Engineering Computations

Module 2: Take off with stats

Lorena A. Barba and Natalia C. Clementi

Lesson 1: Cheers! Stats with Beers

Welcome to the second module of our course in *Engineering Computations*, for undergraduate engineering students. This module explores practical statistical analysis with Python.

This first lesson explores how we can answer questions using data combined with practical methods from statistics. We'll need some fun data to work with. We found a neat data set of canned craft beers in the US, scraped from the web and cleaned up by Jean-Nicholas Hould (@NicholasHould on Twitter)—who we want to thank for having a permissive license on his GitHub repository so we can reuse his work!

The data source (@craftcans on Twitter) doesn't say that the set includes *all* the canned beers brewed in the country. So we have to asume that the data is a sample and may contain biases.

We'll manipulate the data using **NumPy**—the array library for Python that we learned about in Module 1, lesson 4. But we'll also learn about a new Python library for data analysis called **pandas**.

`pandas` is an open-source library providing high-performance, easy-to-use data structures and data-analysis tools. Even though `pandas` is great for data analysis, we won't exploit all its power in this lesson. But we'll learn more about it later on!

We'll use `pandas` to read the data file (in `csv` format), display it in a nice table, and extract the columns that we need—which we'll convert to `numpy` arrays to work with.

Let's start by importing the two Python libraries.

```
In [1]: import pandas
        import numpy
```

1 Read the data file

Below, we'll take a peek into the data file, `beers.csv`, using the system command `head` (which we can use with a bang, thanks to IPython).

Note: If you downloaded this notebook alone, rather than the full collection for this course, you may not have the data file on the location we assume below. In that case, you can download the data if you add a code cell, and execute the following code in it:

```
from urllib.request import urlretrieve
URL = 'http://go.gwu.edu/engcomp2data1?accessType=DOWNLOAD'
urlretrieve(URL, 'beers.csv')
```

The data file will be downloaded to your working directory, and you will then need to remove the path information, i.e., the string '../../data/', below.

```
In [2]: !head "../../data/beers.csv"
```

```
,abv,ibu,id,name,style,brewery_id,ounces
0,0.05,,1436,Pub Beer,American Pale Lager,408,12.0
1,0.066,,2265,Devil's Cup,American Pale Ale (APA),177,12.0
2,0.071,,2264,Rise of the Phoenix,American IPA,177,12.0
3,0.09,,2263,Sinister,American Double / Imperial IPA,177,12.0
4,0.075,,2262,Sex and Candy,American IPA,177,12.0
5,0.077,,2261,Black Exodus,Oatmeal Stout,177,12.0
6,0.045,,2260,Lake Street Express,American Pale Ale (APA),177,12.0
7,0.065,,2259,Foreman,American Porter,177,12.0
8,0.055,,2258,Jade,American Pale Ale (APA),177,12.0
```

We can use pandas to read the data from the csv file, and save it into a new variable called beers. Let's then check the type of this new variable—rememeber that we can use the function type() to do this.

```
In [3]: beers = pandas.read_csv("../../data/beers.csv")
```

```
In [4]: type(beers)
```

```
Out[4]: pandas.core.frame.DataFrame
```

This is a new data type for us: a pandas DataFrame. From the pandas documentation: "A DataFrame is a 2-dimensional labeled data structure with columns of potentially different types" [4]. You can think of it as the contens of a spreadsheet, saved into one handy Python variable. If you print it out, you get a nicely laid-out table:

```
In [5]: beers
```

```
Out[5]:       Unnamed: 0    abv    ibu    id  \
        0              0  0.050    NaN  1436
        1              1  0.066    NaN  2265
        2              2  0.071    NaN  2264
        3              3  0.090    NaN  2263
        4              4  0.075    NaN  2262
        5              5  0.077    NaN  2261
        6              6  0.045    NaN  2260
        7              7  0.065    NaN  2259
        8              8  0.055    NaN  2258
        9              9  0.086    NaN  2131
        10            10  0.072    NaN  2099
        11            11  0.073    NaN  2098
        12            12  0.069    NaN  2097
        13            13  0.085    NaN  1980
        14            14  0.061   60.0  1979
        15            15  0.060    NaN  2318
        16            16  0.060    NaN  2170
```

```
17              17  0.060    NaN  2169
18              18  0.060    NaN  1502
19              19  0.082    NaN  1593
20              20  0.082    NaN  1592
21              21  0.099   92.0  1036
22              22  0.079   45.0  1024
23              23  0.079    NaN   976
24              24  0.044   42.0   876
25              25  0.049   17.0   802
26              26  0.049   17.0   801
27              27  0.049   17.0   800
28              28  0.070   70.0   799
29              29  0.070   70.0   797
...            ...    ...    ...   ...
2380          2380  0.080   31.0   761
2381          2381  0.055    NaN  2149
2382          2382  0.071   60.0  2148
2383          2383  0.052    NaN  2147
2384          2384  0.048   38.0  2146
2385          2385  0.059    NaN  2047
2386          2386  0.062   61.0  1470
2387          2387  0.045   23.0  1469
2388          2388  0.058   72.0  2627
2389          2389  0.045    NaN  2626
2390          2390  0.059  135.0  1676
2391          2391  0.047   15.0  1468
2392          2392  0.050    NaN   822
2393          2393  0.065   82.0  2417
2394          2394  0.028   15.0  2306
2395          2395  0.065   69.0  1697
2396          2396  0.069   69.0  2194
2397          2397  0.045   25.0  1514
2398          2398  0.077   30.0  1513
2399          2399  0.069   69.0  1512
2400          2400  0.060   50.0  1511
2401          2401  0.042    NaN  1345
2402          2402  0.082    NaN  1316
2403          2403  0.055    NaN  1045
2404          2404  0.075    NaN  1035
2405          2405  0.067   45.0   928
2406          2406  0.052    NaN   807
2407          2407  0.055    NaN   620
2408          2408  0.055   40.0   145
2409          2409  0.052    NaN    84

                                                 name  \
0                                            Pub Beer
1                                         Devil's Cup
```

2	Rise of the Phoenix
3	Sinister
4	Sex and Candy
5	Black Exodus
6	Lake Street Express
7	Foreman
8	Jade
9	Cone Crusher
10	Sophomoric Saison
11	Regional Ring Of Fire
12	Garce Selé
13	Troll Destroyer
14	Bitter Bitch
15	Ginja Ninja
16	Cherried Away
17	Rhubarbarian
18	BrightCider
19	He Said Baltic-Style Porter
20	He Said Belgian-Style Tripel
21	Lower De Boom
22	Fireside Chat
23	Marooned On Hog Island
24	Bitter American
25	Hell or High Watermelon Wheat (2009)
26	Hell or High Watermelon Wheat (2009)
27	21st Amendment Watermelon Wheat Beer (2006)
28	21st Amendment IPA (2006)
29	Brew Free! or Die IPA (2008)
...	...
2380	P-51 Porter
2381	#001 Golden Amber Lager
2382	#002 American I.P.A.
2383	#003 Brown & Robust Porter
2384	#004 Session I.P.A.
2385	Tarasque
2386	Ananda India Pale Ale
2387	Tiny Bomb
2388	Train Hopper
2389	Edward's Portly Brown
2390	Troopers Alley IPA
2391	Wolverine Premium Lager
2392	Woodchuck Amber Hard Cider
2393	4000 Footer IPA
2394	Summer Brew
2395	Be Hoppy IPA
2396	Worthy IPA
2397	Easy Day Kolsch
2398	Lights Out Vanilla Cream Extra Stout

2399	Worthy IPA (2013)
2400	Worthy Pale
2401	Patty's Chile Beer
2402	Colorojo Imperial Red Ale
2403	Wynkoop Pumpkin Ale
2404	Rocky Mountain Oyster Stout
2405	Belgorado
2406	Rail Yard Ale
2407	B3K Black Lager
2408	Silverback Pale Ale
2409	Rail Yard Ale (2009)

	style	brewery_id	ounces
0	American Pale Lager	408	12.0
1	American Pale Ale (APA)	177	12.0
2	American IPA	177	12.0
3	American Double / Imperial IPA	177	12.0
4	American IPA	177	12.0
5	Oatmeal Stout	177	12.0
6	American Pale Ale (APA)	177	12.0
7	American Porter	177	12.0
8	American Pale Ale (APA)	177	12.0
9	American Double / Imperial IPA	177	12.0
10	Saison / Farmhouse Ale	177	12.0
11	Saison / Farmhouse Ale	177	12.0
12	Saison / Farmhouse Ale	177	12.0
13	Belgian IPA	177	12.0
14	American Pale Ale (APA)	177	12.0
15	Cider	154	12.0
16	Cider	154	12.0
17	Cider	154	12.0
18	Cider	154	12.0
19	Baltic Porter	368	12.0
20	Tripel	368	12.0
21	American Barleywine	368	8.4
22	Winter Warmer	368	12.0
23	American Stout	368	12.0
24	American Pale Ale (APA)	368	12.0
25	Fruit / Vegetable Beer	368	12.0
26	Fruit / Vegetable Beer	368	12.0
27	Fruit / Vegetable Beer	368	12.0
28	American IPA	368	12.0
29	American IPA	368	12.0
...	...	...	...
2380	American Porter	509	16.0
2381	American Amber / Red Lager	211	12.0
2382	American IPA	211	12.0
2383	American Porter	211	12.0

```
2384                   American IPA        211     12.0
2385          Saison / Farmhouse Ale       239     12.0
2386                   American IPA        239     12.0
2387               American Pilsner        239     12.0
2388                   American IPA         14     12.0
2389             American Brown Ale         14     12.0
2390                   American IPA        344     12.0
2391            American Pale Lager        402     12.0
2392                          Cider        501     12.0
2393                   American IPA        109     12.0
2394               American Pilsner        109     12.0
2395                   American IPA        339     16.0
2396                   American IPA        199     12.0
2397                         Kölsch        199     12.0
2398 American Double / Imperial IPA        199     12.0
2399                   American IPA        199     12.0
2400        American Pale Ale (APA)        199     12.0
2401                     Chile Beer        424     12.0
2402            American Strong Ale        424     12.0
2403                    Pumpkin Ale        424     12.0
2404                 American Stout        424     12.0
2405                    Belgian IPA        424     12.0
2406       American Amber / Red Ale        424     12.0
2407                    Schwarzbier        424     12.0
2408        American Pale Ale (APA)        424     12.0
2409       American Amber / Red Ale        424     12.0

[2410 rows x 8 columns]
```

Inspect the table above. The first column is a numbering scheme for the beers. The other columns contain the following data:

- `abv`: Alcohol-by-volume of the beer.
- `ibu`: International Bittering Units of the beer.
- `id`: Unique identifier of the beer.
- `name`: Name of the beer.
- `style`: Style of the beer.
- `brewery_id`: Unique identifier of the brewery.
- `ounces`: Ounces of beer in the can.

2 Explore the data

In the field of statistics, Exploratory Data Analysis (EDA) has the goal of summarizing the main features of our data, and seeing what the data can tell us without formal modeling or hypothesis-testing. [2]

Let's start by extracting the columns with the `abv` and `ibu` values, and converting them to NumPy arrays. One of the advantages of data frames in `pandas` is that we can access a column simply using its header, like this:

```
data_frame['name_of_column']
```

The output of this action is a pandas Series. From the documentation: "a Series is a 1-dimensional labeled array capable of holding any data type." [4]

Check the type of a column extracted by header:

```
In [6]: type(beers['abv'])
```

```
Out[6]: pandas.core.series.Series
```

Of course, you can index and slice a data series like you know how to do with strings, lists and arrays. Here, we display the first ten elements of the abv series:

```
In [7]: beers['abv'][:10]
```

```
Out[7]: 0    0.050
        1    0.066
        2    0.071
        3    0.090
        4    0.075
        5    0.077
        6    0.045
        7    0.065
        8    0.055
        9    0.086
        Name: abv, dtype: float64
```

Inspect the data in the table again: you'll notice that there are NaN (not-a-number) elements in both the abv and ibu columns. Those values mean that there was no data reported for that beer. A typical task when cleaning up data is to deal with these pesky NaNs.

Let's extract the two series corresponding to the abv and ibu columns, clean the data by removing all NaN values, and then access the values of each series and assign them to a NumPy array.

```
In [8]: abv_series = beers['abv']
```

```
In [9]: len(abv_series)
```

```
Out[9]: 2410
```

Another advantage of pandas is that it has the ability to handle missing data. The data-frame method dropna() returns a new data frame with only the good values of the original: all the null values are thrown out. This is super useful!

```
In [10]: abv_clean = abv_series.dropna()
```

Check out the length of the cleaned-up abv data; you'll see that it's shorter than the original. NaNs gone!

```
In [11]: len(abv_clean)
```

```
Out[11]: 2348
```

Remember that a a pandas *Series* consists of a column of values, and their labels. You can extract the values via the `series.values` attribute, which returns a `numpy.ndarray` (multidimensional array). In the case of the `abv_clean` series, you get a one-dimensional array. We save it into the variable name `abv`.

```
In [12]: abv = abv_clean.values
```

```
In [13]: print(abv)
```

```
[ 0.05    0.066   0.071 ...,   0.055   0.055   0.052]
```

```
In [14]: type(abv)
```

```
Out[14]: numpy.ndarray
```

Now we repeat the whole process for the `ibu` column: extract the column into a series, clean it up removing `NaN`s, extract the series values as an array, check how many values we lost.

```
In [15]: ibu_series = beers['ibu']

         len(ibu_series)
```

```
Out[15]: 2410
```

```
In [16]: ibu_clean = ibu_series.dropna()

         ibu = ibu_clean.values

         len(ibu)
```

```
Out[16]: 1405
```

Exercise Write a Python function that calculates the percentage of missing values for a certain data series. Use the function to calculate the percentage of missing values for the `abv` and `ibu` data sets.

For the original series, before cleaning, remember that you can access the values with `series.values` (e.g., `abv_series.values`).

```
In [ ]:
```

Important: Notice that in the case of the variable `ibu` we are missing almost 42% of the values. This is important, because it will affect our analysis. When we do descriptive statistics, we will ignore these missing values, and having 42% missing will very likely cause bias.

3 Ready, stats, go!

Now that we have NumPy arrays with clean data, let's see how we can manipulate them to get some useful information.

Focusing on the numerical variables `abv` and `ibu`, we'll walk through some "descriptive statistics," below. In other words, we aim to generate statistics that summarize the data concisely.

3.1 Maximum and minimum

The maximum and minimum values of a dataset are helpful as they tell us the *range* of our sample: the range gives some indication of the *variability* in the data. We can obtain them for our `abv` and `ibu` arrays with the `min()` and `max()` functions from NumPy.

abv

```
In [17]: abv_min = numpy.min(abv)
         abv_max = numpy.max(abv)
```

```
In [18]: print('The minimum value for abv is: ', abv_min)
         print('The maximum value for abv is: ', abv_max)
```

```
The minimum value for abv is:  0.001
The maximum value for abv is:  0.128
```

ibu

```
In [19]: ibu_min = numpy.min(ibu)
         ibu_max = numpy.max(ibu)
```

```
In [20]: print('The minimum value for ibu is: ', ibu_min)
         print('The maximum value for ibu is: ', ibu_max)
```

```
The minimum value for ibu is:  4.0
The maximum value for ibu is:  138.0
```

3.2 Mean value

The **mean** value is one of the main measures to describe the central tendency of the data: an indication of where's the "center" of the data. If we have a sample of N values, x_i, the mean, $\bar{x}$, is calculated by:

$$\bar{x} = \frac{1}{N}\sum_i x_i$$

In words, that is the sum of the data values divided by the number of values, N.

You've already learned how to write a function to compute the mean in Module 1 Lesson 5, but you also learned that NumPy has a built-in `mean()` function. We'll use this to get the mean of the `abv` and `ibu` values.

```
In [21]: abv_mean = numpy.mean(abv)
         ibu_mean = numpy.mean(ibu)
```

Next, we'll print these two variables, but we'll use some fancy new way of printing with Python's string formatter, `string.format()`. There's a sweet site dedicated to Python's string formatter, called PyFormat, where you can learn lots of tricks!

The basic trick is to use curly brackets `{}` as placeholder for a variable value that you want to print in the middle of a string (say, a sentence that explains what you are printing), and to pass the variable name as argument to `.format()`, preceded by the string.

Let's try something out...

```
In [22]: print('The mean value for abv is {} and for ibu {}'.format(abv_mean,ibu_mean))
```

```
The mean value for abv is 0.059773424190800686 and for ibu 42.71316725978647
```

Ugh! That doesn't look very good, does it? Here's where Python's string formatting gets fancy. We can print fewer decimal digits, so the sentence is more readable. For example, if we want to have four decimal digits, we specify it this way:

```
In [23]: print('The mean value for abv is {:.4f} and for ibu {:.4f}'.format(abv_mean,
ibu_mean))
```

```
The mean value for abv is 0.0598 and for ibu 42.7132
```

Inside the curly brackets—the placeholders for the values we want to print—the `f` is for `float` and the `.4` is for four digits after the decimal dot. The colon here marks the beginning of the format specification (as there are options that can be passed before). There are so many tricks to Python's string formatter that you'll usually look up just what you need. Another useful resource for string formatting is the Python String Format Cookbook. Check it out!

