tutorial018-Pandas-Advanced-WorkingWithData

November 18, 2022

# 1 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.

### 1.0.1 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

## 1.1 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.

[67]:
```
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.

[68]:
```
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`.

[69]:
```
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:

[70]:
```
# 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.

[71]:
```
our_df
```

[71]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| 0 | 127 | 77 | 98.97 | 65 |
| 1 | 125 | 88 | 98.59 | 63 |
| 2 | 122 | 80 | 98.23 | 65 |
| 3 | 129 | 79 | 98.54 | 63 |
| 4 | 132 | 77 | 98.54 | 63 |
| 5 | 131 | 81 | 97.95 | 63 |
| 6 | 121 | 86 | 98.84 | 63 |
| 7 | 128 | 82 | 99.05 | 65 |
| 8 | 129 | 86 | 98.50 | 67 |
| 9 | 126 | 71 | 97.70 | 65 |

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

[72]:
```
num_pts = 16

sys_bp_ex = sys_bp*1.1
dia_bp_ex = dia_bp*0.95
b_oxy_ex = b_oxy*1.02
pulse_ex = pulse*1.04

df_dict = {'systolic BP' : sys_bp_ex,
           'diastolic BP' : dia_bp_ex,
           'blood oxygenation' : b_oxy_ex,
           'pulse rate' : pulse_ex
           }

df2 = pd.DataFrame(df_dict)
df2
```

[72]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| 0 | 139.7 | 73.15 | 100.9494 | 67.60 |
| 1 | 137.5 | 83.60 | 100.5618 | 65.52 |
| 2 | 134.2 | 76.00 | 100.1946 | 67.60 |
| 3 | 141.9 | 75.05 | 100.5108 | 65.52 |
| 4 | 145.2 | 73.15 | 100.5108 | 65.52 |
| 5 | 144.1 | 76.95 | 99.9090 | 65.52 |
| 6 | 133.1 | 81.70 | 100.8168 | 65.52 |
| 7 | 140.8 | 77.90 | 101.0310 | 67.60 |
| 8 | 141.9 | 81.70 | 100.4700 | 69.68 |
| 9 | 138.6 | 67.45 | 99.6540 | 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).

[73]:
```
our_array = np.transpose(np.vstack((sys_bp, dia_bp, b_oxy, pulse)))
our_array
#np.vstack?
```

[73]:
```
array([[127.  ,  77.  ,  98.97,  65.  ],
       [125.  ,  88.  ,  98.59,  63.  ],
       [122.  ,  80.  ,  98.23,  65.  ],
```

```
       [129.  ,  79.  ,  98.54,  63.  ],
       [132.  ,  77.  ,  98.54,  63.  ],
       [131.  ,  81.  ,  97.95,  63.  ],
       [121.  ,  86.  ,  98.84,  63.  ],
       [128.  ,  82.  ,  99.05,  65.  ],
       [129.  ,  86.  ,  98.5 ,  67.  ],
       [126.  ,  71.  ,  97.7 ,  65.  ]])
```

Complete the following exercise.

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

It stacks the array in sequence vertically (row wise).

- 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

[74]:
```
# example 1

english = np.array(['red', 'green', 'pink'])
spanish = np.array(['rojo', 'verde', 'rosa'])

translate_df = pd.DataFrame(np.vstack((english, spanish)).T)
print(translate_df)

# example 2

symbol = np.array(['H', 'He', 'Li', 'Be', 'B', 'C'])
atomic_num = np.array([1, 2, 3, 4, 5, 6])
name = (['Hydrogen', 'Helium', 'Lithium', 'Beryllium', 'Boron', 'Carbon'])

periodic_df = pd.DataFrame(np.vstack((atomic_num, symbol, name)))
print(periodic_df)
```

```
       0      1
0    red   rojo
1  green  verde
2   pink   rosa
          0       1        2          3      4       5
0         1       2        3          4      5       6
1         H      He       Li         Be      B       C
2  Hydrogen  Helium  Lithium  Beryllium  Boron  Carbon
```

I chose these two examples to use `.vstack` to combine arrays to make dataframes easily.

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)

### 1.2 Verbs

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

[75]:
```
our_array.mean()
```

[75]:
```
92.59774999999999
```

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).

[76]:
```
our_array.mean(axis=0)
```

[76]:
```
array([127.   ,  80.7  ,  98.491,  64.2  ])
```

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

[77]:
```
our_df.mean()
```

[77]:
```
systolic BP          127.000
diastolic BP          80.700
blood oxygenation     98.491
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.

[78]:
```
our_df.describe()
```

[78]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| count | 10.00000 | 10.000000 | 10.000000 | 10.000000 |
| mean | 127.00000 | 80.700000 | 98.491000 | 64.200000 |
| std | 3.59011 | 5.121849 | 0.430102 | 1.398412 |
| min | 121.00000 | 71.000000 | 97.700000 | 63.000000 |
| 25% | 125.25000 | 77.500000 | 98.297500 | 63.000000 |
| 50% | 127.50000 | 80.500000 | 98.540000 | 64.000000 |
| 75% | 129.00000 | 85.000000 | 98.777500 | 65.000000 |
| max | 132.00000 | 88.000000 | 99.050000 | 67.000000 |

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

Numpy arrays don't know how to do this.

[79]:
```
our_array.describe()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [79], 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.

