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]:
```
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]:
```
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 [11]:
```
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]:
```
# 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 di
```

And now lets look at it.

In [5]:
```
1 our_df
```

Out[5]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
0	126	84	98.95	63
1	117	86	98.31	63
2	127	78	98.09	67
3	126	70	98.58	64
4	130	81	97.60	65
5	128	80	98.68	68
6	130	81	98.40	62
7	125	87	98.52	63
8	133	79	98.60	63
9	118	81	98.44	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 [14]:
```
num_patients = 16
sys_bp1 = np.int64(((125*.10)+125)  + 5*np.random.randn(num_patients,))
dia_bp1 = np.int64((80-(80*.05))   + 5*np.random.randn(num_patients,))
b_oxy1  = np.round(((98.5*.02)+98.5) + 0.3*np.random.randn(num_patients
pulse1  = np.int64(((65*.04)+65)   + 2*np.random.randn(num_patients,))
df_dict1 = {'systolic BP' : sys_bp1,
            'diastolic BP' : dia_bp1,
            'blood oxygenation' : b_oxy1,
            'pulse rate' : pulse1
            }

our_df1 = pd.DataFrame(df_dict1)
our_df1
```

Out[14]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
0	132	77	100.70	66
1	134	82	100.25	66
2	128	80	100.36	66
3	140	82	100.08	69
4	139	65	100.68	67
5	141	65	100.27	68
6	132	74	101.06	69
7	142	72	100.57	69
8	134	75	100.36	67
9	135	81	100.98	70
10	129	79	100.40	69
11	140	69	100.58	68
12	138	80	100.78	68
13	135	75	100.79	67
14	140	77	100.27	68
15	139	81	100.53	67

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 [24]:
```
our_array = np.transpose(np.vstack((sys_bp, dia_bp, b_oxy, pulse)))
our_array
```

Out[24]:
```
array([[143.  , -84.  , 100.58,  65.  ],
       [140.  , -76.  , 100.02,  63.  ],
       [137.  , -75.  , 100.66,  67.  ],
       [142.  , -72.  , 100.68,  65.  ],
       [130.  , -74.  ,  99.82,  67.  ],
       [140.  , -75.  , 100.32,  64.  ],
       [137.  , -75.  , 100.2 ,  68.  ],
       [138.  , -81.  , 100.2 ,  68.  ],
       [143.  , -72.  , 101.03,  66.  ],
       [136.  , -75.  , 100.4 ,  67.  ],
       [138.  , -66.  , 100.29,  70.  ],
       [132.  , -77.  , 100.36,  71.  ],
       [137.  , -65.  , 100.49,  67.  ],
       [134.  , -73.  ,  99.83,  70.  ],
       [131.  , -77.  , 100.37,  70.  ],
       [128.  , -78.  , 100.61,  69.  ]])
```

In [16]:
```

```

```
  Input In [16]
    our_array1
    ^
SyntaxError: invalid syntax
```

Complete the following exercise.

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

Type *Markdown* and LaTeX: α^2

- Use the following code cell to show a few examples where you create a numpy array and use vstack to change it, explain why you use chose those operations as examples

In [23]:
```
help(numpy)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [23], in <cell line: 1>()
----> 1 help(numpy)

NameError: name 'numpy' is not defined
```

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 [25]:
```
1 our_array.mean()
```

Out[25]:
```
57.4040625
```

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 [26]:
```
1 our_array.mean(axis=0)
```

Out[26]:
```
array([136.625  , -74.6875 , 100.36625,  67.3125 ])
```

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

In [27]:
```
1 our_df.mean()
```

Out[27]:
```
systolic BP          126.000
diastolic BP          80.700
blood oxygenation     98.417
pulse rate            64.200
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 [28]:
```
1 our_df.describe()
```

Out[28]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
count	10.000000	10.000000	10.000000	10.000000
mean	126.000000	80.700000	98.417000	64.200000
std	5.077182	4.762119	0.366759	1.932184
min	117.000000	70.000000	97.600000	62.000000
25%	125.250000	79.250000	98.332500	63.000000
50%	126.500000	81.000000	98.480000	63.500000
75%	129.500000	83.250000	98.595000	64.750000
max	133.000000	87.000000	98.950000	68.000000

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

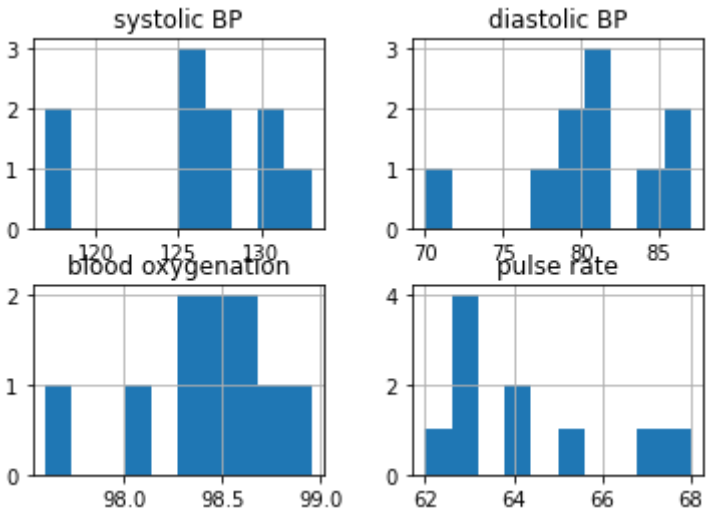
Numpy arrays don't know how to do this.

