Lecture 3 Notebook

August 30, 2018

1 Lecture 3 Notebook

Duncan Callaway August 30 2018

This lecture is a review of variable types and an introduction to the pandas library within python.

My objective is to review enough of the basics to give people a clear sense of what a pandas data frame is and what it can contain.

1.1 Pandas references

(taken from DS100)

Introductory:

- Getting started with Python for research, a gentle introduction to Python in data-intensive research.
- A Whirlwind Tour of Python, by Jake VanderPlas, another quick Python intro (with notebooks).

Core Pandas/Data Science books:

- The Python Data Science Handbook, by Jake VanderPlas.
- Python for Data Analysis, 2nd Edition, by Wes McKinney, creator of Pandas. Companion Notebooks
- Effective Pandas, a book by Tom Augspurger, core Pandas developer.

Complementary resources:

- An introduction to "Data Science", a collection of Notebooks by BIDS' Stéfan Van der Walt.
- Effective Computation in Physics, by Kathryn D. Huff; Anthony Scopatz. Notebooks to accompany the book. Don't be fooled by the title, it's a great book on modern computational practices with very little that's physics-specific.

1.2 Recap last lecture

1.3 A few more shortcuts

`esc` --> command mode for cell
`enter` --> edit mode for cell
`shift-enter` --> run cell and select next cell
`ctrl-enter` --> run cell and keep it selected
in command mode, `a` creates a new cell above
in command mode, `b` creates a new cell below
in command mode, `dd` deletes cell
in command mode, `m` turns cell into markdown
in command mode, `y` turns cell to code
in command mode, `h` shows keyboard shortcuts
`shift-tab` gives information on a function or method
`tab` gives options to add a method or function to an object.

1.4 Object, method, function?

These phrases come up repeatedly in python.
Q: What are they?
Go here for more

1. Object: virtually anything with attributes in python.
    1. Objects usually belong to classes and have attributes
    2. For example an object might belong to a bike class. The attributes of the class would be material, wheel size, number of gears, etc.
2. Method: A function associated with an object.
    a. for example `object.method` applies the method to the object.
3. Function: performs an action using some set of input parameters.
    b. for example `function(object)` applies the function to the object.

1.5 Basic data, or variable, types

Q: what are some variable types native to Python?

1. `string`
2. `numbers`
    1. `int`
    2. `float`
3. `bool`

An interesting note about strings: though we can index their contents, they are *immutable*, meaning you can't change individual elements. Instead you need to do a wholesale reassignment.

1.6 Data structures or "containers"

Q: what are some data structures native to Python?

1. `dict`
2. `list`
3. `tuple`
4. `set`

Let's explore these a little further.

1.6.1 Lists

```
In [ ]: squares = [1, 5, 9, 16, 25] # do this in lecture
        squares
```

We can index elements with the usual process:

```
In [ ]: print(squares[0]) # do this in lecture; zero is the first element
        print(squares[-1]) # do this in lecture; -1 gives the last
        print(squares[2:]) # do this in lecture; you can slice...
```

What does "mutable" mean?
...that you can change individual entries of the data structure.
Lists are mutable:

```
In [ ]: squares[1] = 4
        squares
```

That's better!
We can also append:

```
In [ ]: squares.append(6**2)
        squares
```

Finally, we can nest lists

```
In [ ]: x = [1,2,3]
        y = [4,6]
```

```
In [ ]: A = [x,y]
        A
```

Note that the number of elements in the two lists within the list did not need to be equal.
Note also that we can also assign different variable types to the nested list:

```
In [ ]: a = ['abc', 'def']
```

```
In [ ]: A = [x,a]
        A
```

Careful though, it's tough to index these things.
Think about how you might get the entry 'def' from the list, by integer indexes.

```
In [ ]: A[1,1]
```

That didn't work! (We have to wait for numpy and pandas to be able to do that.)

```
In [ ]: A[1][1]
```

Note, pandas data frames are a lot like lists in this way. You first index which "sublist" you want, then you index into elements of that. Numpy arrays (as we'll see) are much easier to index. Frankly, the indexing with pandas is pretty annoying, but there are some workarounds that we'll discuss.

1.6.2 Sets

Whereas lists are defined with square brackets, sets use curlies:

```
In [ ]: basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

What happened there?
...I wrote orange and apple twice. Let's look at the set:

```
In [ ]: print(basket)
```

Pretty smart -- no duplicates!
What might you do if you have a variable called basket but you don't know what's in it?
Now we can query the set for membership:

```
In [ ]: 'orange' in basket
```

```
In [ ]: 'kiwi' in basket
```

Note, pandas data frames are like lists in that there are a number of entries that we can very quickly query. But there are even more similarities with `dict` data structures.

1.6.3 Dict

Now let's talk about dicts. We are inching closer to the kind of structure we have in a pandas data frame.
Dicts associate one value with another.