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

[ ]:
```
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.

[ ]:
```
our_df.
```

Complete the following exercise.

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

One method was `.append` which appends rows to the end of the caller, returning a new object. Another method is `.asfreq` which converts time series to specified frequency. It returns the original data conformed to a new index with the specificed frequency.

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.

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

[ ]:
```
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!*

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

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`

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

## 1.3 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…

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

Compare that to doing it the pandas way:

[ ]:
```
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.

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:

[ ]:
```
# numpy
dia_bp_mean = np_style_means[1]
print(dia_bp_mean)

# pandas
our_means['diastolic BP']
```

### 1.3.1 Accessing data using square brackets

Let's look ot our litte data frame again.

[ ]:
```
our_df
```

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

[80]:
```
our_df['pulse rate']
```

[80]:
```
0    65
1    63
```

```
2    65
3    63
4    63
5    63
6    63
7    65
8    67
9    65
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:

[81]:
```
our_df['pulse rate'].mean()    # creates a series, then makes it compute its own␣
 ↪mean
```

[81]:
```
64.2
```

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

[82]:
```
needed_cols = ['diastolic BP', 'systolic BP']    # make a list
our_df[needed_cols]                              # use the list to grab columns
```

[82]:
```
   diastolic BP  systolic BP
0            77          127
1            88          125
2            80          122
3            79          129
4            77          132
5            81          131
6            86          121
7            82          128
8            86          129
9            71          126
```

We could also do this in one step.

[83]:
```
our_df[['diastolic BP', 'systolic BP']]  # the inner brackets define our list
```

[83]:
```
   diastolic BP  systolic BP
0            77          127
1            88          125
2            80          122
3            79          129
4            77          132
5            81          131
6            86          121
7            82          128
```

```
8               86              129
9               71              126
```

(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

[84]:
```
our_df[['blood oxygenation','pulse rate']]
```

[84]:
```
   blood oxygenation  pulse rate
0              98.97          65
1              98.59          63
2              98.23          65
3              98.54          63
4              98.54          63
5              97.95          63
6              98.84          63
7              99.05          65
8              98.50          67
9              97.70          65
```

### 1.3.2 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:

[85]:
```
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, but...)
```

Let's take a look at this index:

[86]:
```
print(my_ind)
```

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

[87]:
```
our_df.index = my_ind
```

[88]:
```
our_df
```

[88]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| 1 | 127 | 77 | 98.97 | 65 |
| 2 | 125 | 88 | 98.59 | 63 |
| 3 | 122 | 80 | 98.23 | 65 |
| 4 | 129 | 79 | 98.54 | 63 |
| 5 | 132 | 77 | 98.54 | 63 |
| 6 | 131 | 81 | 97.95 | 63 |
| 7 | 121 | 86 | 98.84 | 63 |
| 8 | 128 | 82 | 99.05 | 65 |
| 9 | 129 | 86 | 98.50 | 67 |
| 10 | 126 | 71 | 97.70 | 65 |

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

[89]:
```
new_ind = np.linspace(5,15,10)
new_ind = np.int64(new_ind)

print(new_ind)
```

```
[ 5  6  7  8  9 10 11 12 13 15]
```

#### 1.3.3 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:

[90]:
```
our_df.loc[3]
```

[90]:
```
systolic BP          122.00
diastolic BP          80.00
blood oxygenation     98.23
pulse rate            65.00
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:

[91]:
```
our_df.loc[3:6]
```

[91]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| 3 | 122 | 80 | 98.23 | 65 |
| 4 | 129 | 79 | 98.54 | 63 |
| 5 | 132 | 77 | 98.54 | 63 |
| 6 | 131 | 81 | 97.95 | 63 |

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:

[92]:
```
our_df.loc[3:6, 'blood oxygenation']
```

[92]:
```
3    98.23
4    98.54
5    98.54
6    97.95
Name: blood oxygenation, dtype: float64
```

For a single column, or:

[93]:
```
our_df.loc[3:6,'systolic BP':'blood oxygenation']
```

[93]:

| | systolic BP | diastolic BP | blood oxygenation |
|---|---|---|---|
| 3 | 122 | 80 | 98.23 |
| 4 | 129 | 79 | 98.54 |
| 5 | 132 | 77 | 98.54 |
| 6 | 131 | 81 | 97.95 |

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!):

[94]:
```
names = np.array(['John Doe', 'Jane Smith', 'Patrick Wu', 'Jenny Romeo',␣
↪'Richard Martin',
                  'John Green', 'Hank Green', 'David Valorz', 'Domenica␣
↪Aburto', 'Marietta Aburto'])

df2 = our_df.copy()
df2.index = names
df2
```

[94]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| John Doe | 127 | 77 | 98.97 | 65 |
| Jane Smith | 125 | 88 | 98.59 | 63 |
| Patrick Wu | 122 | 80 | 98.23 | 65 |
| Jenny Romeo | 129 | 79 | 98.54 | 63 |
| Richard Martin | 132 | 77 | 98.54 | 63 |
| John Green | 131 | 81 | 97.95 | 63 |
| Hank Green | 121 | 86 | 98.84 | 63 |
| David Valorz | 128 | 82 | 99.05 | 65 |
| Domenica Aburto | 129 | 86 | 98.50 | 67 |
| Marietta Aburto | 126 | 71 | 97.70 | 65 |

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

[95]:
```
our_sum = our_df.describe()
our_sum
```

[95]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| count | 10.00000 | 10.000000 | 10.000000 | 10.000000 |
| mean | 127.00000 | 80.700000 | 98.491000 | 64.200000 |
| std | 3.59011 | 5.121849 | 0.430102 | 1.398412 |
| min | 121.00000 | 71.000000 | 97.700000 | 63.000000 |
| 25% | 125.25000 | 77.500000 | 98.297500 | 63.000000 |
| 50% | 127.50000 | 80.500000 | 98.540000 | 64.000000 |
| 75% | 129.00000 | 85.000000 | 98.777500 | 65.000000 |
| max | 132.00000 | 88.000000 | 99.050000 | 67.000000 |

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

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

[96]: pandas.core.frame.DataFrame

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

[100]:
```
our_sum.loc['mean']
```

[100]:
```
systolic BP          127.000
diastolic BP          80.700
blood oxygenation     98.491
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:

