Preliminaries

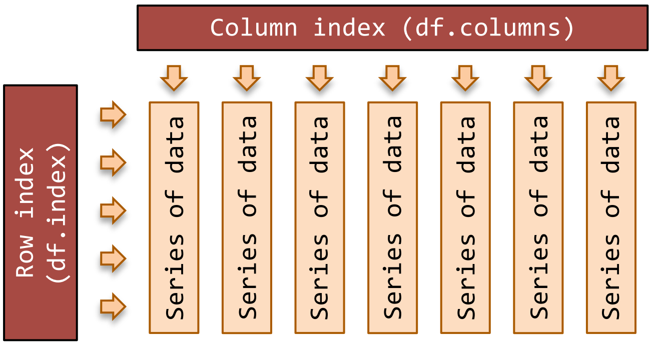
Start by importing these Python modules

```
import pandas as pd                       # required
from pandas import DataFrame, Series # useful
import numpy as np                        # required
```

Conventions (in these notes)

Name	Description
df	A pandas DataFrame object
s	A pandas Series object
index	A pandas index object

Overview: the conceptual DataFrame model



Series object: an ordered array of data with an index. Series arithmetic is vectorised after first aligning the Series (row) index of 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
s4 = Series(['a','b'])*3 # --> 'aaa', 'bbb'
```

DataFrame object: is a table of data with column and row indexes. The columns are made up of pandas Series objects.

Get your data into a DataFrame

Play data (useful for testing)

```
df = DataFrame(np.random.randn(26,5),
    columns=['col'+str(i) for i in range(5)],
    index=list("ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
df['cat'] = list('aaaabbbccddef' * 2)
```

Get a DataFrame from a CSV file

```
df = pd.read_csv('file.csv')
```

Get a DataFrame from a Microsoft Excel file

```
# put each Excel workbook in a dictionary
workbook = pd.ExcelFile('file.xls')
dictionary = {}
for name in workbook.sheet_names:
    df = workbook.parse(name)
    dictionary[name] = df
```

Get a DataFrame from a Python dictionary

```
d = { 'col1' : [1.0, 2.0, 3.0, 4.0],
      'col3' : [100, 200, 300, 400]  }
df = pd. DataFrame.from_dict(d)
```

Working with indexes

The Index object

Both the column index and the row index are a pandas Index object. Broadly speaking, this will be one of a number of pandas data types:

1. Int64Index – integer indexes,
2. Float64Index – float indexes
3. DatetimeIndex – a timestamp/point in time
4. PeriodIndex – a timespan/period of time. This is the preferred index for time series data where the period is longer than ms.
5. Index – of any other hashable Python object (often a string, can be anything hashable).
6. MultiIndex – for Hierarchical indexing (not covered in these notes)

Hint: Typically, the column index is a list of strings (the observed variable names) or integers. As a guide, the row index might be

- Integers - for case or row numbers
- Strings – for case names
- DatetimeIndex or PeriodIndex – for time series data (more on these indexes below)

Get column labels

```
index = df.columns      # get column labels
label = df.column[0]    # the 1st column label
```

Change column labels

```
df.columns = ['a','b'] # set all column names
df.rename(columns={'old':'new'},inplace=True)
# more than one col can be changed via dict
df = df.rename(columns = {'a':'a1','b':'b2'})
```

Get the (row) index

```
index = df.index                # get index
label = df.index[0]             # 1st row label
index_list = df.index.tolist() # get as list
```

Change the (row) index

```
df.index = index
df.index = range(len(df))      # set with list
df = df.reset_index()
df = df.reindex(index=range(len(df)))
df = df.set_index(keys='col1') # set with col
df = df.set_index(keys=['col1','col2','etc'])
df.rename(index={'old':'new'}, inplace=True)
```

The default (row) index

An integer index numbered (0, 1, 2 ... n-1)

Working with the whole DataFrame

Peek at the DataFrame

```
summary_df = df.describe()
head_df = df.head();    tail_df = df.tail()
top_left_corner_df = df.iloc[:5, :5]
```

Transpose rows and columns