```
In [ ]: cars = {'Prius':'Toyota', 'Volt':'Chevy', 'Model 3': 'Tesla'}
        cars
```

```
In [ ]: cars['Prius']
```

In this case 1. 'Prius' is the **key** of the dict. 2. 'Toyota' is the associated **value**.
We can add new entries very easily:

```
In [ ]: cars['Leaf'] = 'Nissan'
```

```
In [ ]: cars
```

1.7 Numpy

Before moving into pandas, let's do a quick review of numpy. A numpy array is a grid of elements *of the same data type*. (Pandas is like numpy but handles different data types.) The beauty of numpy, like matlab, is that you don't need to specify what sort of data type you're working with -- it will guess for you.

```
In [ ]: import numpy as np
        a = np.array([[1,2,3], [4,5,6]])
        a
```

That's a numpy array.
Q: What have we seen before that looks a lot like this?
A: a nested list.
A big difference, though, is that we can index numpy arrays more cleanly can we can lists:

```
In [ ]: a[1,1]
```

What is the first index entry calling? If you're not sure, play around with `a` to see if you can figure it out.
Answer: The row.
And so the second entry is calling the column. Note this is the opposite of the order we used when we indexed into the nested list.
What happens when we try to add a different data type to the array?

```
In [ ]: a[1,1] = 'g'
        a
```

We can't make the assignment. However if we do it in the original definition, we can

```
In [ ]: a = np.array([[1,2,3], [4,5,'g']])
        a
```

We can make the assignment -- but python interprets the data type as '<U21', which is a mixed data type.
What happens if we try to do mathematical operations on this type?

```
In [ ]: a[1,0]+1
```

Ok. What's a slice?
It's a portion of the array that we call with expressions involving a colon : in the index locations.

```
In [ ]: b = a[0:1, 0:2]
        b
```

Perhaps you remember that the indexing is not *inclusive* meaning it does not include data associated with the end index.

```
In [ ]: c = a[:,-1:0:-1]
        c
```

If python's not inclusive, how to I include entries at the ends?

If we want to get all the way to the end of the array, just drop the second entry in the slice:

```
In [ ]: c = a[:, -1::-1]
        c
```

1.8 Pandas!

Ok, finally we're here.

In Data8 you used the `tables` library to organize and manipulate data. I've not used tables but it has been described to me as a 'light' version of pandas. So this will be somewhat familiar to you.

Pandas has several features that make it immediately better than numpy for organizing data. 1. You can have different data types (string, int, float) in each column 2. You can label columns (headers) 3. You can label rows (index)

We call the data structure that holds all these things together a **data frame**

Ambiguity alert: up to now we've talked about indexing to access individual elements of a data structure. The index in pandas has a slightly more specific meaning, in that it references the rows of the data frame. Pandas documentation talks about "location" or "position" in place using "index" as we did with the other data structures. I'll try my best to disambiguate, but also rely on context to clarify the meaning of the terms.

At its core, the data frame structure is what draws us to use pandas. But it also has a bunch of great built-in functions you can use to manipulate data once it is loaded in.

```
In [ ]: import pandas as pd
```

I'm going to define a simple data frame in a way that you can see the connections to existing Python data types and structures.

First let's define a dict of fruit information:

```
In [ ]: fruit_info_dict = {'fruit':['apple','banana','orange','raspberry'],
                           'color':['red','yellow','orange','pink'],
                           'weight':[120,150,250,15]}
        fruit_info_dict
```

```
In [ ]: fruit_info_df = pd.DataFrame(
            data={'fruit':['apple','banana','orange','raspberry'],
                           'color':['red','yellow','orange','pink'],
                           'weight':[120,150,250,15]
                 })
        fruit_info_df
```

Some notes: 1. You can see that we put the data inside curly brackets much like we do in a dict.

1. We defined it as if fruit, color and weight are keys. 1. In fact, we'll be calling these column names. 2. The pandas data structure is called the data frame.

```
In [ ]: type(fruit_info_df)
```

We can call the values associated with each column name (like the dict key) in much the same way that we did with the dict:

```
In [ ]: fruit_info_dict['color']
```

```
In [ ]: fruit_info_df['color']
```

But notice above in the dataframe we don't just have a list of colors. Instead we have something called a **series**. This is a pandas object that is analogous to a numpy series.

