Pandas DataFrames

The official project homepage

- Goal
  - Extend what we learned about Series objects in the previous tutorial to their 2D counterpart - DataFrames
  - Develop some tools for dealing with missing data (not exhaustive, but a start)

DataFrames

Pandas quick start guide for DataFrames

- A DataFrame (DF) is a labeled data struture that can be thought of as a 2D extension of the Series objects that we discussed in the first part of the tutorial
- A DF can accept many types of input, multiple Series, a dict of 1D arrays, another DF, etc
- Like a Series, DFs contain data values and their labels. Because we're now dealing with a 2D structure, we call the **row labels the index argument** and the **column labels the column argument**.
  - Like a Series, if you don't explicitly assign row and column labels, then they will be auto-generated (but not as useful as specifying the labels yourself!)

Much of what we learned about Series objects will generalize to DFs, so here we'll focus on some of key functionality that might not be obvious based on the first part of the tutorial.

One more quick note: if using an older version of Python (earlier than 3.6) and Pandas (earlier than 0.23) and you create a DF from a dict without explicitly specifying column names, then the column names will be entered into the DF based on lexical order

Import libs

In [ ]:
```
# import a generic pandas object and also a few specific functions that we'll use
import pandas as pd
from google.colab import files
```

Upload a file to the /content folder on google colab

- Select the file you want to upload (the csv file that I sent out)
- It will load into your 'contents' folder
- Then you can interact with it just like a normal file on your hardrive

In [ ]:
```
%ls
```

In [ ]:
```
files.upload()
```

Remove unwanted files...

In [ ]:
```
%ls
```

In [ ]:
```
%rm *.csv
```

Make a DataFrame object to hold the contents of the data set

DataFrame help page

- Just like with the pd.Series call, you can specify the data, index labels (row labels in this case)
- In addition to row labels, you can also specify column labels (with 'columns')
- Can also specify data type (default is inferred)
- If you read in the data from a csv file, you will be able to inherit row and column labels (if they are specified in the file).

In [ ]:
```
# make the call to pd.DataFrames to create the DF - usage much like pd.Series
df = pd.read_csv('annual_temp_csv2.csv')
```

In [ ]:
```
# take a look at the output...
# compare to print(df) - looks nicer with display thanks to iPython backend
display(df)
```

In [ ]:
```
# another handy display function...good for large dfs that are too big to fit -
# at least you can get an idea of the overall structure
df.head()
```

Get a high-level summary of the data using built-in functionality of DataFrame object

API reference page

- What do you notice about the two counts for Year and for Mean

In [ ]:
```
df.describe()
```

Just like with Series object, can compute mean, std, etc

In [ ]:
```
df['Year'].mean()
```

remember that you can also call by field...I prefer by name like ['Mean'] to avoid confusion with built in methods/functions, but either will work

In [ ]:
```
df.Mean.std()
```

By default, mean, std etc will skip (ignore) missing values (NaNs)

- Sometimes, its good to do a sanity check if you think there are missing values.
- Can do this by chosing to NOT skip the NANs...in which case if they exist you'll get back NaN as the answer!
- Then you know that there are NaNs in the data set.

In [ ]:
```
df['Mean'].mean(skipna=False)
```

Find missing values in your data and deal with them (NaNs)

- Can apply to just one column at a time
- note that you can call the isna method from the object directly
- To make this work, you index into the data frame where 'Mean' is a Nan

In [ ]:
```
# isolate just the rows (indicies) where Mean is NaN
df['Mean'].isna()
```

In [ ]:
```
# now index into df using the true/false sequence from above!
df[df['Mean'].isna()]
```

Or do the opposite, isolate just the rows where Mean is not NaN (i.e. its a real number)

In [ ]:
```
df[df['Mean'].notna()]
```

Can deal with NaNs lots of ways...

- Can make a new DF without them
- can assign the mean of all of the data to NaNs

In [ ]:
```
# make a new df, but only keep the non-NaN entries
df2 = df[df['Mean'].notna()]
df2.head()
```

Fill the NaN with the mean of the column!

- Or any other value...just pass it into fillna
- see also 'interpolate' for more functions like this
- NEED TO ASSIGN output to apply changes..e.g df = df.fillna(....)

