# Pandas

Today we are going to really start using pandas. Lets review the libraries or packages and what they have done so far.

1. We always start byt typing %matploltlib inline. This is a built in magic command that enables us to plot the data right into our ipython notebook https://ipython.org/ipython-doc/3/interactive/magics.html
2. we import matplotlib.pyplot as plt. This turns on all the graphing capabilities and then uses the shorthand plt. for when we call functions from matplotlib. This used to be called pylab but was updated to pyplot. Then we say fig,ax=plt.subplots() and all the plot functions go into fig and ax
3. we import numpy as np. This turns on math functions and we use the shorthand np.
4. from scipy we import stats. scipy gives us a lot of analysis functions and we use linear regression from stats.
5. Now we are also going to use pandas. Pandas is database management. It lets us take complicated datasets and anlyze them. You can think of it like a supercharged excel where you combine the organization of excel with the power of a programming language. It can do amazing things and I am still learning every day. So lets get started!
6. What is pandas? http://pandas.pydata.org/index.html and here is the documentation http://pandas.pydata.org/pandas-docs/stable/
7. import pandas as pd!!!!!!
8. On a final note you can see I made a numbered list in markdown. To do that you type a number a period and then two spaces.
9. Also in terms of line numnbers. I turn my line numbers on so it is easier to debug. Do this under view

In [2]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

## Importing files

We are going to start by finding our csv file and reading it in

I only want to list the csv files so I can see what I can read in. so I will do ls *.csv the star is a wildcard that means everything and then .csv is only ones that end in .csv. Today we are going to look at data from well water chemistry in Bangladesh. Specifically arsenic concentrations and if people drink the water. We will also look at the rest of the chemistry. We are looking at well water arsenic because drinking water with arsenic has negative long term health impacts. The US standard for arsenic is 10 ppb or 10 ug/L. The bangladesh standard is 50 ppb. Lets see what we can learn! We are going to try and learn about how many people drink water with 10 or 50 ppb arsenic. (to show the star I had to type *)

The data is on the edblogs siteh ttps://edblogs.columbia.edu/eescx3050-001-2015-3/category/classes/class-10-start-pandas/ or on the github site https://github.com/bmaillou/BigDataPython/blob/master/well_data.csv.

In [4]:
```
pwd
```

Out[4]:
```
'/Users/bmaillou/Documents/work-teaching/python/fall21/BigDataPython'
```

In [5]:
```
ls *.csv
```

```
Brian.csv                     gdp2 - Copy.csv
CoreEM09GC01-extra-line.csv   gdp2.csv
CoreEM09GC01.csv              gdp2015.csv
GDP-Lifespan - Copy.csv       gdp_and_lifespan.csv
GDP-Lifespan.csv              gdp_only.csv
GDP_Lifespan_all_data.csv     gdp_only_download.csv
Libby_Thesis_Data.csv         mystery.csv
Well-As.csv                   twoD1.csv
central_park.csv              weekly_mlo.csv
fldav_ljo.csv                 well_data.csv
fldav_ljo_Yasna.csv           well_sites.csv
gdp.csv
```

now we read in a well_data.csv. But I want to use pandas and not numpy.

But we are going to read in some data and try to analyze it. open the well_data.csv. It is for wells from Bangladesh. every well has an id#, a latitude and longitude, Depth, if people drink it and then some concentration data. lets use readcsv to get read in. In Pandas you are trying to get your data into a dataframe which is like an excel sheet. It will have column titles and an index for rows. It is all about the dataframes. When using pandas people name things 'df' a lot. That is shorthand for dataframe. I am not a good namer.

I am going to just name it df today.

In [3]:
```
df=pd.read_csv('well_data.csv')
```

The data is now magically in the computers memory even if we can't see it we can access it!

# This is important. Your output may not look like my output. It changes between computers depending on default settings when you installed. Don't worry. If you see data of descriptions you are fine.

just typing well_data will give us some descriptions of what we got! It used the first row for column names!

In [4]:
```
df
```

Out[4]:

| | Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 2 | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN |
| **1** | 14 | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN |

| | Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | |
|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 23 | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | |
| **3** | 83 | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1 |
| **4** | 84 | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **754** | 12516 | 24.71 | 90.41 | 160 | Y | 32379.64000 | 0.197380 | 3669.430000 | 39790.24000 | 0 |
| **755** | 12654 | 24.36 | 91.27 | 60 | Y | 25561.12000 | 0.090570 | 13771.370000 | 57630.63000 | 1. |
| **756** | 72641 | 24.38 | 90.90 | 45 | N | 31319.48000 | 1.162550 | 38.300000 | 60905.16000 | 22 |
| **757** | 76175 | 23.90 | 90.65 | 60 | N | 30605.53000 | 1.556120 | 4168.520000 | 66756.16000 | 12 |
| **758** | 141499 | 23.60 | 91.34 | 50 | N | NaN | NaN | NaN | NaN | |

759 rows × 21 columns

In [ ]:

Since we didn't set an index it just numbers each row and calls that the index. But that doesn't help us. I think we could set the well_id to the index. When you look at your data above. see how the numbers on the left have no title but are a little offset. That is the index. But what is an index. I am not sure. It is sort of like a master column that helps us organize the data. It will make more sense when we get to timeseries analysis. That is where pandas shines even more. But lets set and index and use well_id as that is the most important factor.

In [5]:
```
df=df.set_index('Well_ID')
```

In [6]:
```
df
```

Out[6]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | |
|---|---|---|---|---|---|---|---|---|---|
| **2** | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN | Na |
| **14** | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN | Na |
| **23** | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | Na |
| **83** | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1.2600 |
| **84** | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | Na |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **12516** | 24.71 | 90.41 | 160 | Y | 32379.64000 | 0.197380 | 3669.430000 | 39790.24000 | 0.3412( |
| **12654** | 24.36 | 91.27 | 60 | Y | 25561.12000 | 0.090570 | 13771.370000 | 57630.63000 | 1.4983 |
| **72641** | 24.38 | 90.90 | 45 | N | 31319.48000 | 1.162550 | 38.300000 | 60905.16000 | 22.4175 |
| **76175** | 23.90 | 90.65 | 60 | N | 30605.53000 | 1.556120 | 4168.520000 | 66756.16000 | 12.7931( |
| **141499** | 23.60 | 91.34 | 50 | N | NaN | NaN | NaN | NaN | Na |

759 rows × 20 columns

In [ ]:
```
```

we can undue the index

In [7]:
```
df=df.reset_index()
```

In [8]:
```
df #since we have an index it prints the index name on its own row.
```

Out[8]:

| | Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2 | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN | |
| **1** | 14 | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN | |
| **2** | 23 | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | |
| **3** | 83 | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1 |
| **4** | 84 | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **754** | 12516 | 24.71 | 90.41 | 160 | Y | 32379.64000 | 0.197380 | 3669.430000 | 39790.24000 | 0 |
| **755** | 12654 | 24.36 | 91.27 | 60 | Y | 25561.12000 | 0.090570 | 13771.370000 | 57630.63000 | 1. |
| **756** | 72641 | 24.38 | 90.90 | 45 | N | 31319.48000 | 1.162550 | 38.300000 | 60905.16000 | 22 |
| **757** | 76175 | 23.90 | 90.65 | 60 | N | 30605.53000 | 1.556120 | 4168.520000 | 66756.16000 | 12 |
| **758** | 141499 | 23.60 | 91.34 | 50 | N | NaN | NaN | NaN | NaN | |

759 rows × 21 columns

Or we could just read in the data with the index set.

In [9]:
```
df=pd.read_csv('well_data.csv',index_col='Well_ID')
```

If you don't know the column name you can use the column number!

In [10]:
```
df=pd.read_csv('well_data.csv',index_col=0)
```

# The first great trick of pandas!

The describe function. It gives you amazing summary statistics lickety-split!

In [11]:
```
df.describe()
```

Out[11]:

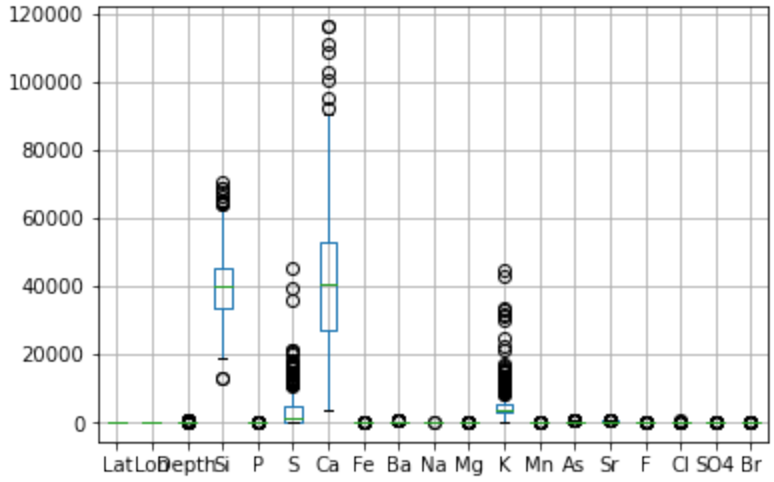
| | Lat | Lon | Depth | Si | P | S | |
|---|---|---|---|---|---|---|---|
| **count** | 759.000000 | 759.000000 | 759.000000 | 407.000000 | 407.000000 | 407.000000 | 407.00 |
| **mean** | 23.789249 | 90.641199 | 65.554677 | 40101.151444 | 0.809323 | 3407.292389 | 41129.29 |
| **std** | 0.578493 | 0.578800 | 42.186161 | 10117.680290 | 0.902860 | 5364.247733 | 20161.13 |
| **min** | 22.780000 | 89.610000 | 0.000000 | 12605.576700 | 0.008210 | -41.390000 | 3577.16 |
| **25%** | 23.285000 | 90.155000 | 45.000000 | 33200.310900 | 0.151957 | 149.635000 | 26996.27 |
| **50%** | 23.790000 | 90.650000 | 50.000000 | 40021.490000 | 0.507850 | 1220.877945 | 40166.83 |

| | Lat | Lon | Depth | Si | P | S | |
|---|---|---|---|---|---|---|---|
| **75%** | 24.300000 | 91.130000 | 70.000000 | 45369.825000 | 1.189271 | 4341.695000 | 52976.45 |
| **max** | 24.770000 | 91.650000 | 523.000000 | 70304.057950 | 5.477616 | 45035.460000 | 116040.62 |

A hint of what is to come! But we just got all of our summary statistics.