3.3 Variance and standard deviation

While the mean indicates where's the center of your data, the **variance** and **standard deviation** describe the *spread* or variability of the data. We already mentioned that the *range* (difference between largest and smallest data values) is also an indication of variability. But the standard deviation is the most common measure of variability.

We really like the way Prof. Kristin Sainani, of Stanford University, presents this in her online course on Statistics in Medicine. In her lecture "Describing Quantitative Data: Whhat is the variability in the data?", available on YouTube, she asks: *What if someone were to ask you to devise a statistic that gives the avarage distance from the mean?* Think about this a little bit.

The distance from the mean, for any data value, is $x_i - \bar{x}$. So what is the average of the distances from the mean? If we try to simply compute the average of all the values $x_i - \bar{x}$, some of which are negative, you'll just get zero! It doesn't work.

Since the problem is the negative distances from the mean, you might suggest using absolute values. But this is just mathematically inconvenient. Another way to get rid of negative values is to take the squares. And that's how we get to the expression for the *variance*: it is the average of the squares of the deviations from the mean. For a set of N values,

$$\text{var} = \frac{1}{N}\sum_i (x_i - \bar{x})^2$$

The variance itself is hard to interpret. The problem with it is that the units are strange (they are the square of the original units). The **standard deviation**, the square root of the variance, is more meaningful because it has the same units as the original variable. Often, the symbol σ is used for it:

$$\sigma = \sqrt{\text{var}} = \sqrt{\frac{1}{N}\sum_i (x_i - \bar{x})^2}$$

3.4 Sample vs. population

The above definitions are used when N (the number of values) represents the entire population. But if we have a *sample* of that population, the formulas have to be adjusted: instead of dividing by N we divide by $N-1$. This is important, especially when we work with real data since usually we have samples of populations.

The **standard deviation** of a sample is denoted by s, and the formula is:

$$s = \sqrt{\frac{1}{N-1}\sum_i (x_i - \bar{x})^2}$$

Why? This gets a little technical, but the reason is that if you have a *sample* of the population, you don't know the *real* value of the mean, and $\bar{x}$ is actually an *estimate* of the mean. That's why you'll often find the symbol μ used to denote the population mean, and distinguish it with the sample mean, $\bar{x}$. Using $\bar{x}$ to compute the standard deviation introduces a small bias: $\bar{x}$ is computed *from the sample values*, and the data are on average (slightly) closer to $\bar{x}$ than the population is to μ. Dividing by $N-1$ instead of N corrects this bias!

Prof. Sainani explains it by saying that we lost one degree of freedom when we estimated the mean using $\bar{x}$. For example, say we have 100 people and I give you their mean age, and the actual age for 99 people from the sample: you'll be able to calculate the age of that 100th person. Once we calculated the mean, we only have 99 degrees of freedom left because that 100th person's age is fixed.

3.5 Let's code!

Now that we have the math sorted out, we can program functions to compute the variance and the standard deviation. In our case, we are working with samples of the population of craft beers, so we need to use the formulas with $N-1$ in the denominator.

```
In [24]: def sample_var(array):
             """ Calculates the variance of an array that contains values of a
              sample of a population.
```

```
    Arguments
    ---------
    array : array, contains sample of values.

    Returns
    -------
    var   : float, variance of the array .
    """

    sum_sqr = 0
    mean = numpy.mean(array)

    for element in array:
        sum_sqr += (element - mean)**2

    N = len(array)
    var = sum_sqr / (N - 1)

    return var
```

Notice that we used `numpy.mean()` in our function: do you think we can make this function even more Pythonic?

Hint: Yes!, we totally can.

Exercise: Re-write the function `sample_var()` using `numpy.sum()` to replace the `for`-loop. Name the function `var_pythonic`.

In [ ]:

We have the sample variance, so we take its square root to get the standard deviation. We can make it a function, even though it's just one line of Python, to make our code more readable:

```
In [25]: def sample_std(array):
    """ Computes the standard deviation of an array that contains values
    of a sample of a population.

    Arguments
    ---------
    array : array, contains sample of values.

    Returns
    -------
    std   : float, standard deviation of the array.
    """

    std = numpy.sqrt(sample_var(array))
```

```
            return std
```

Let's call our brand new functions and assign the output values to new variables:

```
In [26]: abv_std = sample_std(abv)
         ibu_std = sample_std(ibu)
```

If we print these values using the string formatter, only printing 4 decimal digits, we can display our descriptive statistics in a pleasant, human-readable way.

```
In [27]: print('The standard deviation for abv is {:.4f} and for ibu {:.4f}'.format(
abv_std, ibu_std))
```

```
The standard deviation for abv is 0.0135 and for ibu 25.9541
```

These numbers tell us that the `abv` values are quite concentrated around the mean value, while the `ibu` values are quite spread out from their mean. How could we check these descriptions of the data? A good way of doing so is using graphics: various types of plots can tell us things about the data.

We'll learn about *histograms* in this lesson, and in the following lesson we'll explore *box plots*.

4 Distribution plots

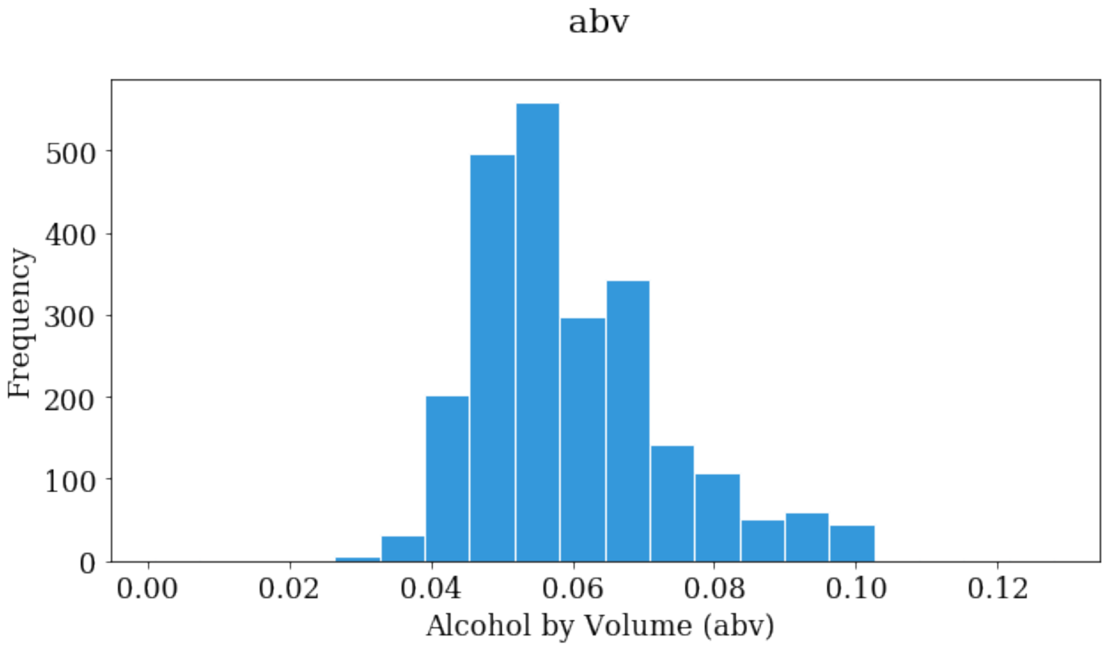
Every time that we work with data, visualizing it is very useful. Visualizations give us a better idea of how our data behaves. One way of visualizing data is with a frequency-distribution plot known as **histogram**: a graphical representation of how the data is distributed. To make a histogram, first we need to "bin" the range of values (divide the range into intervals) and then we count how many data values fall into each interval. The intervals are usually consecutive (not always), of equal size and non-overlapping.

Thanks to Python and Matplotlib, making histograms is easy. We recommend that you always read the documentation, in this case about histograms. We'll show you here an example using the `hist()` function from `pyplot`, but this is just a starting point.

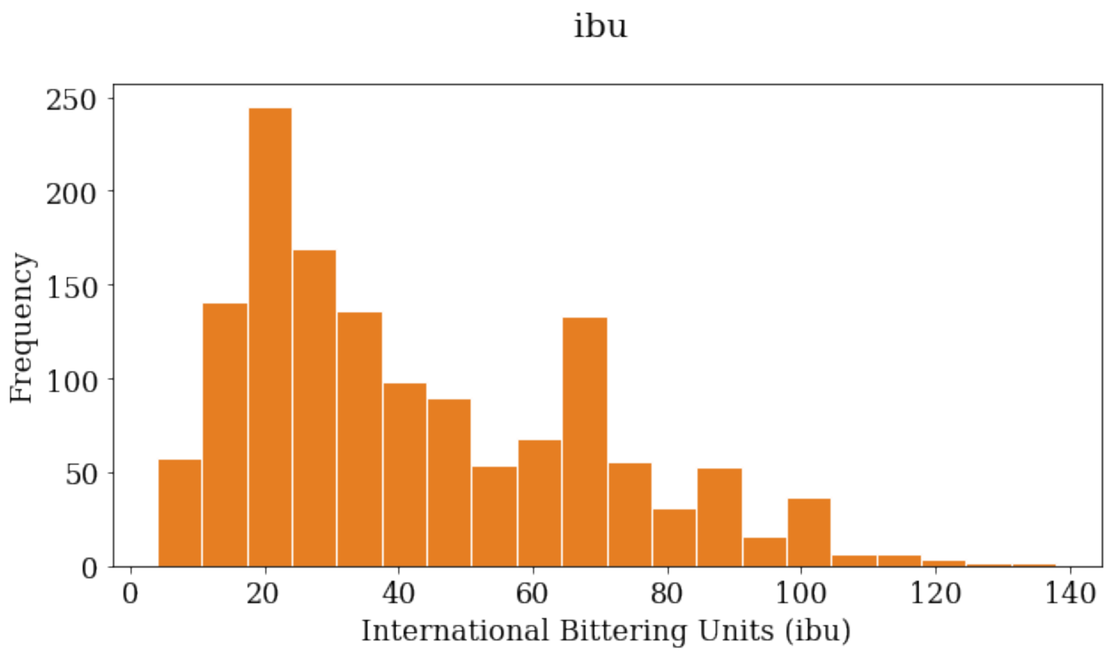
Let's import the libraries that we need for plotting, as you learned in Module 1 Lesson 5, then study the plotting commands used below. Try changing some of the plot options and seeing the effect.

```
In [28]: from matplotlib import pyplot
         %matplotlib inline

         #Import rcParams to set font styles
         from matplotlib import rcParams

         #Set font style and size
         rcParams['font.family'] = 'serif'
         rcParams['font.size'] = 16
```

In [29]:
```python
#You can set the size of the figure by doing:
pyplot.figure(figsize=(10,5))

#Plotting
pyplot.hist(abv, bins=20, color='#3498db', histtype='bar', edgecolor='white')
#The \n is to leave a blank line between the title and the plot
pyplot.title('abv \n')
pyplot.xlabel('Alcohol by Volume (abv) ')
pyplot.ylabel('Frequency');
```



In [30]:
```python
#You can set the size of the figure by doing:
pyplot.figure(figsize=(10,5))

#Plotting
pyplot.hist(ibu, bins=20, color='#e67e22', histtype='bar', edgecolor='white')
#The \n is to leave a blanck line between the title and the plot
pyplot.title('ibu \n')
pyplot.xlabel('International Bittering Units (ibu)')
pyplot.ylabel('Frequency');
```



Exploratory exercise: Play around with the plots, change the values of the bins, colors, etc.

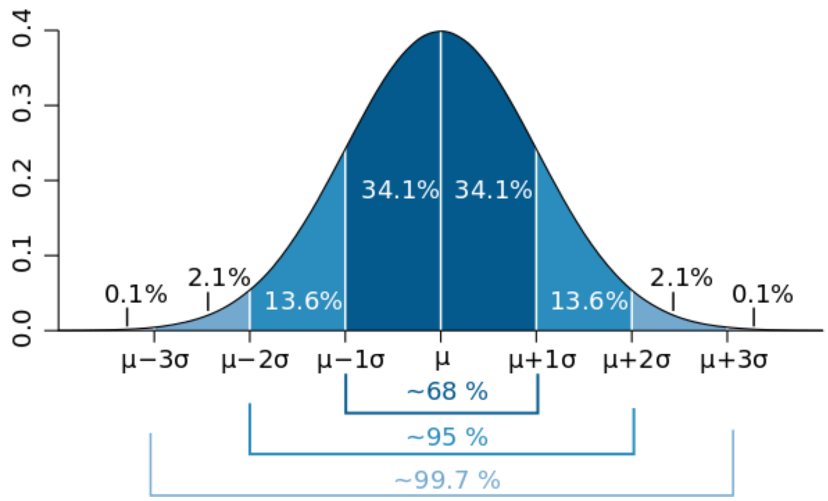
4.1 Comparing with a normal distribution

A **normal** (or Gaussian) distribution is a special type of distrubution that behaves as shown in the figure: 68% of the values are within one standard deviation σ from the mean; 95% lie within 2σ; and at a distance of $\pm 3\sigma$ from the mean, we cover 99.7% of the values. This fact is known as the 3-σ rule, or 68-95-99.7 (empirical) rule.

Notice that our histograms don't follow the shape of a normal distribution, known as *Bell Curve*. Our histograms are not centered in the mean value, and they are not symetric with respect to it. They are what we call **skewed** to the right (yes, to the *right*). A right (or positive) skewed distribution looks like it's been pushed to the left: the right tail is longer and most of the values are concentrated on the left of the figure. Imagine that "right-skewed" means that a force from the right pushes on the curve.

Discuss with your neighbor

- How do you think that skewness will affect the percentages of coverage by standard deviation compared to the Bell Curve?
- Can we calculate those percentages?



Standard deviation and coverage in a normal distribution. Modified figure based on original from Wikimedia Commons, the free media repository.

Spoiler alert! (and Exercise) Yes we can, and guess what: we can do it in a few lines of Python. But before doing that, we want you to explain in your own words how the following piece of code works.

Hints:

1. Check what the boolean operation `(1 < x) & (x < 4)` returns.
2. Check what happens if you sum booleans. For example, `True + True`, `True + False` and so on.

```
In [31]: x = numpy.array([1,2,3,4])
         num_ele = ((1 < x) & (x < 4)).sum()
         print(num_ele)
```

```
2
```

Now, using the same idea, we will calculate the number of elements in each interval of width $(1\sigma, 2\sigma, 3\sigma)$, and get the corresponding percentage.

Since we want to compute this for both of our variables, `abv` and `ibu`, we'll write a function to do so. Study carefully the code below. Better yet, explain it to your neighbor.

```
In [32]: def std_percentages(x, x_mean, x_std):
             """ Computes the percentage of coverage at 1std, 2std and 3std from the
             mean value of a certain variable x.

             Arguments
             ---------
             x      : array, data we want to compute on.
             x_mean : float, mean value of x array.
             x_std  : float, standard deviation of x array.
```

```
    Returns
    -------

    per_std_1 : float, percentage of values within 1 standard deviation.
    per_std_2 : float, percentage of values within 2 standard deviations.
    per_std_3 : float, percentage of values within 3 standard deviations.
    """

    std_1 = x_std
    std_2 = 2 * x_std
    std_3 = 3 * x_std

    elem_std_1 = (((x_mean - std_1) < x) & (x < (x_mean + std_1))).sum()
    per_std_1 = elem_std_1 * 100 / len(x)

    elem_std_2 = (((x_mean - std_2) < x) & (x < (x_mean + std_2))).sum()
    per_std_2 = elem_std_2 * 100 / len(x)

    elem_std_3 = (((x_mean - std_3) < x) & (x < (x_mean + std_3))).sum()
    per_std_3 = elem_std_3 * 100 / len(x)

    return per_std_1, per_std_2, per_std_3
```

Let's compute the percentages next. Notice that the function above returns three values. If we want to assign each value to a different variable, we need to follow a specific syntax. In our example this would be:

abv

```
In [33]: abv_std1_per, abv_std2_per, abv_std3_per = std_percentages(abv, abv_mean, abv_std)
```

Let's pretty-print the values of our variables so we can inspect them:

```
In [34]: print('The percentage of coverage at 1 std of the abv_mean is : {:.2f} %'.format(
         abv_std1_per))
         print('The percentage of coverage at 2 std of the abv_mean is : {:.2f} %'.format(
         abv_std2_per))
         print('The percentage of coverage at 3 std of the abv_mean is : {:.2f} %'.format(
         abv_std3_per))
```

```
The percentage of coverage at 1 std of the abv_mean is : 74.06 %
The percentage of coverage at 2 std of the abv_mean is : 94.34 %
The percentage of coverage at 3 std of the abv_mean is : 99.79 %
```

ibu

```
In [35]: ibu_std1_per, ibu_std2_per, ibu_std3_per = std_percentages(ibu, ibu_mean, ibu_std)
```

```
In [36]: print('The percentage of coverage at 1 std of the ibu_mean is : {:.2f} %'.format(
         ibu_std1_per))
```

```
print('The percentage of coverage at 2 std of the ibu_mean is : {:.2f} %'.format(
ibu_std2_per))
print('The percentage of coverage at 3 std of the ibu_mean is : {:.2f} %'.format(
ibu_std3_per))
```

```
The percentage of coverage at 1 std of the ibu_mean is : 68.11 %
The percentage of coverage at 2 std of the ibu_mean is : 95.66 %
The percentage of coverage at 3 std of the ibu_mean is : 99.72 %
```

Notice that in both cases the percentages are not that far from the values for normal distribution (68%, 95%, 99.7%), especially for 2σ and 3σ. So usually you can use these values as a rule of thumb.

