Pandas II - working with data

Last time, we met some of the basic data structures in pandas.

Basic pandas objects:

- Index
- Series
- Data Frame

We also learned how these three things are related. Namely, we can think of a pandas `DataFrame` as being composed of several *named columns*, each of which is like a `Series`, and a special `Index` column along the left-hand side.

In this tutorial, we'll learn about

- some common methods that pandas objects have – the "verbs" that make them do useful things
- accessing row/column subsets fo data
- working with grouped data: aggregation and pivot tables

Make a data frame to play with

Let's build a little data frame and take look at it to remind ourselves of this structure. We'll build one similar to a data frame we played with last time.

It will have 5 columns co

First, import `pandas` because of course, and `numpy` in order to simulate some data.

```
In [ ]: import pandas as pd
        import numpy as np      # to make some simulated data
```

Now we can make the data frame. It will have 4 variables of cardiovascular data for a number of patients that we can specify. Because a pandas `DataFrame` has that special index column, we'll just use it to correspond to "patient ID" instead of making a fifth variable.

In [ ]:
```
num_patients = 10    # specify the number of patients

# make some simulated data
sys_bp = np.int64(125 + 5*np.random.randn(num_patients,))
dia_bp = np.int64(80 + 5*np.random.randn(num_patients,))
b_oxy = np.round(98.5 + 0.3*np.random.randn(num_patients,), 2)
pulse = np.int64(65 + 2*np.random.randn(num_patients,))

# Make a dictionary with a "key" for each variable name, and
# the "values" being the num_patients long data vectors
df_dict = {'systolic BP' : sys_bp,
           'diastolic BP' : dia_bp,
           'blood oxygenation' : b_oxy,
           'pulse rate' : pulse
           }

our_df = pd.DataFrame(df_dict)    # Now make a data frame out of the dictio
```

And now lets look at it.

In [ ]:
```
our_df
```

Now we can see the nice structure of the `DataFrame` object. We have four columns corresponding to our measurement variables, and each row is an "observation" which, in the case, corresponds to an individual patient.

To appreciate some of the features of a pandas `DataFrame` , let's compare it with a numpy `Array` holding the same information. (Which we can do because we're only dealing with numbers here - one of the main features of a pandas data frame is that it can hold non-numeric information too).

In [ ]:
```
our_array = np.transpose(np.vstack((sys_bp, dia_bp, b_oxy, pulse)))
our_array
```

We can see here that our array, `our_array` , contains exactly the same information as our dataframe, `our_df` . There are 3 main differences between the two:

- they have different verbs – things they know how to do
- we have more ways to access the information in a data frame
- the data frame could contain non-numeric information (e.g. gender) if we wanted

(Also notice that the data frame is just prettier when printed than the numpy array)

Verbs

Let's look at some verbs. Intuitively, it seems like the should both know how to take a mean. Let's see.

In [ ]:
```
our_array.mean()
```

So the numpy array does indeed know how to take the mean of itself, but it takes the mean of the entire array by default, which is not very useful in this case. If we want the mean of each variable, we have to specify that we want the means of the columns (i.e. row-wise means).

In [ ]:
```
our_array.mean(axis=0)
```

But look what happens if we ask for the mean of our data frame:

In [ ]:
```
our_df.mean()
```

Visually, that is much more useful! We have the mean of each of our variables, nicely labled by the variable name.

Data frames can also `describe()` themselves.

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

Gives us a nice summary table of the data in our data frame.

Numpy arrays don't know how to do this.

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

Data frames can also make histograms and boxplots of themselves. They aren't publication quality, but super useful for getting a feel for our data.

In [ ]:
```
our_df.hist();
```

In [ ]:
```
our_df.boxplot();
```

For a complete listing of what our data frame knows how to do, we can type `our_df.` and then hit the tab key.

In [ ]:
```
our_df.
```

Let's return to the `mean()` function, and see what, exactly, it is returning. We can do this by assigning the output to a variable and looking at its type.

In [ ]:
```
our_means = our_df.mean()
our_means
```

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

So it is a pandas series, but, rather than the index being 0, 1, 2, 3, the *index values are actually the names of our variables*.