```
In [ ]: type(fruit_info_df['color'])
```

The series differs from the list in at least one important way: It has numbers directly associated with it that we call the index. (The left column of the Series.)

We can also call columns from the data frame as follows:

```
In [ ]: fruit_info_df.color
```

But as we'll see soon, there are alternative ways to get access to the elements of the data frame (`.loc` and `.iloc`) that enable us to work with the frame more as we would a numpy array.

Let me show a quick example of the problems we run into if we *don't* use `.loc` or `.iloc`.

We *can* slice the frame as you might think:

```
In [ ]: fruit_info_df[0:1]
```

But we *can't* slice on columns and rows:

```
In [ ]: fruit_info_df[0:1,0:1]
```

Nor can we index individual entries:

```
In [ ]: fruit_info_df[1,1]
```

If we want individual entries instead we have to first call the column, then the row:

```
In [ ]: fruit_info_df['color'][1]
```

...But that's a little annoying, since the numpy (and most other) conventions is to first reference the row, then the column.

Also, there is no way to slice on columns:

```
In [ ]: fruit_info_df['color':'weight']
```

Stretch Break!

1.8.1 Anatomy of the data frame.

Let's talk a little about the anatomy of the data frame.

We have the following important attributes: 1. Rows 2. Columns 2. Index 3. Column names

The "index" can be numeric, but as we'll see we can also make the indices strings.

1.8.2 Pandas and the CAISO data.

Let's see what's in the current directory. In macos, I type:

```
In [ ]: !ls
```

But if someone picked up my ipynb and ran it on windows, they'd get an error.
Instead we can use the `os` library:

```
In [ ]: import os
        os.listdir()
```

I've already downloaded a file, "CAISO_2017_to_2018.csv", that has one year of renewables production data from CAISO.
Let's load that in as a dataframe and take a look at it.
The simplest command is `pd.read_csv`

```
In [ ]: caiso_data = pd.read_csv('CAISO_2017to2018.csv')
```

Now we can look at the top of the dataframe using the .head method.
Note that you need to put parentheses on the end of the call, otherwise python returns the head "object" in a rather ugly form.

```
In [ ]: caiso_data.head()
```

Pandas loaded the date and time in as a column and put its own row numbers on the data frame.
As an alternative we can actually make the row labels *equal* to whatever column of data we'd like. We'll come back to the notion of the index a little later, but for now let's just reload with the date-time as the index:

```
In [ ]: caiso_data = pd.read_csv('CAISO_2017to2018.csv', index_col=0)
        caiso_data.head()
```

Ok, that looks a little better for now.
As you can see, all the data are the same type of numeric value -- MWh.
In these cases, sometimes it's natural to "stack" the data.
We could do the stacking with a pandas command, `.stack`, but it requires a little more massaging than I want to get into right now. So I've saved a stacked version of the data as a csv:

```
In [ ]: caiso_data_stack = pd.read_csv('CAISO_2017to2018_stack.csv', index_col=0)
        caiso_data_stack.head()
```

You can see now that the category of generation has become a variable rather than a column name.
There are lots of ways to stack data, as you might guess.
Let's work with a few things to explore the data frame.
We've already learned about `.head` but we can use it slightly differently:

```
In [ ]: caiso_data.head(2) # the number in the parens tells pandas how many rows
```

We can also look at the tail!

```
In [ ]: caiso_data.tail()
```

What's the shape of the frame?

```
In [ ]: print('Unstacked shape is ',caiso_data.shape)
        print('Stacked shape is ',caiso_data_stack.shape)
```

The `.shape` method returns a tuple -- number of rows and number of columns.
What about the total number of entries? Use the `.size` method:

```
In [ ]: print('Unstacked size is ',caiso_data.size)
        print('Stacked size is ',caiso_data_stack.size)
```

Both shape and size commands return information about how much data is in the frame. What's the difference?

We can also look at "summary statistics" of the numeric values of the frame:

```
In [ ]: caiso_data.describe()
```

A quirky thing about getting data out of the frame

Do be careful -- pulling data out of the dataframe is sometimes perplexing:

```
In [ ]: caiso_data['GEOTHERMAL'].head()
```

```
In [ ]: caiso_data[['GEOTHERMAL']].head()
```

With only one pair of sq brackets, we got a Series object, but with two sq brackets, we get a frame. One way to think about why this is happening is that if we want to get more than one column, we need to pass a list of column names in, and this would necessarily output a data frame

```
In [ ]: caiso_data[['GEOTHERMAL', 'BIOGAS']].head()
```

Python gets perturbed if you don't pass the labels in as a list.

1.8.3 Indexing and slicing in Pandas

To motivate our interest going forward, let's ask a basic question about the data set, for example:

What hour had the lowest average wind generation in the last year?

Try thinking for a moment about how you'd do this, then we'll try to get there.

First let's figure out how to slice these data frames.

`.iloc` allows us to index and slice on **i**nteger row and column positions:

```
In [ ]: caiso_data.iloc[1,1]
```

But what's nice about `.iloc` is that you can also slice. It works just like numpy.