5 What we've learned

- Read data from a csv file using pandas.
- The concepts of Data Frame and Series in pandas.
- Clean null (NaN) values from a Series using pandas.
- Convert a pandas Series into a numpy array.
- Compute maximum and minimum, and range.
- Revise concept of mean value.
- Compute the variance and standard deviation.
- Use the mean and standard deviation to understand how the data is distributed.
- Plot frequency distribution diagrams (histograms).
- Normal distribution and 3-sigma rule.

6 References

1. Craft beer datatset by Jean-Nicholas Hould.
2. Exploratory Data Analysis, Wikipedia article.
3. *Think Python: How to Think Like a Computer Scientist* (2012). Allen Downey. Green Tea Press. PDF available
4. Intro to data Structures, pandas documentation.
5. *Think Stats: Probability and Statistics for Programmers* version 1.6.0 (2011). Allen Downey. Green Tea Press. PDF available

Recommended viewing

From "Statistics in Medicine,", a free course in Stanford Online by Prof. Kristin Sainani, we highly recommend that you watch these three lectures:

- Describing Quantitative Data: Where is the center?
- Describing Quantitative Data: What is the variability in the data? * Variability in the data, continued: examples, bell curve

Lesson 2: Seeing stats in a new light

Welcome to the second lesson in "Take off with stats," Module 2 of our course in *Engineering Computations*. In the previous lesson, Cheers! Stats with Beers, we did some exploratory data analysis with a data set of canned craft beers in the US [1]. We'll continue using that same data set here, but with a new focus on *visualizing statistics*.

In her lecture "Looking at Data", Prof. Kristin Sainani says that you should always plot your data. Immediately, several things can come to light: are there outliers in your data? (Outliers are data points that look abnormally far from other values in the sample.) Are there data points that don't make sense? (Errors in data entry can be spotted this way.) And especially, you want to get a *visual* representation of how data are distributed in your sample.

In this lesson, we'll play around with different ways of visualizing data. There are so many ways to play! Have a look at the gallery of The Data Viz Project by *ferdio* (a data viz agency in Copenhagen). Aren't those gorgeous? Wouldn't you like to be able to make some pretty pics like that? Python can help!

Let's begin. We'll import our favorite Python libraries, and set some font parameters for our plots to look nicer. Then we'll load our data set for craft beers and begin!

In [1]:
```
import numpy
import pandas
from matplotlib import pyplot
%matplotlib inline

#Import rcParams to set font styles
from matplotlib import rcParams

#Set font style and size
rcParams['font.family'] = 'serif'
rcParams['font.size'] = 16
```

In [2]:
```
# Load the beers data set using pandas, and assign it to a dataframe
beers = pandas.read_csv("../../data/beers.csv")
```

Note: If you downloaded this notebook alone, and don't have the data file on the location we assume above, get it by adding a code cell, and execute the following code in it:

```
from urllib.request import urlretrieve
URL = 'http://go.gwu.edu/engcomp2data1?accessType=DOWNLOAD'
urlretrieve(URL, 'beers.csv')
```

The data file will be downloaded to your working directory, and you will then need to remove the path information, i.e., the string `'../../data/'`, in the code to read it.

OK. Let's have a look at the first few rows of the `pandas` dataframe we just created from the file, and confirm that it's a dataframe using the `type()` function. We only display the first 10 rows to save some space.

```
In [3]: type(beers)
```

```
Out[3]: pandas.core.frame.DataFrame
```

```
In [4]: beers[0:10]
```

```
Out[4]:    Unnamed: 0    abv  ibu    id                 name  \
        0           0  0.050  NaN  1436             Pub Beer
        1           1  0.066  NaN  2265          Devil's Cup
        2           2  0.071  NaN  2264  Rise of the Phoenix
        3           3  0.090  NaN  2263             Sinister
        4           4  0.075  NaN  2262        Sex and Candy
        5           5  0.077  NaN  2261         Black Exodus
        6           6  0.045  NaN  2260  Lake Street Express
        7           7  0.065  NaN  2259              Foreman
        8           8  0.055  NaN  2258                 Jade
        9           9  0.086  NaN  2131         Cone Crusher

                                    style  brewery_id  ounces
        0             American Pale Lager         408    12.0
        1         American Pale Ale (APA)         177    12.0
        2                    American IPA         177    12.0
        3  American Double / Imperial IPA         177    12.0
        4                    American IPA         177    12.0
        5                   Oatmeal Stout         177    12.0
        6         American Pale Ale (APA)         177    12.0
        7                 American Porter         177    12.0
        8         American Pale Ale (APA)         177    12.0
        9  American Double / Imperial IPA         177    12.0
```

1 Quantitative vs. categorical data

As you can see in the nice table that `pandas` printed for the dataframe, we have several features for each beer: the label `abv` corresponds to the acohol-by-volume fraction, label `ibu` refers to the international bitterness unit (IBU), then we have the `name` of the beer and the `style`, the brewery ID number, and the liquid volume of the beer can, in ounces.

Alcohol-by-volume is a numeric feature: a volume fraction, with possible values from 0 to 1 (sometimes also given as a percentage). In the first 10 rows of our dataframe, the `ibu` value is missing (all those `NaN`s), but we saw in the previous lesson that `ibu` is also a numeric feature, with values that go from a minimum of 4 to a maximum of 138 (in our data set). IBU is pretty mysterious: how do you measure the bitterness of beer? It turns out that bitterness is measured as parts per million of *isohumulone*, the acid found in hops [2]. Who knew?

For these numeric features, we learned that we can get an idea of the *central tendency* in the data using the **mean value**, and we get ideas of *spread* of the data with the **standard deviation** (and also with the range, but standard deviation is the most common).

Notice that the beer data also has a feature named `style`: it can be "American IPA" or "American Porter" or a bunch of other styles of beer. If we want to study the beers according to style, we'll have to come up with some new ideas, because you can't take the mean or standard deviation of this feature!

Quantitative data have meaning through a numeric feature, either on a continuous scale (like a fraction from 0 to 1), or a discrete count. **Categorical data**, in contrast, have meaning through a qualitative feature (like the style of beer). Data in this form can be collected in groups (categories), and then we can count the number of data items in that group. For example, we could ask how many beers (in our set) are of the style "American IPA," or ask how many beers we have in each style.

2 Visualizing quantitative data

In the previous lesson, we played around a bit with the `abv` and `ibu` columns of the dataframe `beers`. For each of these columns, we extracted it from the dataframe and saved it into a `pandas` series, then we used the `dropna()` method to get rid of null values. This "clean" data was our starting point for some exploratory data analysis, and for plotting the data distributions using **histograms**. Here, we will add a few more ingredients to our recipes for data exploration, and we'll learn about a new type of visualization: the **box plot**.

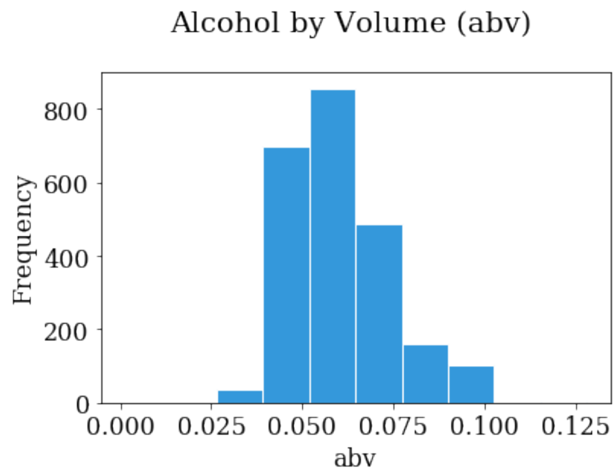
Let's repeat here the process for extracting and cleaning the two series, and getting the values into NumPy arrays:

In [5]:
```
#Repeat cleaning values abv
abv_series = beers['abv']
abv_clean = abv_series.dropna()
abv = abv_clean.values
```

In [6]:
```
#Repeat cleaning values ibu
ibu_series = beers['ibu']
ibu_clean = ibu_series.dropna()
ibu = ibu_clean.values
```

Let's also repeat a histogram plot for the `abv` variable, but this time choose to plot just 10 bins (you'll see why in a moment).

In [7]:
```
pyplot.figure(figsize=(6,4))
pyplot.hist(abv, bins=10, color='#3498db', histtype='bar', edgecolor='white')
pyplot.title('Alcohol by Volume (abv) \n')
pyplot.xlabel('abv')
pyplot.ylabel('Frequency');
```



You can tell that the most frequent values of `abv` fall in the bin just above 0.05 (5% alcohol), and the bin below. The mean value of our data is 0.06, which happens to be within the top-frequency bin, but data is not always so neat (sometimes, extreme values weigh heavily on the mean). Note also that we have a *right skewed* distribution, with higher-frequency bins occuring in the lower end of the range than in the higher end.

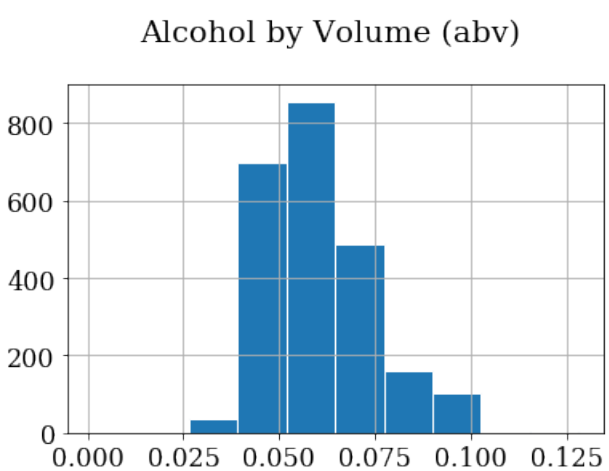
If you played around with the bin sizes in the previous lesson, you might have noticed that with a lot of bins, it becomes harder to visually pick out the patterns in the data. But if you use too few bins, the plot is also unhelpful. What number of bins is just right? Well, it depends on your data, so you'll just have to experiment and use your best judgement.

Let's learn a new trick. It turns out that `pandas` has built-in methods to make histograms directly from columns of a dataframe! (It uses Matplotlib internally for that.) The syntax is short and sweet:

`dataframe.hist(column='label')`

And `pandas` plots a pretty nice histogram without help. You can add optional parameters to set these to your liking; see the documentation. Check it out, and compare with our previous plot.

In [8]:
```
beers.hist(column='abv', edgecolor='white')
pyplot.title('Alcohol by Volume (abv) \n');
```



Which one do you like better? Well, the `pandas` histogram took fewer lines of code to create. And it doesn't look bad at all. But we do have more fine-grained control with Matplotlib. Which method you choose in a real situation will just depend on the situation and your preference.

2.1 Exploring quantitative data (continued)

In the previous lesson, you learned how to compute the mean of the data using `numpy.mean()`. How easy is that? But then we wrote our own custom functions to compute variance or standard deviation. It shouldn't surprise you by now that there are also NumPy functions for that!

Exercise:

- Go to the documentation of `numpy.var()` and analyze if this function is computing the *sample variance*. **Hint**: Check what it says about the "data degrees of freedom."

If you did the reading, you might have noticed that, by default, the argument `ddof` in `numpy.var()` is set to zero. If we use the default option, then we are not really calculating the sample variance. Recall from the previous lesson that the **sample variance** is:

$$\text{var}_{sample} = \frac{1}{N-1}\sum_i (x_i - \bar{x})^2$$

Therefore, we need to be explicit about the division by $N-1$ when calling `numpy.var()`. How do we do that? We explicitly set `ddof` to `1`.

For example, to compute the sample variance for our `abv` variable, we do:

```
In [9]: var_abv = numpy.var(abv, ddof=1)
        print(var_abv)
```

```
0.000183378552053
```

Now we can compute the standard deviation by taking the square root of `var_abv`:

```
In [10]: std_abv = numpy.sqrt(var_abv)
         print(std_abv)
```

```
0.0135417337167
```

You might be wondering if there is a built-in function for the standard deviation in NumPy. Go on and search online and try to find something.

Spoiler alert! You will.

Exercise:

1. Read the documentation about the NumPy standard deviation function, compute the standard deviation for `abv` using this function, and check that you obtained the same value than if you take the square root of the variance computed with NumPy.

2. Compute the variance and standard deviation for the variable `ibu`.

2.2 Median value

So far, we've learned to characterize quantitative data using the mean, variance and standard deviation.

If you watched Prof. Sainani's lecture Describing Quantitative Data: Where is the center? (recommended in our previous lesson), you'll recall that she also introduced the **median**: the middle value in the data, the value that separates your data set in half. (If there's an even number of data values, you take the average between the two middle values.)

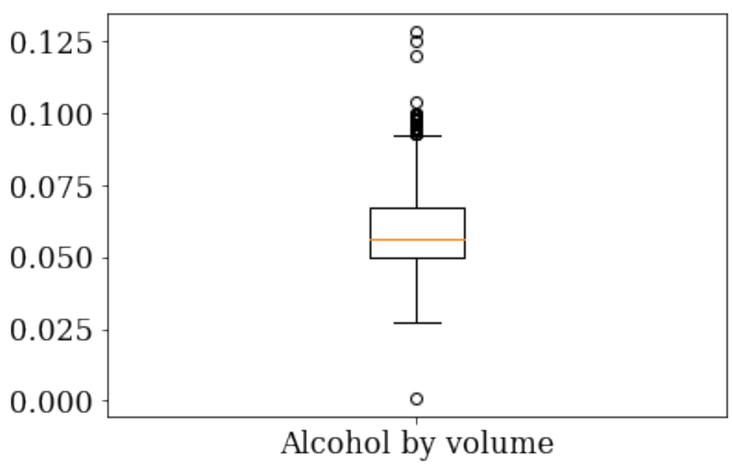
As you may anticipate, NumPy has a built-in function that computes the median, helpfully named `numpy.median()`.

Exercise: Using NumPy, compute the median for our variables `abv` and `ibu`. Compare the median with the mean, and look at the histogram to locate where the values fall on the x-axis.

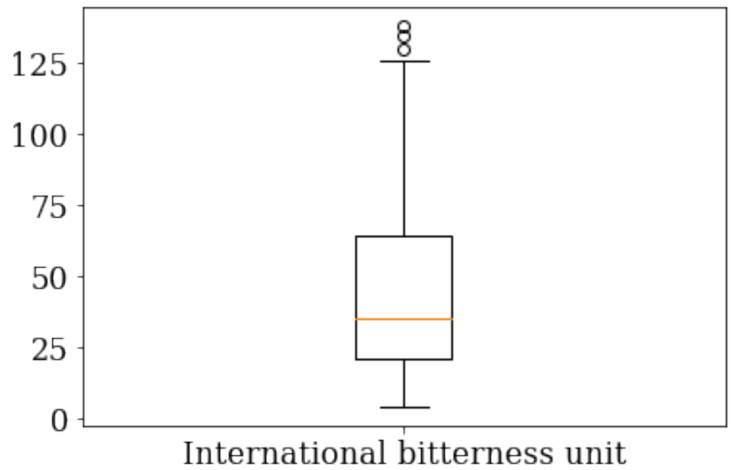
2.3 Box plots

Another handy way to visualize the distribution of quantitative data is using **box plots**. By "distribution" of the data, we mean some idea of the dataset's "shape": where is the center, what is the range, what is the variation in the data. Histograms are the most popular type of plots in exploratory data analysis. But check out box plots: they are easy to make with `pyplot`:

```
In [11]: pyplot.boxplot(abv, labels=['Alcohol by volume']);
```



```
In [12]: pyplot.boxplot(ibu, labels=['International bitterness unit']);
```



What is going on here? Obviously, there is a *box*: it represents 50% of the data in the middle of the data range, with the line across it (here, in orange) indicating the *median*.

The bottom of the box is at the 25th *percentile*, while the top of the box is at the 75th percentile. In other words, the bottom 25% of the data falls below the box, and the top 25% of the data falls above the box.

Confused by percentiles? The Nth percentile is the value below which N% of the observations fall.