If we want the mean pulse rate, *we can actually ask for it by name!*

In [ ]:
```
our_means['pulse rate']
```

This introduces another key feature of pandas: **you can access data by name**.

Accessing data

Accessing data by name is kind of a big deal. It makes code more readable and faster and easier to write.

So, for example, let's say we wanted the mean pulse rate for our patients. Using numpy, we would have to remember or figure our which column of our numpy array was pulse rate. And we'd have to remember that Python indexes start at 0. *And* we'd have to remember that we have to tell numpy to take the mean down the columns explicitly. Ha.

So our code might look something like...

In [ ]:
```
np_style_means = our_array.mean(axis=0)
pulse_mean = np_style_means[3]
pulse_mean
```

Compare that to doing it the pandas way:

In [ ]:
```
our_means = our_df.mean()
our_means['pulse rate']
```

The pandas way makes it very clear what we are doing! People like things to have names and, in pandas, things have names.

Accessing data using square brackets

Let's look ot our litte data frame again.

In [ ]:
```
our_df
```

We can grab a column (variable) by name if we want:

In [ ]:
```
our_df['pulse rate']
```

Doing this creates another `DataFrame` (or `Series`), so it knows how to do stuff to. This allows us to do things like, for example, compute the mean pulse rate in one step instead of two. Like this:

```
In [ ]: our_df['pulse rate'].mean()   # creates a series, then makes it compute its
```

We can grab as many columns as we want by using a list of column names.

```
In [ ]: needed_cols = ['diastolic BP', 'systolic BP']   # make a list
        our_df[needed_cols]                              # use the list to grab colu
```

We could also do this in one step.

```
In [ ]: our_df[['diastolic BP', 'systolic BP']]  # the inner brackets define our li
```

(although the double brackets might look a little confusing at first)

Getting row and row/column combinations of data: "indexing"

Terminology Warning! "Indexing" is a general term which means "accessing data by location". In pandas, as we have seen, objects like DataFrames also have an "index" which is a special column of row identifiers. So, in pandas, we can index data using column names, row names (indexing using the index), or both. (We can also index into pandas data frames as if they were numpy arrays, which sometimes comes in handy.)

Changing the index to make (row) indexing more intuitive

Speaking of indexes, it's a little weird to have our patient IDs start at "0". Both because "patient zero" has a special meaning and also because it's just not intuitive to number a sequence of actual things starting at "0".

Fortunately, pandas `DataFrame` (and `Series`) objects allow you to customize their index column fairly easily.

Let's set the index to start at 1 rather than 0:

```
In [ ]: my_ind = np.linspace(1, 10, 10)  # make a sequence from 1 to 10
        my_ind = np.int64(my_ind)        # change it from decimal to integer (not r
        our_df.index = my_ind
```

```
In [ ]: our_df
```

Accessing data using `pd.DataFrame.loc[]`

In the section above, we saw that you can get columns of data our of a data frame using square brackets `[]`. Pandas data frames also know how to give you subsets of rows or row/column combinations.

The primary method for accessing specific bits of data from a pandas data frame is with the `loc[]` verb. It provides an easy way to get rows of data based upon the index column. In other words, `loc[]` is the way we use the data frame index as an index!

So this will give us the data for patient number 3:

In [ ]:
```
our_df.loc[3]
```

Note! The above call did **not** behave like a Python or numpy index! If it had, we would have gotten the data for patient number 4 because Python and numpy use *zero based indexing*.

But using the `loc[]` function gives us back the row "named" 3. We literally get what we asked for! Yay!

We can also *slice* out rows in chunks:

In [ ]:
```
our_df.loc[3:6]
```

Which, again, gives us what we asked for without having to worry about the zero-based business.

But `.loc[]` also allows us to get specfic columns too. Like:

In [ ]:
```
our_df.loc[3:6, 'blood oxygenation']
```

For a single column, or:

In [ ]:
```
our_df.loc[3:6,'systolic BP':'blood oxygenation']
```

for multiple columns.

In summary, there are 3 main ways to get chunks of data out of a data frame "by name".

