PIC 16, Fall 2016 – Preparation 5W

Assigned 10/24/2016. To be completed by 12 p.m. 10/26/2016.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- identify the appropriate `pandas` data structure (`Series` or a `DataFrame`) to contain data,
- read and write Excel worksheets with `pandas`,
- use IPython tab completion to view the attributes of an object,
- select `DataFrame` data by lists/slices of row/column indices/labels,
- select `DataFrame` data by element values using boolean indexing,
- modify selected `DataFrame` data using the assignment operator (as usual), and
- add data by concatenation and remove data by copying the desired data into a new `DataFrame`.

Tasks

- ☐ Watch 10 Minute Tour of Pandas
- ☐ Read the `pandas` Overview until "Getting Support". Most of it may look like gibberish at this point, so focus on "Data Structures at a Glance" and "Mutability and copying of data". This is what you should get now:
  - We will be working with two kinds of `pandas` data structures: `Series` (one dimensional, homogeneously-typed arrays) and `DataFrames` (2D arrays with column and row labels, each column of which is typically a `Series`). You can think of a `Series` as a single column from a table, and a `DataFrame` as a table (composed of several columns), but they are separate kinds of objects, each with its own methods.
  - It notes that it is conceivable to have a single class of objects capable of storing either 1D, 2D, or 3D (some sort of "*N*-D array" object, or a list of lists of lists, for instance) but the designers of `pandas` chose not to do that. Instead there is one class for 1D data (`Series`), one class for 2D data (`DataFrame`), and one class for 3D data (which we won't use).
  - Once created, the values in a `pandas` data structure (a `Series` or `DataFrame`) can be edited, but it might not be possible to add *new*, additional values. To accomplish the same goal, you'd have to copy all the data into a new, larger data structure. It's like we're back in C++, when we can change the values of an array, but not change its length.
- ☐ Read 10 minutes to `pandas` up to "Object Creation". All we need from here is how to import the `pandas` module. We'll use the other modules later in the course.
- ☐ Skip way ahead to `Getting Data In/Out - Excel`. If you're interested, read the part above about CSV files for an alternative to the `csv` module from earlier in the course.
- ☐ Using the example as a guide, but modifying arguments as needed, load the only sheet of `data.xlsx` into a variable (a `DataFrame`).
  - If you don't understand what a worksheet is, that work*book* files can contain multiple work*sheets*, or why the name of a worksheet needs to be specified, read this. Note that the second argument to the `read_excel` method accepts integer arguments as an alternative to the worksheet name.
  - If you left the index column as `None`, a column of integers has been added to the left of the data in the worksheet. We'll learn that we can use the elements in this column to access rows of the `DataFrame`, just like we use integers to access elements of a list. However, note that the 0th column in the original excel worksheet can (and should, in this

case) be used as an index. Check out what happens when you change the index column to 0. The integer index column is *not* created; instead we can use the Member ID, a string, to refer to a particular row, just a key is used to access an element of a dictionary.

- ☐ Go way back to read Object Creation.
- ☐ We haven't seen the numpy (`np`) module yet, so let's run through what's going on:
    - The first command should be pretty intuitive. We use the `Series` constructor to convert a regular Python list into a `pandas Series` object. When printed, the `Series` object shows its index column, a list of integers by default, to the left of the actual data. `np.nan` is simply an object called "Not a Number" used to represent a quantitative concept that is not actually a number, like infinity. You don't need to know this.
    - The second command takes a bit more thinking. Based on `Out[7]`, we can infer that the `date_range` function generates a range of dates. This is also implied by the name of the function… Apparently the first argument is the initial date as a string in a "YYYYMMDD" format, and the next argument is the total number of consecutive dates we want to be generated. Presumably, the default separation between values in the range is one day.
    - Now we know how to use `pandas date_range` function. If you ever need more information about it, you can look it up. But now you know that it exists, what it does, and basic use. You're going to have to compare inputs and outputs like this to figure out what is going on throughout the tutorial. It will not spoon-feed the information to you.
    - I'll give you a hint at the next one – the command `np.random.rand(6,4)` generates an object representing a two dimensional (six rows, four columns) array of random numbers between zero and one. We create a `DataFrame` object out of it. But rather than using an integer index to refer to a particular row, we want to use a *date* to pick out rows. Apparently each date corresponds with four numbers, which could represent, for example, the average temperature, humidity, rainfall, and wind speed on that day.
    - Please take some time to work out what's going on with the next example on your own. Maybe it would help to individually call some of the commands you don't understand and print out the results to see what each does before piecing together the whole puzzle.
- ☐ You don't need to create data to work with as you already imported some from Excel, but I suggest you try out the IPython tab completion technique mentioned at the end of the section to check out of few attributes. That is, in IPython, type the name of the `DataFrame` variable followed by a "`.`", press the tab key on your keyboard, and note the selectable list of attributes that appears.
- ☐ Read Viewing Data
- ☐ Try all the attributes (methods and instance variables) mentioned in the section on the data you imported from excel.
    - You're going to see a lot of `u"string"`; the `u` indicates that the string is encoded using Unicode rather than ASCII. You don't need to know this.
    - When you get to `sort_index`, try different values for `axis` and `ascending` to explore what is going on. Can you reverse the order of the rows? Columns?
    - Can you sort the data by First Name or Last Name instead of by Member Number?
- ☐ Read Getting
- ☐ Try the following:
    - select column `First` using the `["Column_Name"]` notation
    - select column `First` using the `.Column_Name` notation
    - select rows 3 - 6 by slicing based on the row number (zero-indexed). Note that row number slicing excludes the row corresponding with the second argument, just like regular Python list slicing.

