Day 12

Python for Data Science

Pandas

Preliminaries

Always start by importing these Python modules

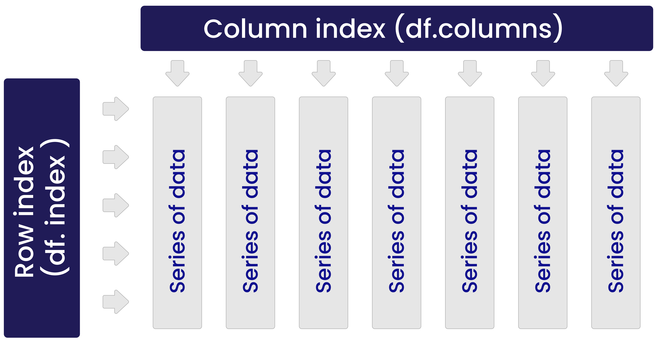
```
import pandas as pd
from pandas import DataFrame, Series
```

In the code examples

- s to represent a pandas Series object;
- df to represent a pandas DataFrame object;
- idx to represent a pandas Index object.
- Also: t – tuple, l – list, b – Boolean, i – integer,a – numpy array, st – string, d – dictionary, etc.

The conceptual model

DataFrame object: is a two-dimensional table of data with column and row indexes (something like a spread sheet). The columns are made up of Series objects.



A DataFrame has two Indexes:

* Typically, the column index (df.columns) is a list of strings (variable names) or (less commonly) integers

* Typically, the row index (df.index) might be:

  Integers - for case or row numbers; Strings – for case names; or DatetimeIndex or PeriodIndex – for time series

Series object: an ordered, one-dimensional array of data with an index. All the data in a Series is of the same data type. Series arithmetic is vectorised after first aligning the Series index for each of the operands.

```
s1 = Series(range(0,4)) # -> 0, 1, 2, 3
s2 = Series(range(1,5)) # -> 1, 2, 3, 4
s3 = s1 + s2 # -> 1, 3, 5, 7
```

Get your data into a DataFrame

Instantiate an empty DataFrame

```
df = DataFrame()
```

Load a DataFrame from a CSV file

```
df = pd.read_csv('file.csv') # often works
df = pd.read_csv('file.csv', header=0,
 index_col=0, quotechar='"', sep=':',
 na_values = ['na', '-', '.', ''])
```

Note: refer to pandas docs for all arguments

Load DataFrames from a Microsoft Excel file

```
# Each Excel sheet in a Python dictionary
workbook = pd.ExcelFile('file.xlsx')
```

Load a DataFrame from a MySQL database

```
import pymysql
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://'
 +'USER:PASSWORD@HOST/DATABASE')
df = pd.read_sql_table('table', engine)
```

Get a DataFrame from a Python dictionary

```
# default --- assume data is in columns
df = DataFrame({
 'col0' : [1.0, 2.0, 3.0, 4.0],
 'col1' : [100, 200, 300, 400]
 })
df
```

	col0	col1
0	1.0	100
1	2.0	200
2	3.0	300
3	4.0	400

Saving a DataFrame

Saving a DataFrame to a CSV file

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

Saving a DataFrame to 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')
```

Saving to Python objects

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

Working with the whole DataFrame

Peek at the DataFrame contents/structure

```
df.info() # index & data types
dfh = df.head(2) # get first i rows
dft = df.tail(2) # get last i rows
dfs = df.describe() # summary stats cols
top_left_corner_df = df.iloc[:4, :4]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 2 columns):
 #    Column    Non-Null Count    Dtype
---   ------    --------------    -----
 0    col0      4 non-null        float64
 1    col1      4 non-null        int64
dtypes: float64(1), int64(1)
memory usage: 192.0 bytes
```

DataFrame non-indexing attributes

```
df = df.T # transpose rows and cols
l = df.axes # list row and col indexes
(r_idx, c_idx) = 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
```

DataFrame utility methods

```
df = pd.DataFrame({'a':[7,9,8,10],'b':[4,3,2,1]})
df
```

	a	b
0	7	4
1	9	3
2	8	2
3	10	1

```
df = df.copy() # copy a DataFrame
df = df.rank() # rank each col (default)
df = df.sort_values(by='a')
df = df.sort_values(by=['a', 'b'])
df = df.sort_index()
df = df.astype(dtype) # type conversion
```

DataFrame iteration methods

```
df
```

	a	b
0	7	4
1	9	3
2	8	2
3	10	1

```
df.iteritems() # (col-index, Series) pairs
df.iterrows() # (row-index, Series) pairs
# example ... iterating over columns ...
for (name, series) in df.iteritems():
  print('\nCol name: ' + str(name))
  print('1st value: ' + str(series.iat[0]))
```

Maths on the whole DataFrame (not a complete list)

