Python Statistical Concepts Exercises

Numpy handles many statistics functions.

Below we have mean and median. Unfortunately, just like in R, there is no mode command, but we can fake it using Set.

```
x = [2,5,7,9,10,15,16]
np.mean(x)
```

```
9.1428571428571423
```

```
np.median(x)
```

```
9.0
```

using set to fake mode

```
from sets import Set
y = [2,5,2,8,2,10]
```

```
max(Set(y), key=y.count)
```

```
2
```

Numpy can also be used to find range, variance, and standard deviation.

```
: #range (max - min)
  np.ptp(x)
: 22.122448979591834

: #variance
  np.var(x)
: 4.7034507523298075

: #standard deviation
  np.std(x)
: 14
```

Frequency Distribution

To understand the Central Limit Theorem, first you need to be familiar with the concept of Frequency Distribution.

Let's look at this Python code below. Here I am importing the module random from numpy. I then use the function random_integers from random. Here is the syntax:

```
random.random_integers(Max value, number of elements)
```

So `random.random_integers(10, size =10)` would produce a list of 10 numbers between 1 and 10.

Below I selected 20 numbers between 1 and 5

```
: from numpy import random
  x = random.random_integers(5, size = 20)
  x
: array([4, 1, 1, 5, 3, 2, 2, 3, 5, 5, 1, 4, 1, 3, 2, 5, 5, 1, 1, 1])
```

Now, since I am talking about a Frequency Distribution, I'd bet you could infer that I am concerned with Frequency. And you would be right. Looking at the data above, this is what I have found.

I create a table of the integers 1 – 5 and I then count the number of time (frequency) each number appears in my list above.

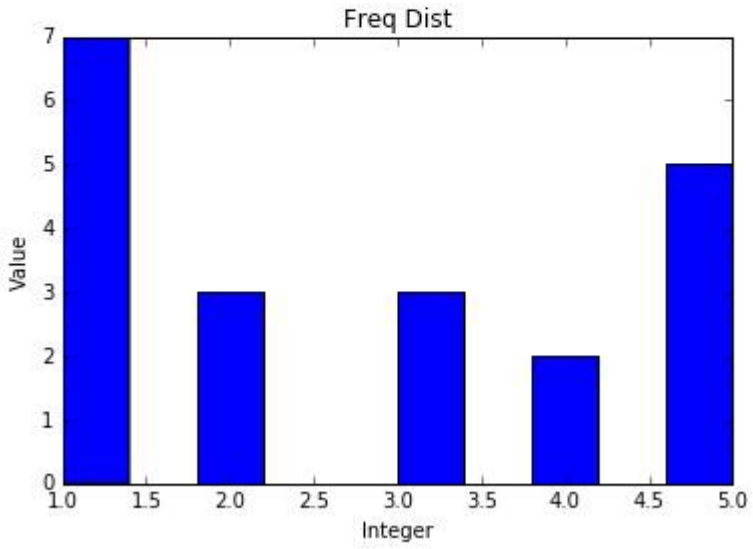
	A	B
	Number	**Freq**
2	1	7
3	2	3
4	3	3
5	4	2
6	5	5
7		

Histogram

Using my Frequency table above, I can easily make a bar graph commonly known as a histogram. Let's use matplotlab to build this.

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.hist(x)
plt.title('Freq Dist')
plt.xlabel('Integer')
plt.ylabel('Value')
```

