fabsta / pandas cheat sheet.md

Created 6 years ago • Report abuse

Star

Code Revisions 1 Stars 6 Forks 8

pandas cheat sheet

pandas cheat sheet.md

[TOC]

Preliminaries/Import

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas import DataFrame, Series

%matplotlib inline
%load_ext autoreload
%autoreload 2

# from __future__ import division
from import_file import *
from helpers.parallel_helper import *
load_libs()



# normal
import numpy as np
import pandas as pd
import time
import warnings
warnings.filterwarnings('ignore')
from __future__ import division # allows float division


# plotting
import matplotlib.pyplot as plt
%matplotlib inline
import plotly.offline as py
py.init_notebook_mode(connected=True)
import plotly.graph_objs as go
import plotly.tools as tls
import seaborn as sns


# with user code
import sys
sys.path.append('../shared/')
import plotting
import utils
import skutils


# plotting
from altair import *
```

(Back to top)

Input/Output

Input

Internal

```
# From Dictionary
df = DataFrame({
        'col0' : [1.0, 2.0, 3.0, 4.0],
        'col1' : [100, 200, 300, 400]
})

# use helper method for data in rows
df = DataFrame.from_dict({ # data by row
   # rows as python dictionaries
        'row0' : {'col0':0, 'col1':'A'},
        'row1' : {'col0':1, 'col1':'B'}
        }, orient='index')
df = DataFrame.from_dict({ # data by row
   # rows as python lists
        'row0' : [1, 1+1j, 'A'],
        'row1' : [2, 2+2j, 'B']
        }, orient='index')


# Testing
df = DataFrame(np.random.rand(50,5))
```

```
# with a time-stamp row index:
df = DataFrame(np.random.rand(500,5))
df.index = pd.date_range('1/1/2005',
periods=len(df), freq='M')

# with alphabetic row and col indexes and a "groupable" variable
import string
import random
r = 52 # note: min r is 1; max r is 52
c = 5
df = DataFrame(np.random.randn(r, c),
        columns = ['col'+str(i) for i in range(c)],
        index = list((string. ascii_uppercase+ string.ascii_lowercase)[0:r]))
        df['group'] = list(''.join(random.choice('abcde')
        for _ in range(r)) )
```

External

```
df = DataFrame()

#CSV
df = pd.read_csv('file.csv')
df = pd.read_csv('file.csv', header=0, index_col=0, quotechar='"',sep=':', na_val
# specifying "." and "NA" as missing values in the Last Name column and "." as mi
df = pd.read_csv('../data/example.csv', na_values={'Last Name': ['.', 'NA'], 'Pre
df = pd.read_csv('../data/example.csv', na_values=sentinels, skiprows=3)    # ski
df = pd.read_csv('../data/example.csv', thousands=',')  # interpreting "," in str

# CSV (Inline)
from io import StringIO
data = """, Animal, Cuteness, Desirable
row-1, dog, 8.7, True
row-2, cat, 9.5, True
row-3, bat, 2.6, False"""
df = pd.read_csv(StringIO(data),
        header=0, index_col=0,
        skipinitialspace=True)

# JSON
import json
json_data = open('data-text.json').read()
data = json.loads(json_data)
for item in data:
    print item

# XML
from xml.etree import ElementTree as ET
tree = ET.parse('../../data/chp3/data-text.xml')
root = tree.getroot()
```

~~~
print root
data = root.find('Data')
all_data = []
for observation in data:
    record = {}
    for item in observation:
        lookup_key = item.attrib.keys()[0]
        if lookup_key == 'Numeric':
            rec_key = 'NUMERIC'
            rec_value = item.attrib['Numeric']
        else:
            rec_key = item.attrib[lookup_key]
            rec_value = item.attrib['Code']
        record[rec_key] = rec_value
    all_data.append(record)
print all_data

# Excel
workbook = pd.ExcelFile('file.xlsx')
d = {} # start with an empty dictionary
for sheet_name in workbook.sheet_names:   # Each Excel sheet in a Python dictiona
        df = workbook.parse(sheet_name)
        d[sheet_name] = df

# MySQL
import pymysql
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://'
+'USER:PASSWORD@HOST/DATABASE')
df = pd.read_sql_table('table', engine)
(<a href="#top">Back to top</a>)



### Combine DataFrame
Data in Series then combine into a DataFrame
```python
# Example 1 ...
s1 = Series(range(6))
s2 = s1 * s1
s2.index = s2.index + 2# misalign indexes
df = pd.concat([s1, s2], axis=1)
# Example 2 ...
s3 = Series({'Tom':1, 'Dick':4, 'Har':9})
s4 = Series({'Tom':3, 'Dick':2, 'Mar':5})
df = pd.concat({'A':s3, 'B':s4 }, axis=1)
~~~