Recall the bell curve from our previous lesson: we said that 95% of the data falls at a distance $\pm 2\sigma$ from the mean. This implies that 5% of the data (the rest) falls in the (symmetrical) tails, which in turn implies that the 2.5 percentile is at -2σ from the mean, and the 97.5 percentile is at $+2\sigma$ from the mean.

The percentiles 25, 50, and 75 are also named *quartiles*, since they divide the data into quarters. They are named first (Q1), second (Q2 or median) and third quartile (Q3), respectively.

Fortunately, NumPy has a function to compute percentiles and we can do it in just one line. Let's use `numpy.percentile()` to compute the `abv` and `ibu` quartiles.

abv quartiles

```
In [13]: Q1_abv = numpy.percentile(abv, q=25)
         Q2_abv = numpy.percentile(abv, q=50)
         Q3_abv = numpy.percentile(abv, q=75)

         print('The first quartile for abv is {}'.format(Q1_abv))
         print('The second quartile for abv is {}'.format(Q2_abv))
         print('The third quartile for abv is {}'.format(Q3_abv))
```

```
The first quartile for abv is 0.05
The second quartile for abv is 0.056
The third quartile for abv is 0.067
```

ibu quartiles

You can also pass a list of percentiles to `numpy.percentile()` and calculate several of them in one go. For example, to compute the quartiles of `ibu`, we do:

```
In [14]: quartiles_ibu = numpy.percentile(ibu, q=[25, 50, 75])

         print('The first quartile for ibu is {}'.format(quartiles_ibu[0]))
         print('The second quartile for ibu is {}'.format(quartiles_ibu[1]))
         print('The third quartile for ibu is {}'.format(quartiles_ibu[2]))
```

```
The first quartile for ibu is 21.0
The second quartile for ibu is 35.0
The third quartile for ibu is 64.0
```

OK, back to box plots. The height of the box—between the 25th and 75th percentile—is called the *interquartile range* (IQR). Outside the box, you have two vertical lines—the so-called "whiskers" of the box plot—which used to be called "box and whiskers plot" [3].

The whiskers extend to the upper and lower extremes (short horizontal lines). The extremes follow the following rules:

- Top whisker: lower value between the **maximum** and `Q3 + 1.5 x IQR`.
- Bottom whisker: higher value between the **minimum** and `Q1 - 1.5 x IQR`

Any data values beyond the upper and lower extremes are shown with a marker (here, small circles) and are an indication of outliers in the data.

Exercise: Calculate the end-points of the top and bottom whiskers for both the `abv` and `ibu` variables, and compare the results with the whisker end-points you see in the plot.

A bit of history: "Box-and-whiskers" plots were invented by John Tukey over 45 years ago. Tukey was a famous mathematician/statistician who is credited with coining the words *software* and *bit* [4]. He was active in the efforts to break the *Enigma* code durig WWII, and worked at Bell Labs in the first surface-to-air guided missile ("Nike"). A classic 1947 work on early design of the electonic computer acknowledged Tukey: he designed the electronic circuit for computing addition. Tukey was also a long-time advisor for the US Census Bureau, and a consultant for the Educational Testing Service (ETS), among many other contributions [5].

Note: Box plots are also drawn horizontally. Often, several box plots are drawn side-by-side with the purpose of comparing distributions.

3 Visualizing categorical data

The typical method of visualizing categorical data is using **bar plots**. These show visually the frequency of appearance of items in each category, or the proportion of data in each category.

Suppose we wanted to know how many beers of the same style are in our data set. Remember: the *style* of the beer is an example of *categorical data*. Let's extract the column with the style information from the beers dataframe, assign it to a variable named style_series, check the type of this variable, and view the first 10 elements.

```
In [15]: style_series = beers['style']

In [16]: type(style_series)

Out[16]: pandas.core.series.Series

In [17]: style_series[0:10]

Out[17]: 0               American Pale Lager
         1           American Pale Ale (APA)
         2                      American IPA
         3    American Double / Imperial IPA
         4                      American IPA
         5                     Oatmeal Stout
         6           American Pale Ale (APA)
         7                   American Porter
         8           American Pale Ale (APA)
         9    American Double / Imperial IPA
         Name: style, dtype: object
```

Already in the first 10 elements we see that we have two beers of the style "American IPA," two beers of the style "American Pale Ale (APA)," but only one beer of the style "Oatmeal Stout." The question is: how many beers of each style are contained in the whole series?

Luckily, pandas has a built-in function to answer that question: series.value_counts() (where series is the variable name of the pandas series you want the counts for). Let's try it on our style_series, and save the result in a new variable named style_counts.

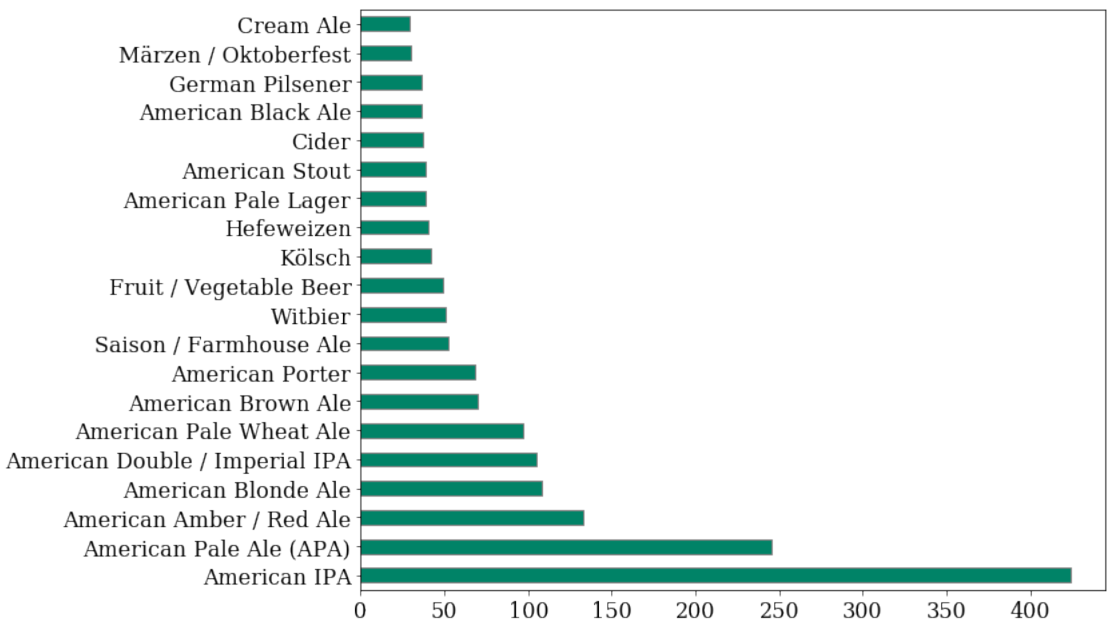
```
In [18]: style_counts = style_series.value_counts()
         style_counts[0:5]

Out[18]: American IPA                      424
         American Pale Ale (APA)           245
         American Amber / Red Ale          133
         American Blonde Ale               108
         American Double / Imperial IPA    105
         Name: style, dtype: int64

In [19]: type(style_counts)

Out[19]: pandas.core.series.Series

In [20]: len(style_counts)

Out[20]: 99
```

The len() function tells us that style_counts has 99 elements. That is, there are a total of 99 styles of beer in our data set. Wow, that's a lot!

Notice that `value_counts()` returned the counts sorted in decreasing order: the most popular beer in our data set is "American IPA" with 424 entries in our data. The next-most popular beer is "American Pale Ale (APA)" with a lot fewer entries (245), and the counts decrease sharply after that. Naturally, we'd like to know how much more popular are the top-2 beers from the rest. Bar plot to the rescue!

Below, we'll draw a horizontal bar plot directly with `pandas` (which uses Matplotlib internally) using the `plot.barh()` method for series. We'll only show the first 20 beers, because otherwise we'll get a huge plot. This plot gives us a clear visualization of the popularity ranking of beer styles in the US!

```
In [21]: style_counts[0:20].plot.barh(figsize=(10,8), color='#008367', edgecolor='gray');
```



4 Visualizing multiple data

These visualizations are really addictive! We're now getting ambitious: what if we wanted to show more than one feature, together on the same plot? What if we wanted to get insights about the relationship between two features through a multi-variable plot?

For example, don't you want to know if the bitterness of beers is associated with the alcohol-by-volume fraction? We do!

4.1 Scatter plots

Maybe we can do this: imagine a plot that has the alcohol-by-volume on the absissa, and the IBU value on the ordinate. For each beer, we can place a dot on this plot with its `abv` and `ibu` values as (x, y) coordinates. This is called a **scatter plot**.

We run into a bit of a problem, though. The way we handled the beer data above, we extracted the column for `abv` into a series, dropped the null entries, and saved the values into a NumPy array. We then repeated this process for the `ibu` column. Because a lot more `ibu` values are missing, we ended up with two arrays of different length: 2348 entries for the `abv` series, and 1405 entries for the `ibu` series. If we want to make a scatter plot with these two features, we'll need series (or arrays) of the same length.

Let's instead clean the whole `beers` dataframe (which will completely remove any row that has a null entry), and *then* extract the values of the two series into NumPy arrays.

```
In [22]: beers_clean = beers.dropna()
```

```
In [23]: ibu = beers_clean['ibu'].values
         len(ibu)
```

```
Out[23]: 1403
```

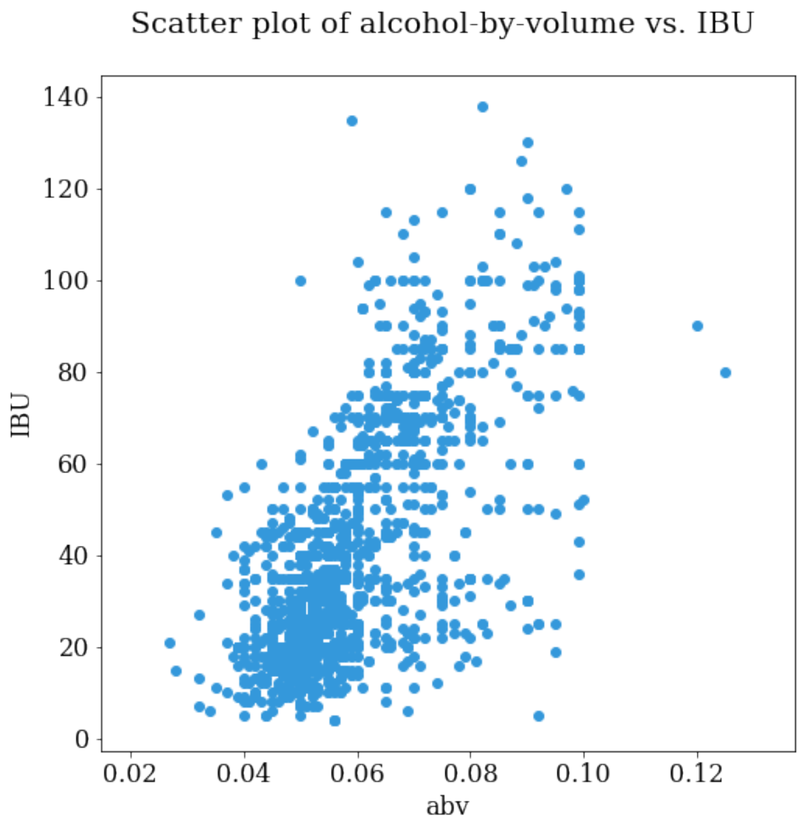
```
In [24]: abv = beers_clean['abv'].values
         len(abv)
```

```
Out[24]: 1403
```

Notice that both arrays now have 1403 entries—not 1405 (the length of the clean `ibu` data), because two rows that had a non-null `ibu` value *did* have a null `abv` value and were dropped.

With the two arrays of the same length, we can now call the `pyplot.scatter()` function.

```
In [25]: pyplot.figure(figsize=(8,8))
         pyplot.scatter(abv, ibu, color='#3498db')
         pyplot.title('Scatter plot of alcohol-by-volume vs. IBU \n')
         pyplot.xlabel('abv')
         pyplot.ylabel('IBU');
```



Hmm. That's a bit of a mess. Too many dots! But we do make out that the beers with low alcohol-by-volume tend to have low bitterness. For higher alcohol fraction, the beers can be anywhere on the bitterness scale: there's a lot of vertical spread on those dots to the right of the plot.

An idea! What if the bitterness has something to do with *style*? Neither of us knows much about beer, so we have no clue. Could we explore this question with visualization? We found a way!

4.2 Bubble chart

What we imagined is that we could group together the beers by style, and then make a new scatter plot where each marker corresponds to a style. The beers within a style, though, have many values of alcohol fraction and bitterness: we have to come up with a "summary value" for each style. Well, why not the *mean*... we can calculate the average `abv` and the average `ibu` for all the beers in each style, use that pair as (x, y) coordinate, and put a dot there representing the style.

Better yet! We'll make the size of the "dot" proportional to the popularity of the style in our data set! This is called a **bubble chart**.

How to achieve this idea? We searched online for "mean of a column with pandas" and we landed in `dataframe.mean()`. This could be helpful... But we don't want the mean of a *whole* column—we want the mean of the column values grouped by *style*. Searching online again, we landed in `dataframe.groupby()`. This is amazing: `pandas` can group a series for you!

Here's what we want to do: group beers by style, then compute the mean of `abv` and `ibu` in the groups. We experimented with `beers_clean.groupby('style').mean()` and were amazed...

However, one thing was bothersome: `pandas` computed the mean (by style) of every column, including the `id` and `brewery_id`, which have no business being averaged. So we decided to first drop the columns we don't need, leaving only `abv`, `ibu` and `style`. We can use the `dataframe.drop()` method for that. Check it out!

```
In [26]: beers_styles = beers_clean.drop(['Unnamed: 0','name','brewery_id','ounces',
        'id'], axis=1)
```

```
In [27]: beers_styles[0:10]
```

```
Out[27]:       abv   ibu                    style
         14  0.061  60.0  American Pale Ale (APA)
         21  0.099  92.0      American Barleywine
         22  0.079  45.0            Winter Warmer
         24  0.044  42.0  American Pale Ale (APA)
         25  0.049  17.0   Fruit / Vegetable Beer
         26  0.049  17.0   Fruit / Vegetable Beer
         27  0.049  17.0   Fruit / Vegetable Beer
         28  0.070  70.0             American IPA
         29  0.070  70.0             American IPA
         30  0.070  70.0             American IPA
```

We now have a dataframe with only the numeric features `abv` and `ibu`, and the categorical feature `style`. Let's find out how many beers we have of each style—we'd like to use this information to set the size of the style bubbles.

```
In [28]: style_counts = beers_styles['style'].value_counts()
```

```
In [29]: style_counts[0:10]
```

```
Out[29]: American IPA                      301
         American Pale Ale (APA)           153
         American Amber / Red Ale           77
         American Double / Imperial IPA     75
         American Blonde Ale                61
         American Pale Wheat Ale            61
         American Porter                    39
         American Brown Ale                 38
         Fruit / Vegetable Beer             30
         Hefeweizen                         27
         Name: style, dtype: int64
```

```
In [30]: type(style_counts)
```

```
Out[30]: pandas.core.series.Series
```

```
In [31]: len(style_counts)
```

```
Out[31]: 90
```

The number of beers in each style appears on each row of `style_counts`, sorted in decreasing order of count. We have 90 different styles, and the most popular style is the "American IPA," with 301 beers...

Discuss with your neighbor:

- What happened? We used to have 99 styles and 424 counts in the "American IPA" style. Why is it different now?

OK. We want to characterize each style of beer with the *mean values* of the numeric features, `abv` and `ibu`, within that style. Let's get those means.

```
In [32]: style_means = beers_styles.groupby('style').mean()
```

```
In [33]: style_means[0:10]
```

```
Out[33]:                                  abv        ibu
         style
         Abbey Single Ale           0.049000  22.000000
         Altbier                    0.054625  34.125000
         American Adjunct Lager     0.046545  11.000000
         American Amber / Red Ale   0.057195  36.298701
         American Amber / Red Lager 0.048063  23.250000
         American Barleywine        0.099000  96.000000
         American Black Ale         0.073150  68.900000
         American Blonde Ale        0.050148  20.983607
         American Brown Ale         0.057842  29.894737
         American Dark Wheat Ale    0.052200  27.600000
```

Looking good! We have the information we need: the average `abv` and `ibu` by style, and the counts by style. The only problem is that `style_counts` is sorted by decreasing count value, while `style_means` is sorted alphabetically by style. Ugh.

Notice that `style_means` is a dataframe that is now using the style string as a *label* for each row. Meanwhile, `style_counts` is a `pandas` series, and it also uses the style as label or index to each element.

More online searching and we find the `series.sort_index()` method. It will sort our style counts in alphabetical order of style, which is what we want.