In [29]:
```
1 our_array.describe()
```

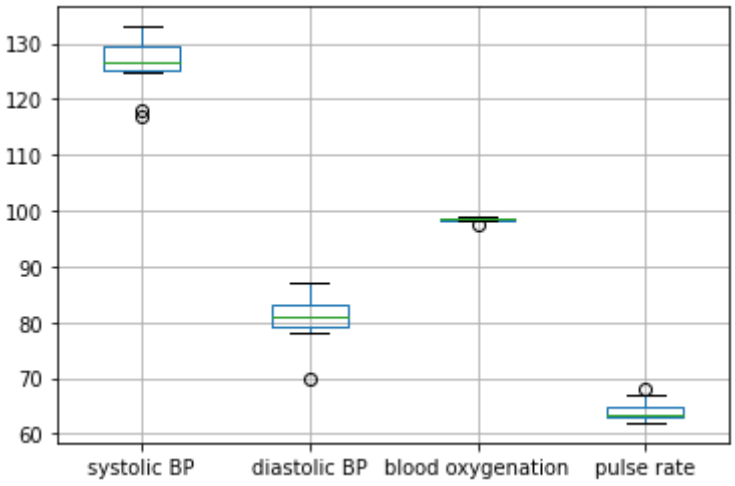
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [29], in <cell line: 1>()
----> 1 our_array.describe()

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 [30]:
```
1 our_df.hist();
```



In [31]:
```
1 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 [46]:
```
1 our_df.dtypes
```

Out[46]:
```
systolic BP            int64
diastolic BP           int64
blood oxygenation    float64
pulse rate             int64
dtype: object
```

Complete the following exercise.

- Use the next cell to report and describe two methods of `our_df` , explain why you chose those two. columns gives you the column names, this could be important for finding descriptors dtypes gives you the type of each variable group

Type *Markdown* and LaTeX: α^2

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 [47]:
```
1 our_means = our_df.mean()
2 our_means
```

Out[47]:
```
systolic BP          126.000
diastolic BP          80.700
blood oxygenation     98.417
pulse rate            64.200
dtype: float64
```

In [48]:
```
1 type(our_means)
```

Out[48]:
```
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 [49]:
```
1 our_means['pulse rate']
```

Out[49]:
```
64.2
```

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 [50]:
```
1 our_means['diastolic BP']
```

Out[50]: 80.7

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 [51]:
```
1 np_style_means = our_array.mean(axis = 0)
2 pulse_mean = np_style_means[3]
3 pulse_mean
```

Out[51]: 67.3125

Compare that to doing it the pandas way:

In [52]:
```
1 our_means = our_df.mean()
2 our_means['pulse rate']
```

Out[52]: 64.2

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 [55]:
```
1 np_style_means = our_array.mean(axis = 0)
2 print(np_style_means[1])
```

```
-74.6875
```

Accessing data using square brackets

Let's look ot our litte data frame again.

In [56]:
```
1 our_df
```

Out[56]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
0	126	84	98.95	63
1	117	86	98.31	63
2	127	78	98.09	67
3	126	70	98.58	64
4	130	81	97.60	65
5	128	80	98.68	68
6	130	81	98.40	62
7	125	87	98.52	63
8	133	79	98.60	63
9	118	81	98.44	64