(Back to top)

Output

```
# CSV
df.to_csv('name.csv', encoding='utf-8')

# Excel

from pandas import ExcelWriter
writer = ExcelWriter('filename.xlsx')
df1.to_excel(writer,'Sheet1')
df2.to_excel(writer,'Sheet2')
writer.save()

# MySQL
import pymysql
from sqlalchemy import create_engine
e = create_engine('mysql+pymysql://' +
        'USER:PASSWORD@HOST/DATABASE')
df.to_sql('TABLE',e, if_exists='replace')

# Python object
d = df.to_dict() # to dictionary
str = df.to_string() # to string
m = df.as_matrix() # to numpy matrix
```

(Back to top)

Summary: Selecting using Index

Select columns

Using the DataFrame index to select columns

```
s = df['col_label']  # returns Series
df = df[['col_label']] # return DataFrame
df = df[['L1', 'L2']] # select with list
df = df[index] # select with index
df = df[s] #select with Series
```

Note: the difference in return type with the first two examples above based on argument type (scalar vs list).

```
df.iloc[:,:2] # Select the first 2 columns
feature_cols = ['TV','Radio','Newspaper']
x = data[feature_cols]
```

```
data[['TV','Radio','Newspaper']]

# by column labels
df.loc[:,['A','B']]  # syntax is: df.loc[rows_index, cols_index]
# conditional
df.filter(like='data')
df['preTestScore'].where(df['postTestScore'] > 50) # Find where a value exists ir
```

(Back to top)

Select rows

Using the DataFrame index to select rows

```
df = df['from':'inc_to']# label slice
df = df[3:7] # integer slice
df = df[df['col'] > 0.5]# Boolean Series
df = df.loc['label'] # single label
df = df.loc[container] # lab list/Series
df = df.loc['from':'to']# inclusive slice
df = df.loc[bs] # Boolean Series
df = df.iloc[0] # single integer
df = df.iloc[container] # int list/Series
df = df.iloc[0:5] # exclusive slice
df = df.ix[x] # loc then iloc

# by index
df.iloc[:2] # rows by row number
df.iloc[1:2] # Select the second and third row
df.iloc[2:] # Select every row after the third row
df.iloc[3:6,0:3] #

# by label
df.loc[:'Arizona'] # all rows by index label
df.ix[['Arizona', 'Texas']]   # .ix is the combination of both .loc and .iloc. Ir

# conditional
df.query('A > C')
df.query('A > 0')
df.query('A > 0 & A < 1')
df.query('A > B | A > C')
df[df['coverage'] > 50] # all rows where coverage is more than 50
df[(df['deaths'] > 500) | (df['deaths'] < 50)]
df[(df['score'] > 1) & (df['score'] < 5)]
df[~(df['regiment'] == 'Dragoons')] # Select all the regiments not named "Dragoor
df[df['age'].notnull() & df['sex'].notnull()]  # ignore the missing data points

# is in
df[df.name.isin(value_list)]   # value_list = ['Tina', 'Molly', 'Jason']
```

```
df[~df.name.isin(value_list)]

# partial matching
df2[df2.E.str.contains("tw|ou")]

# Regex
df['raw'].str.contains('....-..-..', regex=True)  # regex

# where cells are arrays
df[df['country'].map(lambda country: 'Syria' in country)]

# take random columns
df.take(np.random.permutation(len(df))[:2])
```

(Back to top)

Select a cross-section

Using the DataFrame index to select a cross-section

```
# r and c can be scalar, list, slice
df.loc[r, c] # label accessor (row, col)
df.iloc[r, c]# integer accessor
df.ix[r, c] # label access int fallback
df[c].iloc[r]# chained - also for .loc
```

(Back to top)

Select a cell

Using the DataFrame index to select a cell

```
# r and c must be label or integer
df.at[r, c] # fast scalar label accessor
df.iat[r, c] # fast scalar int accessor
df[c].iat[r] # chained - also for .at

df.ix['Arizona', 2]  # Select the third cell in the row named Arizona
df.ix[2, 'deaths']  # Select the third cell down in the column named deaths

df.ix['Yuma', 'coverage'] # view the value based on a row and column
```

(Back to top)

DataFrame indexing methods

```
v = df.get_value(r, c) # get by row, col
df = df.set_value(r,c,v)# set by row, col
df = df.xs(key, axis) # get cross-section
df = df.filter(items, like, regex, axis)
df = df.select(crit, axis)
```

Note: the indexing attributes (.loc, .iloc, .ix, .at .iat) can be used to get and set values in the DataFrame.

