Pandas, indexing and other advanced data manipulation features

The past few tutorials were focussed on `Pandas`. 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.

This tutorial focuses on more advanced `pandas` options to accessing, addressing (indexing) and manipulating data.

Learning goals:

- advanced pandas objects methods – the "verbs" that make them do useful things
- indexing and accessing row/column subsets fo data
- grouped data: aggregation and pivot tables

Make a data frame to play with

To get started this time instead of loading data from file, we will build a little data frame and take look at it to remind ourselves of this structure. We'll build a data frame similar to a data set mentioned in a previosu tutorial.

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

In [1]:

```python
import pandas as pd
import numpy as np     # to make the simulated data
```

Now we can make the data frame. It will have 4 variables of cardiovascular data for a number of patients (the number of patiencts can be specified):

- systolic blood pressure
- diastolic blood pressure
- blood oxygenation
- pulse rate

Given that Pandas `DataFrame` s have a special `index` column, we'll just use the `index` as "patient ID" instead of making a fifth variable dedicated to it.

In [2]:

```python
num_patients = 10    # specify the number of patients
```

We will use `Numpy` to simulate data by choosing a mean for each variable and a standard deviation. More specifically, the systolic blood pressure will have a mean of `125` and a standard deviation of `5` . The diastolic pressure will have a lower mean (`80`) but the same standard deviation, the blood oxygenation will have a mean of `98.5` and a smaller standard deviation of `0.3` . Finally, the pulse rate will have a mean of `65` abd a standard deviation of `2` .

In [3]:

```python
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,))
```

We will build the data frame using a dictionary:

In [4]:

```python
# 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 dictionary
```

And now lets look at it.

In [5]:

```python
our_df
```

Out[5]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
0	124	73	98.67	66
1	136	90	98.44	65
2	113	82	98.41	66
3	120	82	98.83	62
4	122	77	98.73	65
5	116	84	97.87	64
6	132	78	98.44	66
7	117	82	98.81	66
8	128	80	98.59	67
9	121	81	98.60	64

Complete the following exercise.

- Use the cell below to create a dataframe with the following data:
  - 16 patients
  - systolic blood pressure 10% higher than the current
  - diastolic blood pressure 5% lower
  - blood oxygenation 2% higher
  - a 4% higher pulse rate

In [6]:

```
num_patients_2 = 16
sys_bp_2 = np.int64(125  + 5*np.random.randn(num_patients_2,))
dia_bp_2 = np.int64(80 + 5*np.random.randn(num_patients_2,))
b_oxy_2  = np.round(98.5 + 0.3*np.random.randn(num_patients_2,), 2)
pulse_2  = np.int64(65   + 2*np.random.randn(num_patients_2,))
```

In [7]:

```
df_dict_2 = {'systolic BP' : sys_bp_2 * 1.1,
           'diastolic BP' : dia_bp_2 * 0.95,
           'blood oxygenation' : b_oxy_2 * 1.02,
           'pulse rate' : pulse_2 * 1.04
           }

our_df_2 = pd.DataFrame(df_dict_2)
```

In [8]:
```
our_df_2
```

Out[8]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
0	133.1	80.75	100.3272	69.68
1	138.6	70.30	100.8678	66.56
2	143.0	76.95	100.1946	67.60
3	137.5	84.55	100.5618	67.60
4	132.0	82.65	99.8682	67.60
5	135.3	83.60	100.0110	65.52
6	145.2	78.85	100.6740	65.52
7	139.7	70.30	100.7352	67.60
8	138.6	84.55	100.2456	65.52
9	148.5	71.25	100.3782	67.60
10	136.4	74.10	100.0926	71.76
11	136.4	70.30	100.6740	68.64
12	143.0	79.80	100.7556	67.60
13	145.2	72.20	99.9396	68.64
14	143.0	76.00	100.3476	65.52
15	137.5	71.25	100.3884	67.60

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 [9]:
```
our_array = np.transpose(np.vstack((sys_bp, dia_bp, b_oxy, pulse)))
our_array
```

Out[9]:
```
array([[124.  ,  73.  ,  98.67,  66.  ],
       [136.  ,  90.  ,  98.44,  65.  ],
       [113.  ,  82.  ,  98.41,  66.  ],
       [120.  ,  82.  ,  98.83,  62.  ],
       [122.  ,  77.  ,  98.73,  65.  ],
       [116.  ,  84.  ,  97.87,  64.  ],
       [132.  ,  78.  ,  98.44,  66.  ],
       [117.  ,  82.  ,  98.81,  66.  ],
       [128.  ,  80.  ,  98.59,  67.  ],
       [121.  ,  81.  ,  98.6 ,  64.  ]])
```

Complete the following exercise.

- Explore what `.vstack` does, use the `markdown` cell below to explain what it does in your own words

The `.vstack` function is used to stack arrays in sequence vertically.

In [12]:

```
t_1 = np.int64(100  + 5*np.random.randn(10,))
t_2 = np.int64(120   + 5*np.random.randn(10,))

df_dict_3 = {'Test 1' : t_1,
           'Test 2' : t_2
           }

our_df_3 = pd.DataFrame(df_dict_3)
our_df_3
```

Out[12]:

	Test 1	Test 2
0	95	122
1	102	122
2	94	117
3	108	121
4	99	120
5	102	120
6	102	112
7	101	113
8	93	119
9	98	120

In [11]:
```python
our_array_2 = np.transpose(np.vstack((t_1, t_2)))
our_array_2
```

Out[11]:
```
array([[ 93, 128],
       [101, 118],
       [ 97, 118],
       [ 94, 124],
       [101, 119],
       [107, 114],
       [105, 123],
       [109, 117],
       [103, 115],
       [ 98, 121]])
```

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 both variables should *know* how to take a mean. Let's see.