In [12]:
```
df.boxplot()
```

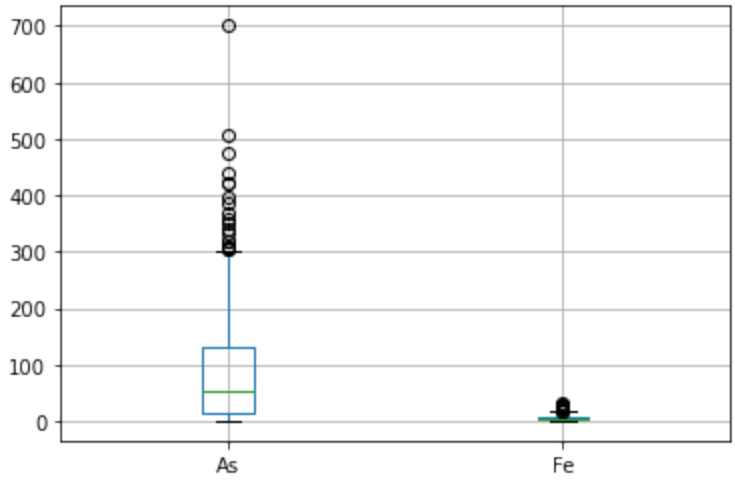
Out[12]: <AxesSubplot:>




That boxplot was hard to see. What if we just look at As and Fe?

In [13]:
```
fig,ax=plt.subplots()
df.boxplot(column=['As','Fe'],ax=ax)
```

Out[13]: <AxesSubplot:>




this plotting is a little diffrent then how we have been plotting. Pandas has some built in plotting so you can make some really nice and quick plots. But these plots are a little harder to customize. So we will be doing both types of plotting depending on the goal. The goal could be a quick view versus a profesional looking plot.

We can also just get a list of our columns.

In [14]:
```
df.columns
```

Out[14]:
```
Index(['Lat', 'Lon', 'Depth', 'Drink', 'Si', 'P', 'S', 'Ca', 'Fe', 'Ba', 'Na',
       'Mg', 'K', 'Mn', 'As', 'Sr', 'F', 'Cl', 'SO4', 'Br'],
      dtype='object')
```

Why did the columns not have parantheses? I am learning this. But each dataframe has attributes and methods. Methods uses paranthesis. Think of it as having to do something. An attribute just tells you about the dataframe and doesn't need parantheses. Methods can take extra arguments.

Remember NaN is not a number. We are going to use this to our advantage!

shape still gives us the shape. We can call it two different ways

In [15]:
```
df.shape
```

Out[15]:
```
(759, 20)
```

In [16]:
```
np.shape(df)
```

Out[16]:
```
(759, 20)
```

# Stop and think for a second. What does this shape mean?

It means we are starting to analyze a lot of data. It is a dataset with 759 rows or wells and 20 columns or different parameters. This will already get hard to deal with in excel!

# We have to slow down and learn some Pandas basics. this is a critical section. Take your time

Now how do we get at our data. How do we slice it. There are many ways. lets go through them all.

.ix

.loc

.iloc

[]

We are going to do a lot of practice and then I tried to make a cheat sheet/table. Take lots of notes.

[] works like normal except you can only use integers on rows and names on columns. you can't use integers on both rows and columns.

I am putting .head() on the print statements to save paper. You don't need them. It just shows the first 5 rows

In [17]:
```
df[:].head()  #I am including head to shorten my printouts
```

Out[17]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | Fe |
|---|---|---|---|---|---|---|---|---|---|
| **2** | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN | NaN |
| **14** | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **23** | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **83** | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1.260031 |
| **84** | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | NaN |

In [18]:
```
print(df['As'].head())
```

```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

In [19]:
```
print (df[:]['As'].head()) #This is the same as the one above showing the rows
```

```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

In [21]:
```
print (df[30:50]['As'])  #This prints rows 30-50.

#Don't get confused as well_ID is our index and the name of the row
```

```
Well_ID
330     10.233204
333           NaN
342           NaN
356           NaN
374     18.365596
389     59.285003
397    115.834040
398           NaN
402     17.755544
403     81.859568
410           NaN
414           NaN
415     87.102492
417           NaN
418    386.827954
420     79.798479
421    142.409968
434           NaN
475    270.785974
478     56.883257
Name: As, dtype: float64
```

In [22]:
```
print (df[30:50:2]['As']) #we skipped by twos!
```

```
Well_ID
330     10.233204
```

```
342           NaN
374     18.365596
397    115.834040
402     17.755544
410           NaN
415     87.102492
418    386.827954
421    142.409968
475    270.785974
Name: As, dtype: float64
```

But you can pass a list to the columns you want! SEE the double brackets??? It is a list in the brackets!

In [23]:
```
df[30:50:2][['As','Depth']]
```

Out[23]:

| Well_ID | As | Depth |
|---|---|---|
| **330** | 10.233204 | 45 |
| **342** | NaN | 30 |
| **374** | 18.365596 | 45 |
| **397** | 115.834040 | 45 |
| **402** | 17.755544 | 30 |
| **410** | NaN | 60 |
| **415** | 87.102492 | 60 |
| **418** | 386.827954 | 65 |
| **421** | 142.409968 | 150 |
| **475** | 270.785974 | 55 |

And the order doesn't matter. So somehow it is smart about rows and columns

In [25]:
```
df[['As','Depth']][30:50:2]
```

Out[25]:

| Well_ID | As | Depth |
|---|---|---|
| **330** | 10.233204 | 45 |
| **342** | NaN | 30 |
| **374** | 18.365596 | 45 |
| **397** | 115.834040 | 45 |
| **402** | 17.755544 | 30 |
| **410** | NaN | 60 |
| **415** | 87.102492 | 60 |
| **418** | 386.827954 | 65 |
| **421** | 142.409968 | 150 |
| **475** | 270.785974 | 55 |

In [26]:
```
df[['Depth','As']][30:50:2]
```

Out[26]:

| Well_ID | Depth | As |
|---|---|---|
| **330** | 45 | 10.233204 |
| **342** | 30 | NaN |
| **374** | 45 | 18.365596 |
| **397** | 45 | 115.834040 |
| **402** | 30 | 17.755544 |
| **410** | 60 | NaN |
| **415** | 60 | 87.102492 |
| **418** | 65 | 386.827954 |
| **421** | 150 | 142.409968 |
| **475** | 55 | 270.785974 |

In [156…

What I am teaching you is easy and hard at the same time. Take your time. It is a lot. I am showing you how to get at data. I just showed you brackets and now I am going to show you .loc. Also remember I just add .head to shorten the printouts. you can remove it.

.loc only uses names of the index and the columns.

THIS IS DIFFERENT. It is saying if my index matches this name then print it. This is a little confusing our the wells have numbers for names

Sometimes I put print sometimes not. It doesn't always matter and sometimes one looks nicer than the other.

In [28]:
```
df.loc[:].head()  #gives us all rows with all indexes
```

Out[28]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | Fe |
|---|---|---|---|---|---|---|---|---|---|
| **2** | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN | NaN |
| **14** | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **23** | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **83** | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1.260031 |
| **84** | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | NaN |

In [29]:
```
df.loc[101:156]
#gives us all rows with all indexes but the numbers have to match an index. The
```

Out[29]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | F |
|---|---|---|---|---|---|---|---|---|---|
| **101** | 24.40 | 90.26 | 60 | Y | 34311.71514 | 0.117534 | 2618.717799 | 42646.99574 | 1.84315 |
| **107** | 24.02 | 89.67 | 45 | N | NaN | NaN | NaN | NaN | Nal |
| **110** | 23.39 | 91.35 | 45 | Y | 47417.95635 | 1.095644 | 113.180915 | 46848.09017 | 11.74044 |
| **112** | 24.61 | 91.18 | 60 | Y | 37289.99489 | 2.448648 | 13.335397 | 65129.07627 | 8.92346 |
| **116** | 22.96 | 89.77 | 60 | Y | NaN | NaN | NaN | NaN | Nal |
| **130** | 22.94 | 89.97 | 60 | N | 44023.88418 | 1.172086 | 1023.167741 | 80183.25742 | 6.34939 |
| **153** | 24.17 | 90.81 | 45 | Y | 40523.43773 | 0.091676 | 2848.048146 | 40703.88184 | 1.86948 |
| **156** | 22.84 | 91.56 | 60 | N | 48375.82211 | 0.979053 | 1420.255478 | 52694.25919 | 13.02035 |

In [30]:
```
df.loc[101]   # just call one index.  This is well 101
```

Out[30]:
```
Lat          24.4
Lon         90.26
Depth          60
Drink           Y
Si        34311.7
P        0.117534
S         2618.72
Ca          42647
Fe        1.84316
Ba        58.6662
Na         28.281
Mg        22.5784
K             NaN
Mn        1.19269
As        28.0709
Sr        123.043
F          0.1994
Cl        38.1123
SO4         7.518
Br         0.0552
Name: 101, dtype: object
```

In [32]:
```
df.loc[102]   # if the index doesn't exist you get an error
```

In [ ]:
```

```

But we can use column names

In [33]:
```
df.loc[:]['As'].head()
```

Out[33]:
```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

Plus it can seperata with commas as well as multiple brackets

In [34]:
```
df.loc[:,'As'].head()
```

Out[34]:
```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

You can look at the type. It shows if you just get one column then it turns from a dataframe to a series

In [35]:
```
print (type(df.loc[:]['As']))
print (type(df.loc[:,'As']))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
```

This suble difference can sometimes be important. A series is more like a set of values.