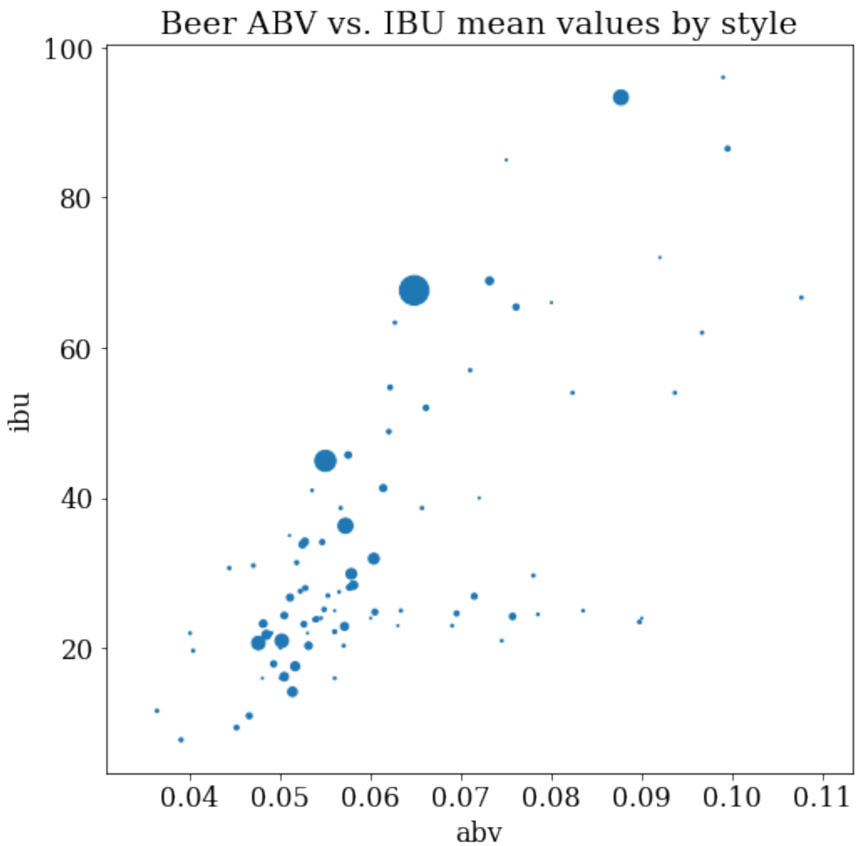
```
In [34]: style_counts = style_counts.sort_index()
```

```
In [35]: style_counts[0:10]
```

```
Out[35]: Abbey Single Ale               2
         Altbier                        8
         American Adjunct Lager        11
         American Amber / Red Ale      77
         American Amber / Red Lager    16
         American Barleywine            2
         American Black Ale            20
         American Blonde Ale           61
         American Brown Ale            38
         American Dark Wheat Ale        5
         Name: style, dtype: int64
```

Above, we used Matplotlib to create a scatter plot using two NumPy arrays as the `x` and `y` parameters. Like we saw previously with histograms, `pandas` also has available some plotting methods (calling Matplotlib internally). Scatter plots made easy!

```
In [36]: style_means.plot.scatter(figsize=(8,8),
                                  x='abv', y='ibu', s=style_counts,
                                  title='Beer ABV vs. IBU mean values by style');
```

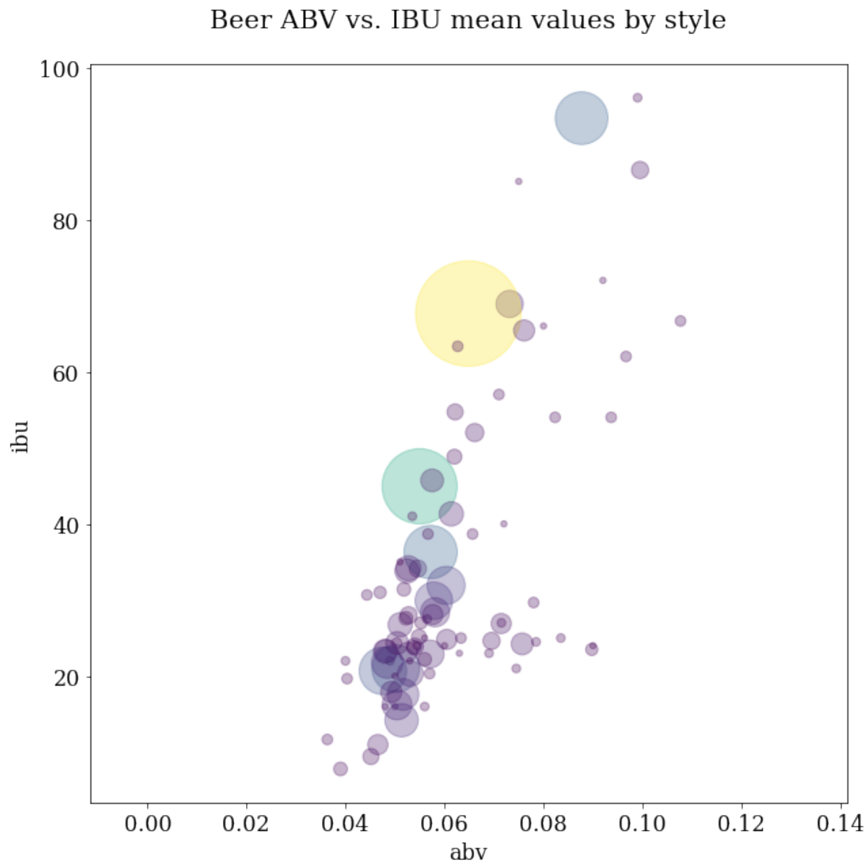


That's rad! Perhaps the bubbles are too small. We could multiply the `style_counts` by a factor of 5, or maybe 10? You should experiment.

But we are feeling gung-ho about this now, and decided to find a way to make the *color* of the bubbles also vary with the style counts. Below, we import the `colormap` module of Matplotlib, and we set our colors using the *viridis* colormap on the values of `style_counts`, then we repeat the plot with these colors on the bubbles and some transparency. *What do you think?*

```
In [37]: from matplotlib import cm
         colors = cm.viridis(style_counts.values)
```

```
In [38]: style_means.plot.scatter(figsize=(10,10),
                                  x='abv', y='ibu', s=style_counts*20, color=colors,
                                  title='Beer ABV vs. IBU mean values by style\n',
                                  alpha=0.3); #alpha sets the transparency
```

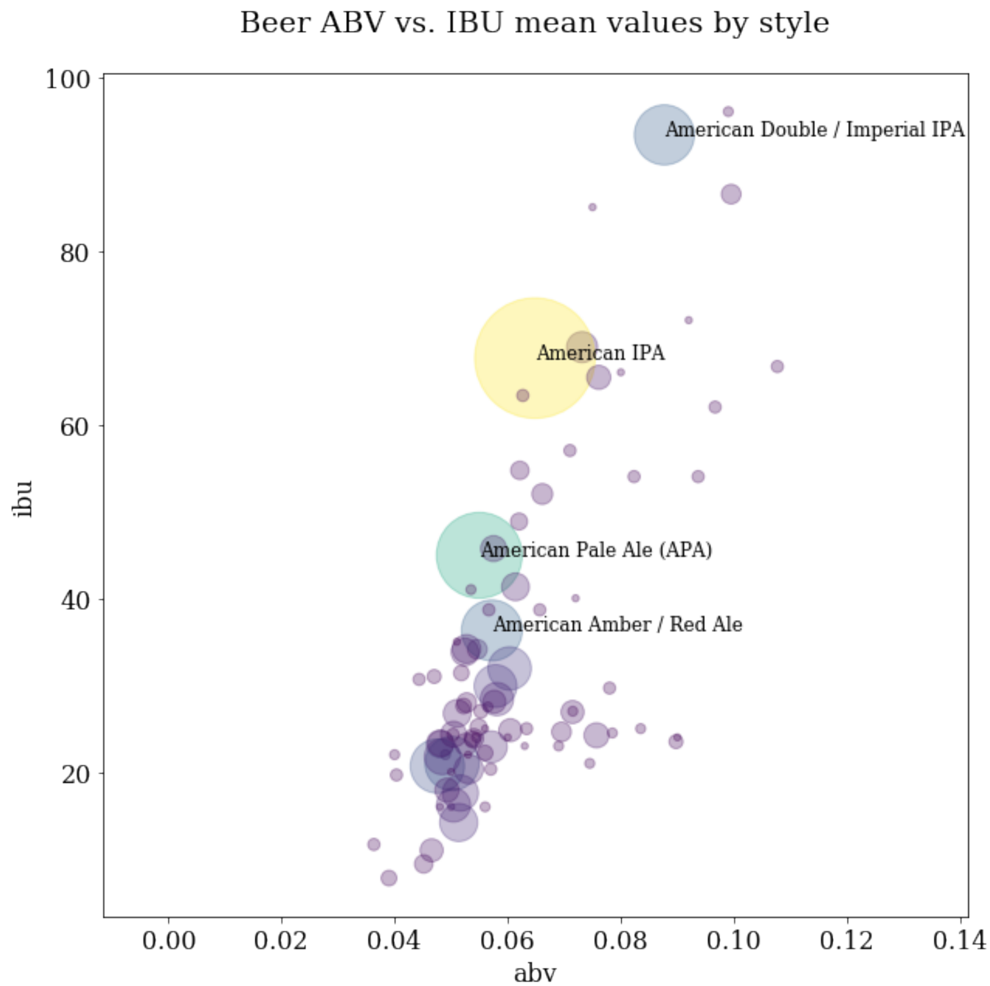


It looks like the most popular beers do follow a linear relationship between alcohol fraction and IBU. We learned a lot about beer without having a sip!

Wait... one more thing! What if we add a text label next to the bigger bubbles, to identify the style?

OK, here we go a bit overboard, but we couldn't help it. We played around a lot to get this version of the plot. It uses `enumerate` to get pairs of indices and values from a list of style names; an `if` statement to select only the large-count styles; and the `iloc[]` slicing method of `pandas` to get a slice based on index position, and extract `abv` and `ibu` values to an (x, y) coordinate for placing the annotation text. *Are we overkeen or what!*

```
In [39]: ax = style_means.plot.scatter(figsize=(10,10),
                                      x='abv', y='ibu', s=style_counts*20, color=colors,
                                      title='Beer ABV vs. IBU mean values by style\n',
                                      alpha=0.3);

         for i, txt in enumerate(list(style_counts.index.values)):
             if style_counts.values[i] > 65:
                 ax.annotate(txt, (style_means.abv.iloc[i],style_means.ibu.iloc[i]),
                 fontsize=12)
```



5 What we've learned

- You should always plot your data.
- The concepts of quantitative and categorical data.
- Plotting histograms directly on columns of dataframes, using `pandas`.
- Computing variance and standard deviation using NumPy built-in functions.
- The concept of median, and how to compute it with NumPy.
- Making box plots using `pyplot`.
- Five statistics of a box plot: the quartiles Q1, Q2 (median) and Q3 (and interquartile range Q3−Q1), upper and lower extremes.
- Visualizing categorical data with bar plots.
- Visualizing multiple data with scatter plots and bubble charts.
- `pandas` is awesome!

6 References

1. Craft beer datatset by Jean-Nicholas Hould.
2. What's The Meaning Of IBU? by Jim Dykstra for The Beer Connoisseur (2015).
3. 40 years of boxplots (2011). Hadley Wickham and Lisa Stryjewski, *Am. Statistician*. PDF available
4. John Wilder Tukey, Encyclopædia Britannica.
5. John W. Tukey: His life and professional contributions (2002). David R. Brillinger, *Ann. Statistics*. PDF available

Recommended viewing

From "Statistics in Medicine,", a free course in Stanford Online by Prof. Kristin Sainani, we highly recommend that you watch this lecture: * Looking at data

Lesson 3: Lead in Lipstick

After completing Lesson 1 and Lesson 2 of "Take off with stats," Module 2 of our course in *Engineering Computations*, here we'll work out a full example of what you can do with all that you've learned.

This example is based on the lecture by Prof. Kristin Sainani at Stanford, "Exploring real data: lead in lipstick," of her online course "Statistics in Medicine,". We followed along her narration, searched online for the sources she cited and the data from the FDA studies, and worked out the descriptive statistics using Python. We hope you'll enjoy it!

1 In the news

In 2007, some alarming reports appeared in the media: a US consumer-rights group had tested 33 brand-name lipsticks, and found that 61% had detectable lead levels of 0.03 to 0.65 parts per million (ppm). A full one-third of the lipsticks tested exceeded the lead level set by the US Food and Drug Administration (FDA) as the limit for candy: 0.1 ppm. Here are some media reports:

- Reuters published on Oct. 12, 2007: Lipsticks contain lead, consumer group says—it quotes a doctor as saying: "Lead builds up in the body over time and lead-containing lipstick applied several times a day, every day, can add up to significant exposure levels."
- CTV.ca News published FDA to examine claim of lead levels in lipstick—it quoted one member of the Campaign for Safe Cosmetics as saying: "We want the companies to immediately re-formulate their products to get the lead out and ultimately, really we need to change the laws and force these companies to be accountable to women's health."
- The New York Times was more measured in The Claim: Some Red Lipstick Brands Contain High Lead Levels (Nov. 13, 2007), concluding: "Studies have found that lead in lipstick is not a cause for concern, but research is continuing."

The FDA did carry out new studies in 2009 and 2012 to try to determine if lead content was a concern for lipstick users. These new studies generated some new scary headlines!

- On the Washington Post: 400 lipsticks found to contain lead, FDA says—the FDA is quoted as stating "We do not consider the lead levels we found in the lipsticks to be a safety concern…"
- In Time Magazine: What's in Your Lipstick? FDA Finds Lead in 400 Shades—a campaigner is quoted as saying: "We want to see the FDA recommend a limit based on the lowest level a company can achieve, like candy manufacturers are required."

Should lipstick users be concerned? Let's fact-check those scary headlines using our stats chops with Python!

2 The FDA studies

We located a web page of the US Food and Drug Administration, titled Limiting Lead in Lipstick and Other Cosmetics, that describes their efforts to assess the safety concerns from lead impurities in cosmetics. The web page includes data tables for the initial study in 2009, with 22 lipsticks, and the expanded study in 2012, with 400 lipsticks.

We copied these tables from the web page and created CSV files with the data. If you have a clone of all our lesson files, you already have the data. But if you downloaded this notebook on its own, you may need to get the data separately. See the Note below.

Let's begin by loading our Python libraries for data analysis: `numpy`, `pandas` and `pyplot`. We'll also load the `rcParams` module for setting Matplotlib's plotting parameters, and set the font family and size to serif 16 points.

In [1]:
```
import numpy
import pandas
from matplotlib import pyplot
%matplotlib inline

#Import rcParams to set font styles
from matplotlib import rcParams

#Set font style and size
rcParams['font.family'] = 'serif'
rcParams['font.size'] = 16
```

Note: We'll be reading the data from CSV files using `pandas`. If you don't have the data files locally, change the code in the cell below to read the data from the files hosted in our repository:

```
URL = 'http://go.gwu.edu/engcomp2data3a'
leadlips2009 = pandas.read_csv(URL)
```

In [2]:
```
# Load the FDA 2009 data set using pandas, and assign it to a dataframe
leadlips2009 = pandas.read_csv("../../data/FDA2009-lipstickdata.csv")
```

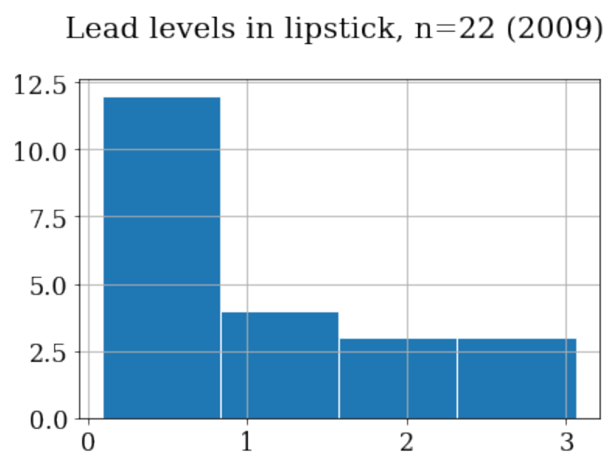
As always, we take a quick peek at the data, now saved in a `pandas` dataframe named `leadlips2009`, and then we get a view of its distribution by plotting a histogram of the column containing the lead content.