```
<matplotlib.text.Text at 0xb764550>
```



Now lets, do it with even more data points (100 elements from 1 to 10 to be exact)

Central Limit Theorem

The Central Limit Theorem is one of core principles of probability and statistics. So much so, that a good portion of inferential statistical testing is built around it. What the Central Limit Theorem states is that, given a data set – let's say of 100 elements if I were to take a random sampling of 10 data points from this sample and take the average (arithmetic mean) of this sample and plot the result on a histogram, given enough samples my histogram would approach what is known as a normal bell curve.

In plain English

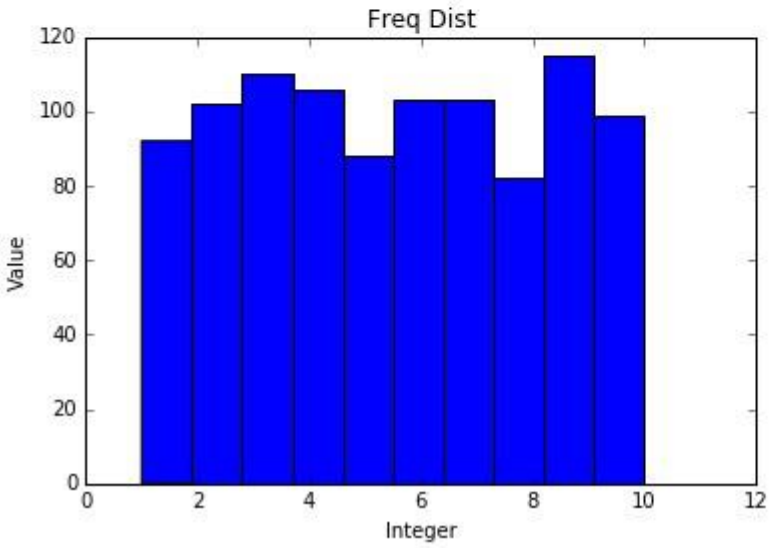
- Take a random sample from your data
- Take the average of your sample
- Plot your sample on a histogram
- Repeat 1000 times
- You will have what looks like a normal distribution bell curve when you are done.

```
from numpy import random
x = random.random_integers(10, size = 100)
x
```

```
array([ 8, 10,  1,  7,  6,  9,  2,  6,  8,  2,  5,  8,  5,  7,  7,  5, 10,
        7, 10,  2,  8,  3,  7,  3,  9,  5,  9,  9,  4,  4,  5,  7,  9,  6,
        3,  4,  7, 10,  8, 10,  1,  3,  7,  7,  5,  3,  4,  8,  9,  8,  4,
        8,  6,  6, 10,  7,  1,  4,  1,  3, 10,  3,  2,  5, 10,  9,  9,  3,
        9,  3, 10,  1,  6,  2,  3,  3,  7,  9,  8,  1,  7,  7,  7,  7,  4,
        4,  4,  1,  4,  9,  6,  8,  8,  5,  6, 10,  6,  5,  7,  2])
```

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.hist(x)
plt.title('Freq Dist')
plt.xlabel('Integer')
plt.ylabel('Value')
```

```
<matplotlib.text.Text at 0xb764f98>
```



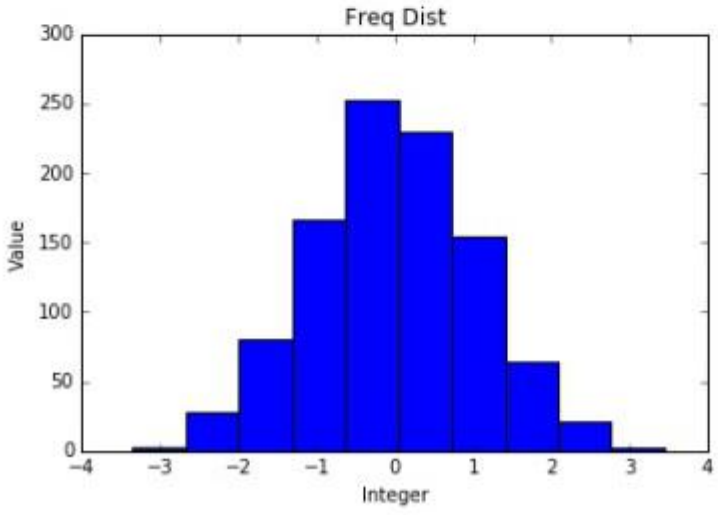
For those who don't know what a normal distribution bell curve looks like, here is an example. I created it using numpy's normal method

```
In [18]: y = random.normal(size=1000)
         y
```

```
Out[18]: array([  8.49812022e-01,   7.62855492e-01,   5.15802467e-01,
                 -5.11359377e-01,  -2.45581108e+00,  -3.00906896e-01,
                 -1.34091706e-01,  -6.44733903e-01,   7.75515121e-02,
                 -7.17617754e-01,  -3.68289876e-01,  -1.68128141e+00,
                  5.83403850e-01,  -2.78844154e-01,   3.31235868e-01,
                  5.68784732e-01,  -6.89099132e-01,  -7.01290269e-01,
                  1.84021627e+00,   1.95352996e+00,   1.91597223e+00,
                  1.41786246e-01,   1.76419156e+00,  -1.79234447e-01,
                  1.13039184e-01,   1.34791933e+00,  -2.43326352e-01,
                 -1.18699617e+00,  -4.86441702e-01,   1.23153332e+00,
                  1.19330106e+00,   1.25940046e+00,  -6.73401916e-01,
                  6.65441640e-02,   6.20138549e-01,  -6.43633964e-02,
                  1.32553900e+00,  -8.50930997e-01,  -3.61375352e-01,
                  2.26123998e+00,  -3.26815970e-01,  -1.68269429e+00,
                 -2.84098582e-03,   7.62992010e-01,  -1.45446759e-02,
                  9.54785029e-01,  -7.40008205e-01,  -6.62882467e-02,
                 -1.70283013e+00,  -1.26253176e+00,   5.64887754e-01,
                 -8.54420681e-01,   9.93698910e-01,   2.23197442e-01,
                 -1.62093995e+00,   2.20648894e-01,   5.93250598e-01,
```

```
In [20]: plt.hist(y)
         plt.title('Freq Dist')
         plt.xlabel('Integer')
         plt.ylabel('Value')
```

```
Out[20]: <matplotlib.text.Text at 0xb895780>
```