In [36]:
```
print (df.loc[330:500:2]['As']  )
```

```
Well_ID
330     10.233204
342           NaN
374     18.365596
397    115.834040
402     17.755544
410           NaN
415     87.102492
418    386.827954
421    142.409968
475    270.785974
481           NaN
488           NaN
500           NaN
Name: As, dtype: float64
```

We can also add a list of names

In [37]:
```
df.loc[330:500:2][['As','Depth']]
```

Out[37]:

| Well_ID | As | Depth |
|---|---|---|
| **330** | 10.233204 | 45 |
| **342** | NaN | 30 |
| **374** | 18.365596 | 45 |
| **397** | 115.834040 | 45 |
| **402** | 17.755544 | 30 |
| **410** | NaN | 60 |
| **415** | 87.102492 | 60 |
| **418** | 386.827954 | 65 |
| **421** | 142.409968 | 150 |
| **475** | 270.785974 | 55 |
| **481** | NaN | 60 |

| Well_ID | As | Depth |
|---|---|---|
| **488** | NaN | 45 |
| **500** | NaN | 60 |

# iloc

iloc only uses integers. So now this is row numbers. NOT the index. look the Well_ID compared to the iloc numberss

In [38]:
```
df.iloc[:].head()
```

Out[38]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | Fe |
|---|---|---|---|---|---|---|---|---|---|
| **2** | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN | NaN |
| **14** | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **23** | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **83** | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1.260031 |
| **84** | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | NaN |

In [39]:
```
df.iloc[101:110:2] #this is row numbers now. so the index is not matching.
```

Out[39]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | Fe |
|---|---|---|---|---|---|---|---|---|---|
| **3058** | 23.92 | 90.47 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **3060** | 23.62 | 91.56 | 60 | Y | 37199.96208 | 0.949837 | 13.694978 | 51862.73844 | 10.061950 |
| **3103** | 23.33 | 90.12 | 130 | Y | NaN | NaN | NaN | NaN | NaN |
| **3112** | 24.43 | 91.05 | 45 | Y | 41513.82677 | 1.697027 | 14.933618 | 47308.53575 | 14.961449 |
| **3179** | 23.18 | 90.78 | 50 | Y | NaN | NaN | NaN | NaN | NaN |

and column number. But we use a column seperator.....

In [40]:
```
df.iloc[101:110:2,5]
```

Out[40]:
```
Well_ID
3058         NaN
3060    0.949837
3103         NaN
3112    1.697027
3179         NaN
Name: P, dtype: float64
```

In [41]:
```
df.iloc[101:110:2,2:5]
```

Out[41]:

| Well_ID | Depth | Drink | Si |
|---|---|---|---|
| **3058** | 60 | Y | NaN |
| **3060** | 60 | Y | 37199.96208 |
| **3103** | 130 | Y | NaN |
| **3112** | 45 | Y | 41513.82677 |
| **3179** | 50 | Y | NaN |

Just to boggles your bind a little.....

In [42]:
```
df.iloc[101:110:2,[2,5,8]]  #I just had it show columns 2,5,8
```

Out[42]:

| Well_ID | Depth | P | Fe |
|---|---|---|---|
| **3058** | 60 | NaN | NaN |
| **3060** | 60 | 0.949837 | 10.061950 |
| **3103** | 130 | NaN | NaN |
| **3112** | 45 | 1.697027 | 14.961449 |
| **3179** | 50 | NaN | NaN |

ix was phased out.

In [43]:
```
print (df.ix[101:110:2,[2,5,8]])
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-43-6c367ed50da5> in <module>
----> 1 print (df.ix[101:110:2,[2,5,8]])

~/anaconda3/lib/python3.8/site-packages/pandas/core/generic.py in __getattr__(se
lf, name)
   5137             if self._info_axis._can_hold_identifiers_and_holds_name(nam
e):
   5138                 return self[name]
-> 5139             return object.__getattribute__(self, name)
   5140
   5141     def __setattr__(self, name: str, value) -> None:

AttributeError: 'DataFrame' object has no attribute 'ix'
```

In [ ]:
```
```

# Dot notation

I am not sure if that is the official name but here is how it works

What I didn't show you is a dot notation.

In [44]:
```
df.As.head()
```

Out[44]:
```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

Dot notation only works if you name your columns well. No minus signs or spaces.

In [45]:
```
df.As[20:30]
```

Out[45]:
```
Well_ID
233          NaN
237          NaN
275          NaN
279          NaN
280     5.364619
283          NaN
287          NaN
290          NaN
292    53.097829
295          NaN
Name: As, dtype: float64
```

All examples in one place so maybe we can make sense of them?

| Name | Description |
|---|---|
| [] | |
| well_data[:] | all data |
| well_data[:]['As'] | all arsenic data |
| well_data.loc[:,'As'] | basically the same as above |
| well_data['As'] | all arsenic data. |
| well_data[1:10]['As'] | arsenic data from rows 1-10 excluding 10 |
| well_data[1:10:2]['As'] | same but skipping by two |
| well_data[1:10:2][['As','Depth']] | for As and depth. note the double brackets. |
| well_data[['As','Depth']][1:10:2] | order doesn't matter |
| well_data[['Depth','As']][1:10:2] | order doesn't matter. |
| you can't use column numbers..... | |

| loc | Description |
|---|---|
| well_data.loc[:] | gives us all rows with all columns |
| well_data.loc[101:156] | needs to be an index Gives us by index number not row number. |
| well_data.loc[101:156:2] | and we can skip |
| well_data.loc[101:156:2]['As'] | and we can do column names |
| well_data.loc[:]['As'] | |

| loc | Description |
|---|---|
| well_data.loc[:,'As'] | is the same as above. I have bugs where one works but other doesn't |
| well_data.loc[101:156:2][['As','Depth']] | we can do multiple columns |
| ..... | . |

| iloc | |
|---|---|
| well_data.iloc[:] | gives it all. |
| well_data.iloc[101:110:2] | does row numbers. |
| well_data.iloc[101:110:2,5] | row number by column number |
| well_data.iloc[101:110:2,2:5] | mulitple row multiple number |
| well_data.iloc[101:110:2,[2,5,8]] | select columns |
| ..... | . |

| ix | phased out |
|---|---|

| Dot notation. | This can be very nice. |
|---|---|
| well_data.As | gives all arsenic data |
| well_data.As[1:5] | gives rows 1-5 |
| ..... | |

## You can use boolean choices to get the data you want. For example I gave the description if people drink or don't drink from their well. Lets count that.

value_counts is a great first function. It just counts for you. Simple but very helpful.

I am going to do the same thing many different ways! value_counts is a function that counts each

In [46]:
```
df['Drink'].value_counts()
```

Out[46]:
```
Y    614
N    144
Name: Drink, dtype: int64
```

is the same as

In [48]:
```
df.Drink.value_counts()
```

Out[48]:
```
Y    614
N    144
Name: Drink, dtype: int64
```

Is the same as (I am trying to teach you pandas)

In [49]:
```
df.iloc[:,3].value_counts()
```

Out[49]:
```
Y    614
N    144
Name: Drink, dtype: int64
```

Is the same as (I am trying to teach you pandas)

In [50]:
```
df.loc[:,'Drink'].value_counts()
```

Out[50]:
```
Y    614
N    144
Name: Drink, dtype: int64
```

Now you should be able to access your data. I always forget the semantics. Look online or back at your cheat sheets. That is why I made the cheat sheet above.

Now we can sub-select data very easily.

We can return a boolean based on results.

In [51]:
```
df['Drink']=='Y'
```

Out[51]:
```
Well_ID
2          True
14         True
23         True
83         True
84         True
          ...
12516      True
12654      True
72641     False
76175     False
141499    False
Name: Drink, Length: 759, dtype: bool
```

Also do it with the dot notation

In [53]:
```
df.Drink=='Y'
```

Out[53]:
```
Well_ID
2          True
14         True
23         True
83         True
84         True
          ...
12516      True
12654      True
72641     False
76175     False
141499    False
Name: Drink, Length: 759, dtype: bool
```

What if we only want data from wells people drink from? we can ask for that. Remember I just added the .head() to save paper

In [59]:
```
df[df.Drink=='Y'].head()
```

Out[59]:

| Well_ID | Lat | Lon | Depth | Drink | Si | P | S | Ca | Fe |
|---|---|---|---|---|---|---|---|---|---|
| **2** | 23.74 | 90.31 | 45 | Y | NaN | NaN | NaN | NaN | NaN |
| **14** | 23.62 | 90.60 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **23** | 23.94 | 91.46 | 60 | Y | NaN | NaN | NaN | NaN | NaN |
| **83** | 23.80 | 91.33 | 50 | Y | 48084.33842 | 0.936358 | 2085.570979 | 54666.48199 | 1.260031 |
| **84** | 23.98 | 90.81 | 150 | Y | NaN | NaN | NaN | NaN | NaN |

What if we only wanted arsenic concentrations where people drink the water?

This is weird again.

You are saying only give me As.

In [60]:
```
df[df.Drink=='Y']['As'].head()
```

Out[60]:
```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

In the crazy world of pandas where you put the Arsenic doesn't matter.