- square brackets (only) gives us columns, e.g. `our_df['systolic BP']`
- `loc[]` with one argument gives us rows, e.g. `our_df.loc[3]`
- `loc[]` with two arguments gives us row-column combinations, e.g. `our_df.loc[3,'systolic BP']`

Additionally, with `loc[]`, we can specify index ranges for the rows or columns or both, e.g. `new_df.loc[3:6,'systolic BP':'blood oxygenation']`

One final thing about using `loc[]` is that the index column in a `DataFrame` doesn't have to be numbers. It can be date/time strings (as we'll see next time), or just plain strings (as we've seen above with `Series` objects).

Let's look at a summary of our data using the `describe()` method:

```
In [ ]: our_sum = our_df.describe()
        our_sum
```

This looks suspiciously like a data frame except the index column looks like they're... er... not indexes. Let's see.

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

Yep, it's a data frame! But let's see if that index column actually works:

```
In [ ]: our_sum.loc['mean']
```

Note that, with a `Series` object, we use square brackets (only) to get rows. With a `DataFrame` , square brackets (only) are used to get columns. It won't work for `DataFrame` objects:

```
In [ ]: our_sum['mean']
```

So, with a `DataFrame` , we have to use `.loc[]` to get rows.

And now we can slice out (get a range of) rows:

```
In [ ]: our_sum.loc['count':'std']
```

Or rows and columns:

```
In [ ]: our_sum.loc['count':'std', 'systolic BP':'diastolic BP']
```

Accessing data using pd.DataFrame.iloc[]

Occasionally, you might want to treat a pandas `DataFrame` as a numpy `Array` and index into it using the *implicit* row and column indexes (which start as zero of course). So support this, pandas `DataFrame` objects also have an `iloc[]` .

Let's look at our data frame again:

```
In [ ]: our_df
```

And let's check its shape:

```
In [ ]: our_df.shape
```

At some level, then, Python considers this to be just a 10x4 array (like a numpy array). This is were `iloc[]` comes in; `iloc[]` will treat the data frame as though it were a numpy array – no names!

So let's index into `our–df` using `iloc[]`:

In [ ]:
```
our_df.iloc[3]  # get the fourth row
```

And compare that to using `loc[]`:

In [ ]:
```
our_df.loc[3]
```

And of course you can slice out rows and columns:

In [ ]:
```
our_df.iloc[2:5, 0:2]
```

Indexing using `iloc[]` is rarely needed on regular data frames (if you're using it, you should probably be working with a numpy `Array`).

It is, however, very handy for pulling data out of summary data tables (see below).

Non-numerical information (categories or factors)

One of the huge benefits of pandas objects is that, unlike numpy arrays, they can contain categorical variables.

Make another data frame to play with

Let's use tools we've learned to make a data frame that has both numerical and categorical variables.

First, we'll make the numerical data:

In [ ]:
```
num_patients = 20    # specify the number of patients

# make some simulated data with realistic numbers.
sys_bp = np.int64(125 + 5*np.random.randn(num_patients,))
dia_bp = np.int64(80 + 5*np.random.randn(num_patients,))
b_oxy = np.round(98.5 + 0.3*np.random.randn(num_patients,), 2)
pulse = np.int64(65 + 2*np.random.randn(num_patients,))
```

(Now we'll make them interesting – this will be clear later)

In [ ]:
```
sys_bp[0:10] = sys_bp[0:10] + 15
dia_bp[0:10] = dia_bp[0:10] + 15
sys_bp[0:5] = sys_bp[0:5] + 5
dia_bp[0:5] = dia_bp[0:5] + 5
sys_bp[10:15] = sys_bp[10:15] + 5
dia_bp[10:15] = dia_bp[10:15] + 5
```

Now let's make a categorical variable indicating whether the patient is diabetic or not. We'll make the first half be diabetic.

In [ ]:
```
diabetic = pd.Series(['yes', 'no'])  # make the short series
diabetic = diabetic.repeat(num_patients/2)       # repeat each over two cel
diabetic = diabetic.reset_index(drop=True)      # reset the series's index
```

Now will make an "inner" gender variable.