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

In [57]:
```
1 our_df['pulse rate']
```

Out[57]:
```
0    63
1    63
2    67
3    64
4    65
5    68
6    62
7    63
8    63
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 [58]:
```
1 our_df['pulse rate'].mean()   # creates a series, then makes it compute
```

Out[58]:
```
64.2
```

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

In [59]:
```
needed_cols = ['diastolic BP', 'systolic BP']   # make a list
our_df[needed_cols]                             # use the list to grab
```

Out[59]:

	diastolic BP	systolic BP
0	84	126
1	86	117
2	78	127
3	70	126
4	81	130
5	80	128
6	81	130
7	87	125
8	79	133
9	81	118

We could also do this in one step.

In [60]:
```
our_df[['diastolic BP', 'systolic BP']]  # the inner brackets define ou
```

Out[60]:

	diastolic BP	systolic BP
0	84	126
1	86	117
2	78	127
3	70	126
4	81	130
5	80	128
6	81	130
7	87	125
8	79	133
9	81	118

(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 [62]:
```
our_df.columns
our_df[['blood oxygenation','pulse rate']]
```

Out[62]:

	blood oxygenation	pulse rate
0	98.95	63
1	98.31	63
2	98.09	67
3	98.58	64
4	97.60	65
5	98.68	68
6	98.40	62
7	98.52	63
8	98.60	63
9	98.44	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 [63]:
```
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 (n
```

Let's take a look at this index:

In [64]:
```
print(my_ind)
```

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

In [65]:
```
our_df.index = my_ind
```

In [66]:
```
our_df
```

Out[66]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
1	126	84	98.95	63
2	117	86	98.31	63
3	127	78	98.09	67
4	126	70	98.58	64
5	130	81	97.60	65
6	128	80	98.68	68
7	130	81	98.40	62
8	125	87	98.52	63
9	133	79	98.60	63
10	118	81	98.44	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 [72]:
```
my_ind = np.linspace(5, 15, 10)
my_ind = np.int64(my_ind)
my_ind
```

Out[72]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 15])
```

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 [73]:
```
1 our_df.loc[3]
```

Out[73]:
```
systolic BP          127.00
diastolic BP          78.00
blood oxygenation     98.09
pulse rate            67.00
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 [74]:
```
1 our_df.loc[3:6]
```

Out[74]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
3	127	78	98.09	67
4	126	70	98.58	64
5	130	81	97.60	65
6	128	80	98.68	68

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 [75]:
```
1 our_df.loc[3:6, 'blood oxygenation']
```

Out[75]:
```
3    98.09
4    98.58
5    97.60
6    98.68
Name: blood oxygenation, dtype: float64
```

For a single column, or:

In [76]:
```
1 our_df.loc[3:6,'systolic BP':'blood oxygenation']
```

Out[76]:

	systolic BP	diastolic BP	blood oxygenation
3	127	78	98.09
4	126	70	98.58
5	130	81	97.60
6	128	80	98.68

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 [91]:
```
Mean =134
SD = 12
p1 = np.int64(Mean  + SD*np.random.randn(10))
p2 = np.int64(Mean   + SD*np.random.randn(10))
p3 = np.int64(Mean + SD*np.random.randn(10))
p4  = np.int64(Mean   + SD*np.random.randn(10))



