Proteomics Informatics (Spring 2014): Week 7
Introductory Pandas hands-on

1. > from pandas import Series, Dataframe
   > import pandas as pd

2. Series
   a. Creation
      > x = Series([10, 20, 30, 40, 50])
      > x?? (to see the code), x? (to see the documentation)
      > x.<tab> (to see the instance methods/attributes)
         a. x.mean(),
         b. x.sort?, x.sort(ascending=False)
      > x.values
      > x.index
      > x = Series([10, 20, 30, 40, 50], index=list('abcde'))
         → you can also assign/re-assign an index after the fact
            > x.index = ['one', 'two', 'three', 'four', 'five']
               → index must be of the same size
   b. Access (Similar to numpy arrays)
      → By index position or index value
         > x[0], x[1] etc.
         > x['a'], x['b']
            → This gives it a dictionary-like interface… So any python function/operation that expects a dictionary argument can be passed a Series object (concept of **delegation**… each object is responsible to implement the interface expected of it…)
            → **Merger of a list and a dictionary**
            → it even has a "keys()" function, just like a dict
               > x.keys()
               > x.keys??
            → Because of this interface, a Series object can also be created by directly passing a dictionary
               > x = Series({'a': 10, 'e': 50, 'c': 30, 'b': 20, 'd': 40}) ## keys will be in sorted order
               > x = Series({'a': 10, 'e': 50, 'c': 30, 'b': 20, 'd': 40}, index=list('aecbd')) ## to preserve key/index order
               > x = Series({'a': 10, 'e': 50, 'c': 30, 'b': 20, 'd': 40}, index=list('aecbdp')) ## extra indices will be assigned NaN, which means missing value or NA in pandas
            → **Missing values can be checked in Pandas using notnull() and isnull() functions…**
               > pd.isnull(x), pd.notnull() ## in pandas
               > x.isnull(), x.notnull() ## as instance method
               → SHOW THIS AFTER ALIGNMENT STEP in point c
      → Slicing
         > x[1:3]
         > x['b': 'd'] ## slicing with index values/labels uses inclusive end-points unlike above, or python list index slicing
         > x['b': 'd'] = 999 ##Assignment operation

→ Boolean indexing:

> x >= 30

> x[x>=30]   ## **Note that** during all these operations, indices are preserved

c. One major utility of having the index objects is automatic alignment across different Series or Dataframe objects

> x = Series({'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50})

> y = Series({'a': 10, 'b': 100})

> x+y   ## well defined

More on data alignment later

→ Index labels can also be used to drop specific entries from the Series object

> x.drop('a')   ## returns a new Series object with dropped entry

3. Dataframe

a. Creation (**numerous ways… Like a list of lists… But not required to discuss… since mostly we'll be building these directly from files… for now, go with this view!!**):

> data = {'a': range(1,11),'b': range(21,31),'c': range(31,41)}   ## From dict of equal-length lists

> dataF = DataFrame(data)

→ has both row and column indexes

→ (row)index automatically populated (can be explicitly provided as additional argument to DataFrame); columns are placed in sorted order (can be altered by explicitly providing columns as argument)

> dataF = DataFrame(data, index=list('pqrstuvwxy'), columns=list('cba'))

> dataF.index = list('pqrstuvwxy')   ## providing the index after the fact

> dataF.values (**List of lists representation**), dataF.columns, dataF.index

b. Access

i. Individual columns as Series objects

> dataF['a']   ## dict-like notation

> dataF.a   ## object attribute dot-notation

→ The series will have the same index as the parent dataframe

ii. Multiple columns (as a dataframe)

> dataF[['a', 'b']]   ## use list of column indices

iii. Special cases (Apply a filter on rows and/or values):

> dataF[:2]   ## Select the first 2 rows (slicing)

> dataF[dataF.a > 5]   ## Boolean indexing on specific rows

> dataF[dataF > 25]   ## Boolean indexing on the whole dataFrame

iv. Indexing on the rows:

→ <DataFrame>**.ix** field… For ex.

> dataF.ix[2] or dataF.ix['r']

→ Slicing

> dataF.ix[2:5] or dataF.ix['r': 't']

→ Selecting rows and columns:

> dataF.ix[['p', 'q', 'r'], ['a', 'b']]

> dataF.ix[2:5, ['a', 'b']]

> dataF.ix[:, ['a', 'b']]   ## all rows and 2 columns

> dataF.ix[dataF.a>5, :2]

c. Manipulation
   i. Each of these methods of access can be used to do bulk-assignment to dataFrame elements with some specific value
```
> dataF.ix[dataF.a>5, :2]  = 9999
```
   ii. Add and delete columns
```
> dataF['new'] = range(41,51)       ## Add a new column to the data frame
> del dataF['new']                  ## Remove a column from parent dataframe
> dataF.drop(['a', 'b'], axis=1)    ## Drop and return a new dataframe
```
   iii. Dropping rows
```
> dataF.drop(['p', 'q'])  ## drop rows (by default)
> data.drop([1, 2])
```

d. Sorting/Ranking operations:
   i. sort lexicographically by row or column index:
```
> <Series>.sort_index()
> dataF.sort_index()     ## Default is to sort the row index
> dataF.index = list('pqrstuvwxy')
> temp = list('pqrstuvwxy')
> np.random.shuffle(temp)
> dataF_new = dataF.reindex(temp)     ## shuffle the rows
> dataF_new = dataF_new[['c', 'a', 'b']] ## shuffle the columns too
> dataF_new.sort_index()
> dataF_new.sort_index(axis=1, ascending=False)       ## in descending order
```
   ii. sort according to values of a column or a set of columns
```
> dataF_new.sort_index(by='c')
> dataF_new.sort_index(by=['a', 'b'])
```
→ <Series>.order() method