```
In [ ]: caiso_data.iloc[:-10:-1, :4]
```

`.loc` is similar to `.iloc`, but it allows you to call the index and column names:

```
In [ ]: caiso_data.loc['2018-08-28 10:00:00':'2018-08-28 24:00:00','GEOTHERMAL':'BIOGAS']
```

If index values are non-unique, the basic process is that `.loc` goes for the first appearance of the left side of the slice and the last appearance of the right side.

Note, this raises again an interesting point about pandas data frames -- the row indices don't need to be integer row numbers.

Let's explore the index labels a little more.

1.8.4 The index.

Let's make the data a little smaller for working with, just choosing the first 24 hours.

```
In [ ]: c_dat = caiso_data.head(24)
        c_dat.shape
```

Now if we wish, we can reassign the index labels to integers 1 through 24:

```
In [ ]: c_dat.index = ['Hour '+str(i) for i in range(1,25)]
        c_dat
```

Now we can slice in `.loc` using a slightly easier index:

```
In [ ]: c_dat.loc['Hour 1':'Hour 4',:]
```

But watch out -- `.iloc` might have different indexing.

```
In [ ]: c_dat.iloc[1:4,:]
```

What's different? 1. for the last entry, `.loc` is inclusive, `.iloc` is not.
2. Also, `.iloc` is doing what we'd expect from python with indexing -- 0 is the first entry, 1 is the second.... On the other hand,

These may all seem inconsistent, but when you think about the application of `.iloc`, you'd want to to follow usual python conventions. But `.loc`, to me at least, feels intuitively better -- especially when you have non-numeric values to identify position -- if it's inclusive.

1.8.5 Logical indexing

Logical indexing is an extremely powerful way to pull data out of a frame.
For example, with the stacked data frame, let's pull out only wind generation.

First, I'll show you a boolean series based on comparisons to the 'Source' data column:

```
In [ ]: (caiso_data_stack['Source']=='WIND TOTAL').head()
```

Now we can embed that inside the `.loc` method:

```
In [ ]: caiso_data_stack.loc[caiso_data_stack['Source']=='WIND TOTAL',:].head()
```

1.9 Let's do the quiz.

1.9.1 Answering our question

What hour had the lowest average hourly wind generation?

```
In [ ]: wind = caiso_data_stack.loc[caiso_data_stack['Source']=='WIND TOTAL',:]
```

What is the data structure of `wind`?

```
In [ ]: type(wind)
```

Next week we'll use pivots to do this better, but for now let's use a for loop to get information by hour.

First thing to do is figure out how to get the hour out of the index.

`datetime.strptime` is useful for this if you're working on individual dates.

But `pd.to_datetime` is even better, especially if you're working on a lot of values in a list (or as the case will be, values in a pandas series).

```
In [ ]: windex = pd.to_datetime(wind.index)
        windex.hour
```

```
In [ ]: wind_ave = [] # initalizes a list to populate
        for i in range(0,24):
            wind_ave.append(np.mean(wind.loc[windex.hour == i,:]))
        wind_ave
```

```
In [ ]: import matplotlib.pyplot as plt
```

```
In [ ]: plt.plot(wind_ave)
```

We can see pretty clearly that the min is 10 or 11...let's dig a little more.

One way to do this is to drop the data into a data frame and then *sort* the data frame.

```
In [ ]: df_wind = pd.DataFrame(wind_ave)
        df_wind
```

I'm going to be adding more MWh values to the data frame in just a moment, so let's be clear that this is the average

```
In [ ]: df_wind.columns = ['Average MWh']
```

```
In [ ]: df_wind.sort_values(by='Average MWh',ascending=True).head()
```

Ok -- so it looks as though mid-day is the minimum *average*.

But what's the range?

```
In [ ]: wind_min = [] # initalizes a list to populate
        wind_max = [] # initalizes a list to populate
        for i in range(0,24):
            wind_min.append(np.min(wind.loc[windex.hour == i,:]))
            wind_max.append(np.max(wind.loc[windex.hour == i,:]))
```

```
In [ ]: wind_max[0]

In [ ]: df_wind['min MWh']=pd.DataFrame(wind_min)['MWh']
        df_wind['max MWh']=pd.DataFrame(wind_max)['MWh']

In [ ]: df_wind

In [ ]: plt.plot(df_wind)
```

1.9.2 Challenge question: What *month* had the highest wind production in the last year?