[101]:
```
our_sum['mean']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File /opt/homebrew/lib/python3.10/site-packages/pandas/core/indexes/base.py:
↪3800, in Index.get_loc(self, key, method, tolerance)
   3799 try:
-> 3800     return self._engine.get_loc(casted_key)
   3801 except KeyError as err:

File /opt/homebrew/lib/python3.10/site-packages/pandas/_libs/index.pyx:138, in␣
↪pandas._libs.index.IndexEngine.get_loc()

File /opt/homebrew/lib/python3.10/site-packages/pandas/_libs/index.pyx:165, in␣
↪pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:5745, in pandas._libs.hashtable.
↪PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:5753, 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)
Cell In [101], line 1
----> 1 our_sum['mean']

File /opt/homebrew/lib/python3.10/site-packages/pandas/core/frame.py:3805, in␣
↪DataFrame.__getitem__(self, key)
   3803 if self.columns.nlevels > 1:
   3804     return self._getitem_multilevel(key)
-> 3805 indexer = self.columns.get_loc(key)
   3806 if is_integer(indexer):
```

```
   3807     indexer = [indexer]

File /opt/homebrew/lib/python3.10/site-packages/pandas/core/indexes/base.py:
 ↪3802, in Index.get_loc(self, key, method, tolerance)
   3800     return self._engine.get_loc(casted_key)
   3801 except KeyError as err:
-> 3802     raise KeyError(key) from err
   3803 except TypeError:
   3804     # If we have a listlike key, _check_indexing_error will raise
   3805     #  InvalidIndexError. Otherwise we fall through and re-raise
   3806     #  the TypeError.
   3807     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:

[102]:
```
our_sum.loc['count':'std']
```

[102]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| count | 10.00000 | 10.000000 | 10.000000 | 10.000000 |
| mean | 127.00000 | 80.700000 | 98.491000 | 64.200000 |
| std | 3.59011 | 5.121849 | 0.430102 | 1.398412 |

Or rows and columns:

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

[103]:

| | systolic BP | diastolic BP |
|---|---|---|
| count | 10.00000 | 10.000000 |
| mean | 127.00000 | 80.700000 |
| std | 3.59011 | 5.121849 |

### 1.3.4 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:

[104]:
```
our_df
```

[104]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| 1 | 127 | 77 | 98.97 | 65 |
| 2 | 125 | 88 | 98.59 | 63 |
| 3 | 122 | 80 | 98.23 | 65 |

```
4           129         79           98.54        63
5           132         77           98.54        63
6           131         81           97.95        63
7           121         86           98.84        63
8           128         82           99.05        65
9           129         86           98.50        67
10          126         71           97.70        65
```

And let's check its shape:

[105]:
```python
our_df.shape
```

[105]:
```
(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[]`:

[106]:
```python
our_df.iloc[3]  # get the fourth row
```

[106]:
```
systolic BP          129.00
diastolic BP          79.00
blood oxygenation     98.54
pulse rate            63.00
Name: 4, dtype: float64
```

And compare that to using `loc[]`:

[107]:
```python
our_df.loc[3]
```

[107]:
```
systolic BP          122.00
diastolic BP          80.00
blood oxygenation     98.23
pulse rate            65.00
Name: 3, dtype: float64
```

And of course you can slice out rows and columns:

[108]:
```python
our_df.iloc[2:5, 0:2]
```

[108]:
```
   systolic BP  diastolic BP
3          122            80
4          129            79
5          132            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).

## 1.4 Non-numerical information (categories or factors)

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

### 1.4.1 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:

[109]:
```
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)

[110]:
```
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.

[111]:
```
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
```

[112]:
```
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.

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

```
[114]: print(sex)
```

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

```
[115]: sex = sex.repeat(num_patients/4)                # repeat each over one cell's␣
         ↪worth of data
```

```
[116]: print(sex)
```

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

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

```
[118]: 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.

[119]:
```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,
           'sex': sex,
           'diabetes': diabetic
           }
```

And turn it into a data frame.

[120]:
```python
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!