In [62]:
```
df['As'][df.Drink=='Y'].head()
```

Out[62]:
```
Well_ID
2          NaN
14         NaN
23         NaN
83    78.97747
84         NaN
Name: As, dtype: float64
```

Say you wanted to do an intervention. You would want to go to the houses with the highest arsenic first. So we could ask what are the well id's for people who drink water and their arsenic is greater than 250 ppb. This would be poeple with high exposure! We would need to use an and statement.

In pandas you do this two ways.

- np.logical_and().
- Else you can use the & but YOU NEED PARANTHESE. **REMEMBER THIS!!!!!** It will come back and help you.

We would try to convince these households to switch. The drinking water standard is 10 ppb. This is really crazy high exposure.

In [64]:
```
df['As'][(df.Drink=='Y') & (df.As>250)]
```

Out[64]:
```
Well_ID
475     270.785974
2821    285.971884
2841    506.750799
2977    282.519542
4545    439.690000
4689    267.553524
4793    271.752307
4987    255.620635
5060    368.900000
5557    351.206317
5717    700.890000
5788    422.070000
6137    309.920000
6583    339.300000
7007    304.690000
8051    308.880000
8522    256.610000
9362    299.530000
Name: As, dtype: float64
```

A second way to do boolean and in pandas. Remember. When you split a line at a comma in a fucntion you don't need to use the . I do this to make the packets print better. You don't need to do it. But also line breaks can just make things cleaner and easier to see

In [67]:
```
df['As'][np.logical_and(df.Drink=='Y'
                        ,df.As>250)]
```

Out[67]:
```
Well_ID
475     270.785974
2821    285.971884
2841    506.750799
2977    282.519542
4545    439.690000
4689    267.553524
4793    271.752307
4987    255.620635
5060    368.900000
5557    351.206317
5717    700.890000
5788    422.070000
6137    309.920000
6583    339.300000
7007    304.690000
8051    308.880000
8522    256.610000
9362    299.530000
Name: As, dtype: float64
```

Can we look at who drinks from their wells and if they don't drink is it beacuse it has more arsenic?

Another way to word this.

What is the average arsenic in wells people drink from?

What is the averarge arsenic in wells people don't drink from.

Use decribe....

In [68]:
```
#wells people drink from
print('Arsenic of wells where people drink')
print (df['As'][df.Drink=='Y'].describe() )

# wells people don't drinkfrom
print ('\nArsenic of wells where people don\'t drink') #to put the ' in the line
print (df['As'][df.Drink=='N'].describe())
```

```
Arsenic of wells where people drink
count    336.000000
mean      72.484421
std       91.571489
min        0.000000
25%        9.962403
50%       39.975000
75%       99.294435
max      700.890000
Name: As, dtype: float64

Arsenic of wells where people don't drink
count     71.000000
mean     171.105792
std      107.308224
min        1.368709
25%       81.614239
50%      150.250000
75%      250.245000
max      473.340000
Name: As, dtype: float64
```

What do the results above show?

I am going to come back to groupby here and there and we will do a whole packet on it. But when your brain can think through groupby it makes things simpler. So here we are going to groupby drink and then describe As. It should do what we just did in one line and make a nicer output.

In [71]:
```
df.groupby('Drink')['As'].describe()
```

Out[71]:

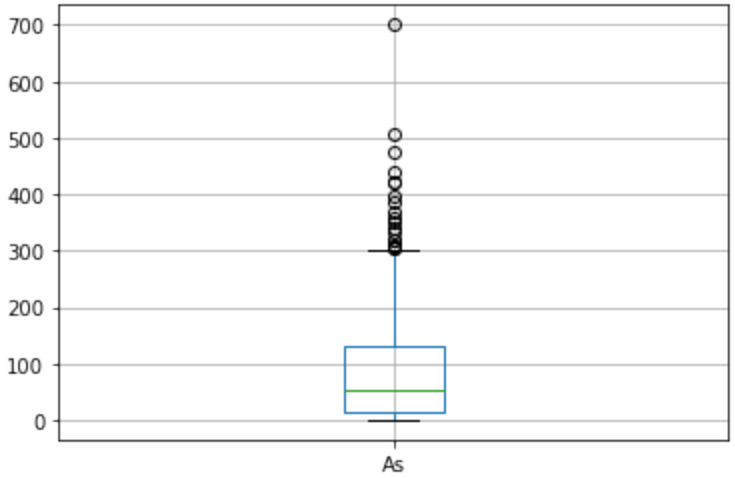
| Drink | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| N | 71.0 | 171.105792 | 107.308224 | 1.368709 | 81.614239 | 150.250 | 250.245000 | 473.34 |
| Y | 336.0 | 72.484421 | 91.571489 | 0.000000 | 9.962403 | 39.975 | 99.294435 | 700.89 |

In [ ]:
```

```

Could we disply this data?

In [72]:
```
df.boxplot(column='As')
```

Out[72]:
```
<AxesSubplot:>
```

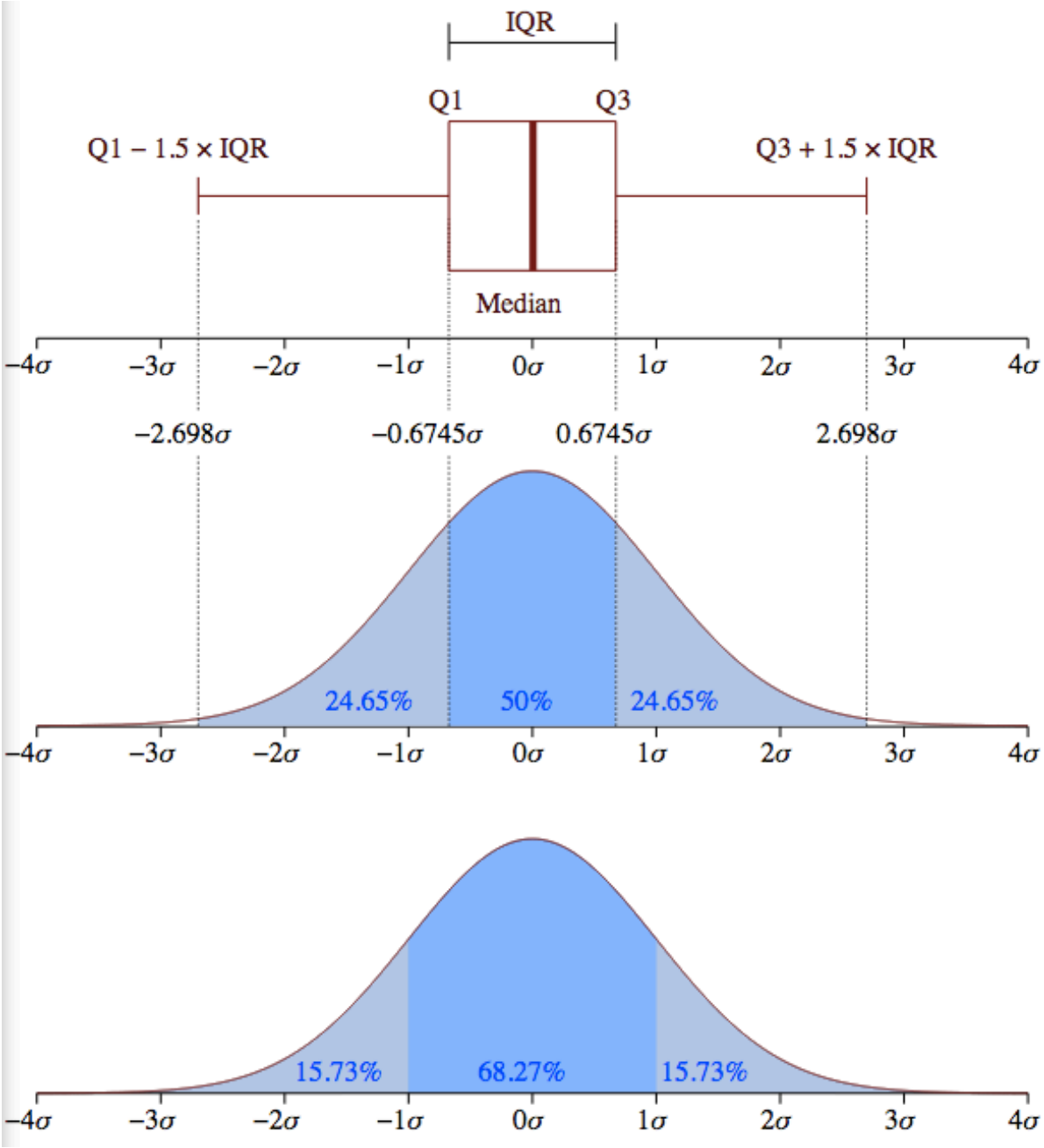



Do you remember what a boxplot shows? I found this next picture on stackoverflow. No need to import. Just for your reference.

http://stackoverflow.com/questions/17725927/boxplots-in-matplotlib-markers-and-outliers

In [67]:
```
from IPython.display import Image
Image(filename='boxplot_structure.png',width=600)
```

Out[67]:




## cool boxplots

But we really want two boxplots. One for people who drink and one for people who don't drink. I wasn't sure how to do it? So I googled pandas boxplot. Here are two of the links I got. See if you can figure it out! If you scroll down on the first link you should find the answer.... You will want your boxplots grouped. (only spend 2 minutes on this and I will come help you. Don't go down a rabbit hole on this. Answer at the ened)