Note: the .loc, iloc and .ix indexing attributes can accept python slice objects. But .at and .iat do not.

Note: .loc can also accept Boolean Series arguments

Avoid: chaining in the form df[col_indexer][row_indexer]

Trap: label slices are inclusive, integer slices exclusive.

(Back to top)

Some index attributes and methods

```
# --- some Index attributes
b = idx.is_monotonic_decreasing
b = idx.is_monotonic_increasing
b = idx.has_duplicates
i = idx.nlevels # num of index levels
# --- some Index methods
idx = idx.astype(dtype)# change data type
b = idx.equals(o) # check for equality
idx = idx.union(o) # union of two indexes
i = idx.nunique() # number unique labels
label = idx.min() # minimum label
label = idx.max() # maximum label
```

Whole DataFrame

Content/Structure

Peek at the DataFrame contents/structure

```
df.info() # index & data types
dfh = df.head(i) # get first i rows
dft = df.tail(i) # get last i rows
dfs = df.describe() # summary stats cols
top_left_corner_df = df.iloc[:4, :4]
data.tail().transpose()
```

(Back to top)

Non-indexing attributes

DataFrame non-indexing attributes

```
dfT = df.T # transpose rows and cols
l = df.axes # list row and col indexes
(r, c) = df.axes # from above
s = df.dtypes # Series column data types
b = df.empty # True for empty DataFrame
i = df.ndim # number of axes (it is 2)
t = df.shape # (row-count, column-count)
i = df.size # row-count * column-count
a = df.values # get a numpy array for df
```

(Back to top)

Utilities

DataFrame utility methods

```
df = df.copy() # copy a DataFrame
df = df.rank() # rank each col (default)
df = df.sort(['sales'], ascending=[False])
df = df.sort_values(by=col)
df = df.sort_values(by=[col1, col2])
df = df.sort_index()
df = df.astype(dtype) # type conversion
```

(Back to top)

Iterations

DataFrame iteration methods

```
df.iteritems()# (col-index, Series) pairs
df.iterrows() # (row-index, Series) pairs
# example ... iterating over columns
for (name, series) in df.iteritems():
        print('Col name: ' + str(name))
        print('First value: ' +
                str(series.iat[0]) + '\n')
```

(Back to top)

Maths

Maths on the whole DataFrame (not a complete list)

```
df = df.abs() # absolute values
df = df.add(o) # add df, Series or value
s = df.count() # non NA/null values
df = df.cummax() # (cols default axis)
df = df.cummin() # (cols default axis)
df = df.cumsum() # (cols default axis)
df = df.diff() # 1st diff (col def axis)
df = df.div(o) # div by df, Series, value
df = df.dot(o) # matrix dot product
s = df.max() # max of axis (col def)
s = df.mean() # mean (col default axis)
s = df.median()# median (col default)
s = df.min() # min of axis (col def)
df = df.mul(o) # mul by df Series val
s = df.sum() # sum axis (cols default)
df = df.where(df > 0.5, other=np.nan)
```

Note: The methods that return a series default to working on columns.

(Back to top)

Select/filter

DataFrame select/filter rows/cols on label values

```
df = df.filter(items=['a', 'b']) # by col
df = df.filter(items=[5], axis=0) #by row
df = df.filter(like='x') # keep x in col
df = df.filter(regex='x') # regex in col
df = df.select(lambda x: not x%5)#5th rows
```

(Back to top)

Columns

Each DataFrame column is a pandas Series object

Info

```
# Index and labels
idx = df.columns # get col index
label = df.columns[0] # first col label
l = df.columns.tolist() # list col labels

# Data type conversions
st = df['col'].astype(str)# Series dtype
a = df['col'].values # numpy array
pl = df['col'].tolist() # python list

//Note: useful dtypes for Series conversion: int, float, str
//Trap: index lost in conversion from Series to array or list

# Common column-wide methods/attributes
value = df['col'].dtype # type of data
value = df['col'].size # col dimensions
value = df['col'].count()# non-NA count
value = df['col'].sum()
value = df['col'].prod()
value = df['col'].min()
value = df['col'].max()
value = df['col'].mean() # also median()
value = df['col'].cov(df['col2'])
s = df['col'].describe()
s = df['col'].value_counts()

# Find index label for min/max values in column
label = df['col1'].idxmin()
label = df['col1'].idxmax()

# Common column element-wise methods
s = df['col'].isnull()
s = df['col'].notnull() # not isnull()
s = df['col'].astype(float)
s = df['col'].abs()
s = df['col'].round(decimals=0)
s = df['col'].diff(periods=1)
s = df['col'].shift(periods=1)
s = df['col'].to_datetime()
s = df['col'].fillna(0) # replace NaN w 0
s = df['col'].cumsum()
s = df['col'].cumprod()
s = df['col'].pct_change(periods=4)
```

```
s = df['col'].rolling_sum(periods=4, window=4)

//Note: also rolling_min(), rolling_max(), and many more.

# Position of a column index label
//Get the integer position of a column index label
j = df.columns.get_loc('col_name')

# Column index values unique/monotonic
if df.columns.is_unique: pass # ...
b = df.columns.is_monotonic_increasing
b = df.columns.is_monotonic_decreasing
```