[121]:
```python
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.

[122]:
```python
new_df.index = my_index
```

Let's look at our creation!

[123]:
```python
new_df
```

[123]:

| | systolic BP | diastolic BP | blood oxygenation | pulse rate | sex | diabetes |
|---|---|---|---|---|---|---|
| patient 1 | 149 | 106 | 98.37 | 66 | male | yes |
| patient 2 | 146 | 96 | 98.89 | 66 | male | yes |
| patient 3 | 148 | 93 | 98.74 | 64 | male | yes |
| patient 4 | 139 | 102 | 98.63 | 60 | male | yes |
| patient 5 | 144 | 96 | 98.52 | 67 | male | yes |
| patient 6 | 143 | 95 | 98.74 | 64 | female | yes |
| patient 7 | 143 | 89 | 98.27 | 62 | female | yes |
| patient 8 | 141 | 98 | 98.40 | 64 | female | yes |
| patient 9 | 136 | 98 | 98.71 | 63 | female | yes |
| patient 10 | 133 | 98 | 98.68 | 64 | female | yes |
| patient 11 | 138 | 86 | 98.21 | 65 | male | no |
| patient 12 | 123 | 90 | 98.56 | 66 | male | no |
| patient 13 | 125 | 91 | 98.52 | 65 | male | no |
| patient 14 | 136 | 85 | 98.43 | 69 | male | no |
| patient 15 | 128 | 81 | 98.43 | 68 | male | no |
| patient 16 | 132 | 82 | 99.07 | 65 | female | no |
| patient 17 | 120 | 73 | 98.34 | 69 | female | no |
| patient 18 | 129 | 77 | 98.57 | 62 | female | no |
| patient 19 | 120 | 82 | 98.52 | 65 | female | no |
| patient 20 | 123 | 80 | 98.79 | 67 | female | no |

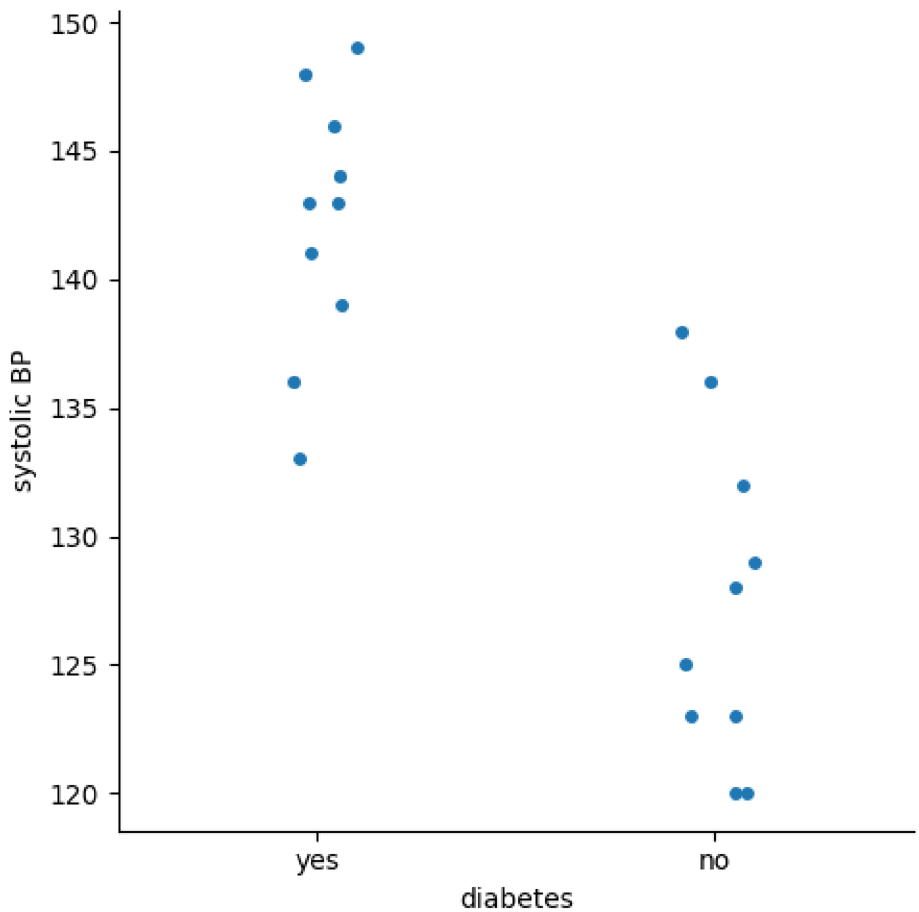
### 1.4.2 Looking at our data

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.

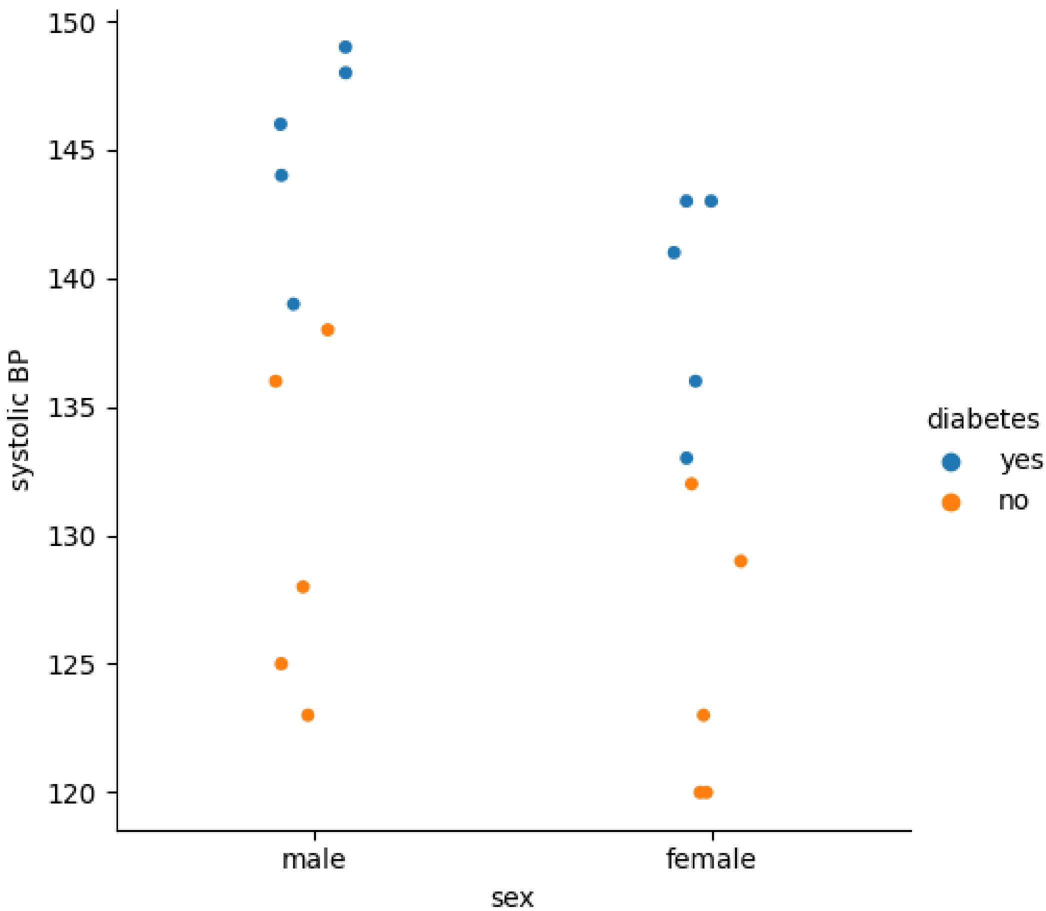
[124]:
```
import seaborn as sns