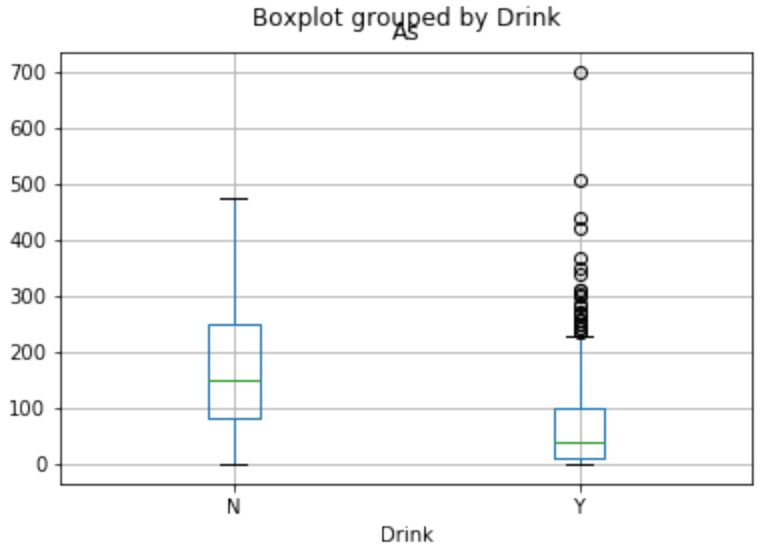
- click on the first link
- scroll down to where it says boxplots.
- Now scroll a little further to where you see the boxplots that say "grouped by x"
- look in the code.
- see if you can find a keyword argument in the parantheses that could hep you and figure out what column to pass

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.boxplot.html

http://stackoverflow.com/questions/23232989/boxplot-stratified-by-column-in-python-pandas

In [65]:

Out[65]: <matplotlib.axes._subplots.AxesSubplot at 0xc0542e8>




What difference do you notice about the arsenic concentrations of peeople drinking from their wells?

## Whenever you are comparing two populations a t-test should pop into your head!

# t-test or student t-test

A t-test tells you if there is a significant difference between two means. Actually it tells you the probability that they are the same. Back to our friend the p-value! The first website seems to have a good explanation.

Whenever you are comparing two means you run a t-test.

I am going to repeat this. If you ever compare two populations with a mean you need to run a t-test to see if the differences are statistically signifcant.

You then need to choose if it is

1. Paired scipy.stats.ttest_rel
2. not paired scipy.stats.ttest_ind

By paired we mean you repeated the measure on the same thing. Can you track the same thing across two samples. For example The exam score of the same person before and after an intervention

By not paired we mean two different populations. Imagine we fed 100 people carrots and 100 people steak and we weighed them and wanted to know if their exam scores was differnt. That is not paired and also the worst experiment ever!

In terms of our arsenic example if we measured the same wells twice it would be paired. if we measured different sets of wells it is unpaired.

Finally, if you are doing unpaired you need to decide if the groups have the same or unequal variance. It is statiscally safer to choose unequal variance. But you can always look at your variance and decide.

Your results are a t statistic and a p-value. We want our p-value less than 0.05 or 0.01 again!

Back to our wells. We will run an unparied t-test with unequal variance.

so lets pass our arrays from aboce with Arsenic for Drink=Y and Drink=N

We are asking if the difference we see with our eyes in the boxplot is statistically significant for arsenic. You need the stats to verify!

## THIS WILL FAIL!

In [74]:
```
stats.ttest_ind(df['As'][df.Drink=='Y']\
                ,df['As'][df.Drink=='N'])
```

Out[74]:
```
Ttest_indResult(statistic=nan, pvalue=nan)
```

It failed b/c we have NaN's in our data (Not a Number). NaN's are nice as they keep track where we don't have data.

But scipy does not handle NaN's well.

So we need to get rid of them then do the math.

In pandas terms we need to drop the NaN's using the function dropna

In [75]:
```
df['As'][df.Drink=='Y'].dropna()
```

Out[75]:
```
Well_ID
83         78.977470
101        28.070949
110        96.885674
112        80.627214
153        39.249817
             ...
12363      26.980000
12440      21.740000
12461     117.820000
12516       0.130000
12654      17.390000
Name: As, Length: 336, dtype: float64
```

So try again!

In [76]:
```
stats.ttest_ind(df['As'][df.Drink=='Y'].dropna()
                ,df['As'][df.Drink=='N'].dropna()
                 ,equal_var=False)
```

Out[76]:
```
Ttest_indResult(statistic=-7.209206229150192, pvalue=1.4829579464861492e-10)
```

That is a small p-value!!!!!

So a signficant difference!

You could say the mean arsenic concentration is lower in well where people drink then where they don't drink (p<0.01)

## What wells do people drink from?

For our final exercise. Lets put it together and get data and then see if we can plot it. I want to know the number of people who are drinking from there wells based on the arsenic concentrations. Can we do the reverse. if the arsenic is <10,10-50, and >50 what is the value counts of drinking and not drinking. I chose these numbers because 10 ppb is the EPA and WHO drinking water limit. 50 ppb is the Bangladesh drinking water limit. We see negative health effects at 10ppb. Drinking water with 10 ppb arsenic is bad for you! It increases your risk of cardiovascular disease, cancers, and death!

I would first just try and break the data into 3 groups and print out the results. So use your way of selecting data and select data based on the levels of arsenic. To do between 10 and 50 you will need to use an and statement and how to do those is different. you need to use a boolean function to choose two data sets! works by taking two arguments and then returning what happens the same way as if you did an and. but it works better. Remember we did this above.

Three Groups

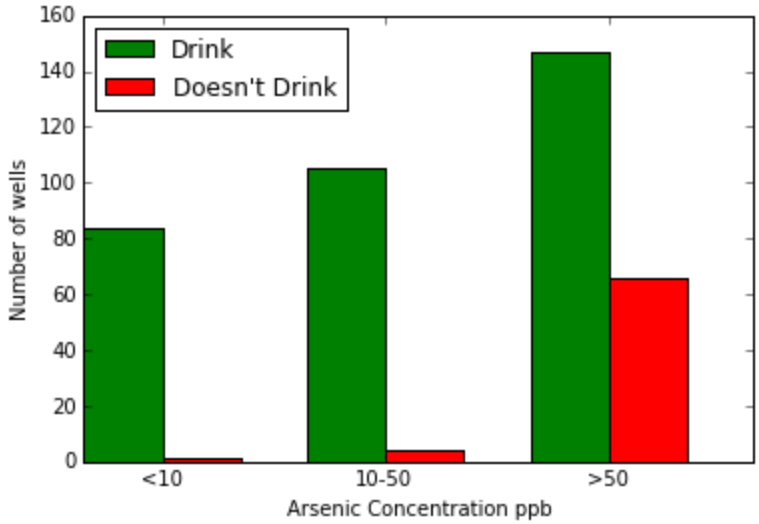
1. <10 ppb arsenic
2. 10-50 ppb arsenic
3. >50 ppb arsenic

4. Print out the number of people drinking from wells with arsenic less than 10. you can use value_counts() and your selection method.

This is the graph we want to make

In [145…

Out[145…
```
<matplotlib.text.Text at 0x11506128>
```




First start by counting who drinks less than 10.

In [77]:
```
print ('people drinking with arsenic <10')

df['Drink'][df.As<=10].value_counts()
```

```
people drinking with arsenic <10
```

Out[77]:
```
Y    84
N     1
Name: Drink, dtype: int64
```

1. Next use determine the people drinking from wells with arsenic more than 50.

In [113…

```
people drinking with >50
Y    147
N     66
dtype: int64
```

1. Now use your logical_and() or & and parantheses to determine between 10 and 50.

In [115…

```
people drinking with 10-50
Y    105
N      4
dtype: int64
```

Looking at the data one by one is painful. Lets work on getting to our bar chart. This is a bad way of looking at the data. I would like to make bar plot. Here is my goal. Can we get there? Follow the next steps after the plot and see how it goes!

python does not make bar plots easy. But let's make one anyway

First lets look up bar plot. http://matplotlib.org/examples/api/barchart_demo.html This is the example on all the web pages. We can make sense of it. Lets do one step at a time. What plt.bar wants is (x,y,width). lets do it for As<10 first. Here is our data again. The x location and width become arbitrary to make it look pretty.