In [ ]:
```
gender = pd.Series(['male', 'female'])                  # make the short series
gender = gender.repeat(num_patients/4)                      # repeat each over o
gender = pd.concat([gender]*2, ignore_index=True)   # stack or "concatenate
```

Now we'll make a dictionary containing all our data.

In [ ]:
```
# Make a dictionary with a "key" for each variable name, and
# the "values" being the num_patients long data vectors
df_dict = {'systolic BP' : sys_bp,
           'diastolic BP' : dia_bp,
           'blood oxygenation' : b_oxy,
           'pulse rate' : pulse,
           'gender': gender,
           'diabetes': diabetic
           }
```

And turn it into a data frame.

In [ ]:
```
new_df = pd.DataFrame(df_dict)    # Now make a data frame out of the dictio
```

Finally, let's up our game and make a more descriptive index column!

In [ ]:
```
basename = 'patient '                        # make a "base" row name
my_index = []                               # make an empty list
for i in range(1, num_patients+1) :         # use a for loop to add
    my_index.append(basename + str(i))       # id numbers so the base name
```

Assign our new row names to the index of our data frame.

In [ ]:
```
new_df.index = my_index
```

Let's look at our creation!

```
In [ ]: new_df
```

Looking at our data

Another really nice thing about pandas `DataFrames` is that they naturally lend themselves to interogation via Seaborn. So let's peek at some stuff.

```
In [ ]: import seaborn as sns

        sns.catplot(data=new_df, x='diabetes', y='systolic BP');
```

Okay, now let's go crazy and do a bunch of plots.

```
In [ ]: sns.catplot(data=new_df, x='gender', y='systolic BP', hue='diabetes');
```

```
In [ ]: sns.catplot(data=new_df, x='gender', y='pulse rate', hue='diabetes');
```

```
In [ ]: sns.catplot(data=new_df, x='gender', y='diastolic BP', hue='diabetes', kind
```

Computing within groups

Now that we have an idea of what's going on, let's look at how we could go about computing things like the mean systolic blood pressure in females vs. males, etc.

Using the `groupby()` method

Data frames all have a `group_by()` method that, as the name implies, will group our data by a categorical variable. Let's try it.

```
In [ ]: new_df.groupby('gender')
```

So this gave us a `DataFrameGroupBy` object which, in and of itself, is very useful. However, *it knows how to do things*!

In general, `GroupBy` objects know how to do pretty much anything that regular `DataFrame` objects do. So, if we want the mean by gender, we can ask the `GroupBy` (for short) object to give us the mean:

```
In [ ]: new_df.groupby('gender').mean()
```

Using the `groupby()` followed by `aggregate()`

More powerfully, we can use a `GroupBy` object's `aggregate()` method to compute many things at once.

In [ ]:
```
new_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])
```

Okay, what's going on here? First, we got a lot of information out. Second, we got a warning because pandas couldn't compute the mean, etc., on the gender variable, which is perfectly reasonable of course.

We can handle this by using our skills to carve out a subset of our data frame – just the columns of interest – and then use `groupby()` and `aggregate()` on that.

In [ ]:
```
temp_df = new_df[['systolic BP', 'diastolic BP', 'diabetes']]        # make
our_summary = temp_df.groupby('diabetes').aggregate(['mean', 'std', 'min',
our_summary
```

Notice here that there are *groups of columns*. Like there are two "meta-columns", each with four data columns in them. This makes getting the actual values out of the table for further computation, etc., kind of a pain. It's called "multi-indexing" or "hierarchical indexing". It's a pain.

Here are a couple examples.

In [ ]:
```
our_summary[("systolic BP", "mean")]
```

In [ ]:
```
our_summary.loc[("no")]
```

Of course, we could do the blood pressure variables separately and store them for later plotting, etc.

In [ ]:
```
temp_df = new_df[['systolic BP', 'diabetes']]        # make a data frame wi
our_summary = temp_df.groupby('diabetes').aggregate(['mean', 'std', 'min',
our_summary
```

But we still have a meta-column label!

Here's were `.iloc[]` comes to the rescue!