```
df = df.T
```

Joining/Combining DataFrames

Merge on columns

```
df_new = pd.merge(left=df1, right=df2,
how='left', left_on='col1', right_on='col2')
```

How: 'left', 'right', 'outer', 'inner'
How: outer=union/all; inner=intersection

Merge on indexes

```
df_new = pd.merge(left=df1, right=df2,
  how='inner', left_index=True,
  right_index=True)
```

Join on indexes (another way of merging)

```
df_new = df1.join(other=df2, how='left')
```

Working with columns (axis=1)

Selecting columns (by label or num)

```
s = df['colName']      # select column by name
df = df[['a','b']]     # select 2 or more cols
df = df[['b','a','c']]# change col order
s = df[df.columns[0]] # select column by num
# cols numbered from 0 to len(df.columns)-1
```

Select a slice of columns by label

```
df = df.loc[:, 'col1':'col2'] #inclusive "to"
```

Can also use df.ix[:, 'col1':'col2']

Select a slice of columns by integer position

```
df = df.iloc[:, 0:2]          #exclusive "to"
```

Can also use df.ix[:, 0:2], but ix will do an inclusive "to" with integer labelled columns.

Dropping columns (by label)

```
df = df.drop('col1', axis=1)
df = df.drop(df.columns[0], axis=1)
df = df.drop(['col1','col2'], axis=1) # multi
s = df.pop('col') # get col; drop from frame
```

Adding new columns

```
df['new_col'] = range(len(df))
df['index_as_column'] = df.index
df['row_sum'] = df.sum(axis=1)
df1[['b','c']] = df2[['e','f']]    # multi add
df3 = df1.append(other=df2)        # multi add
```

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'])
df['rounded'] = np.round(df['col2'], 2)
df['random'] = np.random.rand(len(df))
```

Vectorised if/else on columns (using where)

```
df['col'] = df['col'].where(cond, other=nan)
If condition is true return from the Series;
otherwise from the other (scalar or Series)
l = range(10); s1 = Series(l); # 0 1 2 .. 9
l.reverse(); s2 = Series(l)    # 9 8 7 .. 0
s = s1.where(s1>=5, s2) # 9 8 7 6 5 5 6 7 8 9
```

Column access by Python attributes

```
s = df.a                     # same as s = df['a']
df.old = df.b / df.c
df['new'] = df.b / df.c
```

Trap: column names must be valid identifiers.

Iterating over the Dataframe cols

```
for (index, col) in df.iteritems():
```

Where index is the column label and col is a pandas Series that contains the column data

Common column element-wise methods

```
s = df['col'].to_datetime()
s = df['col1'].isnull()
s = df['col1'].notnull() # not isnull()
s = df['col1'].round(decimals=0)
s = df['col1'].diff(periods=1)
s = df['col1'].shift(periods=1)
```

Common column-wide methods/attributes

```
type =  df['col1'].dtype
value = df['col1'].size       # col dimensions
value = df['col1'].count()    # non-NA count
value = df['col1'].sum()
value = df['col1'].prod()
value = df['col1'].min()
value = df['col1'].max()
value = df['col1'].mean()
value = df['col1'].median()
s =     df['col1'].describe()
```

Group by a column

```
s = df.groupby('cat')['col1'].sum()
dfg = df.groupby('cat').sum()
```

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()
```

Working with rows (axis=0)

Adding rows

```
df = original_df.append(more_rows_in_df)
```

For a new row in a python dictionary or list, convert it to a DataFrame and then append.

Dropping rows (by name)

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

Select a slice of rows by integer position

[inclusive-from : exclusive-to]
[inclusive-from : exclusive-to : step]
default start is 0; default end is len(df)