patients = {"Arthur Winslow": p1, "Clay Sheldon": p2, "William Artright
Patients = pd.DataFrame(patients)
Patients

```

Out[91]:

	Arthur Winslow	Clay Sheldon	William Artright	Severus Snape
0	129	120	134	127
1	124	131	159	147
2	156	124	130	129
3	119	156	156	126
4	134	125	135	146
5	155	153	132	125
6	163	134	127	134
7	141	140	144	112
8	125	118	138	159
9	132	150	131	125

In [93]:
```
Patients.loc[3:6, 'Arthur Winslow']
```

Out[93]:
```
3    119
4    134
5    155
6    163
Name: Arthur Winslow, dtype: int64
```

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

In [94]:
```
1 our_sum = our_df.describe()
2 our_sum
```

Out[94]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
count	10.000000	10.000000	10.000000	10.000000
mean	126.000000	80.700000	98.417000	64.200000
std	5.077182	4.762119	0.366759	1.932184
min	117.000000	70.000000	97.600000	62.000000
25%	125.250000	79.250000	98.332500	63.000000
50%	126.500000	81.000000	98.480000	63.500000
75%	129.500000	83.250000	98.595000	64.750000
max	133.000000	87.000000	98.950000	68.000000

In [95]:
```
1 the_sum = Patients.describe()
2 the_sum
```

Out[95]:

	Arthur Winslow	Clay Sheldon	William Artright	Severus Snape
count	10.000000	10.000000	10.000000	10.000000
mean	137.800000	135.100000	138.600000	133.000000
std	15.295606	14.011503	11.017158	13.792107
min	119.000000	118.000000	127.000000	112.000000
25%	126.000000	124.250000	131.250000	125.250000
50%	133.000000	132.500000	134.500000	128.000000
75%	151.500000	147.500000	142.500000	143.000000
max	163.000000	156.000000	159.000000	159.000000

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

In [96]:
```
1 type(our_sum)
```

Out[96]:
```
pandas.core.frame.DataFrame
```

In [98]:
```
1 type(the_sum)
```

Out[98]:
```
pandas.core.frame.DataFrame
```

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

In [99]:
```
1 the_sum.loc['mean']
```

Out[99]:
```
Arthur Winslow      137.8
Clay Sheldon        135.1
William Artright    138.6
Severus Snape       133.0
Name: mean, dtype: float64
```

In [97]:
```
1 our_sum.loc['mean']
```

Out[97]:
```
systolic BP          126.000
diastolic BP          80.700
blood oxygenation     98.417
pulse rate            64.200
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 [101]:
```
1 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 [101], 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 [102]:
```
1 our_sum.loc['count':'std']
```

Out[102]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
count	10.000000	10.000000	10.000000	10.000000
mean	126.000000	80.700000	98.417000	64.200000
std	5.077182	4.762119	0.366759	1.932184

Or rows and columns:

In [103]:
```
1 our_sum.loc['count':'std', 'systolic BP':'diastolic BP']
```

Out[103]:

	systolic BP	diastolic BP
count	10.000000	10.000000
mean	126.000000	80.700000
std	5.077182	4.762119

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 [104]:
```
1 our_df
```

Out[104]:

	systolic BP	diastolic BP	blood oxygenation	pulse rate
1	126	84	98.95	63
2	117	86	98.31	63
3	127	78	98.09	67
4	126	70	98.58	64
5	130	81	97.60	65
6	128	80	98.68	68
7	130	81	98.40	62
8	125	87	98.52	63
9	133	79	98.60	63
10	118	81	98.44	64

And let's check its shape:

In [105]:
```
1 Patients
```

Out[105]:

	Arthur Winslow	Clay Sheldon	William Artright	Severus Snape
0	129	120	134	127
1	124	131	159	147
2	156	124	130	129
3	119	156	156	126
4	134	125	135	146
5	155	153	132	125
6	163	134	127	134
7	141	140	144	112
8	125	118	138	159
9	132	150	131	125

In [106]:
```
1 our_df.shape
```

Out[106]: (10, 4)

In [107]:
```
1 Patients.shape
```

Out[107]:
```
(10, 4)
```

At some level, then, Python considers this to be just a 10x4 array (like a numpy array). This is where `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 [108]:
```
1 our_df.iloc[3]  # get the fourth row
```

Out[108]:
```
systolic BP          126.00
diastolic BP          70.00
blood oxygenation     98.58
pulse rate            64.00
Name: 4, dtype: float64
```

In [109]:
```
1 Patients.iloc[3]
```

Out[109]:
```
Arthur Winslow      119
Clay Sheldon        156
William Artright    156
Severus Snape       126
Name: 3, dtype: int64
```

In [110]:
```
1 Patients.loc[3]
```

Out[110]:
```
Arthur Winslow      119
Clay Sheldon        156
William Artright    156
Severus Snape       126
Name: 3, dtype: int64
```

And compare that to using `loc[]`:

In [111]:
```
1 our_df.loc[3]
```

Out[111]:
```
systolic BP          127.00
diastolic BP          78.00
blood oxygenation     98.09
pulse rate            67.00
Name: 3, dtype: float64
```

And of course you can slice out rows and columns:

In [112]:
```
1 our_df.iloc[2:5, 0:2]
```

Out[112]:

	systolic BP	diastolic BP
3	127	78
4	126	70
5	130	81

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 [113]:
```
1 num_patients = 20    # specify the number of patients
2
3 # make some simulated data with realistic numbers.
4 sys_bp = np.int64(125 + 5*np.random.randn(num_patients,))
5 dia_bp = np.int64(80 + 5*np.random.randn(num_patients,))
6 b_oxy = np.round(98.5 + 0.3*np.random.randn(num_patients,), 2)
7 pulse = np.int64(65 + 2*np.random.randn(num_patients,))
8
```

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

In [114]:
```
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 [115]:
```
diabetic = pd.Series(['yes', 'no'])  # make the short series
diabetic = diabetic.repeat(num_patients/2)       # repeat each over two
diabetic = diabetic.reset_index(drop=True)      # reset the series's in
```

In [116]:
```
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 [117]:
```
sex = pd.Series(['male', 'female'])             # make the short series
```

In [118]:
```
print(sex)
```

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

In [119]:
```
sex = sex.repeat(num_patients/4)                 # repeat each over one
```

In [120]:
```
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 [121]:
```
sex = pd.concat([sex]*2, ignore_index=True)   # stack or "concatenate"
```

In [122]:
```
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 [123]:
```
# 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 [124]:
```
new_df = pd.DataFrame(df_dict)    # Now make a data frame out of the di
```

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

In [125]:
```
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 nam
```

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

In [126]:
```
new_df.index = my_index
```

Let's look at our creation!