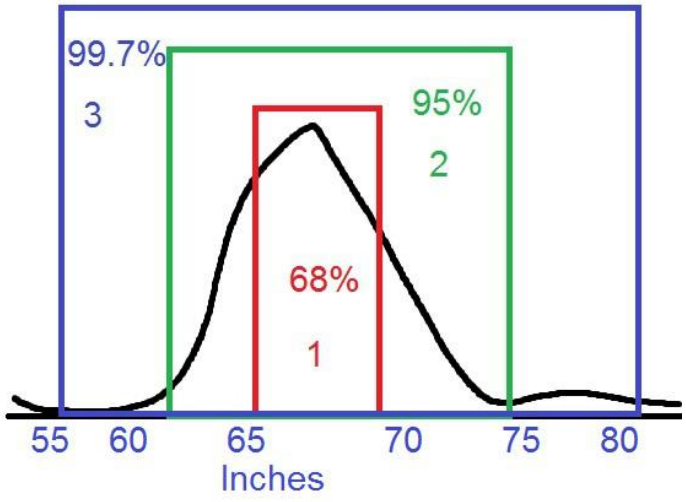
The normal distribution of (Gaussian Distribution – named after the mathematician Carl Gauss) is an amazing statistical tool. This is the powerhouse behind inferential statistics.

The Central Limit Theorem tells me (under certain circumstances), no matter what my population distribution looks like, if I take enough means of sample sets, my sample distribution will approach a normal bell curve.

Once I have a normal bell curve, I now know something very powerful.

Known as the 68,95,99 rule, I know that 68% of my sample is going to be within one standard deviation of the mean. 95% will be within 2 standard deviations and 99.7% within 3.

So let's apply this to something tangible. Let's say I took random sampling of heights for adult men in Ireland. I may get something like this (warning, this data is completely made up)

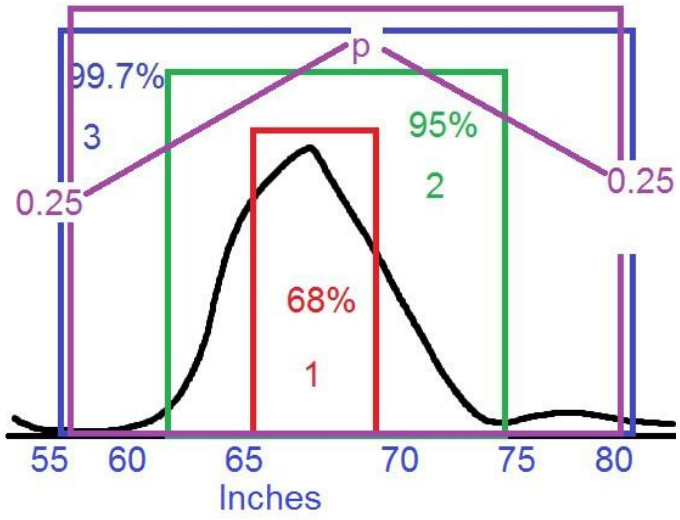


But reading this graph, I can see that 68% of men are between 65 and 70 inches tall. While less than 0.15% of men are shorter than 55 inches or taller than 80 inches.

Now, there are plenty of resources online if you want to dig deeper into the math. However, if you just want to take my word for it and move forward, this is what you need to take away from this lesson:

p value

As we move into statistical testing like Linear Regression, you will see that we are focus on a p value. And generally, we want to keep that p value under 0.5. The purple box below shows a p value of 0.5 – with 0.25 on either side of the curve. A finding with a p value that low basically states that there is only a 0.5% chance that the results of whatever test you are running are a result of random chance. In other words, your results are 99% repeatable and your test demonstrates statistical significance.