In [ ]:
```
print(df['Mean'].mean())
df2=df.fillna(df['Mean'].mean())
df2.head()
```

Pull out selected data and remove from DF

In [ ]:
```
df.head()
```

In [ ]:
```
# or your could self assign df = df[] here to update existing data frame
df2 = df[df['Source'] == 'GISTEMP']
df2.head()
```

Grab a range of rows...across a set of years, for example

In [ ]:
```
df2 = df[df['Year']>1990) & (df['Year']<=2015)]
display(df2)
```

Apply several filters at once!

- Be careful here - readability of code is the prime directive...don't write one-liners that are so dense that nobody can understand them!

In [ ]:
```
df2 = df[df['Source']=='GCAG') & (df['Year']>1990) & (df['Year']<=2015)]
display(df2)
```

More on indexing and selection of specific coordinates in a DF

Row selection - this is a bit more complex as there are many methods

- You can use df.loc to select a row by its label name
- You can use df.iloc to select a row by its integer location (from 0 to length-1 of the axis)
- You can use boolean vectors to select a set of rows that satisfy some condition

Contrary to usual slicing conventions, both the start and the stop indices are included when using the DF.LOC option...see below for demo. This makes sense because you're indexing by label name, not by a zero-based integer index.

In [ ]:
```
#for the next steps, load annual_temp2
files.upload()
```

In [ ]:
```
df = pd.read_csv('annual_temp2.csv', index_col=0)
```

In [ ]:
```
df.head()
```

This returns the data associated with one row

In [ ]:
```
df.loc[2014]
```

this returns the rows associated with a set of years specified in a list

In [ ]:
```
# non-contiguous entries
df.loc[[1999,2015,2016]]
```

In [ ]:
```
# note that years run in descending order...
df.loc[2014:2016]
```

In [ ]:
```
# but this will work...
df.loc[2016:2014]
```

In [ ]:
```
# flip the data frame upside down
df=df.loc[::-1]
```

In [ ]:
```
# now the years run in order
df.loc[2014:2016]
```

iloc does indexing by row location (not label)

- use normal rules of slicing here...start:stop:step

In [ ]:
```
# the first 10
df.iloc[:10]

# # reverse
# df.iloc[::-1]

# # every other
# df.iloc[::2]
```

Adding a column is easy and can be done dynamically (on the fly)

- Make a new column of True and False to mark years above/below mean temp deviation

In [ ]:
```
mean_temp = df['Mean'].mean()

print('mean temp:', mean_temp)

# then populate the new column
df['HighLow'] = df['Mean']>mean_temp

df.head()
```

If you want to convert values in a column, can be a little tricky...

- Use what you might think is the intuitive way to convert True to 1 and False to 0 in our new column 'HighLow'
- **This throws a weird warning because you're trying to modify the thing that you're using as an index!**

In [ ]:
```
df.HighLow[df.HighLow==True] = 1
df.HighLow[df.HighLow==False] = 0
df.head()
```

Solution - use .loc to return the information and then modify it

In [ ]:
```
# reload our df, re-create a new version of our HighLow column

df = pd.read_csv('annual_temp_csv2.csv')

# make our new column again.
mean_temp = df['Mean'].mean()

print('mean temp:', mean_temp)

# then populate the new column
df['HighLow'] = df['Mean']>mean_temp

df.head()
```

like this...using the .loc method ensures that things don't get confused...

In [ ]:
```
df.loc[df['HighLow']==True, 'HighLow'] = 1
df.loc[df['HighLow']==False, 'HighLow'] = 0
df.head()
```

Access specific row and columns!

In [ ]:
```
# flip it over to chronological order so its a little more intuitive to slice
df = df.loc[::-1]
df.head()
```

In [ ]:
```
# just data from one year from just the "Mean" column
df.loc[2016]['Mean']
```

In [ ]:
```
# a range of years
df.loc[1880:1884]['Mean']
```

In [ ]:
```
# range of rows and columns
df.loc[1880:1884][['Mean','HighLow']]
```

In [ ]:
```
# note that the order in which you ask for columns impacts the output
row_ind = [1880,1980]
col_ind = ['HighLow', 'Mean']
df2 = df.loc[row_ind][col_ind]
df2.head()
```