If we look at the shape of the summary:

In [ ]:
```
our_summary.shape
```

We see that, ultimately, the data is just a 2x4 table. So if we want, say, the standard deviation of non-diabetics, we can just do:

In [ ]:
```
our_summary.iloc[0, 1]
```

And we get back a pure number.

We can also do things "backwards", that is, instead of subsetting the data and then doing a `groupby()`, we can do the `groupby()` and then index into it and compute what we want. For example, if we wanted the mean of systolic blood pressure grouped by whether patients had diabetes or not, we could go one of two ways.

We could subset and then group:

```
In [ ]: new_df[['systolic BP', 'diabetes']].groupby('diabetes').mean()
```

Or we could group and then subset:

```
In [ ]: new_df.groupby('diabetes')[['systolic BP']].mean()
```

Okay, first, it's cool that there are multiple ways to do things. Second – **aarrgghh!** – things are starting to get complicated and code is getting hard to read!

Using pivot tables

"Pivot tables" (so named because allow you to look at data along different dimensions or directions) provide a handy solution for summarizing data.

By default, pivot tables tabulate the mean of data. So if we wish to compute the average systolic blood pressure broken out by diabetes status, all we have to do is:

```
In [ ]: new_df.pivot_table('systolic BP', index='diabetes')
```

Here, `index` is used in the "row names" sense of the word.

We can also have another grouping variables map to the columns of the output if we wish:

```
In [ ]: new_df.pivot_table('systolic BP', index='diabetes', columns='gender')
```

Finally, we can specify pretty much any other summary function we want to "aggregate" by:

```
In [ ]: new_df.pivot_table('systolic BP', index='diabetes', columns='gender', aggfu
```

(Where `aggfunc` can me 'min', 'sum', 'std', etc., etc.)

Summary

In this tutorial, we have covered some key aspects of working with data using pandas data frames. These were:

- doing things with data using the methods – the verbs – of pandas objects
- accessing subsets of the data with
  - square brackets
  - the `.loc[]` method
  - the `.iloc[]` method
- assembling data frames and customizing the index
- grouping data and computing summaries using
  - `groupby()` and `aggregate()`
  - pivot tables

Exercise

Let's do the following on our own

1. Make a data frame that has
   - one categorical variable, "bilingual", that splits the data in half ("yes" and "no")
   - two numerical variables, verbal GRE and quant GRE
   - (you can build in, or not, whatever effect of bilingual you wish)
   - (GRE scores have a mean of about 151 and a std. dev. of about 8.5)
2. Set the index to be "Student 1", "Student 2", etc.
3. Do a seaborn plot of verbal GRE vs. bilinguality (is that a word?)
4. Make another one of quant GRE vs. bilingual status
5. Compute the mean and standard *error* of each score separated by bilingual status (using any method you wish!)

In [8]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

In [3]:

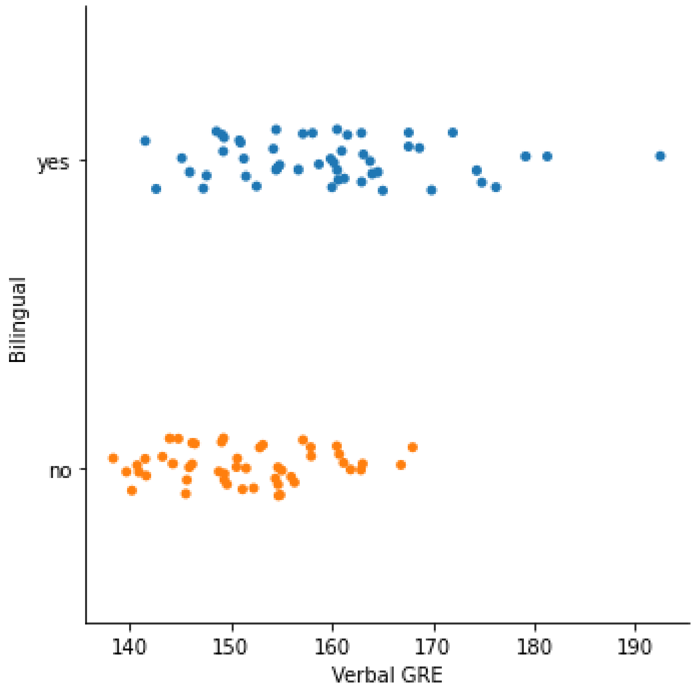
```
N = 100
bilang = ["yes", "no"]
vGRE = 151
qGRE = 151
sd = 8.5
bilingual = []
name = 'Student '
idx = []
verbal_GRE = []
quant_GRE = []
for i in range(1, N + 1):
    bilingual.append(bilang[np.random.randint(2,)])
    idx.append(name + " " + str(i))
    verbal_GRE.append(vGRE + sd*np.random.randn())
    quant_GRE.append(qGRE + sd*np.random.randn())
    if bilingual[-1] == 'yes':
        verbal_GRE[-1] += 3*np.random.randint(5)
        quant_GRE[-1] += 3*np.random.randint(5)
#verbal_GRE = np.int64(vGRE + sd*np.random.randn(N ,))
#quant_GRE = np.int64(qGRE + sd*np.random.randn(N ,))

df_dict = {'Bilingual' : bilingual,
           'Verbal GRE' : verbal_GRE,
           'Quant GRE' : quant_GRE
           }
new_df = pd.DataFrame(df_dict)
new_df.index = idx
new_df
```

