Combining DataFrames

Full Official Guide (Lots of examples!)

https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

---

In [132]:

```
import numpy as np
import pandas as %pd
```

Concatenation

Directly "glue" together dataframes.

In [133]:

```
data_one = {'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3']}
data_two = {'C': ['C0', 'C1', 'C2', 'C3'], 'D': ['D0', 'D1', 'D2', 'D3']}
```

In [134]:

```
one = pd.DataFrame(data_one)
one
```

Out[134]:

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

In [135]:

```
two = pd.DataFrame(data_two)
two
```

Out[135]:

	C	D
0	C0	D0
1	C1	D1
2	C2	D2
3	C3	D3

Axis = 1

Concatenate along columns

In [136]:

```
pd.concat([one,two],axis=1)
```

Out[136]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

Axis = 0

Concatenate along rows

In [137]:

```
pd.concat([one,two],axis=0)
```

Out[137]:

	A	B	C	D
0	A0	B0	NaN	NaN
1	A1	B1	NaN	NaN
2	A2	B2	NaN	NaN
3	A3	B3	NaN	NaN
0	NaN	NaN	C0	D0
1	NaN	NaN	C1	D1
2	NaN	NaN	C2	D2
3	NaN	NaN	C3	D3

In [138]:

```
# If we want data in same column so we have to make column A = C and Column B = to column D
two.columns = one.columns
two
```

Out[138]:

	A	B
0	C0	D0
1	C1	D1
2	C2	D2
3	C3	D3

In [139]:

```
mydf=pd.concat([one,two],axis=0)
mydf
```

Out[139]:

	A	B
0	A0	B0
1	A1	B1

	A	B
2	A2	B2
3	A3	B3
0	C0	D0
1	C1	D1
2	C2	D2
3	C3	D3

In [140]:

```
# Now fixing index numbering
mydf.index
```

Out[140]:

```
Int64Index([0, 1, 2, 3, 0, 1, 2, 3], dtype='int64')
```

In [141]:

```
mydf.index = range(len(mydf))
mydf
```

Out[141]:

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3
4	C0	D0
5	C1	D1
6	C2	D2
7	C3	D3

Merge

Data Tables

In [142]:

```
registrations = pd.DataFrame({'reg_id':[1,2,3,4],'name':['Andrew','Bobo','Claire','David
']})
logins = pd.DataFrame({'log_id':[1,2,3,4],'name':['Xavier','Andrew','Yolanda','Bobo']})
```

In [143]:

```
registrations
```

Out[143]:

	reg_id	name
0	1	Andrew
1	2	Bobo
2	3	Claire
3	4	David

In [144]:

```
logins
```

Out[144]:

	log_id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

pd.merge()

Merge pandas DataFrames based on key columns, similar to a SQL join. Results based on the how parameter.

In [145]:

```
# help(pd.merge)  # help in pd.merge
```

Inner,Left, Right, and Outer Joins

Inner Join

Match up where the key is present in BOTH tables. There should be no NaNs due to the join, since by definition to be part of the Inner Join they need info in both tables. Only Andrew and Bobo both registered and logged in.

In [146]:

```
# Notice pd.merge doesn't take in a list like concat
pd.merge(registrations,logins, how = 'inner', on='name')
```

Out[146]:

	reg_id	name	log_id
0	1	Andrew	2
1	2	Bobo	4

In [147]:

```
# Pandas smart enough to figure out key column (on parameter) if only one column name mat
ches up
pd.merge(registrations,logins,how='inner')
```

Out[147]:

	reg_id	name	log_id
0	1	Andrew	2
1	2	Bobo	4

In [148]:

```
# Pandas reports an error if "on" key column isn't in both dataframes
# pd.merge(registrations,logins,how='inner',on='reg_id')
```

Left Join

Match up AND include all rows from Left Table. Show everyone who registered on Left Table, if they don't have login info, then fill with NaN.

In [149]:

```
pd.merge(left=registrations,right=logins,how='left',on='name')
#pd.merge(registrations,logins,how='left',on='name')
# pd.merge(registrations,logins,how='left') work same
```

Out[149]:

	reg_id	name	log_id
0	1	Andrew	2.0
1	2	Bobo	4.0
2	3	Claire	NaN
3	4	David	NaN

Right Join

Match up AND include all rows from Right Table. Show everyone who logged in on the Right Table, if they don't have registration info, then fill with NaN.