   iii. Ranking
```
> age = Series([22, 22, 16, 13, 29, 16, 28, 32])
>age.rank()       ## By default ties are broken by mean rank of the group
> age.rank(method='first ')       ## ties broken in order in which they appear
                                  ## Other methods: max, min, first, average
> df = DataFrame([[1,2,3], [4,5,6], [7,8,9],[10,11,12],[ 13, 14, 15]],
columns=list('abc'), index=list('pqrst'))
> dataF.rank(), dataF.rank(axis=1)       ## default axis=0
```

e. Vectorized operations (ex. math operations) like in NumPy… everything we studied in numpy applies here too…:
```
> x = Series([10, 20, 30, 40, 50])
> x*2, x**2
> np.sqrt(x)
> df = DataFrame([[1,2,3], [4,5,6], [7,8,9],[10,11,12],[ 13, 14, 15]], columns=list('abc'),
index=list('pqrst'))
> np.sqrt(dataF)
```

f. Summary and Descriptive Statistics
   i. Usage: <Series>.<method>
   ii. Usage: <DataFrame>.<method>

1. skipna=True by default
2. In dataframe, you can provide axis and level (for hierarchically indexed dataframes… > 2-D) arguments also
3. method: sum, mean/median, std/var, mad, count, min/max, idxmin/idxmax, cumsum/cumprod/cummin, describe etc.

iii. <Series>.describe(), <DataFrame>.describe()

1. For non-numeric data also

iv. Correlation and Covariance

> df2 = DataFrame(np.random.normal(size=12).reshape((3,4)), columns=list('abcd'))
> df2.corr()
> df2.cov()
> df3 = DataFrame(np.random.normal(size=12).reshape((3,4)), columns=list('abdq'))
> df2.corrwith(<Series/DataFrame) ## correlation with another Series or DataFrame
Series: pairwise with every column
DataFrame: matching column names

v. Unique values, Value counts, and Membership (**Series' methods**)

> alpha = Series(list('abaabcdbdbdaaabdcd'))
> alpha.unique()
> alpha.value_counts()
> alpha.isin(['a', 'y', 'z']) ## Boolean Series object… can be used for filtering

vi. More Generic functions

1. 'apply' function (apply a function to whole rows or whole columns)

> f = lambda x: x.max() – x.min() ## **Get more examples of lambda**
> dataF.apply(f) ## By default, f is applied to axis 0, i.e., across the rows
> dataF.apply(f, axis=1)
Ex. sum function

2. 'applymap' function (to apply a function to each element of the dataframe)

> format = lambda x: '%.2f'%x
> dataF_new = dataF.applymap(format)
> type(dataF.ix[0,0])
> type(dataF_new.ix[0,0])

Ex. **sqrt function from numpy** (**vectorized**)… will apply here… but not sqrt function from math library…
-> from numpy import sqrt

4. Index objects:
   a. <Series>.index, <DataFrame>.index
   b. Array-like object, but also a fixed-size 'set'
      i. Can be used to do set operations/logic (like intersection, union, difference etc)
      Ex. intersection of indices of two

→ Give an Ex.

c. Used a lot in complex operations like merging of dataframes… will be discussed later

5. Data import and export from/to text files:
   a. dataF.to_csv("/Users/hgrover/Desktop/dataF.txt")
      i. By default, index column is also written
      ii. dataF.to_csv("/Users/hgrover/Desktop/dataF.txt", index=False)
   b. If index col is written:
      i. pd.read_csv("/Users/hgrover/Desktop/dataF.txt", index_col = 0)
      ii. Otherwise:
         1. pd.read_csv("/Users/hgrover/Desktop/dataF.txt")

   c. Read from a file, URL or file-like object:
      i. read_csv(): default delimiter is ","
         Create a simple file with data:
         ID, C1, C2, C3, C4
         a, 1, 2, 3, 4
         b, 5, 6, 7, 8
         c, 9, 10, 11, 12
         d, 13, 14, 15, 16
         e, 17, 18, 19, 20

         > dataF = pd.read_csv(<fileName>)
         > dataF = pd.read_table(<fileName>, sep=', ')
         > dataF = pd.read_csv(<fileName>, header=None)
         > dataF = pd.read_csv(<fileName>, header=None, skiprows=1)
         or
         dataF = pd.read_csv(<fileName>, header=None, skiprows=[0,1])
         → skiprows can be an integer number of rows to be skipped in the beginning, or a list of row numbers, starting from 0
         > dataF = pd.read_csv(<fileName>, header=None, skiprows=1, names=['ID', 'C1', 'C2', 'C3', 'C4'])
         > dataF = pd.read_csv(<fileName>, index_col='ID')
         → Writing to a file:
            > dataF.to_csv(<fileName>)
            > dataF.to_csv(<fileName>, header=False)
            > dataF.to_csv(<fileName>, index=False, cols=['ID', 'C1', 'C2'])

      ii. Lots of other options in parser functions:
         1. Regular Expression delimiters
         2. Skip specific rows
         3. Identify and/or fill in missing values (each column can have its own markers for missing values)
         4. Specify marker to split comments (at the end of lines)
            > dataF = pd.read_csv(<fileName>, comment='#')
         5. nrows: to read nrows from the beginning
         6.
      iii. read_table(): default delimiter is "\t"