```
copy_df = df[:]         # copy DataFrame
rows_df = df[0:2]       # rows 0 and 1
rows_df = df[-1:]       # the last row
rows_df = df[2:3]       # row 2 (the third row)
rows_df = df[:-1]       # all but the last row
rows_df = df[::2]       # every 2nd row (0 2 ..)
```

Trap: a single integer without a colon is an index and not a slice. Furthermore it will return a column and not a row.

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

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

Select rows by value in a column
(row selection from a Boolean Series)

```
rows_df = df[df['col2'] >= 0.0]
df = df[(df['col3']>=1.0) | (df['col1']<0.0)]
```

Trap: bitwise "or" and "and" co-opted to be Boolean operators on a Series of Boolean --> also note parentheses around comparisons.

Iterating over DataFrame rows

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

Trap: row data type may be coerced.

Sorting DataFrame rows by column values

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

Working with rows and columns

Select cell by integer position (using .iloc)

```
value = df.iloc[0, 0]            # [row, col]
value = df.iloc[9, 3]            # [row, col]
value = df.iloc[len(df), len(df.columns)]
```

Slicing by integer position (using .iloc)

```
top_left_corner_df = df.iloc[:5, :5]
# a size safe top-left-corner print follows:
print (df.iloc[:min(5, len(df)),
               :min(5, len(df.columns))])
s = df.iloc[0, :5]   # row specific/col slice
s = df.iloc[:5, 0]   # row slice/col specific
```

Note: exclusive "to" – same as list slicing.

Selecting and slicing on labels (with .loc)

```
df = df.loc['row1':'row3', 'col1':'col3']
```

Note: the "to" on this slice is inclusive.

Hybrid selecting and slicing (with .ix)

```
df = df.ix[0:5, 'col1':'col3']
```

Trap: integer indexes treated as labels

Views and copies
From the manual: The rules about when a view on the data is returned are entirely dependent on NumPy. Whenever an array of labels or a boolean vector are involved in the indexing operation, the result will be a copy. With single label / scalar indexing and slicing, e.g. df.ix[3:6] or df.ix[:, 'A'], a view will be returned.

Working with dates, times and their indexes

Dates and time – points and spans
With its focus on time-series data, pandas provides a suite of tools for managing dates and time: either as a point in time (a Timestamp) or as a span of time (a Period).

```
timestamp = pd.Timestamp('2013-01-01')
period = pd.Period('2013-01-01', freq='M')
```

Dates and time – stamps and spans as indexes
An index of Timestamps is a DatetimeIndex; and an index of Periods is a PeriodIndex. These can be constructed as follows:

```
date_strs = ('2013-10-01', '2013-11-01',
             '2013-12-01', '2014-01-01')
tstamp = pd.to_datetime(pd.Series(date_strs))
dt_idx = pd.DatetimeIndex(tstamp, freq='MS')
prd_idx = pd.PeriodIndex(tstamp, freq='M')
spi = Series([1,2,3,4], index=prd_idx)
sdi = Series([1,2,3,4], index=dt_idx)
# Also: index changed through its attribute
spi.index = dt_idx # change to time stamps
spi.index = range(len(spi)) # to integers
```

From DatetimeIndex and PeriodIndex and back

```
spi = sdi.to_period(freq='M')# to PeriodIndex
sdi = spi.to_timestamp()   # to DatetimeIndex
```

Note: from period to timestamp defaults to the point in time at the start of the period.

More examples on working with dates/times

```
d = pd.to_datetime(['04-01-2012'],
      dayfirst=True) # Australian date format
t = pd.to_datetime(['2013-04-01 15:14:13.1'])
```

DatetimeIndex can be converted to an array of Python native datetime.datetime objects using the to_pydatetime() method.

Error handling with dates

```
# first example returns string not Timestamp
s = pd.to_datetime('2014-02-30')
# second example returns NaT (not a time)
n = pd.to_datetime('2014-02-30', coerce=True)
# NaT is like NaN ... tests True for isnull()
b = pd.isnull(n) # --> True
```

Creating date/period indexes from scratch