In [127]:
```
1 new_df
```

Out[127]:

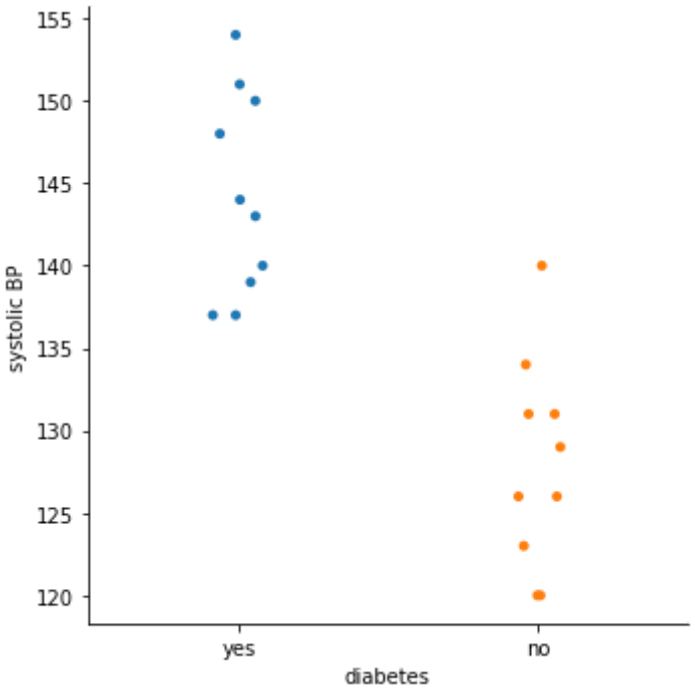
	systolic BP	diastolic BP	blood oxygenation	pulse rate	sex	diabetes
patient 1	154	97	98.35	66	male	yes
patient 2	150	103	98.56	64	male	yes
patient 3	144	92	98.97	64	male	yes
patient 4	148	99	98.18	64	male	yes
patient 5	151	104	98.78	60	male	yes
patient 6	139	90	97.89	64	female	yes
patient 7	143	95	98.84	61	female	yes
patient 8	140	96	98.20	66	female	yes
patient 9	137	95	98.48	66	female	yes
patient 10	137	97	98.42	69	female	yes
patient 11	123	89	98.67	60	male	no
patient 12	131	88	98.41	64	male	no
patient 13	126	86	98.43	65	male	no
patient 14	131	86	98.48	65	male	no
patient 15	134	85	98.22	65	male	no
patient 16	120	77	98.59	65	female	no
patient 17	129	76	98.53	64	female	no
patient 18	140	82	98.73	65	female	no
patient 19	120	87	98.58	61	female	no
patient 20	126	80	98.02	66	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).

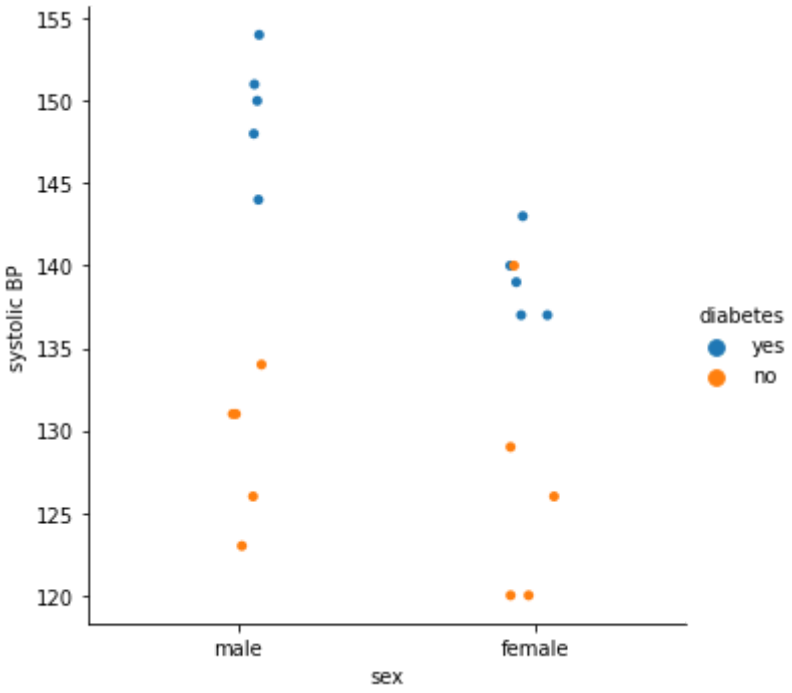
So let's peek at some stuff.

In [128]:
```
1 import seaborn as sns
2
3 sns.catplot(data=new_df, x='diabetes', y='systolic BP');
```