In [150]:

```
pd.merge(left=registrations,right=logins,how='right')
#pd.merge(registrations,logins,how='right',on='name')
```

Out[150]:

	reg_id	name	log_id
0	NaN	Xavier	1
1	1.0	Andrew	2
2	NaN	Yolanda	3
3	2.0	Bobo	4

Outer Join

Match up on all info found in either Left or Right Table. Show everyone that's in the Log in table and the registrations table. Fill any missing info with NaN

In [151]:

```
pd.merge(registrations, logins,how='outer',on='name')
```

Out[151]:

	reg_id	name	log_id
0	1.0	Andrew	2.0
1	2.0	Bobo	4.0
2	3.0	Claire	NaN
3	4.0	David	NaN
4	NaN	Xavier	1.0
5	NaN	Yolanda	3.0

In [152]:

```
# Here we set name as index
```

```
registrations = registrations.set_index('name')
registrations
```

Out[152]:

name	reg_id
Andrew	1
Bobo	2
Claire	3
David	4

In [153]:

```
logins
```

Out[153]:

	log_id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

In [154]:

```
# Now we are going to merge on the bases of registratins index thats name we just asign s
o we set left_index=True or we can
# use left_on and right_on for common link
pd.merge(registrations , logins, left_index=True, right_on='name', how='inner')
```

Out[154]:

	reg_id	log_id	name
1	1	2	Andrew
3	2	4	Bobo

In [155]:

```
registrations
```

Out[155]:

name	reg_id
Andrew	1
Bobo	2
Claire	3
David	4

Join on Index or Column

Use combinations of left_on,right_on,left_index,right_index to merge a column or index on each other

In [156]:

```
# reset the index
```

```
registrations = registrations.reset_index()
registrations
```

Out[156]:

	name	reg_id
0	Andrew	1
1	Bobo	2
2	Claire	3
3	David	4

In [160]:

```
registrations.columns=['reg_name','reg_id']
registrations
```

Out[160]:

	reg_name	reg_id
0	Andrew	1
1	Bobo	2
2	Claire	3
3	David	4

In [162]:

```
logins
```

Out[162]:

	log_id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

Dealing with differing key column names in joined tables

In [169]:

```
result=pd.merge(registrations, logins, left_on='reg_name',right_on='name', how='inner')
result
```

Out[169]:

	reg_name	reg_id	log_id	name
0	Andrew	1	2	Andrew
1	Bobo	2	4	Bobo

In [170]:

```
result.drop('reg_name',axis=1)
```

Out[170]:

	reg_id	log_id	name
0	1	2	Andrew

1 reg_id 2 log_id 4 Bobo name

In [174]:

```
registrations.columns = ['name','id']
logins.columns = ['id','name']
```

In [176]:

```
logins
```

Out[176]:

	id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

In [177]:

```
registrations
```

Out[177]:

	name	id
0	Andrew	1
1	Bobo	2
2	Claire	3
3	David	4

In [182]:

```
# now we have same column name in both tables
pd.merge(registrations, logins, how='inner', on = 'name')
# _x is for left
# _y is for right
```

Out[182]:

	name	id_x	id_y
0	Andrew	1	2
1	Bobo	2	4

Here we see that id_x and id_y add automatically that help to identify that belong to which table like x is from left and y is from right table we can set out suffixes too

In [181]:

```
pd.merge(registrations,logins,how='inner',on='name',suffixes=['_reg','_log'])
```

Out[181]:

	name	id_reg	id_log
0	Andrew	1	2
1	Bobo	2	4

Text Methods

A normal Python string has a variety of method calls available:

In [184]:

```
email = 'niko@email.com'
```

In [186]:

```
email.split('@')
```

Out[186]:

```
['niko', 'email.com']
```

In [190]:

```
names =pd.Series(['andrew','bobo','claire','david','5'])
```

Pandas and Text

Pandas can do a lot more than what we show here. Full online documentation on things like advanced string indexing and regular expressions with pandas can be found here:
https://pandas.pydata.org/docs/user_guide/text.html

Text Methods on Pandas String Column

In [191]:

```
names
```

Out[191]:

```
0    andrew
1      bobo
2    claire
3     david
4         5
dtype: object
```

In [193]:

```
names.str.upper()
```

Out[193]:

```
0    ANDREW
1      BOBO
2    CLAIRE
3     DAVID
4         5
dtype: object
```

In [195]:

```
email.isdigit() # we tab after . we will get all possible options
# Here i false for email because it is string
```