In [3]:
```
leadlips2009[0:5]
```

Out[3]:

	count	Sample	Brand	Parent company	Pb ppm
0	1	1a	Cover Girl	Procter & Gamble	3.06
1	2	1b	Cover Girl	Procter & Gamble	3.05
2	3	2	Revlon	Revlon	2.38
3	4	3	Cover Girl	Procter & Gamble	2.24
4	5	4	Body Shop	L'Oreal	1.79

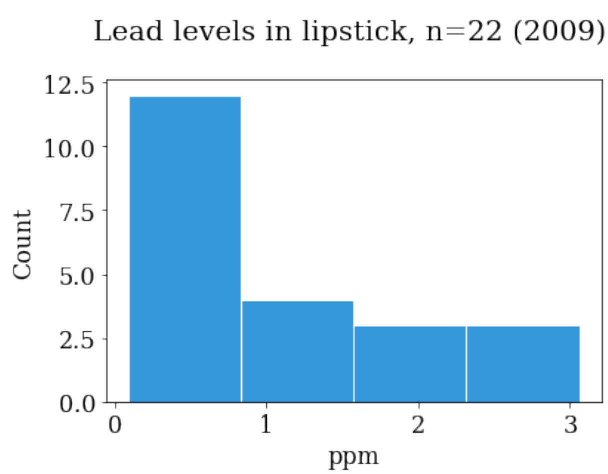
In [4]:
```
leadlips2009.hist(column='Pb ppm', bins=4, edgecolor='white')
pyplot.title('Lead levels in lipstick, n=22 (2009) \n');
```



Above, we used the built-in plotting capability of pandas. Just for kicks, let's get the same plot but using `pyplot` directly. To do that, remember that we need the data in a NumPy array, for which we use the `Series.values` method.

```
In [5]: lead2009 = leadlips2009['Pb ppm'].values
```

```
In [6]: pyplot.figure(figsize=(6,4))
        pyplot.hist(lead2009, bins=4, color='#3498db', histtype='bar', edgecolor='white')
        pyplot.title('Lead levels in lipstick, n=22 (2009) \n')
        pyplot.xlabel('ppm')
        pyplot.ylabel('Count');
```



Nothing new here: the histograms look the same, except for style. If you are following along with Sainani's lecture, however, you'll note some differences. We confirm that the data is the same by getting the descriptive statistics shown 4-min into the video:

```
In [7]: print('The mean value is {:.2f}'.format(leadlips2009['Pb ppm'].mean()))
        print('The median is {:.2f}'.format(leadlips2009['Pb ppm'].median()))
        print('The standard deviation is {:.2f}'.format(leadlips2009['Pb ppm'].std()))
        print('The maximum value is {:.2f}'.format(leadlips2009['Pb ppm'].max()))
```

```
The mean value is 1.07
The median is 0.73
```

```
The standard deviation is 0.96
The maximum value is 3.06
```

All of these match the statistics shown in the video. We do see some slight differences in the percentile values, however. Check them out:

```
In [8]: print('The 99 percentile is {:.2f}'.format(leadlips2009['Pb ppm'].quantile(.99)))
        print('The 95 percentile is {:.2f}'.format(leadlips2009['Pb ppm'].quantile(.95)))
        print('The 90 percentile is {:.2f}'.format(leadlips2009['Pb ppm'].quantile(.90)))
        print('The 75 percentile is {:.2f}'.format(leadlips2009['Pb ppm'].quantile(.75)))
```

```
The 99 percentile is 3.06
The 95 percentile is 3.02
The 90 percentile is 2.37
The 75 percentile is 1.69
```

Challenge question Despite the small difference in some percentile values from those shown on the video, we do think this is the same data that Sainani uses in her example. Look carefully at the histograms: can you explain the differences? (Play around with the plots here as much as you need to explain it.)

OK. Let's load the data for the extended study in 2012.

Note: If you don't have the data files locally, change the code in the cell below to read the data from the files hosted in our repository:

```
URL = 'http://go.gwu.edu/engcomp2data3b'
leadlips2012 = pandas.read_csv(URL)
```

```
In [9]: # Load the FDA 2012 data set using pandas, and assign it to a dataframe
        leadlips2012 = pandas.read_csv("../../data/FDA2012-lipstickdata.csv")
```

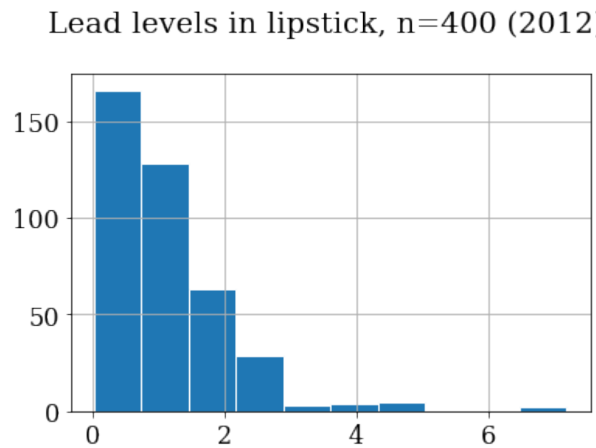
Take a quick peek at the first few rows of the dataframe we just created, and then make a histogram of the column containing the lead values (notice that it has a different label than the previous dataframe).

```
In [10]: leadlips2012[0:5]
```

Out[10]:

	Sample #	Brand	Parent company	Lead (ppm)
0	1	Maybelline	L'Oreal USA	7.19
1	2	L'Oreal	L'Oreal USA	7.00
2	3	NARS	Shiseido	4.93
3	4	Cover Girl Queen	Procter & Gamble	4.92
4	5	NARS	Shiseido	4.89

```
In [11]: leadlips2012.hist(column='Lead (ppm)', bins=10, edgecolor='white')
         pyplot.title('Lead levels in lipstick, n=400 (2012) \n');
```



Now, let's get the descriptive statistics for this data set, and confirm that they match with those shown in Dr. Sainani's video.

```
In [12]: print('The mean value is {:.2f}'.format(leadlips2012['Lead (ppm)'].mean()))
         print('The median is {:.2f}'.format(leadlips2012['Lead (ppm)'].median()))
         print('The standard deviation is {:.2f}'.format(leadlips2012['Lead (ppm)'].std()))
         print('The maximum value is {:.2f}'.format(leadlips2012['Lead (ppm)'].max()))
```

```
The mean value is 1.11
The median is 0.89
The standard deviation is 0.97
The maximum value is 7.19
```

The mean value, median, and standard deviation did not change much between the 2009 and 2012 studies, even though the earlier study only tested 22 samples. As Prof. Sainani points out, this goes to show that you can begin to describe a feature even with modest sample sizes.

The maximum value in the second study was a lot higher: 7.19 compared to 3.06. The reason for seeing this higher maximum value in the later study is that, for a *right skewed* distribution like this one, there are infrequent occurrences of a higher concentration of lead. These start to be detected with larger sample sizes.

Next, we compute a few percentiles (noticing slight differences with the values shown by Sainani).

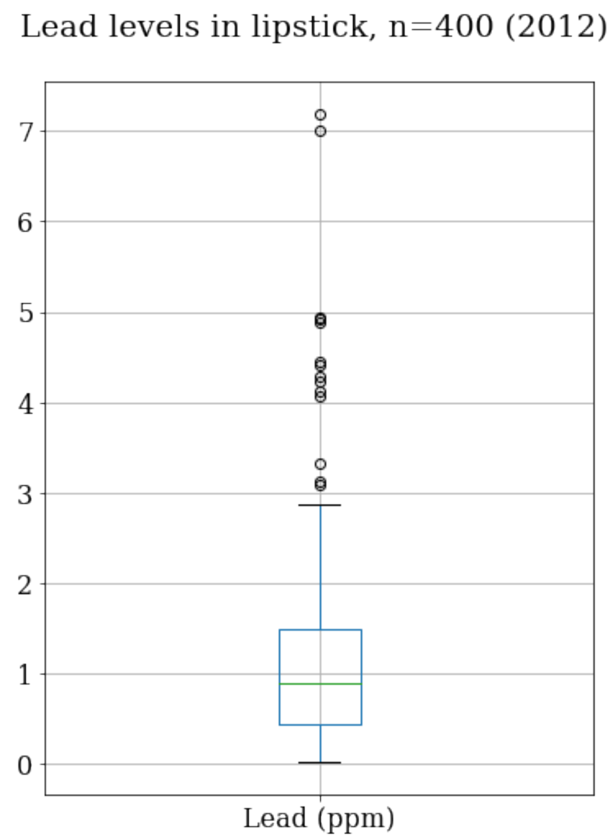
```
In [13]: print('The 99 percentile is {:.2f}'.format(leadlips2012['Lead (ppm)']
         .quantile(.99)))
         print('The 95 percentile is {:.2f}'.format(leadlips2012['Lead (ppm)']
         .quantile(.95)))
         print('The 90 percentile is {:.2f}'.format(leadlips2012['Lead (ppm)']
         .quantile(.90)))
         print('The 75 percentile is {:.2f}'.format(leadlips2012['Lead (ppm)']
         .quantile(.75)))
```

```
The 99 percentile is 4.89
The 95 percentile is 2.74
The 90 percentile is 2.22
```

```
The 75 percentile is 1.49
```

In the previous lesson, you learned to make box plots using `pyplot`, which requires extracting the values of the data series of interest into NumPy arrays. It turns out, `pandas` can make box plots directly with a column of the dataframe.

```
In [14]: leadlips2012.boxplot(column='Lead (ppm)', figsize=(6,8))
         pyplot.title('Lead levels in lipstick, n=400 (2012) \n');
```



The box plot also indicates a right skewed distribution, and shows a number of outliers on the high end of the range: some lipsticks have an especially high level of lead.

3 Lead exposure from lipstick

A European study of exposure to various cosmetic products [Ref. 2] offers some useful statistics about lipstick use. In figure 6, the paper shows a histogram of lipstick applied by the participants in the study. The distribution is right skewed: most users apply a moderate amount of lipstick daily, but there are a few heavy users in the tail of the distribution. The number of participants was 30,000, and the summary statistics are:

- mean value = 24.61 mg/day,
- median = 17.11 mg/day,
- minimum = 0.13 mg/day,

- maximum = 217.53 mg/day
- 95th percentile = 72.51 mg/day

Prof. Sainani suggests the following exercise: suppose that users ingest half of the lipstick they apply daily—seems like a conservative estimate, given that some lipstick will end up on cups, napkins, and (as Sainani amusingly points out) other people. We'd like to calculate:

1. the typical lead exposure from lipstick, using the medians
2. the highest daily lead exposure from lipstick, using the maxima

From the 2012 FDA study of lead in lipstick: the median is 0.89 ppm (μg/g) and the maximum is 7.19 ppm. From the European study on exposure to cosmetics, the median daily usage of lipstick is 17.11 mg, and the maximum is 217.53. Now… keep your units straight!

$1\mu\text{g} = 10^{-3}\text{mg} = 10^{-6}\text{g}$

```
In [15]: # Typical user: 0.89 µg/g * 17.11 mg/day (divide by 1000 to get µg)
         print('The typical daily exposure to lead from lipstick is {:.4f} µg/day.'
         .format(0.89 *17.11/1000))
         print('Half of this amount is ingested: {:.4f} µg/day.'
         .format(0.89 *17.11/1000/2))
```

```
The typical daily exposure to lead from lipstick is 0.0152 µg/day.
Half of this amount is ingested: 0.0076 µg/day.
```

```
In [16]: # Maximum usage: 7.19 µg/g * 217.53 mg/day / 1000 to get µg
         print('The maximum daily exposure to lead from lipstick is {:.2f} µg/day.'
         .format(7.19 *217.53/1000))
         print('Half of this amount is ingested: {:.2f} µg/day.'
         .format(7.19 *217.53/1000/2))
```

```
The maximum daily exposure to lead from lipstick is 1.56 µg/day.
Half of this amount is ingested: 0.78 µg/day.
```

The maximum daily exposure is 100 times larger than the typical exposure, based on the median. Note that this maximum occurs for one user over 30,000 (the size of the study sample), and one lipstick over 400—so it's a chance of one in 12 million!

4 Is this bad?

The US Food and Drug Administration provides a recommended *maximum* lead level of 0.1 ppm in candy to be consumed by small children [3]. But most food products are well below the maximum. For example, the FDA data on 40 samples of milk chocolate in the years 1991–2002 showed a mean lead level of 0.025 ppm [4]. That's of course much lower than the concentration of lead in lipstick, but the *consumption* of chocolate is much higher! Forbes reported that the average American eats about 9.5 lbs (4.3 kg) of chocolate each year [6].

```
In [17]: print('The average American consumes {:.1f} grams of chocolate per day.'
         .format(4.3*1000/365))
```

```
print('This amounts to {:.2f} μg of lead exposure from chocolate (mean of FDA data).'.format(4.3*1000/365*0.025))
```

```
The average American consumes 11.8 grams of chocolate per day.
This amounts to 0.29 μg of lead exposure from chocolate (mean of FDA data).
```

Compared to the median exposure to lead from lipstick of 0.0076 μg per day, the exposure from chocolate is almost 40 times higher!

Clearly the consumer group that generated all those headlines was scaremongering. And now you have the tools to fact-check many of those scary health-related "fake news."

5 References

1. Limiting Lead in Lipstick and Other Cosmetics, US Food and Drug Administration.
2. European consumer exposure to cosmetic products, a framework for conducting population exposure assessments (2007). Hall, B., et al., *Food and Chemical Toxicology* **45**(11): 2097-2108. Available on PubMed.
3. US FDA Guidance for Industry: Lead in Candy Likely To Be Consumed Frequently by Small Children: Recommended Maximum Level and Enforcement Policy (2005, revised 2006).
4. US FDA Supporting Document for Recommended Maximum Level for Lead in Candy Likely To Be Consumed Frequently by Small Children (2006).
5. The World's Biggest Chocolate Consumers, Forbes, July 22nd 2015.

5.1 Recommended viewing

This lesson was based on the followign lecture from "Statistics in Medicine,", a free course in Stanford Online by Prof. Kristin Sainani: Exploring real data: lead in lipstick

Lesson 4: Life expectancy and wealth

Welcome to **Lesson 4** of the second module in *Engineering Computations*. This module gives you hands-on data analysis experience with Python, using real-life applications. The first three lessons provide a foundation in data analysis using a computational approach. They are:

1. Lesson 1: Cheers! Stats with beers.
2. Lesson 2: Seeing stats in a new light.
3. Lesson 3: Lead in lipstick.

You learned to do exploratory data analysis with data in the form of arrays: NumPy has built-in functions for many descriptive statistics, making it easy! And you also learned to make data visualizations that are both good-looking and effective in communicating and getting insights from data.

But NumPy can't do everything. So we introduced you to `pandas`, a Python library written *especially* for data analysis. It offers a very powerful new data type: the *DataFrame*—you can think of it as a spreadsheet, conveniently stored in one Python variable.

In this lesson, you'll dive deeper into `pandas`, using data for life expectancy and per-capita income over time, across the world.

1 The best stats you've ever seen

Hans Rosling was a professor of international health in Sweeden, until his death in Februrary of this year. He came to fame with the thrilling TED Talk he gave in 2006: "The best stats you've ever seen" (also on YouTube, with ads). We highly recommend that you watch it!

In that first TED Talk, and in many other talks and even a BBC documentary (see the trailer on YouTube), Rosling uses data visualizations to tell stories about the world's health, wealth, inequality and development. Using software, he and his team created amazing animated graphics with data from the United Nations and World Bank.

According to a blog post by Bill and Melinda Gates after Prof. Rosling's death, his message was simple: *"that the world is making progress, and that policy decisions should be grounded in data."*

In this lesson, we'll use data about life expectancy and per-capita income (in terms of the gross domestic product, GDP) around the world. Visualizing and analyzing the data will be our gateway to learning more about the world we live in.

Let's begin! As always, we start by importing the Python libraries for data analysis (and setting some plot parameters).

```
In [1]: import numpy
        import pandas
        from matplotlib import pyplot
        %matplotlib inline

        #Import rcParams to set font styles
        from matplotlib import rcParams

        #Set font style and size
        rcParams['font.family'] = 'serif'
        rcParams['font.size'] = 16
```

2 Load and inspect the data

We found a website called The Python Graph Gallery, which has a lot of data visualization examples. Among them is a Gapminder Animation, an animated GIF of bubble charts in the style of Hans Rosling. We're not going to repeat the same example, but we do get some ideas from it and re-use their data set. The data file is hosted on their website, and we can read it directly from there into a `pandas` dataframe, using the URL.

```
In [2]: # Read a dataset for life expectancy from a CSV file hosted online
        url = 'https://python-graph-gallery.com/wp-content/uploads/gapminderData.csv'
        life_expect = pandas.read_csv(url)
```

The first thing to do always is to take a peek at the data. Using the `shape` attribute of the dataframe, we find out how many rows and columns it has. In this case, it's kind of big to print it all out, so to save space we'll print a small portion of `life_expect`. You can use a slice to do this, or you can use the `DataFrame.head()` method, which returns by default the first 5 rows.

```
In [3]: life_expect.shape

Out[3]: (1704, 6)

In [4]: life_expect.head()
```

Out[4]:

	country	year	pop	continent	lifeExp	gdpPercap
0	Afghanistan	1952	8425333.0	Asia	28.801	779.445314
1	Afghanistan	1957	9240934.0	Asia	30.332	820.853030
2	Afghanistan	1962	10267083.0	Asia	31.997	853.100710
3	Afghanistan	1967	11537966.0	Asia	34.020	836.197138
4	Afghanistan	1972	13079460.0	Asia	36.088	739.981106

You can see that the columns hold six types of data: the country, the year, the population, the continent, the life expectancy, and the per-capita gross domestic product (GDP). Rows are indexed from 0, and the columns each have a **label** (also called an index). Using labels to access data is one of the most powerful features of `pandas`.