    - select Members A1002 - A1005 by slicing based on the `DataFrame` index, in this case the member number. Note that index slicing is *inclusive* of the second argument.
    - select ONLY row 3 / Member only by slicing based on the row index (zero-indexed) or the member number. Note that you MUST slice like "[<start>:<end>]"; a single row index number or label does not work.
- Read Selection by Label. By "label", it means row label/index (Member #) or column label text rather than number.
- Exercises:
    - Print out `df.loc`. Apparently it's some sort of special `pandas` object. It seems the operator `[]` has been overloaded (by writing a `__getitem__` method) for this object to provide a different means of accessing data from the `DataFrame`. According to the note at the beginning of the Selection section, it is somehow optimized to be faster than the regular `DataFrame` indexing above.
    - Use `.loc` to select member A1002. Apparently the `.loc` object doesn't require slicing like the `DataFrame` indexing.
    - Select members A1002 - A1005 first and last names (no phone numbers).
    - Can you select based on a list of member numbers (e.g. `["A1002", "A1003"]` instead of slicing like `"A1002":"A1003"`)?
    - Can you select by slicing column names (e.g. `"First":"Last"` instead of `["First","Last"]`)
- Read Selection by Position
- Try the same exercises as you did selecting by label, but this time using column and row numbers with `.iloc` (instead of row/column names with `.loc`) Note that like regular Python list slicing, indexing is 0-based (not including the row or column labels), the start bound is included in the returned data, and the end bound is *excluded* (as usual).
- Read Boolean Indexing. Note that the operation `df.A > 0` in the example returns a series of Boolean values. Each element in the series corresponds with whether the value in column A of `df` is greater than zero. `df[df.A > 0]` selects *only* the rows where the Boolean value is true (the value in column A is greater than zero); it leaves out the rows in which the Boolean value is false (the value in column A is less than zero).
- Exercises:
    - Select the rows corresponding with all First names starting with "M-Z"
    - Select the rows with all First names that are "Meredith" or "Summer" (that is, are in the list ["Meredith", "Summer"])
    - Select the rows with all First names that are "Meredith" or "Summer" and whose phone numbers start with 3-9. Note that you can perform element-wise logical operations with the & and | symbols, but you should be careful to group operations with parentheses.
    - Select the phone numbers (only) in all the rows in which the first names are "Meredith" or "Summer". The reading doesn't show you explicitly, but you can use the logical series returned by an operation in conjunction with `.loc` and `.iloc`....
- Now that you can select data, not only can you return it - you can change it! You can change the value of all entries you have selected (to the same new value) simultaneously using the assignment operator. For instance:

```
my_data_frame.loc["A1001","Phone"] = "888-888-8888"
my_data_frame.iloc[0:1,0:1] = "111-111-1111"
my_data_frame.loc[my_data_frame.Phone == "111-111-1111", "Phone"] =
"222-222-2222"
```

- Try changing all the First Names in the `DataFrame` that start with the letter "A" to "Unicorn"
- You can remove data by selecting what you want and saving that to a new variable.
  For instance, to remove the row A1001:

```
y = my_data_frame[1:]
y = my_data_frame["A1002":]
y = my_data_frame[my_data_frame.index != "A1001"]
```

- Read Concat. We may practice this later, but you can see that with this you could add new data to the same `DataFrame`.
- Try saving your modified spreadsheet in data2.xlsx.
- Feel free to skim everything else, but I think this is a good start to manipulating two-dimensional data using `pandas`.