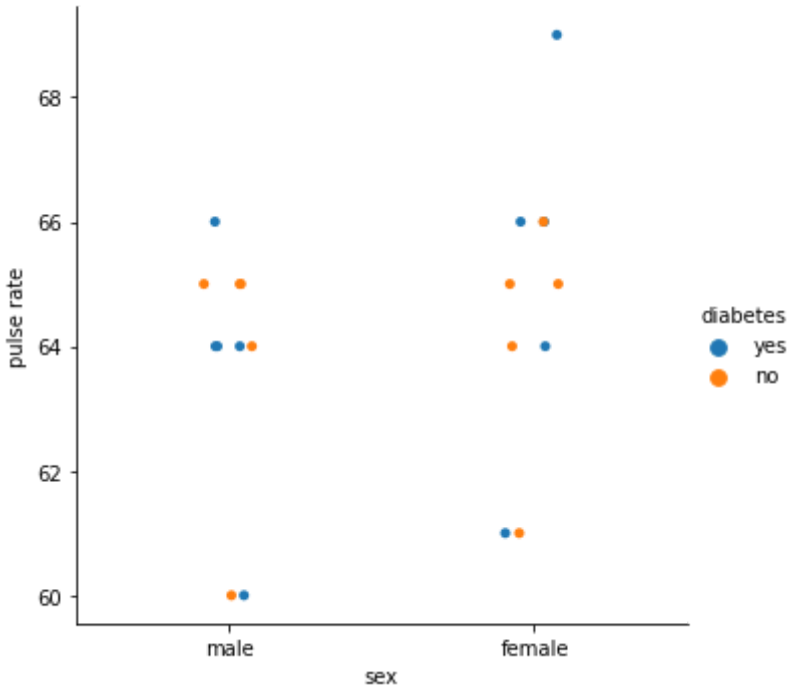


Okay, now let's go crazy and do a bunch of plots.

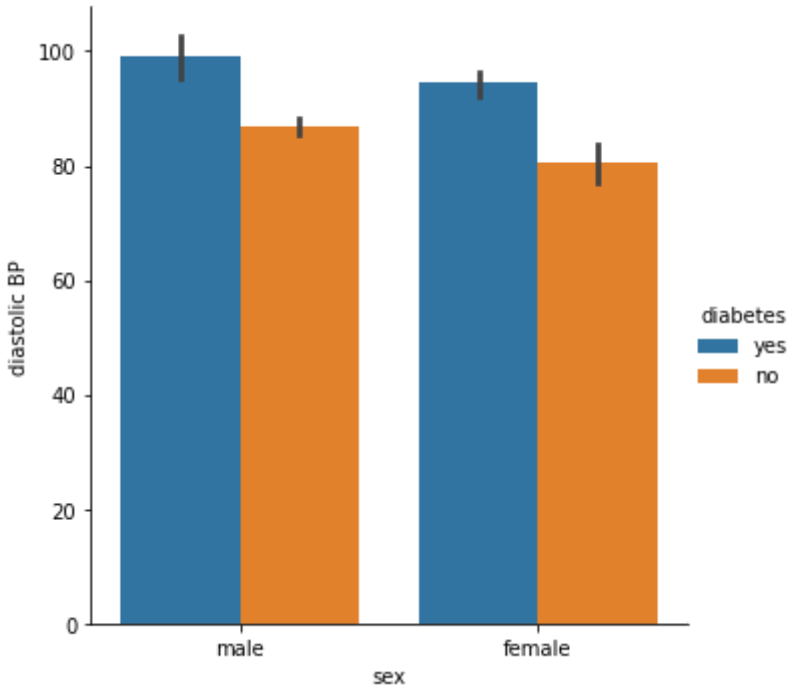
In [129]:
```
1 sns.catplot(data=new_df, x='sex', y='systolic BP', hue='diabetes');
```



In [130]:
```
1 sns.catplot(data=new_df, x='sex', y='pulse rate', hue='diabetes');
```



In [131]:
```
1 sns.catplot(data=new_df, x='sex', y='diastolic BP', hue='diabetes', kin
```



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 [132]:
```
1 new_df.groupby('sex')
```

Out[132]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fee10d43c40>
```

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 [133]:
```
1 new_df.groupby('sex').mean()
```

Out[133]:

sex	systolic BP	diastolic BP	blood oxygenation	pulse rate
female	133.1	87.5	98.428	64.7
male	139.2	92.9	98.505	63.7

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

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

In [134]:
```
1 new_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])
```

```
/var/folders/y2/12krz73x2mz0yfzf5kjb51_m0000gn/T/ipykernel_16436/2935691488.py:1: FutureWarning: ['sex'] did not aggregate successfully. If any error is raised this will raise in a future version of pandas. Drop these columns/ops to avoid this warning.
  new_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])
```

Out[134]:

	systolic BP				diastolic BP				blood oxygenation			
	mean	std	min	max	mean	std	min	max	mean	std	min	max
diabetes												
no	128.0	6.324555	120	140	83.6	4.599517	76	89	98.466	0.212875	98.02	98.73
yes	144.3	6.147267	137	154	96.8	4.366539	90	104	98.467	0.333568	97.89	98.97

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 [135]:
```
temp_df = new_df[['systolic BP', 'diastolic BP', 'diabetes']]        #
our_summary = temp_df.groupby('diabetes').aggregate(['mean', 'std', 'mi
our_summary
```

Out[135]:

	systolic BP				diastolic BP			
	mean	std	min	max	mean	std	min	max
diabetes								
no	128.0	6.324555	120	140	83.6	4.599517	76	89
yes	144.3	6.147267	137	154	96.8	4.366539	90	104

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 [136]:
```
our_summary[("systolic BP", "mean")]
```

Out[136]:
```
diabetes
no     128.0
yes    144.3
Name: (systolic BP, mean), dtype: float64
```

In [137]:
```
our_summary.loc[("no")]
```

Out[137]:
```
systolic BP   mean    128.000000
              std       6.324555
              min     120.000000
              max     140.000000
diastolic BP  mean     83.600000
              std       4.599517
              min      76.000000
              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 [138]:
```
temp_df = new_df[['systolic BP', 'diabetes']]        # make a data fram
our_summary = temp_df.groupby('diabetes').aggregate(['mean', 'std', 'mi
our_summary
```