In the first five rows, we see that the country repeats (Afghanistan), while the year jumps by five. We guess that the data is arranged in blocks of rows for each country.

We can get a useful summary of the dataframe with the `DataFrame.info()` method: it tells us the number of rows and the number of columns (matching the output of the `shape` attribute) and then for each column, it tells us the number of rows that are populated (have non-null entries) and the type of the entries; finally it gives a breakdown of the types of data and an estimate of the memory used by the dataframe.

```
In [5]: life_expect.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
country      1704 non-null object
year         1704 non-null int64
pop          1704 non-null float64
continent    1704 non-null object
lifeExp      1704 non-null float64
gdpPercap    1704 non-null float64
dtypes: float64(3), int64(1), object(2)
memory usage: 80.0+ KB
```

The dataframe has 1704 rows, and every column has 1704 non-null entries, so there is no missing data. Let's find out how many entries of the same year appear in the data. In Lesson 1 of this module, you already learned to extract a column from a data frame, and use the `series.value_counts()` method to answer our question.

```
In [6]: life_expect['year'].value_counts()
```

```
Out[6]: 2007    142
        2002    142
        1997    142
        1992    142
        1987    142
        1982    142
        1977    142
        1972    142
        1967    142
        1962    142
        1957    142
        1952    142
        Name: year, dtype: int64
```

We have an even 142 occurrences of each year in the dataframe. The distinct entries must correspond to each country. It also is clear that we have data every five years, starting 1952 and ending 2007. We think we have a pretty clear picture of what is contained in this data set. What next?

3 Grouping data for analysis

We have a dataframe with a `country` column, where countries repeat in blocks of rows, and a `year` column, where sets of 12 years (increasing by 5) repeat for every country. Tabled data commonly has this interleaved structure. And data analysis often involves grouping the data in various ways, to transform it, compute statistics, and visualize it.

With the life expectancy data, it's natural to want to analyze it by year (and look at geographical differences), and by country (and look at historical differences).

In Lesson 2 of this module, we already learned how useful it was to group the beer data by style, and calculate means within each style. Let's get better acquainted with the powerful `groupby()` method for dataframes. First, grouping by the values in the `year` column:

```
In [7]: by_year = life_expect.groupby('year')
```

```
In [8]: type(by_year)
```

```
Out[8]: pandas.core.groupby.DataFrameGroupBy
```

Notice that the type of the new variable `by_year` is different: it's a *GroupBy* object, which—without making a copy of the data—is able to apply operations on each of the groups.

The `GroupBy.first()` method, for example, returns the first row in each group—applied to our grouping `by_year`, it shows the list of years (as a label), with the first country that appears in each year-group.

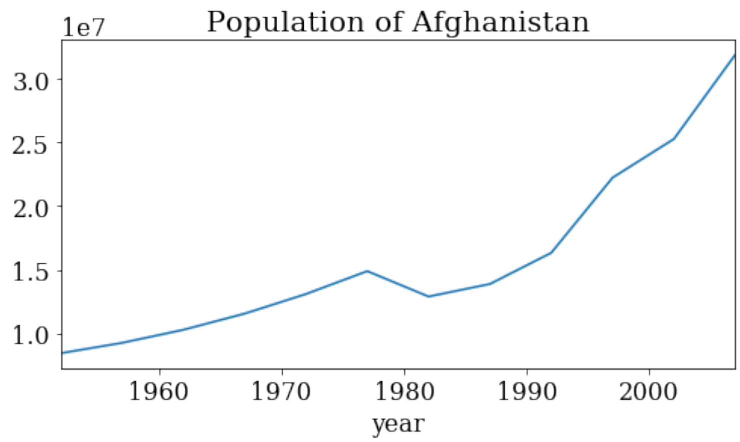
```
In [9]: by_year.first()
```

```
Out[9]:          country         pop continent  lifeExp   gdpPercap
        year
        1952  Afghanistan   8425333.0      Asia   28.801  779.445314
        1957  Afghanistan   9240934.0      Asia   30.332  820.853030
        1962  Afghanistan  10267083.0      Asia   31.997  853.100710
        1967  Afghanistan  11537966.0      Asia   34.020  836.197138
        1972  Afghanistan  13079460.0      Asia   36.088  739.981106
        1977  Afghanistan  14880372.0      Asia   38.438  786.113360
        1982  Afghanistan  12881816.0      Asia   39.854  978.011439
        1987  Afghanistan  13867957.0      Asia   40.822  852.395945
        1992  Afghanistan  16317921.0      Asia   41.674  649.341395
        1997  Afghanistan  22227415.0      Asia   41.763  635.341351
        2002  Afghanistan  25268405.0      Asia   42.129  726.734055
        2007  Afghanistan  31889923.0      Asia   43.828  974.580338
```

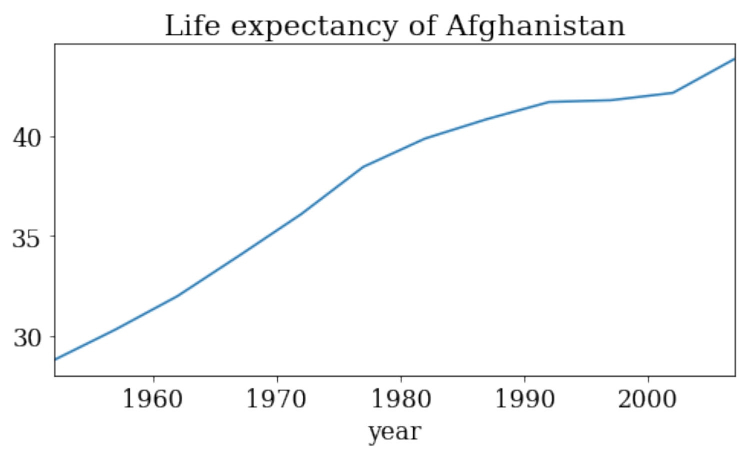
All the year-groups have the same first country, Afghanistan, so what we see is the population, life expectancy and per-capita income in Afghanistan for all the available years. Let's save that into a new dataframe, and make a line plot of the population and life expectancy over the years.

```
In [10]: Afghanistan = by_year.first()
```

```
In [11]: Afghanistan['pop'].plot(figsize=(8,4),
                                 title='Population of Afghanistan');
```




```
In [12]: Afghanistan['lifeExp'].plot(figsize=(8,4),
                                     title='Life expectancy of Afghanistan');
```



Do you notice something interesting? It's curious to see that the population of Afghanistan took a fall after 1977. We have data every 5 years, so we don't know exactly when this fall began, but it's not hard to find the answer online. The USSR invaded Afghanistan in 1979, starting a conflict that lasted 9 years and resulted in an estimated death toll of one million civilians and 100,000 fighters [1]. Millions fled the war to neighboring countries, which may explain why we se a dip in population, but not a dip in life expectancy.

We can also get some descriptive statistics in one go with the `DataFrame.describe()` method of pandas.

```
In [13]: Afghanistan.describe()
```

```
Out[13]:                pop    lifeExp   gdpPercap
         count  1.200000e+01  12.000000   12.000000
         mean   1.582372e+07  37.478833  802.674598
         std    7.114583e+06   5.098646  108.202929
         min    8.425333e+06  28.801000  635.341351
         25%    1.122025e+07  33.514250  736.669343
         50%    1.347371e+07  39.146000  803.483195
         75%    1.779529e+07  41.696250  852.572136
```

```
max    3.188992e+07  43.828000  978.011439
```

Let's now group our data by country, and use the `GroupBy.first()` method again to get the first row of each group-by-country. We know that the first year for which we have data is 1952, so let's immediately save that into a new variable named `year1952`, and keep playing with it. Below, we double-check the type of `year1952`, print the first five rows using the `head()` method, and get the minimum value of the population column.

```
In [14]: by_country = life_expect.groupby('country')
```

The first year for all groups-by-country is 1952. Let's save that first group into a new dataframe, and keep playing with it.

```
In [15]: year1952 = by_country.first()

In [16]: type(year1952)

Out[16]: pandas.core.frame.DataFrame

In [17]: year1952.head()

Out[17]:             year         pop  continent  lifeExp    gdpPercap
         country
         Afghanistan  1952   8425333.0       Asia   28.801   779.445314
         Albania      1952   1282697.0     Europe   55.230  1601.056136
         Algeria      1952   9279525.0     Africa   43.077  2449.008185
         Angola       1952   4232095.0     Africa   30.015  3520.610273
         Argentina    1952  17876956.0   Americas   62.485  5911.315053

In [18]: year1952['pop'].min()

Out[18]: 60011.0
```

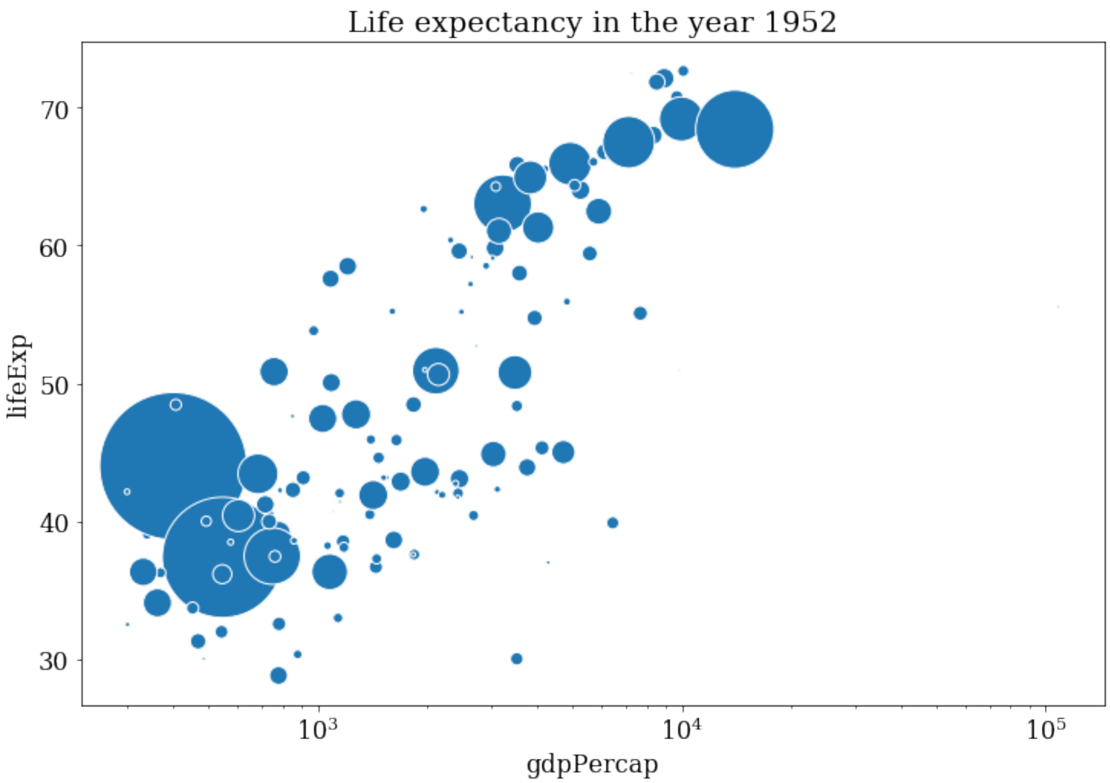
4 Visualizing the data

In Lesson 2 of this module, you learned to make bubble charts, allowing you to show at least three features of the data in one plot. We'd like to make a bubble chart of life expectancy vs. per-capita GDP, with the size of the bubble proportional to the population. To do that, we'll need to extract the population values into a NumPy array.

```
In [19]: populations = year1952['pop'].values
```

If you use the `populations` array unmodified as the size of the bubbles, they come out *huge* and you get one solid color covering the figure (we tried it!). To make the bubble sizes reasonable, we divide by 60,000—an approximation to the minimum population—so the smallest bubble size is about 1 pt. Finally, we choose a logarithmic scale in the absissa (the GDP). Check it out!

```
In [20]: year1952.plot.scatter(figsize=(12,8),
                               x='gdpPercap', y='lifeExp', s=populations/60000,
                               title='Life expectancy in the year 1952',
                               edgecolors="white")
         pyplot.xscale('log');
```



That's neat! But the Rosling bubble charts include one more feature in the data: the continent of each country, using a color scheme. Can we do that?

Matplotlib colormaps offer several options for *qualitative* data, using discrete colors mapped to a sequence of numbers. We'd like to use the `Accent` colormap to code countries by continent. But we need a numeric code to assign to each continent, so it can be mapped to a color.

The Gapminder Animation example at The Python Graph Gallery has a good tip: using the `pandas` *Categorical* data type, which associates a numerical value for each category in a column containing qualitative (categorical) data.

Let's see what we get if we apply `pandas.Categorical()` to the `continent` column:

```
In [21]: pandas.Categorical(year1952['continent'])
```

```
Out[21]: [Asia, Europe, Africa, Africa, Americas, ..., Asia, Asia, Asia, Africa, Africa]
         Length: 142
         Categories (5, object): [Africa, Americas, Asia, Europe, Oceania]
```

Right. We see that the `continent` column has repeated entries of 5 distinct categories, one for each continent. In order, they are: Africa, Americas, Asia, Europe, Oceania.

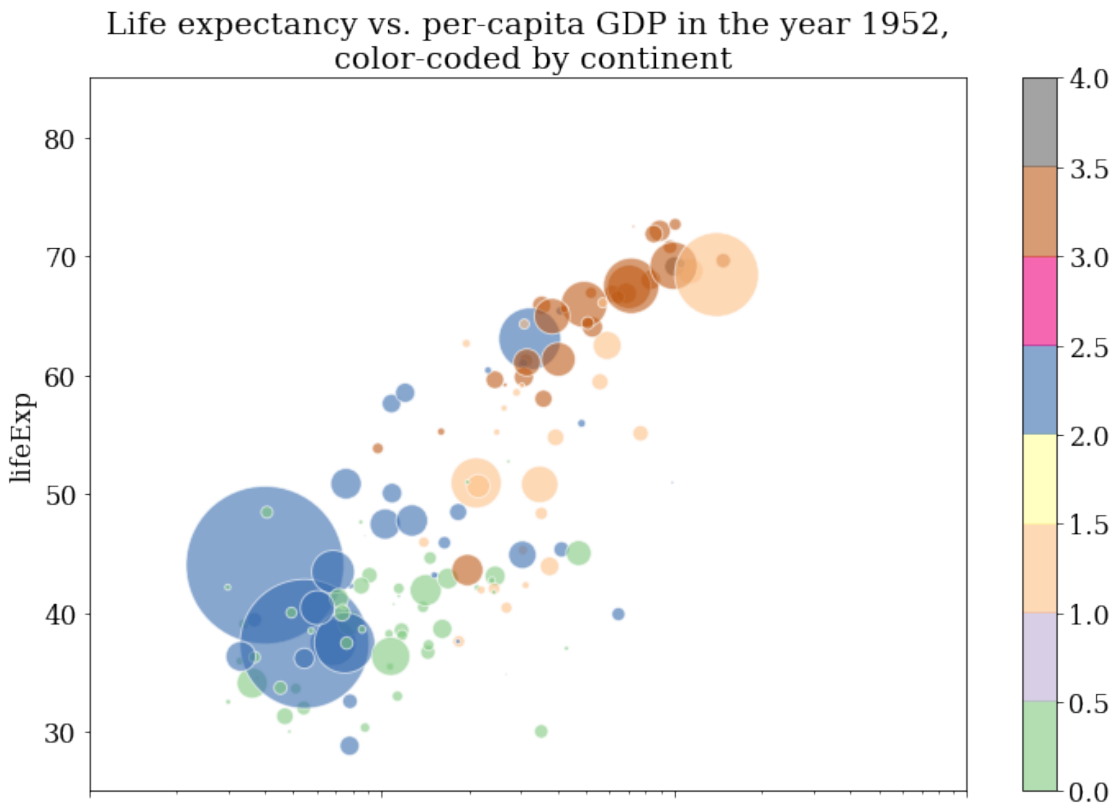
Applying `pandas.Categorical()` to the `continent` column will create an integer value—the *code* of the category—associated to each entry. We can then use these integer values to map to the colors in a colormap. The trick will be to extract the `codes` attribute of the *Categorical* data and save that into a new variable named `colors` (a NumPy array).

```
In [22]: colors = pandas.Categorical(year1952['continent']).codes

In [23]: type(colors)

Out[23]: numpy.ndarray

In [24]: len(colors)

Out[24]: 142

In [25]: print(colors)

[2 3 0 0 1 4 3 2 2 3 0 1 3 0 1 3 0 0 2 0 1 0 0 1 2 1 0 0 0 1 0 3 1 3 3 0 1
 1 0 1 0 0 0 3 3 0 0 3 0 3 1 0 0 1 1 2 3 3 2 2 2 2 3 2 3 1 2 2 0 2 2 2 2 0
 0 0 0 0 2 0 0 0 1 2 3 0 0 2 0 2 3 4 1 0 0 3 2 2 1 1 1 2 3 3 1 0 3 0 0 2 0
 3 0 2 3 3 0 0 3 2 0 0 3 3 2 2 0 2 0 1 0 3 0 3 1 1 1 2 2 2 0 0]
```

You see that `colors` is a NumPy array of 142 integers that can take the values: `0, 1, 2, 3, 4`. They are the codes to `continent` categories: `Africa, Americas, Asia, Europe, Oceania`. For example, the first entry is 2, corresponding to Asia, the continent of Afghanistan.