In [85]:
```
df['Drink'][df.As<=10].value_counts()
```

Out[85]:
```
Y    84
N     1
Name: Drink, dtype: int64
```

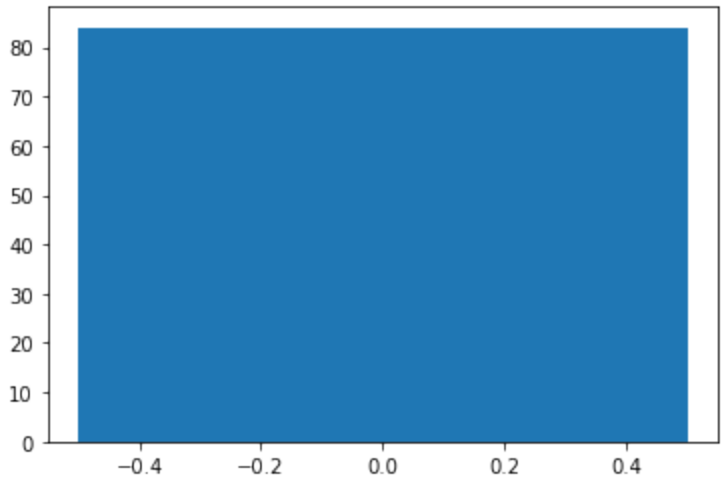
So we want to plot 84 Drink, 1 Doesn't drink. I will do it longhand first time.

remember it is ax.bar(x,y,width)

x=where to plot it and is a bit of a dumy variable y=the height of the bar width=how wide you want the bars

In [86]:
```
# bar for people who drink
fig,ax=plt.subplots()
ax.bar(0,84,1)
```

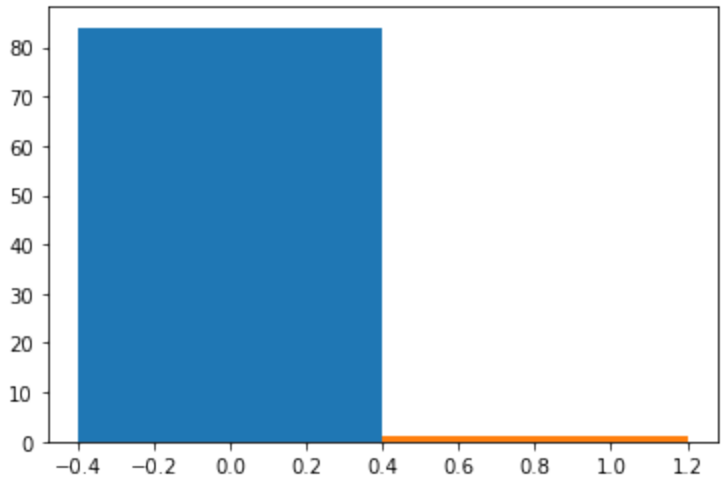
Out[86]:
```
<BarContainer object of 1 artists>
```




The default width is 0.8 starting from 0. Now we need to add the doesn't drink.

In [88]:
```
# bar for people who drink and don't drink
fig,ax=plt.subplots()
ax.bar(0,84,.8)
ax.bar(0.8,1,.8)
```

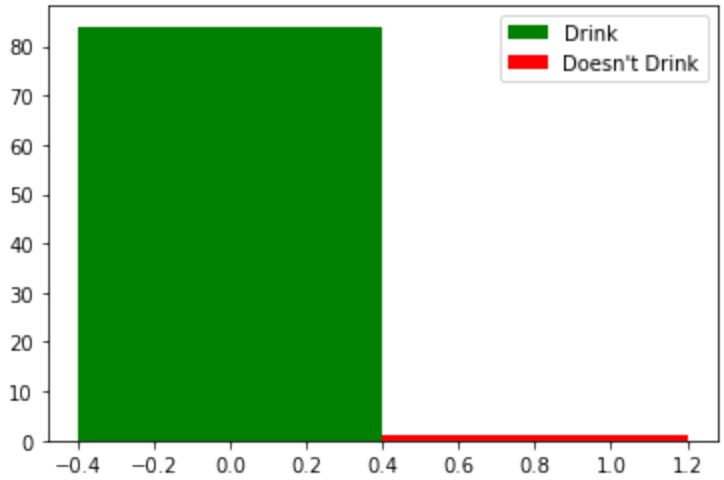
Out[88]:
```
<BarContainer object of 1 artists>
```



Now we need to add colors and labels for a legend.

In [87]:
```
fig,ax=plt.subplots()
ax.bar(0,84,0.8,color='g',label='Drink')
ax.bar(0.8,1,0.8,color='r',label="Doesn't Drink")
#I did double quotes so I could print the single quote
ax.legend(loc='best')
```

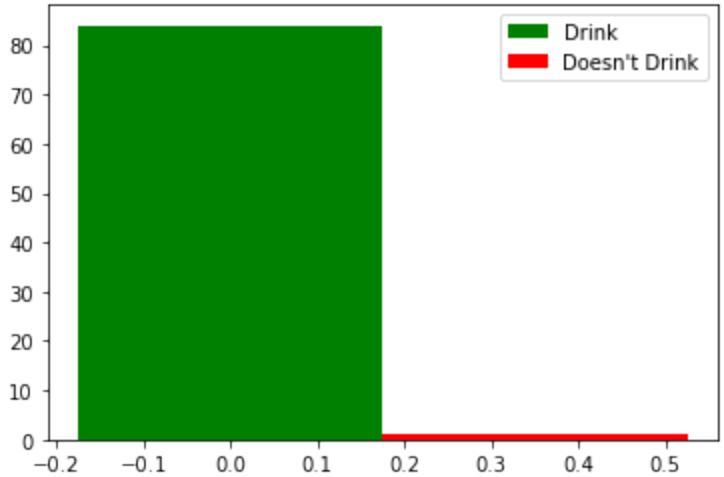
Out[87]:
```
<matplotlib.legend.Legend at 0x7f8862c5afd0>
```




this is a disaster. We can't hard wire it all. We need to be better in our programming and be what people call pythonic. now instead of setting the x-axis to zero lets use np.arange. Then also lets set the width. We will also make the second bar start at one width

In [79]:
```
fig,ax=plt.subplots()
width=0.35
xvalues=np.arange(1)
ax.bar(xvalues,84,width,color='g',label='Drink')
ax.bar(xvalues+width,1,width,color='r',label="Doesn't Drink")
#I did double quotes so I could print the single quote
ax.legend(loc='best')
```

Out[79]:
```
<matplotlib.legend.Legend at 0x7f7fdf6d2880>
```




Now we are starting to make progress. But we need the other two sets of bars. We will need a set of yes and no values. so we need yes[0],yes[1],yes[2] representing our values. I would make

a nump array of zeros and then fill it in. So to make a numpy array of zeros. then fill in the array. we know the length has to be three.

In [90]:
```
yes=np.zeros(3)
print (yes)
```

```
[0. 0. 0.]
```

Now do the same for no. then set each one equal to the correct result that you have above where you printed out the results. don't print the results like you did above. set them to yes,no given the correct array spot. At the end you should now have yes and no set for the three levels.

In [85]:
```

```

```
[ 0.  0.  0.]
```

Now I will show you how to add the first yes and no

In [91]:
```
yes=np.zeros(3)
no=np.zeros(3)
yes[0],no[0]=df['Drink'][df.As<=10].value_counts()
print ('yes',yes)
print ('no',no)
```

```
yes [84.  0.  0.]
no [1. 0. 0.]
```

Now can you do the other two?

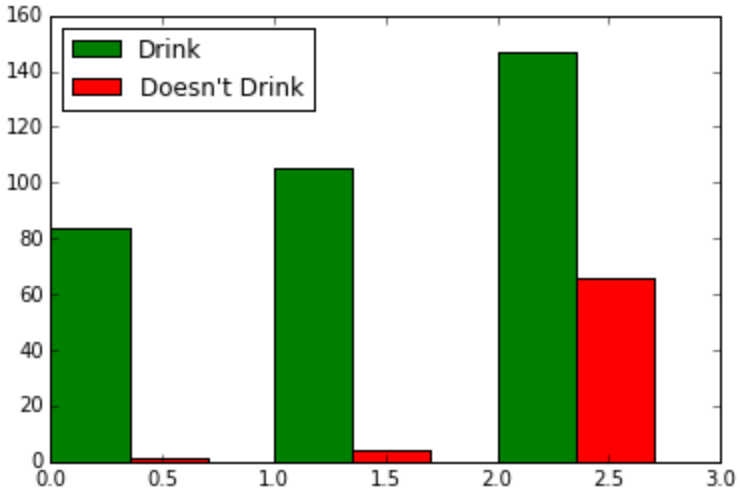
In [140…
```

```

```
[  84.  105.  147.] [  1.   4.  66.]
```

Now we can do a bar plot of yes and no. Go copy and past your barplot code from above. but now make the x-axis have an np.arange of 3 b/c we want 3 locations. And don't use the hardwired number put in your new yes and no arrays you just made.

In [141…
```

```

Out[141… <matplotlib.legend.Legend at 0x106ed908>

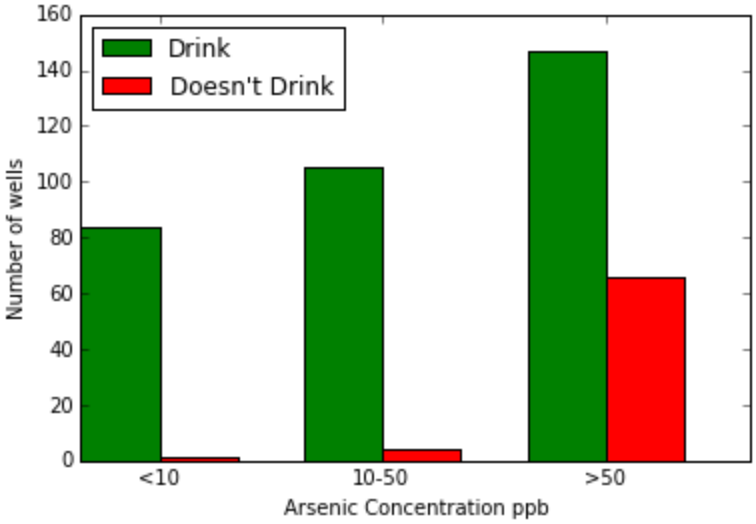



Now you are looking great with a wonderful graph. lets label everything. We just need an x-axis labeled correctly. Also, I would put all the code in one cell so it always works smoothly. If we go

back to our webpage with the example we can use ax.set_xticks(xvalues+width/2) to get us the xticks we want. then we can add ax.set_xticklabels(('names','names','names')). We can also use ax.set_xlabel() and ax.set_ylabel()