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

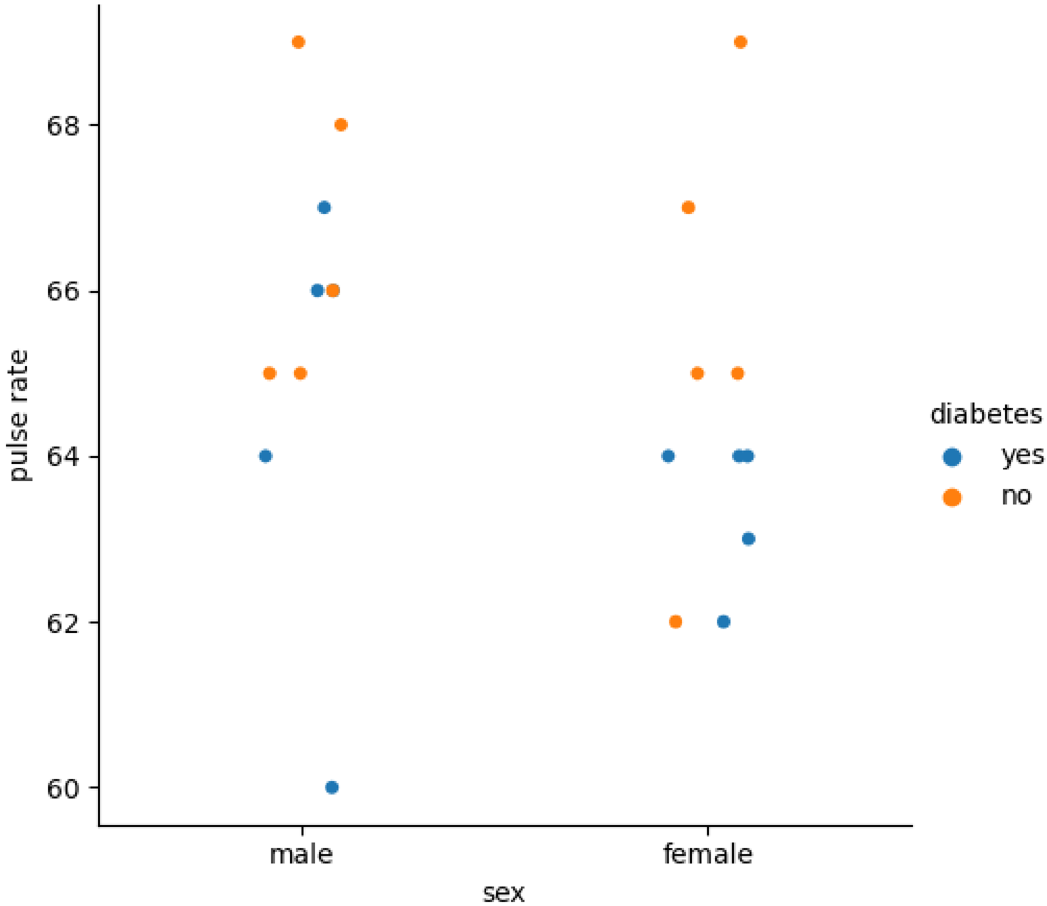



Okay, now let's go crazy and do a bunch of plots.