In [13]:
```python
our_array.mean()
```

Out[13]:
```
91.85974999999999
```

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 [14]:
```
our_array.mean(axis=0)
```

Out[14]:
```
array([122.9  ,  80.9  ,  98.539,  65.1  ])
```

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

In [15]:
```
our_df.mean()
```

Out[15]:
```
systolic BP          122.900
diastolic BP          80.900
blood oxygenation     98.539
pulse rate            65.100
dtype: float64
```

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

Data frames can also `describe()` themselves.

In [16]:
```
our_df.describe()
```

Out[16]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
count	10.000000	10.000000	10.000000	10.000000
mean	122.900000	80.900000	98.539000	65.100000
std	7.264067	4.508018	0.279263	1.449138
min	113.000000	73.000000	97.870000	62.000000
25%	117.750000	78.500000	98.440000	64.250000
50%	121.500000	81.500000	98.595000	65.500000
75%	127.000000	82.000000	98.715000	66.000000
max	136.000000	90.000000	98.830000	67.000000

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

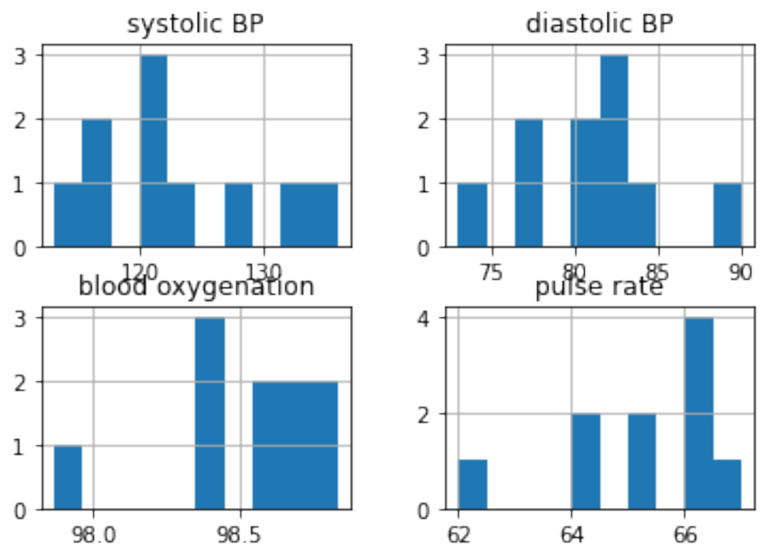
Numpy arrays don't know how to do this.

In [17]:
```
our_array.describe()
```

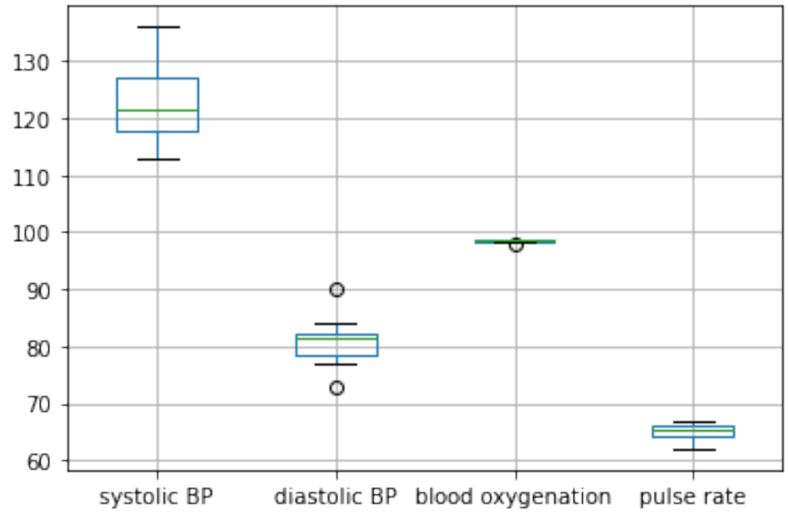
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [17], in <cell line: 1>()
----> 1 our_array.describe()

AttributeError: 'numpy.ndarray' object has no attribute '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 [18]:
```
our_df.hist();
```