Out[195]:

```
False
```

In [199]:

```
'5'.isdigit() # here '5' is still digit
```

Out[199]:

```
True
```

In [200]:

```
# Same we will try on name
names.str.isdigit()
```

Out[200]:

```
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

Splitting , Grabbing, and Expanding

In [201]:

```
tech_finance = ['GOOG,APPL,AMZN','JPM,BAC,GS']
```

In [202]:

```
len(tech_finance)
```

Out[202]:

```
2
```

In [204]:

```
tickers = pd.Series(tech_finance)
```

In [205]:

```
tickers.str.split(',')
```

Out[205]:

```
0    [GOOG, APPL, AMZN]
1        [JPM, BAC, GS]
dtype: object
```

In [206]:

```
tech = 'GOOG,APPL,AMZN'
```

In [208]:

```
# split data to check
tech.split(',')
```

Out[208]:

```
['GOOG', 'APPL', 'AMZN']
```

In [210]:

```
# Now calling first item in list
tech.split(',')[0]
```

Out[210]:

```
'GOOG'
```

In [217]:

```
# we will split tickers, and in this way it return first item of both rows
tickers.str.split(',').str[0]
```

Out[217]:

```
0    GOOG
```

```
1      JPM
dtype: object
```

In [220]:

```
# we can create table by expending
tickers.str.split(',',expand=True)
```

Out[220]:

	0	1	2
0	GOOG	APPL	AMZN
1	JPM	BAC	GS

Cleaning or Editing Strings

In [221]:

```
messy_names = pd.Series(["andrew  ","bo;bo","  claire  "])
```

In [223]:

```
messy_names[0]
```

Out[223]:

```
'andrew  '
```

In [224]:

```
messy_names
```

Out[224]:

```
0      andrew
1         bo;bo
2      claire
dtype: object
```

In [225]:

```
# Remove ; by using replace
messy_names.str.replace(';','')
```

Out[225]:

```
0      andrew
1          bobo
2      claire
dtype: object
```

In [226]:

```
# now we will remove space at end by using strip
messy_names.str.replace(';','').str.strip()
```

Out[226]:

```
0    andrew
1      bobo
2    claire
dtype: object
```

In [228]:

```
# Here we see that space from the end of andrew is removed
messy_names.str.replace(';','').str.strip()[0]
```

Out[228]:

```
'andrew'
```

In [229]:

```
# Here next we going to capitalize first letter by capitalize
messy_names.str.replace(';','').str.strip().str.capitalize()
```

Out[229]:

```
0    Andrew
1      Bobo
2    Claire
dtype: object
```

Alternative with Custom apply() call

In [230]:

```
def cleanup(name):
    name = name.replace(";","")
    name = name.strip()
    name = name.capitalize()
    return name
```

In [231]:

```
messy_names
```

Out[231]:

```
0      andrew
1         bo;bo
2      claire
dtype: object
```

In [232]:

```
messy_names.apply(cleanup)
```

Out[232]:

```
0    Andrew
1      Bobo
2    Claire
dtype: object
```

Which one is more efficient?

In [234]:

```
import timeit

# code snippet to be executed only once
setup = '''
import pandas as pd
import numpy as np
messy_names = pd.Series(["andrew  ","bo;bo","  claire  "])
def cleanup(name):
    name = name.replace(";","")
    name = name.strip()
    name = name.capitalize()
    return name
'''

# code snippet whose execution time is to be measured
stmt_pandas_str = '''
messy_names.str.replace(";","").str.strip().str.capitalize()
'''
```

```
stmt_pandas_apply = '''
messy_names.apply(cleanup)
'''

stmt_pandas_vectorize='''
np.vectorize(cleanup)(messy_names)
'''
```

In [235]:

```
timeit.timeit(setup = setup,
                    stmt = stmt_pandas_str,
                    number = 10000)
```

Out[235]:

```
7.5039381999959005
```

In [236]:

```
timeit.timeit(setup = setup,
                    stmt = stmt_pandas_apply,
                    number = 10000)
```

Out[236]:

```
1.873779800000193
```

In [237]:

```
timeit.timeit(setup = setup,
                    stmt = stmt_pandas_vectorize,
                    number = 10000)
```

Out[237]:

```
0.5948379999972531
```

Wow! While .str() methods can be extremely convienent, when it comes to performance, don't forget about np.vectorize()! Review the "Useful Methods" lecture for a deeper discussion on np.vectorize()

In [ ]:

```

```