lot of experience with the DataFrame in the content to come.