Co-variance and Correlation

In this lab, I will using data from a CSV file. You can download the file from the lab folder called : heightWeight

The Data

The data set includes 20 heights (inches) and weights(pounds). Given what you already know, you could tell me the average height and average weight. You could tell me medians, variances and standard deviations.

	A	B
1	68	165
2	71	201
3	61	140
4	69	170
5	71	192
6	58	125
7	72	195
8	73	205
9	58	115
10	74	210
11	61	135
12	59	125
13	69	172
14	68	175
15	64	145
16	69	170
17	72	200
18	66	155
19	65	150
20	69	171

But all of those measurements are only concerned with a single variable. What if I want to see how height interacts with weight? Does weight increase as height increases?

***Note while there are plenty of fat short people and overly skinny tall people, when you look at the population at large, taller people will tend to weigh more than shorter people. This generalization of information is very common as it gives you a big picture view and is not easily skewed by outliers.*

Populate Python with Data

The first thing we are going to focus on is co-variance. Let's start by getting our data in Python.

```
import csv
f = open('C:\\Users\\Benjamin\\Documents\\HeightWeight.csv', 'rb')
rdF = csv.reader(f)

hgt = []
wgt = []

for row in rdF:
    hgt.append(row[0])
    wgt.append(row[1])

print hgt
print wgt
```

```
['68', '71', '61', '69', '71', '58', '72', '73', '58', '74', '61', '59', '69', '68', '64', '69', '72', '66', '65', '6
9']
['165', '201', '140', '170', '192', '125', '195', '205', '115', '210', '135', '125', '172', '175', '145', '170', '20
0', '155', '150', '171']
```

Now there is a small problem. Our lists are filled with strings, not numbers. We can't do calculations on strings.

We can fix this by populating converting the values using int(). Below I created 2 new lists (height and weight), created a for loop counting up to number of values in our lists : range(len(hgt)). Then I filled the new lists using lst.append(int(value))

```
height = []
weight = []
for x in range(len(hgt)):
    height.append(int(hgt[x]))
    weight.append(int(wgt[x]))

print height
print weight
```

```
[68, 71, 61, 69, 71, 58, 72, 73, 58, 74, 61, 59, 69, 68, 64, 69, 72, 66, 65, 69]
[165, 201, 140, 170, 192, 125, 195, 205, 115, 210, 135, 125, 172, 175, 145, 170, 200, 155, 150, 171]
```

***Now I know I could have resolved this in fewer steps, but this is a tutorial, so I want to provide more of a walk through.*

Correlation

We have another measurement known as correlation. A very basic correlation equation divides out the standard deviation of both height and weight. The result of a correlation is between 1 and -1. With -1 being perfect anti-correlation and 1 being perfect correlation. 0 mean no correlation exists.

numpy's corrcoef() shows us a correlation matrix. Ignore the 1's – they are part of what is known as the identity. Instead look at the other numbers = 0.97739. That is about as close to one as you will ever get in reality.

```
np.corrcoef(height,weight)
```

```
array([[ 1.        ,  0.97739957],
       [ 0.97739957,  1.        ]])
```

Hypothesis Testing(T Test)

Hypothesis testing is a first step into really understanding how to use statistics.

The purpose of the test is to tell if there is any significant difference between two data sets.

Consider the follow example:

Let's say I am trying to decide between two computers. I want to use the computer to run advanced analytics, so the only thing I am concerned with is speed.

I pick a sorting algorithm and a large data set and run it on both computers 10 times, timing each run in seconds.

Now I put the results into two lists. A and B

```
a = [10,12,9,11,11,12,9,11,9,9]
b = [13,11,9,12,12,11,12,12,10,11]
```

A quick look at the data makes me think b is slower than a. But is it slower enough to mean something or are these results just a matter of chance (meaning if I ran the test 200 more times would the end result be closer to equal or further apart).

Hypothesis test

To find out, let's do a hypothesis test.

Set our Hypothesis:

- H0 = H1 – there is no significant difference between data sets
- H0 <> H1 – there is a significant difference

To test our hypothesis, let's run a t-test

import stats from scipy and run stats.ttest_ind().

Our output is the z-statistic and the p-value.