Out[138]:

	systolic BP			
	mean	std	min	max
diabetes				
no	128.0	6.324555	120	140
yes	144.3	6.147267	137	154

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 [139]:
```
our_summary.shape
```

Out[139]:
```
(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 [140]:
```
our_summary.iloc[0, 1]
```

Out[140]:
```
6.32455532033676
```

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 [141]:
```
1 new_df[['systolic BP', 'diabetes']].groupby('diabetes').mean()
```

Out[141]:

diabetes	systolic BP
no	128.0
yes	144.3

Or we could group and then subset:

In [142]:
```
1 new_df.groupby('diabetes')[['systolic BP']].mean()
```

Out[142]:

diabetes	systolic BP
no	128.0
yes	144.3

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 [143]:
```
1 new_df.pivot_table('systolic BP', index='diabetes')
```

Out[143]:

diabetes	systolic BP
no	128.0
yes	144.3

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 [144]:
```
1 new_df.pivot_table('systolic BP', index='diabetes', columns='sex')
```

Out[144]:

sex	female	male
diabetes		
no	127.0	129.0
yes	139.2	149.4

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

In [145]:
```
1 new_df.pivot_table('systolic BP', index='diabetes', columns='sex', aggf
```

Out[145]:

sex	female	male
diabetes		
no	126	131
yes	139	150

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 [146]:
```
1 new_df.groupby('diabetes').aggregate(Mean=('systolic BP',"mean"))
```

Out[146]:

	Mean
diabetes	
no	128.0
yes	144.3

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 [147]:
```
new_df.groupby('diabetes').aggregate(Mean=('systolic BP',"mean"),
                                     Standard_Deviation = ('systolic B
```

Out[147]:

diabetes	Mean	Standard_Deviation
no	128.0	6.324555
yes	144.3	6.147267

vs.

In [148]:
```
new_df.groupby('diabetes').aggregate( Mean=('systolic BP',"mean"), STD
```

Out[148]:

diabetes	Mean	STD
no	128.0	6.324555
yes	144.3	6.147267

(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 [164]:
```
num_students = 50
Mean = 151
SD = 8.5
vGRE = np.int64(Mean  + SD*np.random.randn(num_students,))
qGRE = np.int64(Mean   + SD*np.random.randn(num_students,))
```

In [207]:
```


bilingual = pd.Series(["yes","no"])
bilingual = bilingual.repeat(num_students/10)
bilingual = pd.concat([bilingual]*5, ignore_index=True)