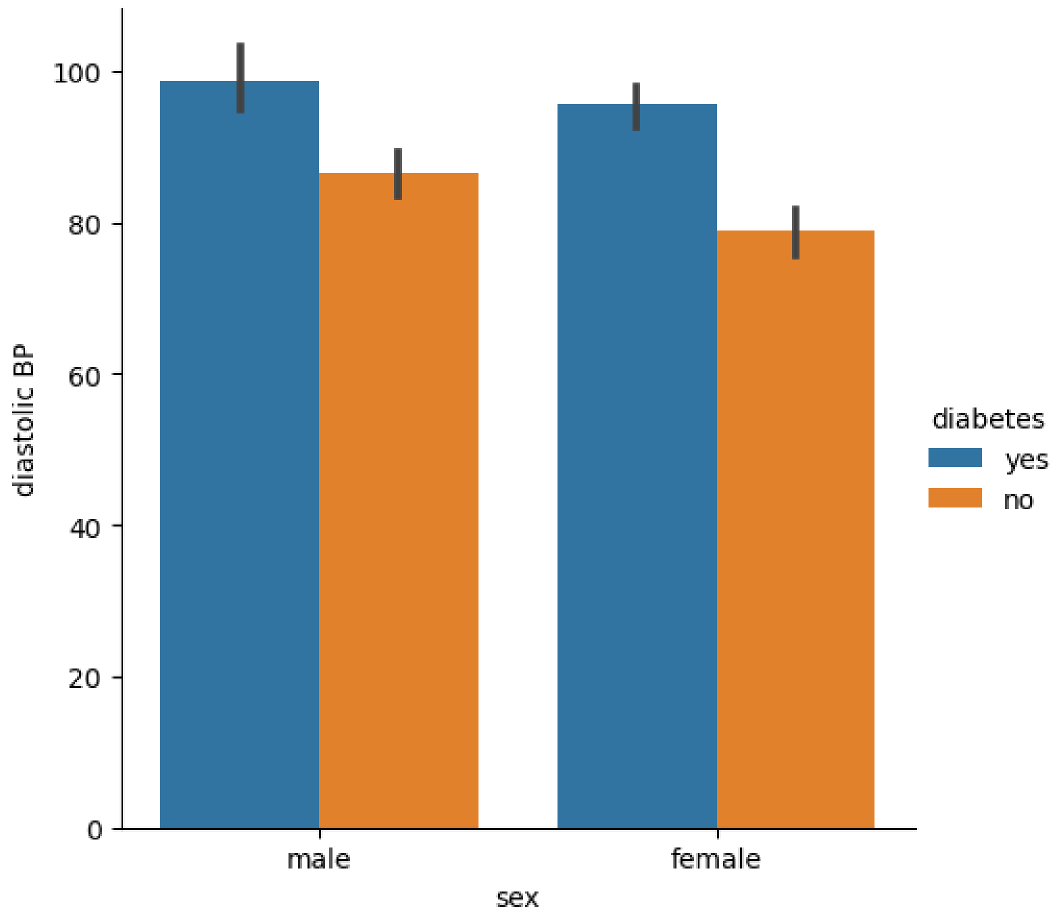
[125]:
```
sns.catplot(data=new_df, x='sex', y='systolic BP', hue='diabetes');
```




[126]:
```
sns.catplot(data=new_df, x='sex', y='pulse rate', hue='diabetes');
```




[127]:
```
sns.catplot(data=new_df, x='sex', y='diastolic BP', hue='diabetes', kind='bar');
```




### 1.4.3 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.

[128]:
```
new_df.groupby('sex')
```

[128]: `<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1281c2680>`

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:

[129]:
```python
new_df.groupby('sex').mean()
```

```
/var/folders/zc/6v283x0929j5f38j6cvlvbwr0000gn/T/ipykernel_10605/4080839992.py:1
: FutureWarning: The default value of numeric_only in DataFrameGroupBy.mean is
deprecated. In a future version, numeric_only will default to False. Either
specify numeric_only or select only columns which should be valid for the
function.
  new_df.groupby('sex').mean()
```

[129]:

| sex | systolic BP | diastolic BP | blood oxygenation | pulse rate |
|---|---|---|---|---|
| female | 132.0 | 87.2 | 98.609 | 64.5 |
| male | 137.6 | 92.6 | 98.530 | 65.6 |

**Using the `groupby()` followed by `aggregate()`** More powerfully, we can use a `GroupBy` object's `aggregate()` method to compute many things at once.

[130]:
```python
new_df.groupby('diabetes').aggregate(['mean', 'std', 'min', 'max'])
```

```
/var/folders/zc/6v283x0929j5f38j6cvlvbwr0000gn/T/ipykernel_10605/2935691488.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'])
```

[130]:

| | systolic BP | | | | diastolic BP | | | | \ |
|---|---|---|---|---|---|---|---|---|---|
| | mean | std | min | max | mean | std | min | max | |
| diabetes | | | | | | | | | |
| no | 127.4 | 6.363088 | 120 | 138 | 82.7 | 5.538752 | 73 | 91 | |
| yes | 142.2 | 5.094660 | 133 | 149 | 97.1 | 4.653553 | 89 | 106 | |

| | blood oxygenation | | | | pulse rate | | | \ |
|---|---|---|---|---|---|---|---|---|
| | mean | std | min | max | mean | std | min | |
| diabetes | | | | | | | | |
| no | 98.544 | 0.240009 | 98.21 | 99.07 | 66.1 | 2.183270 | 62 | |
| yes | 98.595 | 0.197386 | 98.27 | 98.89 | 64.0 | 2.054805 | 60 | |

| | max |
|---|---|
| diabetes | |
| no | 69 |
| yes | 67 |

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.

[131]:
```
temp_df = new_df[['systolic BP', 'diastolic BP', 'diabetes']]         # make a␣
↪data frame with only the columns we want
our_summary = temp_df.groupby('diabetes').aggregate(['mean',
                                                     'std', 'min', 'max'])    ␣
↪# compute stuff on those columns
our_summary
```

[131]:
```
         systolic BP                      diastolic BP
                mean       std  min  max         mean       std min  max
diabetes
no             127.4  6.363088  120  138         82.7  5.538752  73   91
yes            142.2  5.094660  133  149         97.1  4.653553  89  106
```

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.