(Back to top)

Selecting

```
# Columns
s = df['colName'] # select col to Series
df = df[['colName']] # select col to df
df = df[['a','b']] # select 2 or more
df = df[['c','a','b']]# change col order
s = df[df.columns[0]] # select by number
df = df[df.columns[[0, 3, 4]] # by number
s = df.pop('c') # get col & drop from df

# Columns with Python attributes
s = df.a # same as s = df['a']
// cannot create new columns by attribute
df.existing_column = df.a / df.b
df['new_column'] = df.a / df.b

//Trap: column names must be valid identifiers.

# Selecting columns with .loc, .iloc and .ix
df = df.loc[:, 'col1':'col2'] # inclusive
df = df.iloc[:, 0:2] # exclusive

# Conditional selection
df.query('A > C')
df.query('A > 0')
df.query('A > 0 & A < 1')
df.query('A > B | A > C')
df[df['coverage'] > 50] # all rows where coverage is more than 50
df[(df['deaths'] > 500) | (df['deaths'] < 50)]
df[(df['score'] > 1) & (df['score'] < 5)]
df[~(df['regiment'] == 'Dragoons')] # Select all the regiments not named "Dragoor
df[df['age'].notnull() & df['sex'].notnull()]  # ignore the missing data points
```

```
# Is in
df[df.name.isin(value_list)]   # value_list = ['Tina', 'Molly', 'Jason']
df[~df.name.isin(value_list)]

# Partial matching
df2[df2.E.str.contains("tw|ou")]

# Regex
df['raw'].str.contains('....-..-..', regex=True)  # regex
```

Changing

```
# Rename column labels
df.rename(columns={'old1':'new1','old2':'new2'}, inplace=True)
//Note: can rename multiple columns at once.

# Adding new columns to a DataFrame
df['new_col'] = range(len(df))
df['new_col'] = np.repeat(np.nan,len(df))
df['random'] = np.random.rand(len(df))
df['index_as_col'] = df.index
df1[['b','c']] = df2[['e','f']]
df3 = df1.append(other=df2)
// __Trap__: When adding an indexed pandas object as a new
// column, only items from the new series that have a
// corresponding index in the DataFrame will be added.
// The receiving DataFrame is not extended to
// accommodate the new series. To merge, see below.

// __Trap__: when adding a python list or numpy array, the
// column will be added by integer position.


# Vectorised arithmetic on columns
df['proportion']=df['count']/df['total']
df['percent'] = df['proportion'] * 100.0

# Append a column of row sums to a DataFrame
df['Total'] = df.sum(axis=1)

# Apply numpy mathematical functions to columns
df['log_data'] = np.log(df['col1'])
//Note: Many more numpy mathematical functions.
// Hint: Prefer pandas math over numpy where you can.

# Set column values set based on criteria
df['b']=df['a'].where(df['a']>0,other=0)
df['d']=df['a'].where(df.b!=0,other=df.c)
//Note: where other can be a Series or a scalar
```

```
# Swapping
df[['B', 'A']] = df[['A', 'B']]

# Dropping
df = df.drop('col1', axis=1)
df.drop('col1', axis=1, inplace=True)
df = df.drop(['col1','col2'], axis=1)
s = df.pop('col') # drops from frame
del df['col'] # even classic python works
df.drop(df.columns[0], inplace=True)

# drop columns with column names where the first three letters of the column name
cols = [c for c in df.columns if c.lower()[:3] != 'pre']
df=df[cols]

# Multiply every column in DataFrame by Series
df = df.mul(s, axis=0) # on matched rows
//Note: also add, sub, div, etc.
```

(Back to top)

Rows

Info

```
# Get Position
a = np.where(df['col'] >= 2) #numpy array

# DataFrames have same row index  (Test if two DataFrames have same row index)

len(a)==len(b) and all(a.index==b.index)     # Get the integer position of a row
i = df.index.get_loc('row_label')
// Trap: index.get_loc() returns an integer for a unique match. If not a unique m

# Row index values are unique/monotonic
Test if the row index values are unique/monotonic
if df.index.is_unique: pass # ...
b = df.index.is_monotonic_increasing
b = df.index.is_monotonic_decreasing

# Get the row index and labels
idx = df.index      # get row index
label = df.index[0]    # 1st row label
lst = df.index.tolist()     # get as a list
```