In [19]:
```
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 [21]:
```
our_df.plot
```
Out[21]:
```
<pandas.plotting._core.PlotAccessor object at 0x7fd0bc536610>
```

In [22]:
```
our_df.pivot
```
Out[22]:
```
<bound method DataFrame.pivot of    systolic BP  diastolic BP  blood oxygenation  pulse rate
0          124            73              98.67          66
1          136            90              98.44          65
2          113            82              98.41          66
3          120            82              98.83          62
4          122            77              98.73          65
5          116            84              97.87          64
6          132            78              98.44          66
7          117            82              98.81          66
8          128            80              98.59          67
9          121            81              98.60          64>
```

Complete the following exercise.

- Use the next cell to report and describe two methods of `our_df` , explain why you chose those two.

I chose `our_df.plot` and `our_df.pivot` because I think it is interesting the what both function do since they sound very familiar to what we did back in the previous assignments.

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 [23]:
```
our_means = our_df.mean()
our_means
```

Out[23]:
```
systolic BP          122.900
diastolic BP          80.900
blood oxygenation     98.539
pulse rate            65.100
dtype: float64
```

In [24]:
```
type(our_means)
```

Out[24]:
```
pandas.core.series.Series
```

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 [25]:
```
our_means['pulse rate']
```

Out[25]:
```
65.1
```

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

Complete the following exercise.

- Use the cell below to return the diastolic blood pressure from `our_means`

In [26]:
```
our_means['diastolic BP']
```

Out[26]:
```
80.9
```

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 [29]:
```
np_style_means = our_array.mean(axis = 0)
pulse_mean = np_style_means[3]
pulse_mean
```

Out[29]: 65.1

Compare that to doing it the pandas way:

In [28]:
```
our_means = our_df.mean()
our_means['pulse rate']
```

Out[28]: 65.1

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

Complete the following exercise.

- Use the cell below to compute the mean of the `diastolic pressure` both using the `numpy` method and the `pandas` method:

In [30]:
```
# Pandas
np_style_means = our_array.mean(axis = 0)
d_p = np_style_means[1]
d_p
```

Out[30]: 80.9

In [31]:

```
# Numpy
our_means = our_df.mean()
our_means['diastolic BP']
```

Out[31]: 80.9

Accessing data using square brackets

Let's look ot our litte data frame again.

In [32]:

```
our_df
```

Out[32]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
0	124	73	98.67	66
1	136	90	98.44	65
2	113	82	98.41	66
3	120	82	98.83	62
4	122	77	98.73	65
5	116	84	97.87	64
6	132	78	98.44	66
7	117	82	98.81	66
8	128	80	98.59	67
9	121	81	98.60	64

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

In [33]:
```
our_df['pulse rate']
```

Out[33]:
```
0    66
1    65
2    66
3    62
4    65
5    64
6    66
7    66
8    67
9    64
Name: pulse rate, dtype: int64
```

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 [34]:
```
our_df['pulse rate'].mean()   # creates a series, then makes it compute its own mean
```

Out[34]:
```
65.1
```

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

In [35]:

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

Out[35]:

	diastolic BP	systolic BP
0	73	124
1	90	136
2	82	113
3	82	120
4	77	122
5	84	116
6	78	132
7	82	117
8	80	128
9	81	121

We could also do this in one step.

In [36]:
```
our_df[['diastolic BP', 'systolic BP']]  # the inner brackets define our list
```

Out[36]:

	diastolic BP	systolic BP
0	73	124
1	90	136
2	82	113
3	82	120
4	77	122
5	84	116
6	78	132
7	82	117
8	80	128
9	81	121

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

Complete the following exercise.

- Use the cell below to extract blood oxygenation and pulse rate using a single line of code

In [37]:
```
our_df[['blood oxygenation', 'pulse rate']]
```

Out[37]:

	blood oxygenation	pulse rate
0	98.67	66
1	98.44	65
2	98.41	66
3	98.83	62
4	98.73	65
5	97.87	64
6	98.44	66
7	98.81	66
8	98.59	67
9	98.60	64

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 [50]:
```
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 really necessary, l
```

Let's take a look at this index:

In [51]:
```
print(my_ind)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

In [52]:
```
our_df.index = my_ind
```

In [53]:
```
our_df # Creating New Index
```

Out[53]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
1	124	73	98.67	66
2	136	90	98.44	65
3	113	82	98.41	66
4	120	82	98.83	62
5	122	77	98.73	65
6	116	84	97.87	64
7	132	78	98.44	66
8	117	82	98.81	66
9	128	80	98.59	67
10	121	81	98.60	64

Complete the following exercise.

- Use the next cell to create a new index variable using numpy the variable should start at 5 and cintinue to 15 with 10 steps in between

In [44]:
```
my_ind_2 = np.linspace(5, 15, 10)
my_ind_2 = np.int64(my_ind_2)
my_ind_2
```

Out[44]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 15])
```

In [47]:
```
our_df.index = my_ind_2
our_df
```

Out[47]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
5	124	73	98.67	66
6	136	90	98.44	65
7	113	82	98.41	66
8	120	82	98.83	62
9	122	77	98.73	65
10	116	84	97.87	64
11	132	78	98.44	66
12	117	82	98.81	66
13	128	80	98.59	67
15	121	81	98.60	64

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 [54]:
```
our_df.loc[3]
```

Out[54]:
```
systolic BP          113.00
diastolic BP          82.00
blood oxygenation     98.41
pulse rate            66.00
Name: 3, dtype: float64
```

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 [55]:
```
our_df.loc[3:6]
```

Out[55]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
3	113	82	98.41	66
4	120	82	98.83	62
5	122	77	98.73	65
6	116	84	97.87	64

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 [56]:
```
our_df.loc[3:6, 'blood oxygenation']
```

Out[56]:
```
3    98.41
4    98.83
5    98.73
6    97.87
Name: blood oxygenation, dtype: float64
```

For a single column, or:

In [57]:
```
our_df.loc[3:6,'systolic BP':'blood oxygenation']
```

Out[57]:

	systolic BP	diastolic BP	blood oxygenation
3	113	82	98.41
4	120	82	98.83
5	122	77	98.73
6	116	84	97.87

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 later on), or just plain strings (as we've seen above with `Series` objects).

Complete the following exercise.

- Use the next cell to create a data frame of heart measurements where the index is the name of the patients (name and surname, make them up!):

In [74]:

```python
heart_rate = {'Jack': 15,
              'Samira': 14,
              'Samantha': 13,
              'Noah': 13,
              'Jay': 10,
              'Kate': 11,
              'Billy': 12,
              'Tommy': 12,
              'Christ': 9,
              'Nick': 13
             }
heart_rate_Series = pd.Series(heart_rate)

heart_rate_set = {'Heart Measurements': heart_rate_Series}

heart_rate_table = pd.DataFrame(heart_rate_set)

heart_rate_table
```

Out[74]:

	Heart Measurements
Jack	15
Samira	14
Samantha	13
Noah	13
Jay	10
Kate	11
Billy	12
Tommy	12
Christ	9
Nick	13

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

In [75]:
```
our_sum = our_df.describe()
our_sum
```

Out[75]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
count	10.000000	10.000000	10.000000	10.000000
mean	122.900000	80.900000	98.539000	65.100000
std	7.264067	4.508018	0.279263	1.449138
min	113.000000	73.000000	97.870000	62.000000
25%	117.750000	78.500000	98.440000	64.250000
50%	121.500000	81.500000	98.595000	65.500000
75%	127.000000	82.000000	98.715000	66.000000
max	136.000000	90.000000	98.830000	67.000000

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

In [76]:
```
type(our_sum)
```

Out[76]:
```
pandas.core.frame.DataFrame
```

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

In [77]:
```
our_sum.loc['mean']
```

Out[77]:
```
systolic BP          122.900
diastolic BP          80.900
blood oxygenation     98.539
pulse rate            65.100
Name: mean, dtype: float64
```

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 [78]:
```
our_sum['mean']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/indexes/base.py:3621, in Index.get_loc(self, key, method, tolerance)
   3620 try:
-> 3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/_libs/index.pyx:136, in pandas._libs.index.IndexEngine.get_loc()

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/_libs/index.pyx:163, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:5198, in pandas._libs.hashtable.PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:5206, in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'mean'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Input In [78], in <cell line: 1>()
----> 1 our_sum['mean']

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/frame.py:3505, in DataFrame.__getitem__(self, key)
   3503 if self.columns.nlevels > 1:
   3504     return self._getitem_multilevel(key)
-> 3505 indexer = self.columns.get_loc(key)
   3506 if is_integer(indexer):
```

```
   3507     indexer = [indexer]

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/indexes/base.py:3623, in Index.get
_loc(self, key, method, tolerance)
   3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:
-> 3623     raise KeyError(key) from err
   3624 except TypeError:
   3625     # If we have a listlike key, _check_indexing_error will raise
   3626     #  InvalidIndexError. Otherwise we fall through and re-raise
   3627     #  the TypeError.
   3628     self._check_indexing_error(key)

KeyError: '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 [79]:
```
our_sum.loc['count':'std']
```

Out[79]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
count	10.000000	10.000000	10.000000	10.000000
mean	122.900000	80.900000	98.539000	65.100000
std	7.264067	4.508018	0.279263	1.449138

Or rows and columns:

In [80]:
```
our_sum.loc['count':'std', 'systolic BP':'diastolic BP']
```
Out[80]:

	systolic BP	diastolic BP
count	10.000000	10.000000
mean	122.900000	80.900000
std	7.264067	4.508018

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 [81]:

```
our_df
```

Out[81]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
1	124	73	98.67	66
2	136	90	98.44	65
3	113	82	98.41	66
4	120	82	98.83	62
5	122	77	98.73	65
6	116	84	97.87	64
7	132	78	98.44	66
8	117	82	98.81	66
9	128	80	98.59	67
10	121	81	98.60	64

And let's check its shape:

In [82]:

```
our_df.shape
```

Out[82]:

```
(10, 4)
```

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 [83]:
```
our_df.iloc[3]  # get the fourth row
```

Out[83]:
```
systolic BP          120.00
diastolic BP          82.00
blood oxygenation     98.83
pulse rate            62.00
Name: 4, dtype: float64
```

And compare that to using `loc[]` :

In [84]:
```
our_df.loc[3]
```

Out[84]:
```
systolic BP          113.00
diastolic BP          82.00
blood oxygenation     98.41
pulse rate            66.00
Name: 3, dtype: float64
```

And of course you can slice out rows and columns:

In [85]:
```
our_df.iloc[2:5, 0:2]
```

Out[85]:

	systolic BP	diastolic BP
3	113	82
4	120	82
5	122	77

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 [86]:
```
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 [87]:
```
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 [88]:
```
diabetic = pd.Series(['yes', 'no'])  # make the short series
diabetic = diabetic.repeat(num_patients/2)        # repeat each over two cell's worth of data
diabetic = diabetic.reset_index(drop=True)      # reset the series's index value
```

In [89]:
```
print(diabetic)
```

```
0     yes
1     yes
2     yes
3     yes
4     yes
5     yes
6     yes
7     yes
8     yes
9     yes
10     no
11     no
12     no
13     no
14     no
15     no
16     no
17     no
18     no
19     no
dtype: object
```

Now will make an "inner" sex variable.

In [90]:
```
sex = pd.Series(['male', 'female'])             # make the short series
```

In [91]:
```
print(sex)
```

```
0      male
1    female
dtype: object
```

In [92]:

```
sex = sex.repeat(num_patients/4)                    # repeat each over one cell's worth of data
```

In [93]:

```
print(sex)
```

```
0      male
0      male
0      male
0      male
0      male
1    female
1    female
1    female
1    female
1    female
dtype: object
```

In [94]:

```
sex = pd.concat([sex]*2, ignore_index=True)   # stack or "concatenate" two copies
```

In [95]:

```
print(sex)
```

```
0       male
1       male
2       male
3       male
4       male
5     female
6     female
7     female
8     female
9     female
10      male
11      male
12      male
13      male
14      male
15    female
16    female
17    female
18    female
19    female
dtype: object
```

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

In [96]:

```
# 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,
           'sex': sex,
           'diabetes': diabetic
           }

```

And turn it into a data frame.