Now we're ready to re-do our bubble chart, using the array `colors` to set the color of the bubble (according to the continent for the given country).

```
In [26]: year1952.plot.scatter(figsize=(12,8),
                x='gdpPercap', y='lifeExp', s=populations/60000,
                c=colors, cmap='Accent',
title='Life expectancy vs. per-capita GDP in the year 1952,\n color-coded by continent',
                logx = 'True',
                ylim = (25,85),
                xlim = (1e2, 1e5),
                edgecolors="white",
                alpha=0.6);
```



Note: We encountered a bug in `pandas` scatter plots! The labels of the x-axis disappeared when we added the colors to the bubbles. We tried several things to fix it, like adding the line `pyplot.xlabel("GDP per Capita")` at the end of the cell, but nothing worked. Searching online, we found an open issue report for this problem.

Discuss with your neighbor: What do you see in the colored bubble chart, in regards to 1952 conditions in different countries and different continents? Can you guess some countries? Can you figure out which color corresponds to which continent?

5 Spaghetti plot of life expectancy

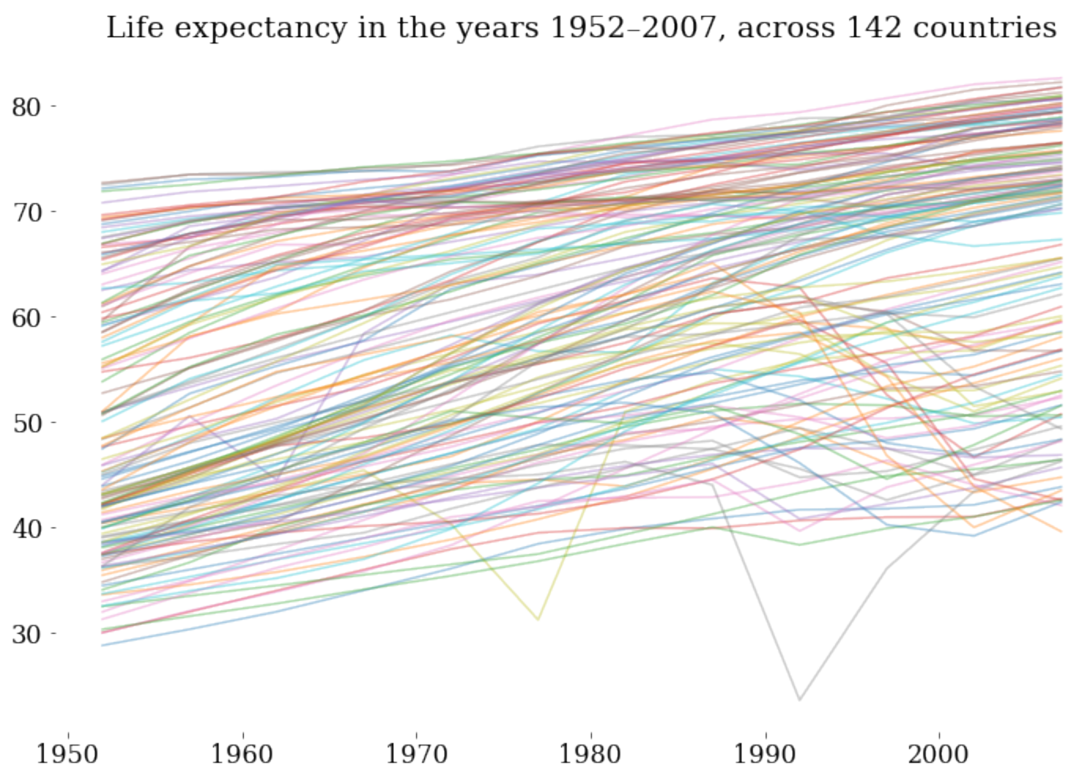
The bubble plot shows us that 1952 life expectancies varied quite a lot from country to country: from a minimum of under 30 years, to a maximum under 75 years. The first part of Prof. Rosling's dying message is *"that the world is making progress*." Is it the case that countries around the world *all* make progress in life expectancy over the years?

We have an idea: what if we plot a line of life expectancy over time, for every country in the data set? It could be a bit messy, but it may give an *overall view* of the world-wide progress in life expectancy.

Below, we'll make such a plot, with 142 lines: one for each country. This type of graphic is called a **spaghetti plot** . . . for obvious reasons!

To add a line for each country on the same plot, we'll use a `for`-statement and the `by_country` groups. For each country-group, the line plot takes the series `year` and `lifeExp` as (x, y) coordinates. Since the spaghetti plot is quite busy, we also took off the box around the plot. Study this code carefully.

```
In [27]: pyplot.figure(figsize=(12,8))

         for key,group in by_country:
             pyplot.plot(group['year'], group['lifeExp'], alpha=0.4)

         pyplot.title('Life expectancy in the years 1952-2007, across 142 countries')
         pyplot.box(on=None);
```



6 Dig deeper and get insights from the data

The spaghetti plot shows a general upwards tendency, but clearly not all countries have a monotonically increasing life expectancy. Some show a one-year sharp drop (but remember, this data jumps every 5 years), while others drop over several years. And something catastrophic happened to one country in 1977, and to another country in 1992. Let's investigate this!

We'd like to explore the data for a particular year: first 1977, then 1992. For those years, we can get the minimum life expectancy, and then find out which country experienced it.

To access a particular group in *GroupBy* data, `pandas` has a `get_group(key)` method, where `key` is the label of the group. For example, we can access yearly data from the `by_year` groups using the year as key. The return type will be a dataframe, containing the same columns as the original data.

```
In [28]: type(by_year.get_group(1977))
```

```
Out[28]: pandas.core.frame.DataFrame
```

```
In [29]: type(by_year['lifeExp'].get_group(1977))
```

```
Out[29]: pandas.core.series.Series
```

Now we can find the minimum value of life expectancy at the specific years of interest, using the `Series.min()` method. Let' do this for 1977 and 1992, and save the values in new Python variables, to reuse later.

```
In [30]: min_lifeExp1977 = by_year['lifeExp'].get_group(1977).min()
         min_lifeExp1977
```

```
Out[30]: 31.219999999999999
```

```
In [31]: min_lifeExp1992 = by_year['lifeExp'].get_group(1992).min()
         min_lifeExp1992
```

```
Out[31]: 23.599
```

Those values of life expectancy are just terrible! Are you curious to know what countries experienced the dramatic drops in life expectancy?

We can find the row *index* of the minimum value, thanks to the `pandas.Series.idxmin()` method. The row indices are preserved from the original dataframe `life_expect` to its groupings, so the index will help us identify the country. Check it out.

```
In [32]: by_year['lifeExp'].get_group(1977).idxmin()
```

```
Out[32]: 221
```

```
In [33]: life_expect['country'][221]
```

```
Out[33]: 'Cambodia'
```

```
In [34]: by_country.get_group('Cambodia')
```

Out[34]:

	continent	gdpPercap	lifeExp	pop	year
216	Asia	368.469286	39.417	4693836.0	1952
217	Asia	434.038336	41.366	5322536.0	1957
218	Asia	496.913648	43.415	6083619.0	1962
219	Asia	523.432314	45.415	6960067.0	1967
220	Asia	421.624026	40.317	7450606.0	1972
221	Asia	524.972183	31.220	6978607.0	1977
222	Asia	624.475478	50.957	7272485.0	1982
223	Asia	683.895573	53.914	8371791.0	1987
224	Asia	682.303175	55.803	10150094.0	1992
225	Asia	734.285170	56.534	11782962.0	1997
226	Asia	896.226015	56.752	12926707.0	2002
227	Asia	1713.778686	59.723	14131858.0	2007

We searched online to learn what was happening in Cambodia to cause such a drop in life expectancy in the 1970s. Indeed, Cambodia experienced a *mortality crisis* due to several factors that combined into a perfect storm: war, ethnic cleansing and migration, collapse of the health system, and cruel famine [2]. It's hard for a country to keep vital statistics under such circumstances, and certainly there are uncertainties in the data for Cambodia in the 1970s. However, various sources report a life expectancy there in 1977 that was *under 20 years*. See, for example, the World Bank's interactive web page on Cambodia.

There is something strange with the data from the The Python Graph Gallery. Is it wrong? Maybe they are giving us *average* life expectancy in a five-year period. Let's look at the other dip in life expectancy, in 1992.

```
In [35]: by_year['lifeExp'].get_group(1992).idxmin()

Out[35]: 1292

In [36]: life_expect['country'][1292]

Out[36]: 'Rwanda'

In [37]: by_country.get_group('Rwanda')

Out[37]:      continent   gdpPercap  lifeExp        pop  year
         1284    Africa  493.323875   40.000  2534927.0  1952
         1285    Africa  540.289398   41.500  2822082.0  1957
         1286    Africa  597.473073   43.000  3051242.0  1962
         1287    Africa  510.963714   44.100  3451079.0  1967
         1288    Africa  590.580664   44.600  3992121.0  1972
         1289    Africa  670.080601   45.000  4657072.0  1977
         1290    Africa  881.570647   46.218  5507565.0  1982
         1291    Africa  847.991217   44.020  6349365.0  1987
         1292    Africa  737.068595   23.599  7290203.0  1992
         1293    Africa  589.944505   36.087  7212583.0  1997
         1294    Africa  785.653765   43.413  7852401.0  2002
         1295    Africa  863.088464   46.242  8860588.0  2007
```

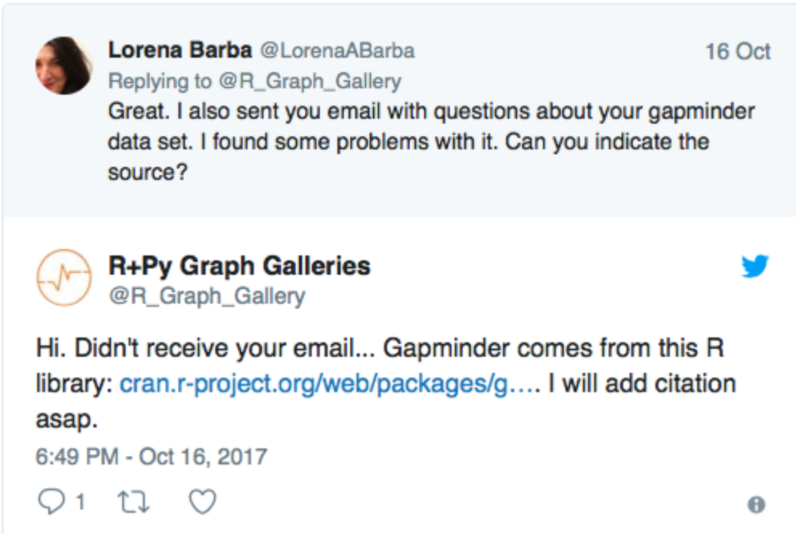
The World Bank's interactive web page on Rwanda gives a life expectancy of 28.1 in 1992, and even lower in 1993, at 27.6 years. This doesn't match the value from the data set we sourced from The Python Graph Gallery, which gives 23.6—and since this value is *lower* than the minimum value given by the World Bank, we conclude that the discepancy is not caused by 5-year averaging.

7 Checking data quality

All our work here started with loading a data set we found online. What if this data set has *quality* problems?

Well, nothing better than asking the author of the web source for the data. We used Twitter to communicate with the author of The Python Graph Gallery, and he replied with a link to *his source*: a data package used for teaching a course in Exploratory Data Analysis at the University of British Columbia.

```
In [38]: %%html
         <blockquote class="twitter-tweet" data-lang="en"><p lang="en" dir="ltr">...
```



Note one immediate outcome of our reaching out to the author of The Python Graph Gallery: he realized he was not citing the source of his data [3], and promised to add proper credit. *It's always good form to credit your sources!*

We visited the online repository of the data source, and posted an issue report there, with our questions about data quality. The author promptly responded, saying that *her* source was the Gapminder.org website—**Gapminder** is the non-profit founded by Hans Rosling to host public data and visualizations. She also said: *"I don't doubt there could be data quality problems! It should definitely NOT be used as an authoritative source for life expectancy."*

So it turns out that the data we're using comes from a set of tools meant for teaching, and is not up-to-date with the latest vital statistics. The author ended up adding a warning to make this clear to visitors of the repository on GitHub.

This is a wonderful example of how people collaborate online via the open-source model.

Note: For the most accurate data, you can visit the website of the World Bank.

8 Using widgets to visualize interactively

One more thing! This whole exploration began with our viewing the 2006 TED Talk by Hans Rosling: "The best stats you've ever seen". One of the most effective parts of the presentation is seeing the *animated* bubble chart, illustrating how countries became healthier and richer over time. Do you want to make something like that?

You can! Introducing Jupyter Widgets. The magic of interactive widgets is that they tie together the running Python code in a Jupyter notebook with Javascript and HTML running in the browser.

You can use widgets to build interactive controls on data visualizations, with buttons, sliders, and more.

To use widgets, the first step is to import the `widgets` module.

```
In [39]: from ipywidgets import widgets
```

After importing `widgets`, you have available several UI (User Interaction) elements. One of our favorites is a *Slider*: an interactive sliding button. Here is a default slider that takes integer values, from 0 to 100 (but does nothing):

```
In [40]: widgets.IntSlider()
```

```
A Jupyter Widget
```

What we'd like to do is make an interactive visualization of bubble charts, with the year in a slider, so that we can run forwards and backwards in time by sliding the button, watching our plot update the bubbles in real time. Sound like magic? It almost is.

The magic happens when you program what should happen when the value in the slider changes. A typical scenario is having a function that is executed with the value in the slider, interactively. To create that, we need two things:

1. A function that will be called with the slider values, and
2. A call to an *interaction* function from the `ipywidgets` package.

Several interaction functions are available, for different actions you expect from the user: a click, a text entered in a box, or sliding the button on a slider. You will need to explore the Jupyter Widgets documentation [4] to learn more.

For this example, we'll be using a slider, a plotting function that makes our bubble chart, and the `.interact()` function to call our plotting function with each value of the slider.

We do everything in one cell below. The first line creates an integer-value slider with our known years—from a minimum 1952, to a maximum 2007, stepping by 5—and assigns it to the variable name `slider`.

Next, we define the function `roslingplot()`, which re-calculates the array of population values, gets the year-group we need from the `by_year` *GroupBy* object, and makes a scater plot of life expectancy vs. per-capita income, like we did above. The `populations` array (divided by 60,000) sets the size of the bubble, and the previously defined `colors` array sets the color coding by continent.

We also removed the colorbar (which added little information), and added the option `sharex=False` following the workaround suggested by someone on the open issue report for the plotting bug we mentioned above.

The last line in the cell below is a call to `.interact()`, passing our plotting function and the slider value assigned to its argument, `year`. Watch the magic happen!

```
In [41]: slider = widgets.IntSlider(min=1952, max=2007, step=5)

         def roslingplot(year):
             populations = by_year.get_group(year)['pop'].values
```

```
        by_year.get_group(year).plot.scatter(figsize=(12,8),
                 x='gdpPercap', y='lifeExp', s=populations/60000,
                 c=colors, cmap='Accent',
                 title='Life expectancy vs per-capita GDP in the year '+ str(year)+'\n',
                 logx = 'True',
                 ylim = (25,85),
                 xlim = (1e2, 1e5),
                 edgecolors="white",
                 alpha=0.6,
                 colorbar=False,
                 sharex=False)
        pyplot.show();

    widgets.interact(roslingplot, year=slider);
```

```
A Jupyter Widget
```

9 References

1. The Soviet War in Afghanistan, 1979-1989, The Atlantic (2014), by Alan Taylor.
2. US National Research Council Roundtable on the Demography of Forced Migration; H.E. Reed, C.B. Keely, editors. Forced Migration & Mortality (2001), National Academies Press, Washington DC; Chapter 5: The Demographic Analysis of Mortality Crises: The Case of Cambodia, 1970-1979, Patrick Heuveline. Available at: https://www.ncbi.nlm.nih.gov/books/NBK223346/
3. gapminder: Data from Gapminder (R data package), by Jennifer (Jenny) Bryan, repository at https://github.com/jennybc/gapminder, v0.3.0 (Version v0.3.0) on Zenodo: https://doi.org/10.5281/zenodo.594018, licensed CC-BY 3.0
4. Jupyter Widgets User Guide