Out[3]:

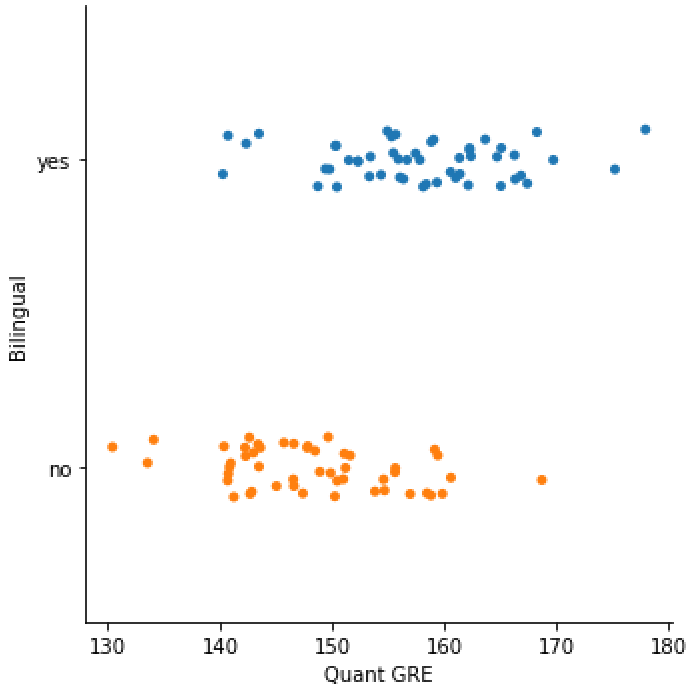
	Bilingual	Verbal GRE	Quant GRE
Student 1	yes	154.444423	155.675815
Student 2	no	160.460967	147.790824
Student 3	yes	149.288668	143.475395
Student 4	yes	167.580101	159.364194
Student 5	no	140.194517	146.598670
...	...	...	...
Student 96	yes	154.176309	157.830526
Student 97	no	154.633148	145.038432
Student 98	yes	163.969972	158.400575
Student 99	yes	159.988640	153.335840
Student 100	no	145.599936	151.007994

100 rows × 3 columns

In [9]:
```python
sns.catplot(data=new_df, x='Verbal GRE', y='Bilingual');
```



In [10]:
```python
sns.catplot(data=new_df, x='Quant GRE', y='Bilingual');
```



In [6]:
```
new_df.groupby('Bilingual')[['Verbal GRE', 'Quant GRE']].mean()
```

Out[6]:

	Verbal GRE	Quant GRE
Bilingual		
no	151.224247	148.197946
yes	159.636419	157.818439

In [7]:
```
new_df.groupby('Bilingual')[['Verbal GRE', 'Quant GRE']].sem()
```

Out[7]:

	Verbal GRE	Quant GRE
Bilingual		
no	1.064314	1.102824
yes	1.470406	1.122460

In [ ]:
```

```