In [97]:
```
new_df = pd.DataFrame(df_dict)    # Now make a data frame out of the dictionary
```

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

In [98]:
```
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 [99]:
```
new_df.index = my_index
```

Let's look at our creation!

In [100]:
```
new_df
```

Out[100]:

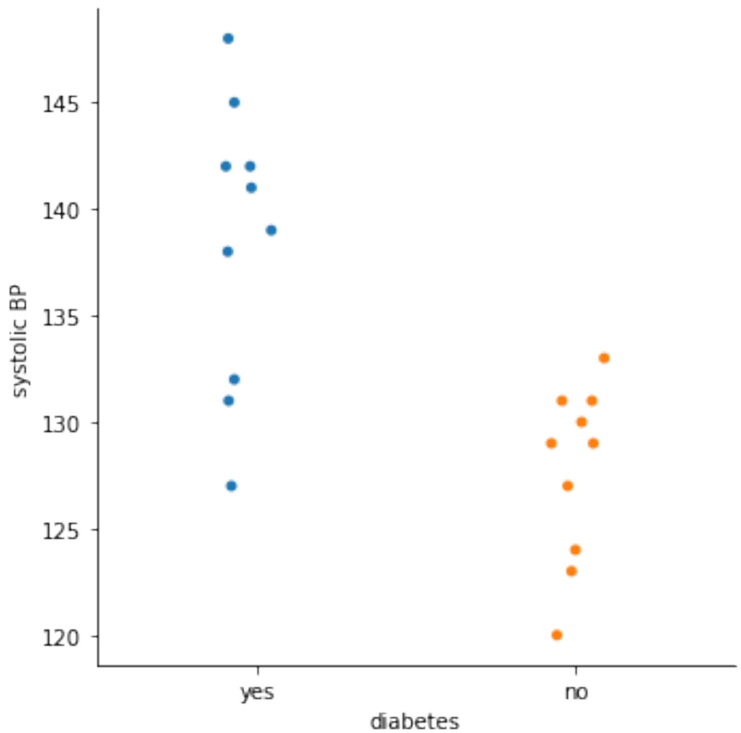
	systolic BP	diastolic BP	blood oxygenation	pulse rate	sex	diabetes
patient 1	148	103	98.74	64	male	yes
patient 2	139	93	98.22	64	male	yes
patient 3	142	95	99.08	64	male	yes
patient 4	132	95	98.26	64	male	yes
patient 5	145	94	98.79	64	male	yes
patient 6	127	90	98.17	64	female	yes
patient 7	131	98	98.48	61	female	yes
patient 8	138	101	98.31	63	female	yes
patient 9	142	90	97.63	66	female	yes
patient 10	141	95	98.53	64	female	yes
patient 11	130	82	98.34	65	male	no
patient 12	129	82	98.44	68	male	no
patient 13	129	88	99.08	65	male	no
patient 14	131	86	98.17	61	male	no
patient 15	133	89	98.49	63	male	no
patient 16	124	79	98.55	66	female	no
patient 17	131	78	98.70	63	female	no
patient 18	120	81	98.94	66	female	no
patient 19	123	82	98.85	64	female	no
patient 20	127	88	98.06	64	female	no

Looking at our data

Another really nice thing about pandas `DataFrames` is that they naturally lend themselves to interrogation via the visualization library `Seaborn` (we will learn about this library more in future tutorials).

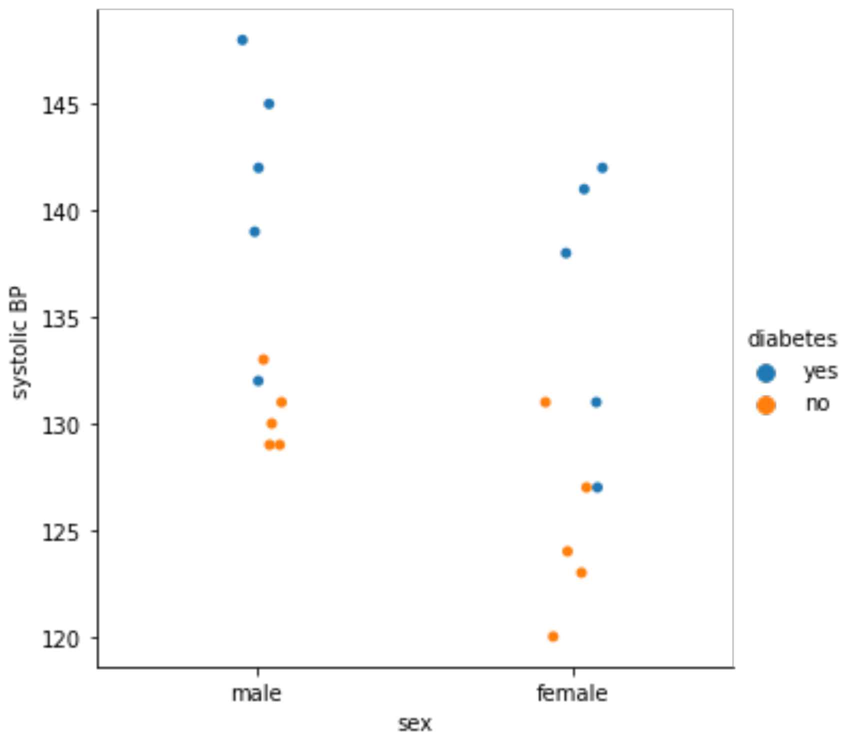
So let's peek at some stuff.

In [101]:
```
import seaborn as sns