```
dt_idx = pd.DatetimeIndex(pd.date_range(
    start='1/1/2011',  periods=12, freq='M'))
p_idx = pd.period_range('1960-01-01',
      '2010-12-31', freq='M')
```

Frequency constants (not a complete list)

Name	Description
U	Microsecond
L	Millisecond
S	Second
T	Minute
H	Hour
D	Calendar day
B	Business day
W-{MON, TUE, …}	Week ending on …
MS	Calendar start of month
M	Calendar end of month
QS-{JAN, FEB, …}	Quarter start with year ending (QS – December)
Q-{JAN, FEB, …}	Quarter end with year ending (Q – December)
AS-{JAN, FEB, …}	Year start (AS - December)
A-{JAN, FEB, …}	Year end (A - December)

Row selection with a time-series index

```
# play data ... start with play data above
idx = pd.period_range('2013-01',
       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)] # note: & not "and"

mayornov_data = df[(df.index.month == 5) |
   (df.index.month == 11)] # note: | not "or"

annual_tot = df.groupby(df.index.year).sum()
```

Also: year, month, day [of month], hour, minute, second, dayofweek [Mon=0 .. Sun=6], weekofmonth, weekofyear [numbered from 1], week starts on Monday], dayofyear [from 1], …
Note: this method works with both Series and DataFrame objects.

The tail of a time-series DataFrame

```
df = df.last("5M")     # the last five months
```

Working with strings

Working with strings

```
# assume that df['col'] is series of strings
s = df['col'].str.lower()
s = df['col'].str.upper()
s = df['col'].str.len()
df['col'] += 'suffix' # add text to each row
df['col'] *= 2        # repeat text
s = df['col1'] + df['col2'] # concatenate
```

Most python string functions are replicated in the pandas DataFrame and Series objects.

Regular expressions

```
s = df['col'].str.contains('text')
s = df['col'].str.startswith('text')
s = df['col'].str.endswith('text')
s = df['col'].str.replace('old', 'new')
```

Working with missing and non-finite data

Working with missing 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 (pd.NaT) construct.

Missing data in a Series

```
s = pd.Series([8,None,float('nan'),np.nan])
# -->         [8,     NaN,  NaN,     NaN]
s.isnull() # --> [False, True, True,  True]
s.notnull()# --> [True, False, False, False]
```

Missing data in a DataFrame

```
df = df.dropna()  # drop all rows with a NaN
df = df.dropna(axis=1) # as above for cols
df=df.dropna(how='all') # only if all in row
df=df.dropna(thresh=2) # at least 2 NaN in r
# only drop row if NaN in a specified 'col'
df = df.dropna(df['col'].notnull())
```

Non-finite numbers

With floating point numbers, pandas provides for positive and negative infinity.

```
s = Series([float('inf'), float('-inf'),
     np.inf, -np.inf]) # inf, -inf, inf, -inf
```

Pandas treats integer comparisons with plus or minus infinity as expected.

Testing for finite numbers

(using the data from the previous example)

```
np.isfinite(s) # False, False, False, False
```

Working with Categorical Data

Categorical data

The pandas Series has an R factors-like data type for encoding categorical data into integers.

```
c = pd.Categorical.from_array(list)
c.levels   # --> the coding frame
c.labels   # --> the encoded integer array
c.describe # --> the values and levels
```

Indexing categorical data

The categorical data can be indexed in a manner conceptually similar to that for Series.iloc[] above:

```
listy = ['a', 'b', 'a', 'b', 'b', 'c']
c = pd.Categorical.from_array(listy)
c.levels       # --> ['a', 'b', 'c']
c.labels       # --> [0, 1, 0, 1, 1, 2]
x = c[1]       # --> 'b'
x = c[[0,1]]   # --> ['a', 'b']
x = c[0:2]     # --> ['a', 'b']
```

Categorical into DataFrame

You can put a column of encoded Categorical data in the DataFrame, but in the process the factor information will be lost; so you will need to hold this factor information outside of the DataFrame.

```
factor = pd.Categorical.from_array(df['cat'])
df['labels'] = factor.labels # integers only
df['cat2'] = factor # converts back to string
```