[132]:
```
our_summary[("systolic BP", "mean")]
```

[132]:
```
diabetes
no     127.4
yes    142.2
Name: (systolic BP, mean), dtype: float64
```

[133]:
```
our_summary.loc[("no")]
```

[133]:
```
systolic BP   mean    127.400000
              std       6.363088
              min     120.000000
              max     138.000000
diastolic BP  mean     82.700000
              std       5.538752
              min      73.000000
              max      91.000000
Name: no, dtype: float64
```

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

[134]:
```
temp_df = new_df[['systolic BP', 'diabetes']]         # make a data frame with␣
↪only the columns we want
our_summary = temp_df.groupby('diabetes').aggregate(['mean',
                                                     'std', 'min', 'max'])    ␣
↪# compute stuff on those columns
our_summary
```

[134]:
```
         systolic BP
                mean       std  min  max
diabetes
no             127.4  6.363088  120  138
yes            142.2  5.094660  133  149
```

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:

[135]:
```
our_summary.shape
```

[135]: `(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:

[136]:
```
our_summary.iloc[0, 1]
```

[136]: `6.363087999461338`

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:

[137]:
```
new_df[['systolic BP', 'diabetes']].groupby('diabetes').mean()
```

[137]:
```
          systolic BP
diabetes
no              127.4
yes             142.2
```

Or we could group and then subset:

[138]:
```
new_df.groupby('diabetes')[['systolic BP']].mean()
```

[138]:
```
          systolic BP
diabetes
no              127.4
yes             142.2
```

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:

[139]:
```
new_df.pivot_table('systolic BP', index='diabetes')
```

[139]:
```
          systolic BP
diabetes
no              127.4
yes             142.2
```

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:

[140]:
```
new_df.pivot_table('systolic BP', index='diabetes', columns='sex')
```

[140]:
```
sex       female   male
diabetes
no         124.8  130.0
yes        139.2  145.2
```

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

[141]:
```
new_df.pivot_table('systolic BP', index='diabetes', columns='sex',␣
 ↪aggfunc='median')
```

[141]:
```
sex       female  male
diabetes
no           123   128
yes          141   146
```

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

[142]:
```
new_df.groupby('diabetes').aggregate(Mean=('systolic BP',"mean"))
```

[142]:
```
           Mean
diabetes
no        127.4
yes       142.2
```

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.

[143]:
```
new_df.groupby('diabetes').aggregate(Mean=('systolic BP',"mean"),
                                     Standard_Deviation = ('systolic␣
↪BP',"std"))
```

[143]:
```
           Mean  Standard_Deviation
diabetes
no        127.4            6.363088
yes       142.2            5.094660
```

### 1.5 vs.

[144]:
```
new_df.groupby('diabetes').aggregate( Mean=('systolic BP',"mean"), STD =␣
↪('systolic BP',"std"))
```

[144]:
```
           Mean       STD
diabetes
no        127.4  6.363088
yes       142.2  5.094660
```

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

### 1.6 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

### 1.7 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!)

[151]:
```python
num_participant = 20

# bilingual array
bilingual = pd.Series(['yes', 'no'])
bilingual = bilingual.repeat(num_participant/2)
bilingual = bilingual.reset_index(drop=True)

# verbal and quant GRE
verbal = np.int64(151 + 8.5 * np.random.randn(num_participant,))
quant = np.int64(151 + 8.5 * np.random.randn(num_participant,))

verbal[0:10] = verbal[0:10] + 6
quant[0:10] = quant[0:10] + 3

# make dict
GRE_bi = {'Verbal GRE': verbal,
         'Quant. GRE': quant,
         'Bilingual': bilingual,
         }
# dataframe
GRE_bi_df = pd.DataFrame(GRE_bi)


# index students
index = np.array([])
for i in range(1, num_participant + 1):
    index = np.append(index, "Student " + str(i))

GRE_bi_df.index = index # specify index
GRE_bi_df
```

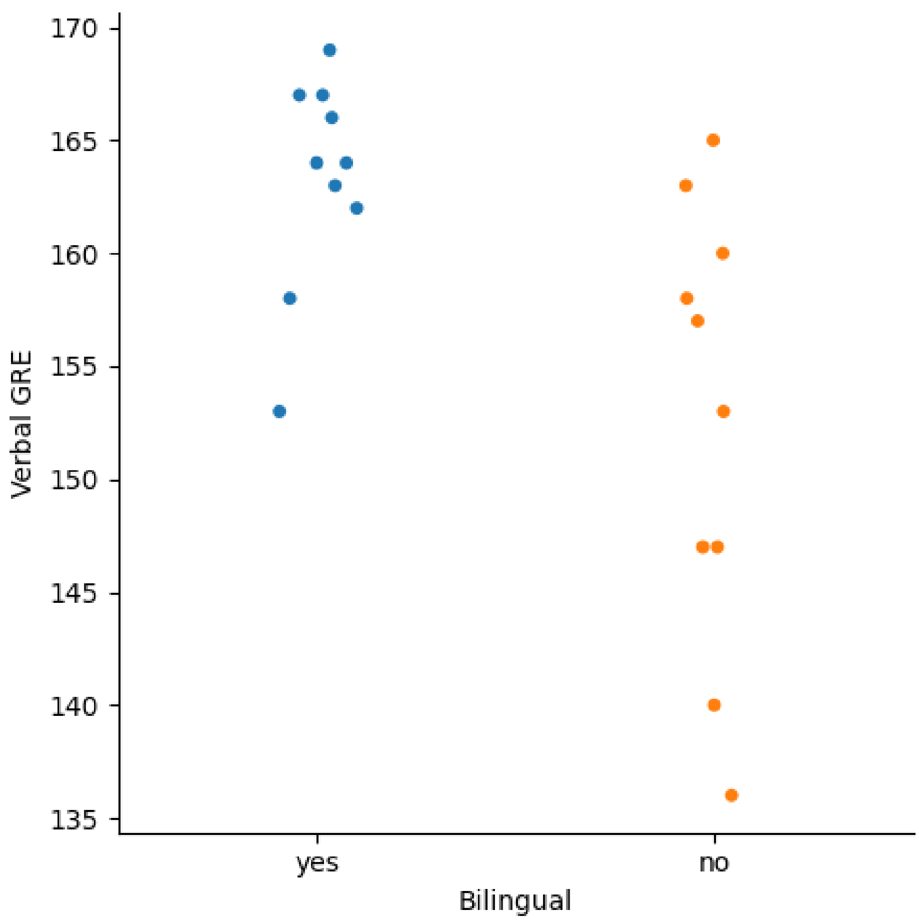
[151]:

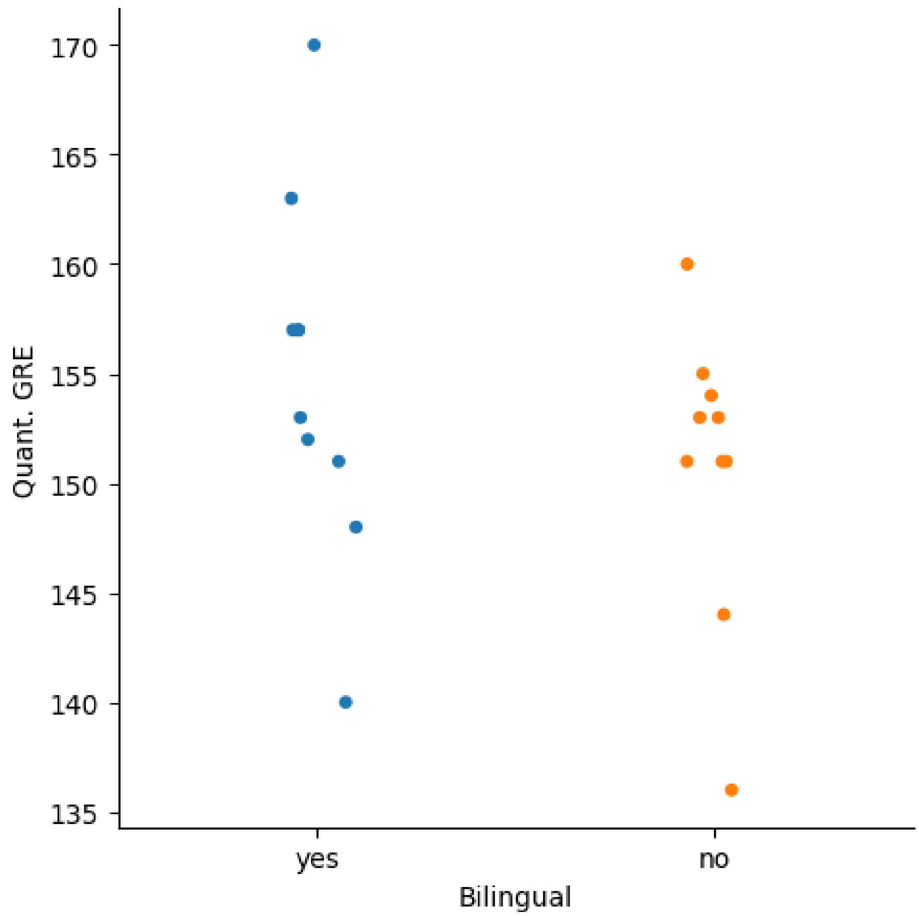
| | Verbal GRE | Quant. GRE | Bilingual |
|---|---|---|---|
| Student 1 | 166 | 152 | yes |
| Student 2 | 164 | 157 | yes |
| Student 3 | 167 | 157 | yes |
| Student 4 | 158 | 170 | yes |
| Student 5 | 162 | 148 | yes |
| Student 6 | 163 | 140 | yes |
| Student 7 | 153 | 151 | yes |
| Student 8 | 167 | 157 | yes |
| Student 9 | 164 | 163 | yes |
| Student 10 | 169 | 153 | yes |
| Student 11 | 147 | 136 | no |
| Student 12 | 153 | 144 | no |
| Student 13 | 140 | 153 | no |
| Student 14 | 160 | 153 | no |
| Student 15 | 136 | 151 | no |

| | | | |
|---|---|---|---|
| Student 16 | 158 | 155 | no |
| Student 17 | 165 | 151 | no |
| Student 18 | 163 | 151 | no |
| Student 19 | 147 | 154 | no |
| Student 20 | 157 | 160 | no |

[153]:
```
# verbal GRE vs. bilingual
sns.catplot(data = GRE_bi_df, x = 'Bilingual', y = 'Verbal GRE', hue =␣
 ↪'Bilingual');

# quant GRE vs. bilingual
sns.catplot(data = GRE_bi_df, x = 'Bilingual', y = 'Quant. GRE', hue =␣
 ↪'Bilingual');
```






[158]:
```
# mean and standard error of each score separated by bilingual status
GRE_sum = GRE_bi_df.groupby('Bilingual').aggregate(['mean', 'std'])
GRE_sum
```

[158]:

| | Verbal GRE | | Quant. GRE | |
|---|---|---|---|---|
| | mean | std | mean | std |
| Bilingual | | | | |
| no | 152.6 | 9.788883 | 150.8 | 6.562520 |
| yes | 163.3 | 4.762119 | 154.8 | 8.189424 |