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



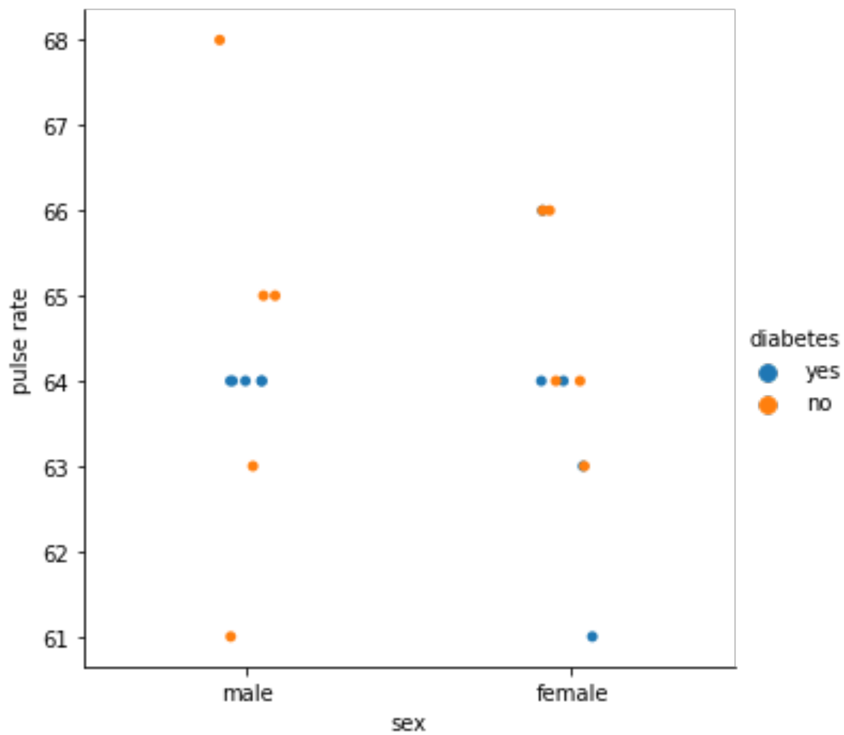
Okay, now let's go crazy and do a bunch of plots.

In [102]:
```
sns.catplot(data=new_df, x='sex', y='systolic BP', hue='diabetes');
```

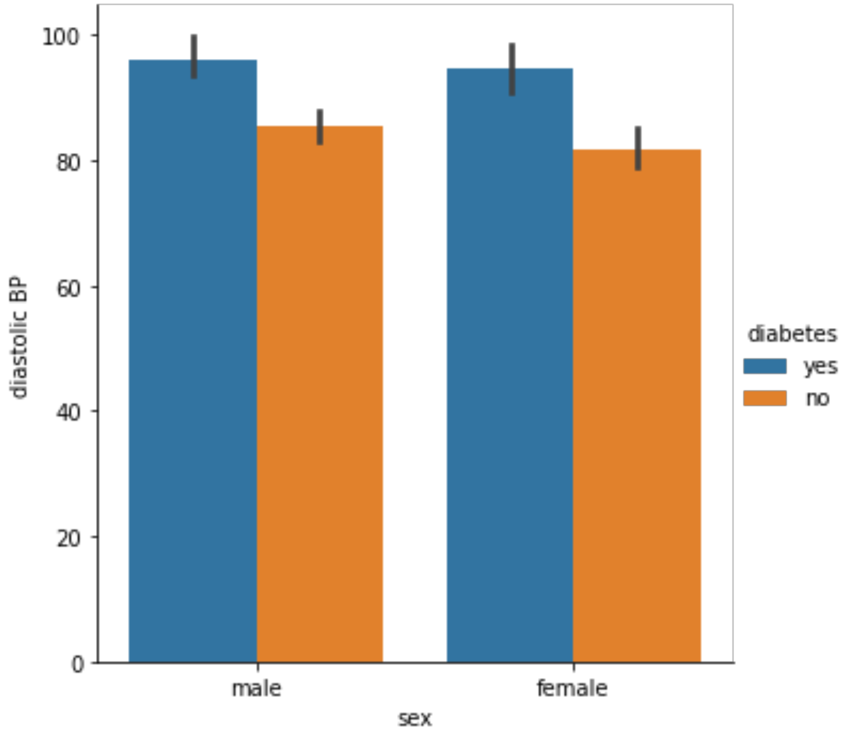


In [103]:

```
sns.catplot(data=new_df, x='sex', y='pulse rate', hue='diabetes');
```



In [104]:
```
sns.catplot(data=new_df, x='sex', y='diastolic BP', hue='diabetes', kind='bar');
```



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 [105]:
```
new_df.groupby('sex')
```

Out[105]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fd0a5b663a0>
```

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 [106]:
```
new_df.groupby('sex').mean()
```

Out[106]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
sex				
female	130.4	88.2	98.422	64.1
male	135.8	90.7	98.561	64.2

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

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

In [107]:

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