Our p-value is 0.08 – greater than the common significance value of 0.05. Since it is greater, we cannot reject H0=H1. This means both computers are effectively the same speed.

```
: from scipy import stats
  c = stats.ttest_ind(a,b)
  c
: Ttest_indResult(statistic=-1.8534061896456464, pvalue=0.080289068931635468)
```

Let's try a third computer – d

```
d = [13,12,9,12,12,13,12,13,10,11]
```

Now, let's run a second T-test. This one comes back with a p-value of 0.026 – under 0.05. This means we can reject our hypothesis that a=d. The speed differences between a and d are significant.

```
c = stats.ttest_ind(a,d)
c
```

```
Ttest_indResult(statistic=-2.4168284181234285, pvalue=0.026494688246160383)
```

Box whisker plot

Box whisker plots are used in stats to graphically view the spread of a data set, as well as to compare data sets.

data set: sensors is available on the lab folder

Using pandas, let's load the data set

```
import pandas as pd
import matplotlib as mp
import matplotlib.pyplot as plt

sensorDF = pd.read_excel("C:\FILE_PATH\sensors.xlsx")
sensorDF.head()
```

Our data set represents monthly readings taken from 4 sensors over the span of a year

```
%matplotlib inline
import pandas as pd
import matplotlib as mp
import matplotlib.pyplot as plt

sensorDF = pd.read_excel("C:\Users\Benjamin\Documents\sensors.xlsx")
sensorDF.head()
```

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	79	83	95	89	89	81	80	94	86	82	96	89
1	90	86	86	91	87	92	83	82	91	84	94	85
2	72	83	69	81	71	85	87	73	78	86	82	75
3	80	95	98	82	98	81	88	79	88	95	94	87

We need to convert the dataframe to a list values for our box plot function.

To do this, first we need to flatten() our dataframe. The flatten() method places all the values from the dataframe into 1 list

```
[16]: sensors = list(sensorDF.values.flatten())
      sensors
```

```
[16]: [79,
       83,
       95,
       89,
       89,
       81,
```

Now let us chop the list into the for sensors represented by the rows in our dataframe

```
s1 = sensors[0:12]
s2 = sensors[12:24]
s3 = sensors[24:36]
s4 = sensors[36:48]
print s1
print s2
print s3
print s4
```

```
[79, 83, 95, 89, 89, 81, 80, 94, 86, 82, 96, 89]
[90, 86, 86, 91, 87, 92, 83, 82, 91, 84, 94, 85]
[72, 83, 69, 81, 71, 85, 87, 73, 78, 86, 82, 75]
[80, 95, 98, 82, 98, 81, 88, 79, 88, 95, 94, 87]
```

Finally, we need to make a list of these lists

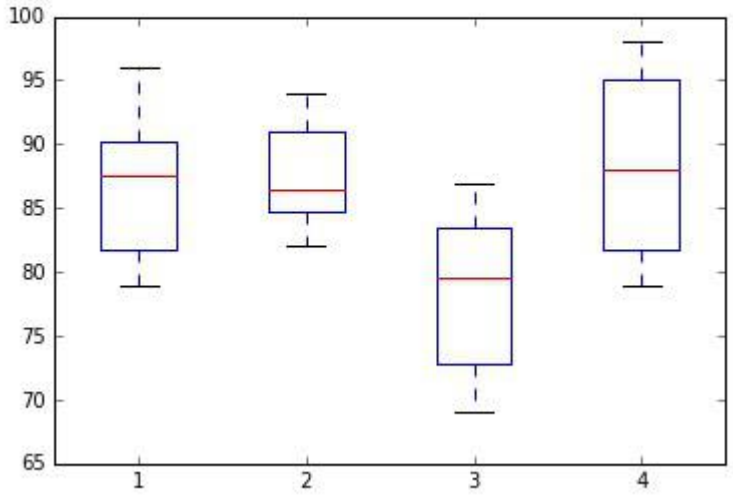
```
sList = [s1,s2,s3,s4]
```

I know that seemed like a lot, but you will spend more time cleaning and prepping data than any other task. It is just the nature of the job.

Let's Plot

The code for creating a boxplot is now easy.

```
]: bp = plt.boxplot(sList)
```



Let’s label our chart a little better now.

```
: fig, ax1 = plt.subplots(figsize=(10, 6))
  ax1.set_title('Comparing Sensor Readings')
  ax1.set_xlabel('Sensors')
  ax1.set_ylabel('Readings')
  bp = ax1.boxplot(sList)
```