```
df = pd.DataFrame({'a':[7,9,8,10],'b':[4,3,2,1]})
df
```

	a	b
0	7	4
1	9	3
2	8	2
3	10	1

```
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
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: methods returning a series default to work on cols

Select/filter rows/cols based on index label values

```
df = pd.DataFrame({'ac_mahi':[7,9,8,10],'ab':[4,3,2,1]})
df
```

	ac_mahi	ab
0	7	4
1	9	3
2	8	2
3	10	1

```
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
```

Working with Columns

Get column index and labels

```
idx = df.columns # get col index
label = df.columns[0] # first col label
l = df.columns.tolist() # list of col labels
a = df.columns.values # array of col labels
```

Change column labels

```
df = df.rename(columns={'old':'new','a':'1'})
df.columns = ['new1', 'new2', 'new3'] # etc.
```

Selecting columns

```
s = df['colName'] # select col to Series
df = df[['colName']] # select col to df
df = df[['a','b']] # select 2-plus cols
df = df[['c','a','b']] # change col order
s = df[df.columns[0]] # select by number
df = df[df.columns[[0, 3, 4]]] # by numbers
df = [df.columns[:-1]] # all but last col
s = df.pop('c') # get & drop from df
```

Selecting 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.

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)
```

Swap column contents

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

Dropping (deleting) columns (mostly by label)

```
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 = df.drop(df.columns[0], axis=1)#first
df = df.drop(df.columns[-1:],axis=1)#last
```

Vectorised arithmetic on columns

```
df['proportion']=df['count']/df['total']
df['percent'] = df['proportion'] * 100.0
```

Apply numpy mathematical functions to columns

```
df['log_data'] = np.log(df['col1'])
```

Note: many many more numpy math 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

Data type conversions

```
st = df['col'].astype(str)# Series dtype
a = df['col'].values # numpy array
l = 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(window=4,
 min_periods=4, center=False).sum()
```

Append a column of row sums to a DataFrame

```
df['Total'] = df.sum(axis=1)
```

Note: also means, mins, maxs, etc.

Multiply every column in DataFrame by Series

```
df = df.mul(s, axis=0) # on matched rows
```

Note: also add, sub, div, etc.

Selecting columns with .loc, .iloc and .ix

```
df = df.loc[:, 'col1':'col2'] # inclusive
df = df.iloc[:, 0:2] # exclusive
```

Get the integer position of a column index label

```
i = df.columns.get_loc('col_name')
```

Working with rows

Get the row index and labels

```
idx = df.index # get row index
label = df.index[0] # first row label
label = df.index[-1] # last row label
l = df.index.tolist() # get as a list
a = df.index.values # get as an array
```

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)
```

Dropping rows (by name)

```
df = df.drop('row_label')
df = df.drop(['row1','row2']) # multi-row
```

Boolean row selection by values in a column

```
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')]
```

Selecting rows 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])
```

Select a slice of rows by integer position [inclusive-from : exclusive-to [: step]] start is 0; end is len(df)

```
df = df[:] # copy entire DataFrame
df = df[0:2] # rows 0 and 1
df = df[2:3] # row 2 (the third row)
df = df[-1:] # the last 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 columns.

Select a slice of rows by label/index [inclusive-from : inclusive–to [: step]]

```
df = df['a':'c'] # rows 'a' through 'c'
```

Trap: cannot work for integer labelled rows – see previous code snippet on integer position slicing.

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)
```

Iterating over DataFrame rows

```
for (index, row) in df.iterrows(): # pass
```

Trap: row data type may be coerced.

Sorting DataFrame rows values

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

Sort DataFrame by its row index

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

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, :] # get copy
```

Note: this randomly selected sample is not sorted

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
```

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 or col index label

```
i = df.index.get_loc('row_label')
```

Trap: index.get_loc() returns an integer for a unique match. If not a unique match, may return a slice/mask.

Get integer position of rows that meet condition

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

Working with cells

Selecting a cell by row and column labels

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

Note: .at[ ] fastest label based scalar lookup

Setting a cell by row and column labels

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

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

```
# 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
```

```
# Selecting a cell 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]
```

```
# Selecting a range of cells by int position
df = df.iloc[2:4, 2:4] # subset of the df
df = df.iloc[:5, :5] # top left corner
s = df.iloc[5, :] # return row as Series
df = df.iloc[5:6, :] # returns row as row
# Note: exclusive "to" – same as python list slicing.
```

```
# 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
```

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

Summary: selecting using the DataFrame index

```
# Using the DataFrame index to select columns
s = df['col_label'] # returns Series
df = df[['col_label']] # returns DataFrame
df = df[['L1', 'L2']] # select cols with list
df = df[index] # select cols with an index
df = df[s] # select with col label Series
# Note: scalar returns Series; list &c returns a DataFrame.
```

```
# 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
# Trap: Boolean Series gets rows, label Series gets cols.
```

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
```

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
```

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.