```
/var/folders/yq/3rc62cqs3nn_n_c8mm6k56jw0000gn/T/ipykernel_959/2935691488.py:1: FutureWarning:
['sex'] did not aggregate successfully. If any error is raised this will raise in a future vers
ion of pandas. Drop these columns/ops to avoid this warning.
  new_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])
```

Out[107]:

	systolic BP				**diastolic BP**				**blood oxygenation**				**pulse rate**			
	mean	**std**	**min**	**max**	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**
diabetes																
no	127.7	4.137901	120	133	83.5	3.951090	78	89	98.562	0.331388	98.06	99.08	64.5	1.957890	61	68
yes	138.5	6.620675	127	148	95.4	4.247875	90	103	98.421	0.402063	97.63	99.08	63.8	1.229273	61	66

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 [108]:

```python
temp_df = new_df[['systolic BP', 'diastolic BP', 'diabetes']]        # make a data frame wit
our_summary = temp_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])     # com
our_summary
```

Out[108]:

	systolic BP				**diastolic BP**			
	mean	**std**	**min**	**max**	**mean**	**std**	**min**	**max**
diabetes								
no	127.7	4.137901	120	133	83.5	3.951090	78	89
yes	138.5	6.620675	127	148	95.4	4.247875	90	103

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 [110]:
```
our_summary[("systolic BP", "mean")]
```

Out[110]:
```
diabetes
no     127.7
yes    138.5
Name: (systolic BP, mean), dtype: float64
```

In [111]:
```
our_summary.loc[("no")]
```

Out[111]:
```
systolic BP   mean    127.700000
              std       4.137901
              min     120.000000
              max     133.000000
diastolic BP  mean     83.500000
              std       3.951090
              min      78.000000
              max      89.000000
Name: no, dtype: float64
```

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

In [112]:

```
temp_df = new_df[['systolic BP', 'diabetes']]        # make a data frame with only the colum
our_summary = temp_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])     # com
our_summary
```

Out[112]:

	systolic BP			
	mean	std	min	max
diabetes				
no	127.7	4.137901	120	133
yes	138.5	6.620675	127	148

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 [113]:

```
our_summary.shape
```

Out[113]: (2, 4)

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 [114]:

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

Out[114]: 4.1379007023153935

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 [115]:
```
new_df[['systolic BP', 'diabetes']].groupby('diabetes').mean()
```

Out[115]:

	systolic BP
diabetes	
no	127.7
yes	138.5

Or we could group and then subset:

In [116]:
```
new_df.groupby('diabetes')[['systolic BP']].mean()
```

Out[116]:

	systolic BP
diabetes	
no	127.7
yes	138.5

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 [117]:
```
new_df.pivot_table('systolic BP', index='diabetes')
```

Out[117]:

	systolic BP
diabetes	
no	127.7
yes	138.5

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 [118]:
```
new_df.pivot_table('systolic BP', index='diabetes', columns='sex')
```

Out[118]:

sex	**female**	**male**
diabetes		
no	125.0	130.4
yes	135.8	141.2

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

In [120]:

```
new_df.pivot_table('systolic BP', index='diabetes', columns='sex', aggfunc='median')
```

Out[120]:

sex	female	male
diabetes		
no	124	130
yes	138	142

If you want to customize the column names using the aggregate function, you can (Though it is somewhat limited)! Look at the example down below for an explanation

In [121]:

```
new_df.groupby('diabetes').aggregate(Mean=('systolic BP',"mean"))
```

Out[121]:

	Mean
diabetes	
no	127.7
yes	138.5

The "Mean" is your new title, while inside the second set of parantheses is where/what you wantthe aggregate function to calculate

However, as you might have noticed, this is fairly limited. It removes the meta column titles, replacing them with the title of your choice. This can make it somewhat dificult to interpret your tables. Additionally, you can't have any spaces in the new title of your choice.

In [122]:

```
new_df.groupby('diabetes').aggregate(Mean=('systolic BP',"mean"),
                                     Standard_Deviation = ('systolic BP',"std"))
```

Out[122]:

	Mean	Standard_Deviation
diabetes		
no	127.7	4.137901
yes	138.5	6.620675

vs.

In [123]:

```
new_df.groupby('diabetes').aggregate( Mean=('systolic BP',"mean"), STD = ('systolic BP',"std'
```

Out[123]:

	Mean	STD
diabetes		
no	127.7	4.137901
yes	138.5	6.620675

(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

Complete the following exercise.

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 [145]:

```
num_student = 20

ver_GRE = np.int64(151  + 8.5*np.random.randn(num_student,))         # Creating GRE Data for
quant_GRE = np.int64(151   + 10*np.random.randn(num_student,))       # Creating GRE Data for

bilin = pd.Series(['yes', 'no'])
bilin = bilin.repeat(20/2)                                  # repeat each over two cell's wo
bilin = bilin.reset_index(drop=True)                        # Creting Bilingual Status

GRE_data = {'Verbal GRE' : ver_GRE,
           'Quant GRE' : quant_GRE,
            'Bilingual' : bilin
           }

GRE = pd.DataFrame(GRE_data)  # Make Data Frame
```

In [141]:

```
basename_GRE = 'Student '                          # make a "base" row name
my_index_stud = []                                 # make an empty list
for i in range(1, num_student+1) :                 # use a for loop to add
    my_index_stud.append(basename_GRE + str(i))     # i
```

In [144]:

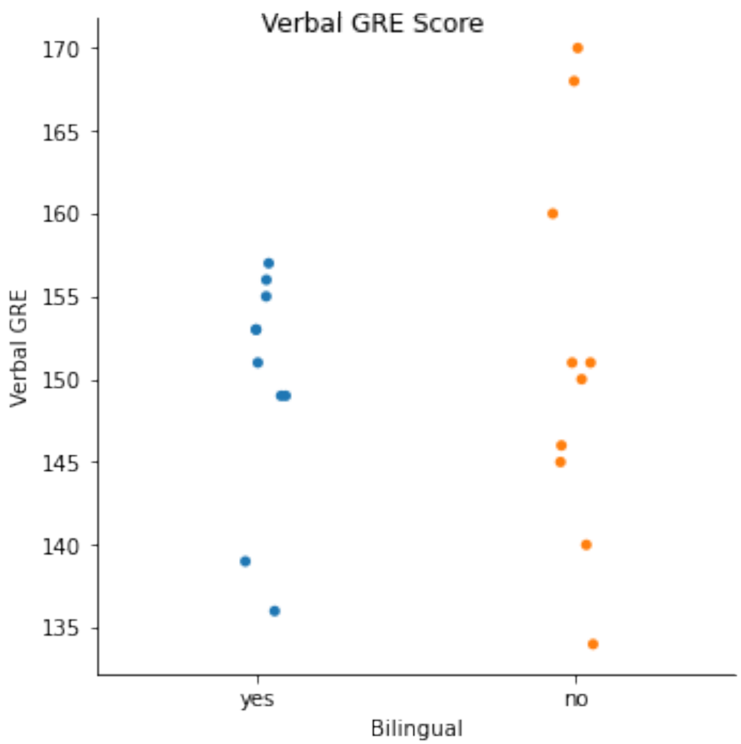
```
GRE.index = my_index_stud    # Adjusting index
```

In [143]:
```
GRE
```
Out[143]:

	Verbal GRE	Quant GRE	Bilingual
Student 1	175	169	yes
Student 2	161	149	yes
Student 3	153	149	yes
Student 4	164	132	yes
Student 5	146	145	yes
Student 6	154	138	yes
Student 7	140	181	yes
Student 8	158	149	yes
Student 9	148	156	yes
Student 10	160	142	yes
Student 11	143	136	no
Student 12	138	141	no
Student 13	151	147	no
Student 14	165	160	no
Student 15	173	132	no
Student 16	162	150	no
Student 17	146	148	no
Student 18	131	155	no
Student 19	160	138	no
Student 20	135	161	no

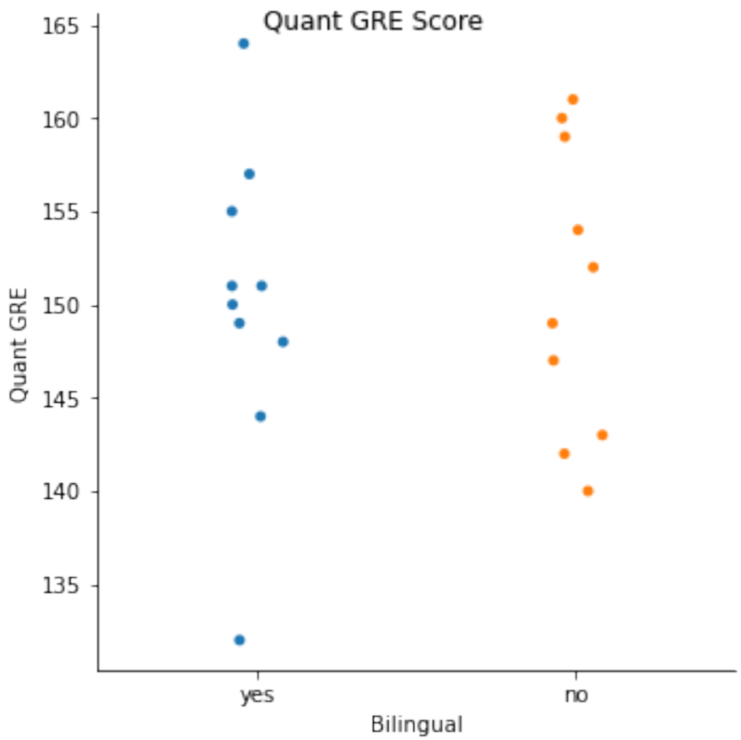
In [156]:
```
ver = sns.catplot(data=GRE, x='Bilingual', y='Verbal GRE');
ver.fig.suptitle('Verbal GRE Score')
```

Out[156]: Text(0.5, 0.98, 'Verbal GRE Score')



In [157]:
```python
quant = sns.catplot(data=GRE, x='Bilingual', y='Quant GRE');
quant.fig.suptitle('Quant GRE Score')
```

Out[157]: Text(0.5, 0.98, 'Quant GRE Score')



In [150]:
```python
# Verbal Score
GRE.groupby('Bilingual').aggregate( Mean=('Verbal GRE',"mean"), STD = ('Verbal GRE',"std"))
```

Out[150]:

	Mean	STD
Bilingual		
no	151.5	11.549411
yes	149.8	7.052186

In [151]:

```
# Quant Score
GRE.groupby('Bilingual').aggregate( Mean=('Quant GRE',"mean"), STD = ('Quant GRE',"std"))
```

Out[151]:

Bilingual	**Mean**	**STD**
no	150.7	7.746684
yes	150.1	8.412293