Change the (row) index

```
df.index = idx     # new ad hoc index
df = df.set_index('A')    # col A new index
df = df.set_index(['A', 'B'])    # MultiIndex
df = df.reset_index()   # replace old w new

// note: old index stored as a col in df
df.index = range(len(df)) # set with list
df = df.reindex(index=range(len(df)))
df = df.set_index(keys=['r1','r2','etc'])
df.rename(index={'old':'new'},inplace=True)
```

(Back to top)

Selecting

```
# By column values
df = df[df['col2'] >= 0.0]
df = df[(df['col3']>=1.0) | (df['col1']<0.0)]
df = df[df['col'].isin([1,2,5,7,11])]
df = df[~df['col'].isin([1,2,5,7,11])]
df = df[df['col'].str.contains('hello')]
// Trap: bitwise "or", "and" “not; (ie. | & ~) co-opted to be Boolean operators c
// Trap: need parentheses around comparisons.

# Using isin over multiple columns

## fake up some data
data = {1:[1,2,3], 2:[1,4,9], 3:[1,8,27]}
df = DataFrame(data)
# multi-column isin
lf = {1:[1, 3], 3:[8, 27]} # look for
f = df[df[list(lf)].isin(lf).all(axis=1)]
Selecting rows using an index
idx = df[df['col'] >= 2].index
print(df.ix[idx])

# Slice of rows by integer position
[inclusive-from : exclusive-to [: step]]
default start is 0; default end is len(df)
df = df[:] # copy DataFrame
df = df[0:2] # rows 0 and 1
df = df[-1:] # the last row
df = df[2:3] # row 2 (the third row)
df = df[:-1] # all but the last row
df = df[::2] # every 2nd row (0 2 ..)
// Trap: a single integer without a colon is a column label for integer numbered
```

```
# Slice of rows by label/index
[inclusive-from : inclusive-to [ : step]]
df = df['a':'c'] # rows 'a' through 'c'
//Trap: doesn't work on integer labelled rows
```

Manipulating

```
# Adding rows
df = original_df.append(more_rows_in_df)
//Hint: convert to a DataFrame and then append. Both DataFrames should have same


# Append a row of column totals to a DataFrame
// Option 1: use dictionary comprehension
sums = {col: df[col].sum() for col in df}
sums_df = DataFrame(sums,index=['Total'])
df = df.append(sums_df)
// Option 2: All done with pandas
df = df.append(DataFrame(df.sum(),
                                columns=['Total']).T)

# Dropping rows (by name)
df = df.drop('row_label')
df = df.drop(['row1','row2']) # multi-row

# Drop duplicates in the row index
df['index'] = df.index # 1 create new col
df = df.drop_duplicates(cols='index',take_last=True)# 2 use new col
del df['index'] # 3 del the col
df.sort_index(inplace=True)# 4 tidy up
```

Iterating over DataFrame rows

```
for (index, row) in df.iterrows(): # pass
//Trap: row data type may be coerced.
```

Sorting

```
# Rows values
df = df.sort(df.columns[0], ascending=False)
df.sort(['col1', 'col2'], inplace=True)
```

```
# By row index
df.sort_index(inplace=True) # sort by row
df = df.sort_index(ascending=False)
```

(Back to top)

Random

Random selection of rows

```
import random as r
k = 20 # pick a number
selection = r.sample(range(len(df)), k)
df_sample = df.iloc[selection, :]
```

Note: this sample is not sorted

Slice off the first k elements of the array returned by permutation, where k is the desired subset size

```
df.take(np.random.permutation(len(df))[:3])
```

(Back to top)

Cells

Selecting

```
# By row and columnvalue = df.at['row', 'col']
value = df.loc['row', 'col']
value = df['col'].at['row'] # tricky
// Note: .at[] fastest label based scalar lookup

# By integer position
value = df.iat[9, 3] # [row, col]
value = df.iloc[0, 0] # [row, col]
value = df.iloc[len(df)-1,
len(df.columns)-1]

# Slice by labels
df = df.loc['row1':'row3', 'col1':'col3']
// Note: the "to" on this slice is inclusive.
```

```
# Slice by Integer Position
df = df.iloc[2:4, 2:4] # subset of the df
df = df.iloc[:5, :5] # top left corner
s = df.iloc[5, :] # returns row as Series
df = df.iloc[5:6, :] # returns row as row
// Note: exclusive "to" - same as python list slicing.

# By label and/or Index
// .ix for mixed label and integer position indexing
value = df.ix[5, 'col1']
df = df.ix[1:5, 'col1':'col3']
```