In [144…

Out[144… <matplotlib.text.Text at 0x10f052e8>



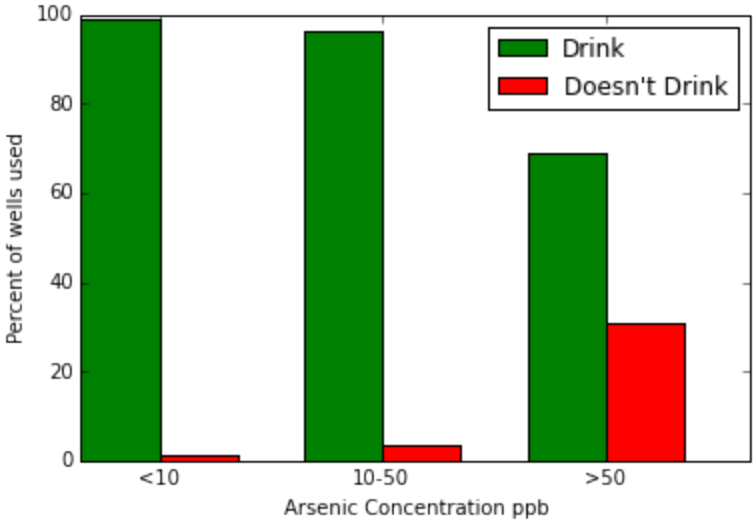

Now that is a great looking graph. You just need to add a figure caption. I would write something like

Number of wells categorized by if the respondents drink or don't drink from the well and stratified by arsenic concentration.

As a total bonus and if you have time you could change it from the number of wells to the proportion of wells in each category.

In [100…

Out[100… <matplotlib.text.Text at 0x19af3cc0>

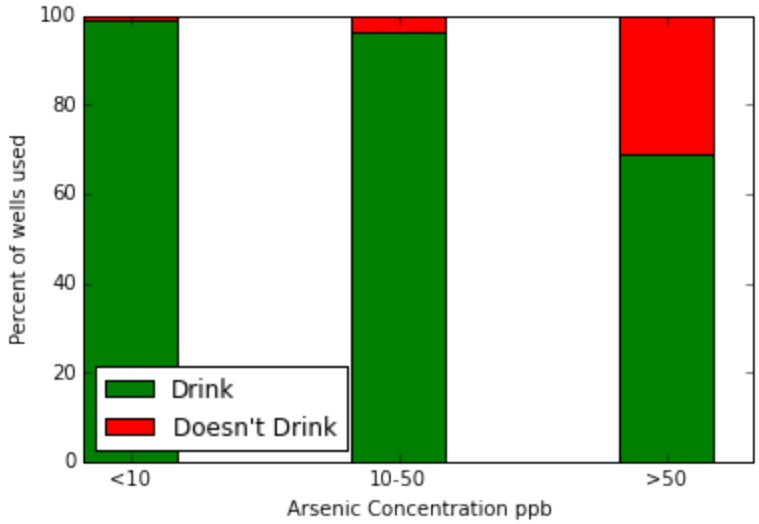



I thought it might be nice to stack the bars since they add up to 100..... See http://matplotlib.org/examples/pylab_examples/bar_stacked.html It is "easy" I used the bottom

keyword. Then I removed the width offset and tweaked a few other things

In [106…

Out[106… <matplotlib.text.Text at 0x1a6b4c18>

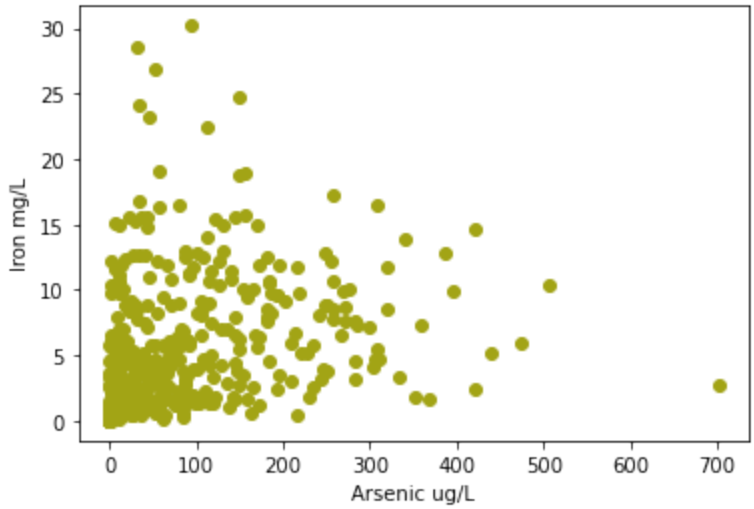



## Some homework hints.

for the homework you will need to make scatter plots. They are easy to make in pandas. Here is one of Arsenic versus Iron. You can label your axes and change the color of your symbols.

In [92]:
```
fig,ax=plt.subplots()
ax.scatter(df['As'],df['Fe'],c='xkcd:vomit')
ax.set_xlabel('Arsenic ug/L')
ax.set_ylabel('Iron mg/L')
```

Out[92]: Text(0, 0.5, 'Iron mg/L')




In [ ]:

In [ ]:

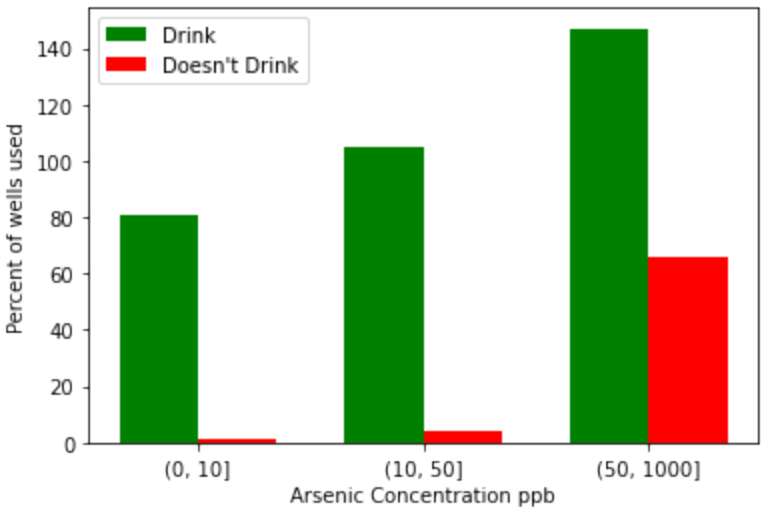
## Below is a simpler method

We will learn more about this but I did advanced Python.

1. I defined bins
2. I used groubpy to group the data by if people drink
3. I added a cut value to also group by the bins I cut and set by.
4. I then filled the group by the counts.
5. I then unstack and transpose and flip the matrix.
6. Then I can plot that new data.
7. The fun part is I can change the bins and it automtacillay updates!

In [109…
```
bins=[0,10,50,1000]
df_No_Yes=df.groupby(['Drink',pd.cut(df['As'],bins)])\
                        .As.count().unstack().transpose()

fig,ax=plt.subplots(1,1)
width=0.35
xvalues=np.arange(df_No_Yes.shape[0])
ax.bar(xvalues,df_No_Yes.Y,width,color='g',label='Drink')
ax.bar(xvalues+width,df_No_Yes.N,width,color='r',label="Doesn't Drink")
#I did double quotes so I could print the single quote

ax.legend(loc='best')
#You can try numbers 1-8 for location. see http://matplotlib.org/1.3.1/users/leg

ax.set_xticks(xvalues+width/2)
ax.set_xticklabels(df_No_Yes.index.values)#('<10','10-50','>50'))
ax.set_xlabel('Arsenic Concentration ppb')
ax.set_ylabel('Percent of wells used')
```

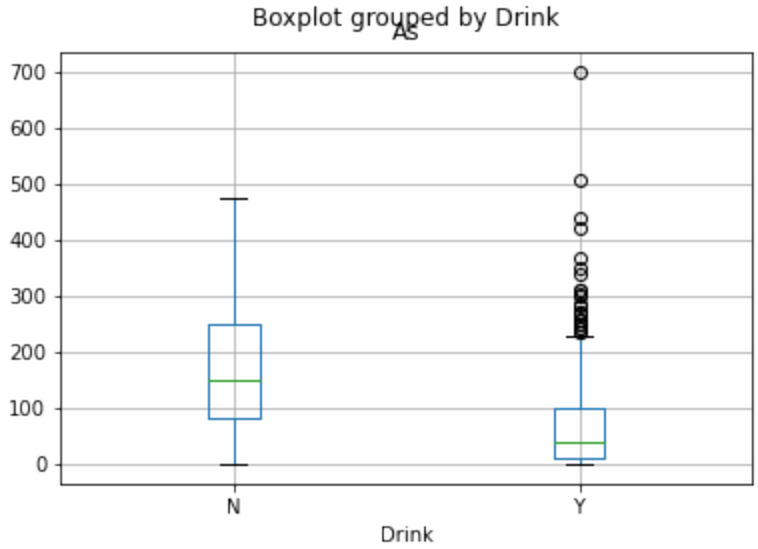
Out[109… Text(0, 0.5, 'Percent of wells used')




# Answers

In [73]:
```
df.boxplot(column='As',by='Drink')
```

Out[73]: <AxesSubplot:title={'center':'As'}, xlabel='Drink'>




In [79]:
```
print ('people drinking with <10')

df['Drink'][df.As<=10].value_counts()
```

```
people drinking with <10
```

Out[79]:
```
Y    84
N     1
Name: Drink, dtype: int64
```

In [80]:
```
print ('\npeople drinking with >50 ')

df['Drink'][df.As>=50].value_counts()
```

```
people drinking with >50
```