bilinguality = pd.Series(["beginner","intermediate","conversational", "
bilinguality = bilinguality.repeat(1)
bilinguality = pd.concat([bilinguality]*10, ignore_index=True)


```

In [218]:
```
basename = 'Student '                          # make a "base" row name
my_index = []                                  # make an empty list
for i in range(1,num_students+1) :             # use a for loop to add
    my_index.append(basename + str(i))
```

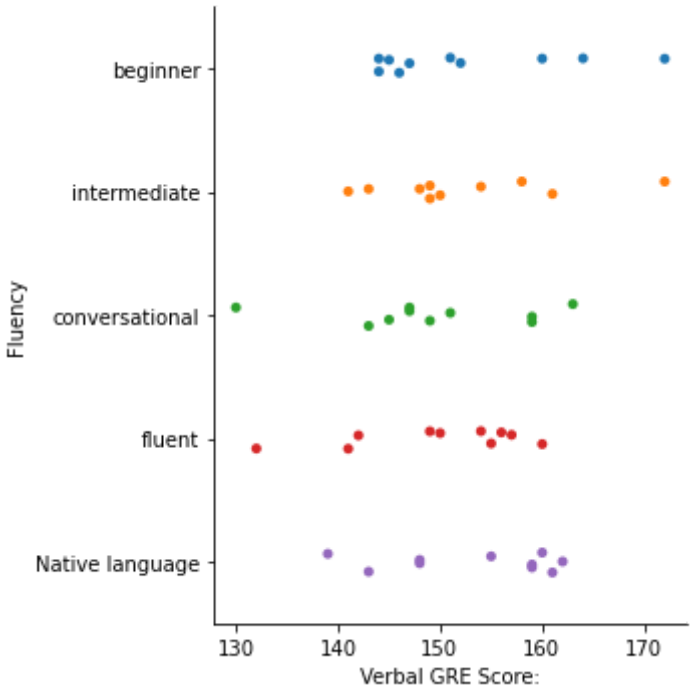
In [226]:
```
scores = {"Verbal GRE Score:": vGRE, "Quant GRE Score:": qGRE, "Bilingu
scores = pd.DataFrame(scores)
scores.index = my_index
scores
```

Out[226]:

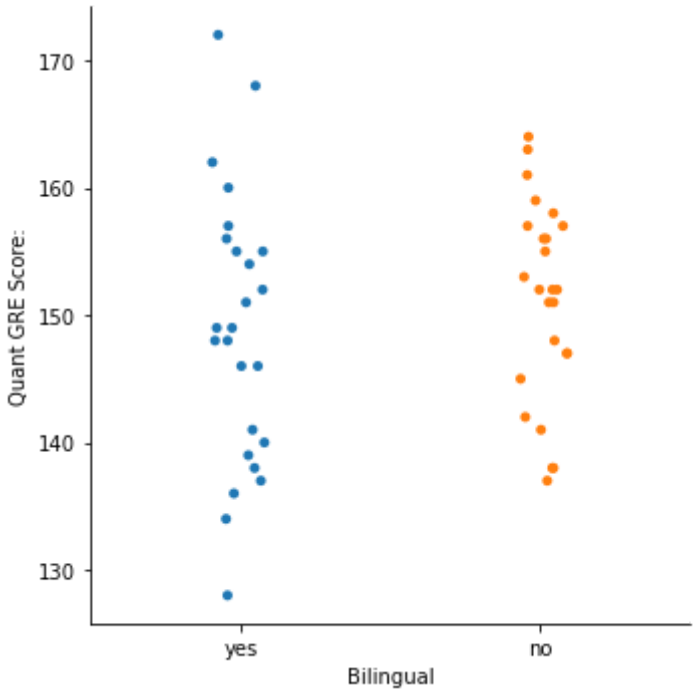
	Verbal GRE Score:	Quant GRE Score:	Bilingual	Fluency
Student 1	160	155	yes	beginner
Student 2	154	137	yes	intermediate
Student 3	159	162	yes	conversational
Student 4	150	152	yes	fluent
Student 5	159	172	yes	Native language
Student 6	147	147	no	beginner
Student 7	172	156	no	intermediate
Student 8	163	145	no	conversational
Student 9	141	151	no	fluent
Student 10	161	152	no	Native language
Student 11	164	148	yes	beginner
Student 12	141	134	yes	intermediate
Student 13	147	149	yes	conversational
Student 14	154	138	yes	fluent
Student 15	139	160	yes	Native language
Student 16	151	138	no	beginner
Student 17	148	155	no	intermediate
Student 18	143	152	no	conversational
Student 19	142	148	no	fluent
Student 20	143	141	no	Native language
Student 21	152	140	yes	beginner
Student 22	143	151	yes	intermediate
Student 23	147	128	yes	conversational
Student 24	155	141	yes	fluent
Student 25	159	154	yes	Native language
Student 26	144	142	no	beginner
Student 27	149	163	no	intermediate
Student 28	145	137	no	conversational
Student 29	149	156	no	fluent
Student 30	160	159	no	Native language

	Verbal GRE Score:	Quant GRE Score:	Bilingual	Fluency
Student 31	146	168	yes	beginner
Student 32	158	139	yes	intermediate
Student 33	130	149	yes	conversational
Student 34	157	156	yes	fluent
Student 35	148	157	yes	Native language
Student 36	172	138	no	beginner
Student 37	161	147	no	intermediate
Student 38	159	157	no	conversational
Student 39	132	157	no	fluent
Student 40	162	164	no	Native language
Student 41	144	148	yes	beginner
Student 42	149	146	yes	intermediate
Student 43	151	155	yes	conversational
Student 44	156	136	yes	fluent
Student 45	148	146	yes	Native language
Student 46	145	153	no	beginner
Student 47	150	151	no	intermediate
Student 48	149	161	no	conversational
Student 49	160	158	no	fluent
Student 50	155	152	no	Native language

In [222]:
```
1 sns.catplot(data=scores, x='Verbal GRE Score:', y='Fluency');
```



In [224]:
```
1 sns.catplot(data=scores, y='Quant GRE Score:', x='Bilingual');
```



In [228]:
```
1 scores.groupby('Verbal GRE Score:').aggregate(Mean=('Bilingual',"mean")
2                                               Standard_Deviation = ('Bilingual'
```

```
  Input In [228]
    Standard_Deviation = ('Bilingual',"std"))
    ^
IndentationError: unexpected indent
```

In [ ]:
```
1
```