(Back to top)

Manipulating

Updating

```
# A cell
df.at['row', 'col'] = value
df.loc['row', 'col'] = value
df['col'].at['row'] = value # tricky

# Setting a cross-section by labels
df.loc['A':'C', 'col1':'col3'] = np.nan
df.loc[1:2,'col1':'col2']=np.zeros((2,2))
df.loc[1:2,'A':'C']=othr.loc[1:2,'A':'C']
// Remember: inclusive "to" in the slice

# Setting cell by integer position
df.iloc[0, 0] = value # [row, col]
df.iat[7, 8] = value

# Setting cell range by integer position
df.iloc[0:3, 0:5] = value
df.iloc[1:3, 1:4] = np.ones((2, 3))
df.iloc[1:3, 1:4] = np.zeros((2, 3))
df.iloc[1:3, 1:4] = np.array([[1, 1, 1],[2, 2, 2]])
// Remember: exclusive-to in the slice
```

(Back to top)

Joining/Combining DataFrames

Three ways to join two DataFrames: • merge (a database/SQL-like join operation) • concat (stack side by side or one on top of the other) • combine_first (splice the two together, choosing values from one over the other)

```
# Merge on indexes
df_new = pd.merge(left=df1, right=df2, how='outer', left_index=True, right_index=
// How: 'left', 'right', 'outer', 'inner'
// How: outer=union/all; inner=intersection

# Merge on columns
df_new = pd.merge(left=df1, right=df2, how='left', left_on='col1', right_on='col2

// Trap: When joining on columns, the indexes on the passed DataFrames are ignore
// Trap: many-to-many merges on a column can result in an explosion of associatec

# Join on indexes (another way of merging)
df_new = df1.join(other=df2, on='col1',how='outer')
df_new = df1.join(other=df2,on=['a','b'],how='outer')
// __Note__: DataFrame.join() joins on indexes by default.
// DataFrame.merge() joins on common columns by default.

# Simple concatenation is often the best
df = pd.concat([df1,df2],axis=0)#top/bottom
df = df1.append([df2, df3]) #top/bottom
df = pd.concat([df1,df2],axis=1)#left/right
// __Trap__: can end up with duplicate rows or cols
// __Note__: concat has an ignore_index parameter


# Combine_first
df = df1.combine_first(other=df2)
// multi-combine with python reduce()
df = reduce(lambda x, y:
x.combine_first(y),
[df1, df2, df3, df4, df5])
//Uses the non-null values from df1. The index of the combined DataFrame will be

# Merge/join multiple
dfs = [df0, df1, df2, dfN]
# Assuming they have some common column, like name in your example, I'd do the fc
dataset = reduce(lambda left,right: pd.merge(left,right,on='Country',how='left'),
[source](http://stackoverflow.com/questions/23668427/pandas-joining-multiple-data
```

(Back to top)

Grouping + Applying functions

The pandas "groupby" mechanism allows us to split the data into groups, apply a function to each group independently and then combine the results.

Grouping

```
# Grouping
gb = df.groupby('cat') # by one columns
gb = df.groupby(['c1','c2']) # by 2 cols
gb = df.groupby(level=0) # multi-index gb
gb = df.groupby(level=['a','b']) # mi gb
print(gb.groups)
// Note: groupby() returns a pandas groupby object
// Note: the groupby object attribute .groups contains a dictionary mapping of th
// Trap: NaN values in the group key are automatically dropped – there will never

# Iterating groups – usually not needed
for name, group in gb:
        print (name)
        print (group)

# Selecting a group
dfa = df.groupby('cat').get_group('a')
dfb = df.groupby('cat').get_group('b')
```

(Back to top)

Aggregating/transormating function

```
# apply to a column ...
s = df.groupby('cat')['col1'].sum()
s = df.groupby('cat')['col1'].agg(np.sum)
# apply to the every column in DataFrame
s = df.groupby('cat').agg(np.sum)
df_summary = df.groupby('cat').describe()
df_row_1s = df.groupby('cat').head(1)

// Note: aggregating functions reduce the dimension by one – they include: mean,

# Applying multiple aggregating functions
gb = df.groupby('cat')
// apply multiple functions to one column
dfx = gb['col2'].agg([np.sum, np.mean])
// apply to multiple fns to multiple cols
dfy = gb.agg({
        'cat': np.count_nonzero,
        'col1': [np.sum, np.mean, np.std],
        'col2': [np.min, np.max]
})
// Note: gb['col2'] above is shorthand for df.groupby('cat')['col2'], without the


# Transforming functions
// transform to group z-scores, which have a group mean of 0, and a std dev of 1.
zscore = lambda x: (x-x.mean())/x.std()
dfz = df.groupby('cat').transform(zscore)
// replace missing data with group mean
mean_r = lambda x: x.fillna(x.mean())
dfm = df.groupby('cat').transform(mean_r)
// Note: can apply multiple transforming functions in a manner similar to multipl

# Applying filtering functions
// Filtering functions allow you to make selections based on whether each group m
// select groups with more than 10 members
eleven = lambda x: (len(x['col1']) >= 11)
df11 = df.groupby('cat').filter(eleven)

# Group by a row index (non-hierarchical index)
df = df.set_index(keys='cat')
s = df.groupby(level=0)['col1'].sum()
dfg = df.groupby(level=0).sum()
```

(Back to top)

Dates+times (and their indexes)

```
# Dates and time - points and spans
With its focus on time-series data, pandas has a suite of tools for managing date
t = pd.Timestamp('2013-01-01')
t = pd.Timestamp('2013-01-01 21:15:06')
t = pd.Timestamp('2013-01-01 21:15:06.7')
p = pd.Period('2013-01-01', freq='M')
// Note: Timestamps should be in range 1678 and 2261 years. (Check Timestamp.max

# A Series of Timestamps or Periods
ts = ['2015-04-01 13:17:27', '2014-04-02 13:17:29']
// Series of Timestamps (good)
s = pd.to_datetime(pd.Series(ts))
// Series of Periods (often not so good)
s = pd.Series( [pd.Period(x, freq='M') for x in ts] )
s = pd.Series(pd.PeriodIndex(ts,freq='S'))
// Note: While Periods make a very useful index; they may be less useful in a Ser

# From non-standard strings to Timestamps
t = ['09:08:55.7654-JAN092002', '15:42:02.6589-FEB082016']
```

```
s = pd.Series(pd.to_datetime(t, format="%H:%M:%S.%f-%b%d%Y"))
// Also: %B = full month name; %m = numeric month; %y = year without century; anc

# Dates and time - stamps and spans as indexes
An index of Timestamps is a DatetimeIndex.
An index of Periods is a PeriodIndex.

date_strs = ['2014-01-01', '2014-04-01','2014-07-01', '2014-10-01']
dti = pd.DatetimeIndex(date_strs)
pid = pd.PeriodIndex(date_strs, freq='D')
pim = pd.PeriodIndex(date_strs, freq='M')
piq = pd.PeriodIndex(date_strs, freq='Q')
print (pid[1] - pid[0]) # 90 days
print (pim[1] - pim[0]) # 3 months
print (piq[1] - piq[0]) # 1 quarter
time_strs = ['2015-01-01 02:10:40.12345', '2015-01-01 02:10:50.67890']
pis = pd.PeriodIndex(time_strs, freq='U')
df.index = pd.period_range('2015-01', periods=len(df), freq='M')
dti = pd.to_datetime(['04-01-2012'], dayfirst=True) # Australian date format
pi = pd.period_range('1960-01-01','2015-12-31', freq='M')
// Hint: unless you are working in less than seconds, prefer PeriodIndex over Dat

# From DatetimeIndex to Python datetime objects
dti = pd.DatetimeIndex(pd.date_range(
start='1/1/2011', periods=4, freq='M'))
s = Series([1,2,3,4], index=dti)
na = dti.to_pydatetime() #numpy array
na = s.index.to_pydatetime() #numpy array
```

Timestamps -> Python dates or times

```
df['date'] = [x.date() for x in df['TS']]
df['time'] = [x.time() for x in df['TS']]
// Note: converts to datatime.date or datetime.time. But does not convert to date

# From DatetimeIndex to PeriodIndex and back
df = DataFrame(np.random.randn(20,3))
df.index = pd.date_range('2015-01-01', periods=len(df), freq='M')
dfp = df.to_period(freq='M')
dft = dfp.to_timestamp()
// Note: from period to timestamp defaults to the point in time at the start of t

# Working with a PeriodIndex
pi = pd.period_range('1960-01','2015-12',freq='M')
na = pi.values # numpy array of integers
lp = pi.tolist() # python list of Periods
sp = Series(pi)# pandas Series of Periods
ss = Series(pi).astype(str) # S of strs
ls = Series(pi).astype(str).tolist()
```

```
# Get a range of Timestamps
dr = pd.date_range('2013-01-01', '2013-12-31', freq='D')

# Error handling with dates
// 1st example returns string not Timestamp
t = pd.to_datetime('2014-02-30')
// 2nd example returns NaT (not a time)
t = pd.to_datetime('2014-02-30',coerce=True)
// NaT like NaN tests True for isnull()
b = pd.isnull(t) # --> True

# The tail of a time-series DataFrame
df = df.last("5M") # the last five months


# Upsampling and downsampling
// upsample from quarterly to monthly
pi = pd.period_range('1960Q1', periods=220, freq='Q')
df = DataFrame(np.random.rand(len(pi),5), index=pi)
dfm = df.resample('M', convention='end')
// use ffill or bfill to fill with values
// downsample from monthly to quarterly
dfq = dfm.resample('Q', how='sum')


# Time zones
t = ['2015-06-30 00:00:00','2015-12-31 00:00:00']
dti = pd.to_datetime(t).tz_localize('Australia/Canberra')
dti = dti.tz_convert('UTC')
ts = pd.Timestamp('now', tz='Europe/London')
# get a list of all time zones
import pyzt
for tz in pytz.all_timezones:
        print tz

//Note: by default, Timestamps are created without time zone information.

# Row selection with a time-series index
// start with the play data above
idx = pd.period_range('2015-01', periods=len(df), freq='M')
df.index = idx
february_selector = (df.index.month == 2)
february_data = df[february_selector]
q1_data = df[(df.index.month >= 1) & (df.index.month <= 3)]
mayornov_data = df[(df.index.month == 5) | (df.index.month == 11)]
totals = df.groupby(df.index.year).sum()
// Also: year, month, day [of month], hour, minute, second, dayofweek [Mon=0 .. S

# The Series.dt accessor attribute
// DataFrame columns that contain datetime-like objects can be manipulated with t

t = ['2012-04-14 04:06:56.307000', '2011-05-14 06:14:24.457000', '2010-06-14 08:2
```

```
// a Series of time stamps
s = pd.Series(pd.to_datetime(t))
print(s.dtype) # datetime64[ns]
print(s.dt.second) # 56, 24, 7
print(s.dt.month) # 4, 5, 6
// a Series of time periods
s = pd.Series(pd.PeriodIndex(t,freq='Q'))
print(s.dtype) # datetime64[ns]
print(s.dt.quarter) # 2, 2, 2
print(s.dt.year) # 2012, 2011, 2010
```

(Back to top)

Missing / non-finite data

Pandas uses the not-a-number construct (np.nan and float('nan')) to indicate missing data. The Python None can arise in data as well. It is also treated as missing data; as is the pandas not-a-time construct (pandas.NaT).

```
# Missing data in a Series
s = Series( [8,None,float('nan'),np.nan])
//[8, NaN, NaN, NaN]
s.isnull() #[False, True, True, True]
s.notnull()#[True, False, False, False]
s.fillna(0)#[8, 0, 0, 0]

# Missing data in a DataFrame
df = df.dropna() # drop all rows with NaN
df = df.dropna(axis=1) # same for cols
df=df.dropna(how='all') #drop all NaN row
df=df.dropna(thresh=2) # drop 2+ NaN in r
// only drop row if NaN in a specified col
df = df.dropna(df['col'].notnull())

# Recoding/Replacing missing data
df.fillna(0, inplace=True) # np.nan -> 0
s = df['col'].fillna(0) # np.nan -> 0
df = df.replace(r'\s+', np.nan,regex=True) # white space -> np.nan

# Non-finite numbers
// With floating point numbers, pandas provides for positive and negative infinit

s = Series([float('inf'), float('-inf'),np.inf, -np.inf])
// Pandas treats integer comparisons with plus or minus infinity as expected.
```

```
# Testing for finite numbers using the data from the previous example
b = np.isfinite(s)
```

(Back to top)

Working with Categorical Data

```
# Categorical data
The pandas Series has an R factors-like data type for encoding categorical data.
s = Series(['a','b','a','c','b','d','a'],
dtype='category')
df['B'] = df['A'].astype('category')
// Note: the key here is to specify the "category" data type.
// Note: categories will be ordered on creation if they are sortable. This can be

# Convert back to the original data type
s = Series(['a','b','a','c','b','d','a'], dtype='category')
s = s.astype('string')

## Ordering, reordering and sorting
s = Series(list('abc'), dtype='category')
print (s.cat.ordered)
s=s.cat.reorder_categories(['b','c','a'])
s = s.sort()
s.cat.ordered = False
// Trap: category must be ordered for it to be sorted

# Renaming categories
s = Series(list('abc'), dtype='category')
s.cat.categories = [1, 2, 3] # in place
s = s.cat.rename_categories([4,5,6])
// using a comprehension ...
s.cat.categories = ['Group ' + str(i)
        for i in s.cat.categories]
// Trap: categories must be uniquely named

# Adding new categories
s = s.cat.add_categories([4])

# Removing categories
s = s.cat.remove_categories([4])
s.cat.remove_unused_categories() #inplace
```

(Back to top)