Out[80]:
```
Y    147
N     66
Name: Drink, dtype: int64
```

In [84]:
```
print ('people drinking with 10-50')

df['Drink'][(df.As<=50)&(df.As>=10)].value_counts()

# using np.logical_and
#df['Drink'][np.logical_and(df.As<=50,df.As>=10)].value_counts()
```

```
people drinking with 10-50
```

Out[84]:
```
Y    105
N      4
Name: Drink, dtype: int64
```

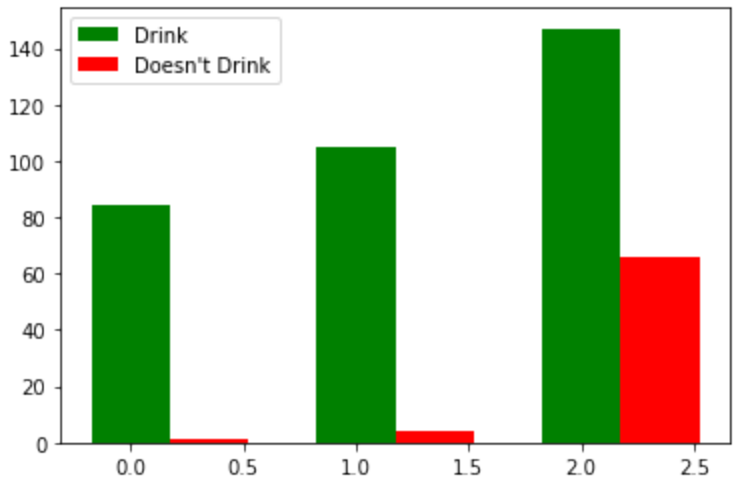
In [94]:
```
yes=np.zeros(3)
no=np.zeros(3)
yes[0],no[0]=df['Drink'][df.As<=10].value_counts()
yes[1],no[1]=df['Drink'][(df.As<=50)&(df.As>=10)].value_counts()
yes[2],no[2]=df['Drink'][df.As>=50].value_counts()
print (yes,no)
```

```
[ 84. 105. 147.] [ 1.  4. 66.]
```

In [95]:
```
fig,ax=plt.subplots(1,1)
width=0.35
xvalues=np.arange(3)
ax.bar(xvalues,yes,width,color='g',label='Drink')
```

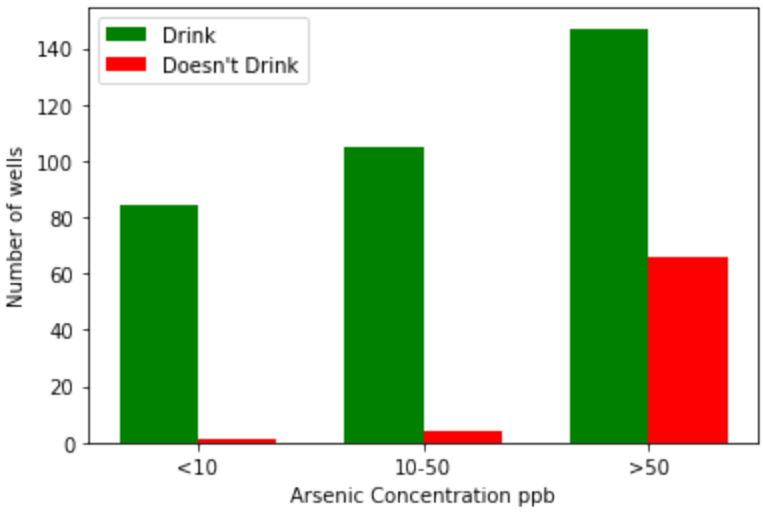
```
ax.bar(xvalues+width,no,width,color='r',label="Doesn't Drink") #I did double quo
ax.legend(loc='best')
```

Out[95]: <matplotlib.legend.Legend at 0x7f8862f94070>



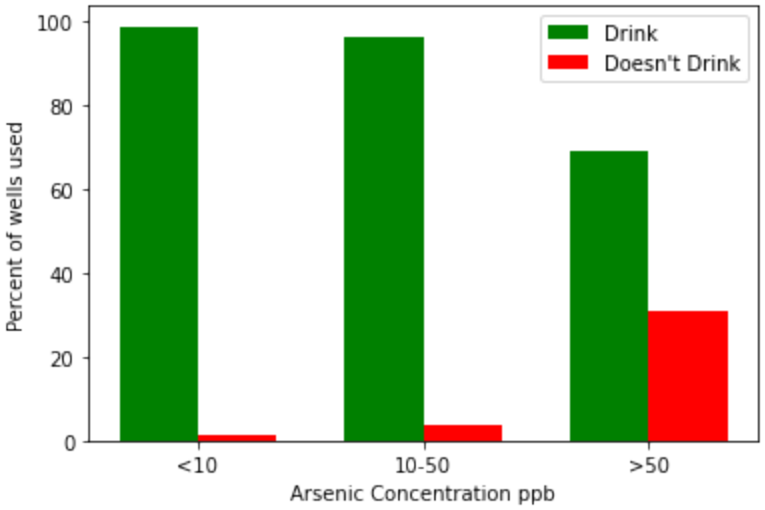

In [96]:
```
yes=np.zeros(3)
no=np.zeros(3)
yes[0],no[0]=df['Drink'][df.As<=10].value_counts()
yes[1],no[1]=df['Drink'][(df.As<=50)&(df.As>=10)].value_counts()
yes[2],no[2]=df['Drink'][df.As>=50].value_counts()


fig,ax=plt.subplots()
width=0.35
xvalues=np.arange(3)
ax.bar(xvalues,yes,width,color='g',label='Drink')
ax.bar(xvalues+width,no,width,color='r',label="Doesn't Drink")
#I did double quotes so I could print the single quote
ax.legend(loc='best')
ax.set_xticks(xvalues+width/2)
ax.set_xticklabels(('<10','10-50','>50'))
ax.set_xlabel('Arsenic Concentration ppb')
ax.set_ylabel('Number of wells')
```

Out[96]: Text(0, 0.5, 'Number of wells')



In [105…
```
yes=np.zeros(3)
no=np.zeros(3)
yes[0],no[0]=df['Drink'][df.As<=10].value_counts()\
            /df['Drink'][df.As<=10].count()*100

yes[1],no[1]=df['Drink'][(df.As<=50)&(df.As>=10)].value_counts()\
            /df['Drink'][(df.As<=50)&(df.As>=10)].count()*100

yes[2],no[2]=df['Drink'][df.As>=50].value_counts()\
            /df['Drink'][df.As>=50].count()*100


fig,ax=plt.subplots(1,1)
width=0.35
xvalues=np.arange(3)
ax.bar(xvalues,yes,width,color='g',label='Drink')
ax.bar(xvalues+width,no,width,color='r',label="Doesn't Drink") #I did double quo
ax.legend(loc='best')
ax.set_xticks(xvalues+width/2)
ax.set_xticklabels(('<10','10-50','>50'))
ax.set_xlabel('Arsenic Concentration ppb')
ax.set_ylabel('Percent of wells used')
```

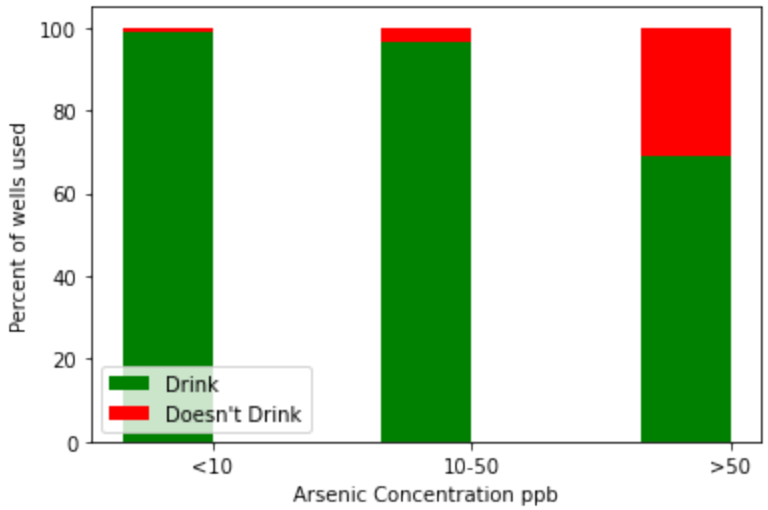
Out[105… Text(0, 0.5, 'Percent of wells used')




In [107…
```
yes=np.zeros(3)
no=np.zeros(3)
yes[0],no[0]=df['Drink'][df.As<=10].value_counts()\
            /df['Drink'][df.As<=10].count()*100

yes[1],no[1]=df['Drink'][(df.As<=50)&(df.As>=10)].value_counts()\
            /df['Drink'][(df.As<=50)&(df.As>=10)].count()*100

yes[2],no[2]=df['Drink'][df.As>=50].value_counts()\
            /df['Drink'][df.As>=50].count()*100


fig,ax=plt.subplots(1,1)
width=0.35
xvalues=np.arange(3)
ax.bar(xvalues,yes,width,color='g',label='Drink')
ax.bar(xvalues,no,width,color='r',bottom=yes,label="Doesn't Drink")
```

```
#I did double quotes so I could print the single quote
ax.legend(loc=3)  #You can try numbers 1-8 for location. see http://matplotlib.o
ax.set_xticks(xvalues+width/2)
ax.set_xticklabels(('<10','10-50','>50'))
ax.set_xlabel('Arsenic Concentration ppb')
ax.set_ylabel('Percent of wells used')
```

Out[107… Text(0, 0.5, 'Percent of wells used')




In [ ]:

In [ ]: