Module 6

Sample Mean and Sample Standard Deviation

In [47]:
```
import numpy as np
from math import sqrt
```

Pb 6.1

Setup

In [52]:
```
X = [500, 2000, 2500, 5000, 10000]
n = [96, 107, 130, 89, 78]
n_groups = len(n)
n_people = sum(n)
print("n_groups=", n_groups, " n_people=", n_people)
```

```
n_groups= 5  n_people= 500
```

Sample Mean

In [88]:
```
x_bar = sum([Xi*ni for Xi, ni in zip(X, n)])/n_people
print("x_bar=", x_bar)
```

```
x_bar= 3624.0
```

Equation 6.3 from Montgomery probably looks tempting, right? It turns out you can't use this directly on the table of values given. Why? Because group data is given, whereas Equation 6.3 sums over individuals.

Sample Standard Deviation (Long Way)

A simple approach is to convert the group data to data of individuals. This is not the best approach, but it illustrates use of Eq 6.3.

In [89]:
```
X[0]
```

Out[89]:
```
500
```

In [90]:
```
[X[0]]*5
```

Out[90]:
```
[500, 500, 500, 500, 500]
```

In [91]:
```
print([X[0]]*n[0])
```

```
[500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 50
0, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 5
00, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500]
```

In [92]:
```
print([[xsub]*nsub for xsub, nsub in zip(X, n)])
```

```
[[500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 50
0, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 5
00, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500], [2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000,
2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2
000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 20
00, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000], [2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 25
00, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500], [5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 50
00, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000], [10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 1000
0, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10
000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 1000
0, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10
000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 1000
0]]
```

This gives us a (ragged) list of lists; however, a flattened version will be easier to deal with. From a quick Google search "flatten ragged list of lists", we find an example of how to do this and some sources:

https://stackabuse.com/python-how-to-flatten-list-of-lists/ (https://stackabuse.com/python-how-to-flatten-list-of-lists/)

https://stackoverflow.com/questions/9057379/correct-and-efficient-way-to-flatten-array-in-numpy-in-python (https://stackoverflow.com/questions/9057379/correct-and-efficient-way-to-flatten-array-in-numpy-in-python)

https://stackoverflow.com/questions/2158395/flatten-an-irregular-list-of-lists (https://stackoverflow.com/questions/2158395/flatten-an-irregular-list-of-lists)

In [93]:
```python
def flatten(regular_list):
    return [item for sublist in regular_list for item in sublist]
```

In [94]:
```python
print(flatten(individuals))
```

```
[500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 50
0, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 5
00, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
500, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 20
00, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 200
0, 2000, 2000, 2000, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 2500, 250
0, 2500, 2500, 2500, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 500
0, 5000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10
000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 1000
0, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10
000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 1000
0, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000]
```

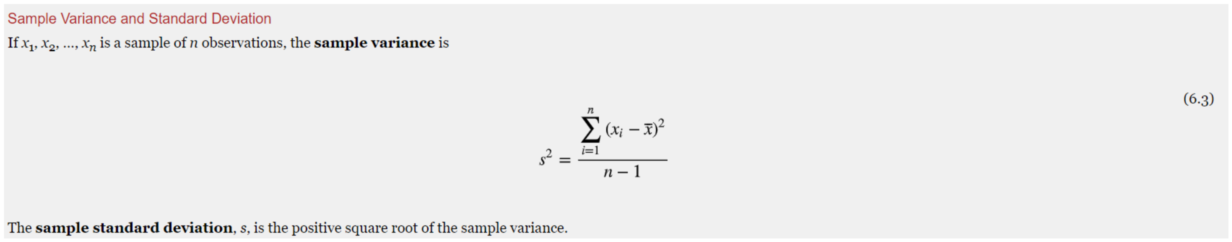
All together, this would look like:

In [108]:
```python
def expand_groups(X, n):
    persons = flatten([[xsub]*nsub for xsub, nsub in zip(X, n)])
    return persons
```

In [132]:
```python
x = expand_groups(X, n)
print("length=",len(x))
print("mean=",np.mean(x))
```

```
length= 500
mean= 3624.0
```

Now that we have a flat list of all 500 people and their corresponding deposits, we can plug this directly into Eq 6.3 from the book:



Sample Variance and Standard Deviation

If $x_1, x_2, ..., x_n$ is a sample of n observations, the **sample variance** is

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1} \quad (6.3)$$

The **sample standard deviation**, s, is the positive square root of the sample variance.

In [110]:
```python
std = sqrt(sum([(xi-x_bar)**2 for xi in x])/(n_people-1))
print("sample standard deviation=", std)
```

```
sample standard deviation= 3076.451382065451
```

Sample Standard Deviation (Better Way)

Analytical Formula

If you keep reading in Section 6.1 of the textbook, however, it turns there is an easier way to calculate this (Eq 6.4).

$$s^2 = \frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}$$

Now that the x_i no longer appear directly next to $\bar{x}$, and because for a single group with m people, we can write:

$\sum_{i=1}^{m} x^2 = mx^2$

we can rewrite the following in terms of "groups":

$$\sum_{i=1}^{n_{people}} x_i^2 = \sum_{i=1}^{n_{groups}} n_i X_i^2$$

and

$$\sum_{i=1}^{n_{people}} x_i = \sum_{i=1}^{n_{groups}} n_i X_i$$

Substituting this back in, we get:

$$\frac{\sum_{i=1}^{n_{people}} x_i^2 - \frac{\left(\sum_{i=1}^{n_{people}} x_i\right)^2}{n_{people}}}{n_{people}-1} = \frac{\sum_{i=1}^{n_{groups}} n_i X_i^2 - \frac{\left(\sum_{i=1}^{n_{groups}} (X_i n_i)\right)^2}{n_{people}}}{n_{people}-1}$$

Coded Version

And finally, to convert the RHS of this equation to code:

In [129]:
```
def frequency_sample_std_dev(X, n):
    """Sample standard deviation for X and n,
    where X[i] is the quantity each person in group i has,
    and n[i] is the number of people in group i."""
    n_groups = len(n)
    n_people = sum(n)
    lhs_numerator = sum([ni*Xi**2 for Xi, ni in zip(X, n)])
    rhs_numerator = sum([Xi*ni for Xi, ni in zip(X,n)])**2/n_people
    denominator = n_people-1
    var = (lhs_numerator - rhs_numerator) / denominator
    std = sqrt(var)
    return std
```

In [130]:
```
std = frequency_sample_std_dev(X, n)
print(std)
```

```
3076.451382065451
```

Example with More Data

If we had an entirely different set of values or even a different problem with the same format, we can compute this simply as:

In [131]:
```
X2 = [500, 2000, 2500, 5000, 10000, 12000, 16000, 20000, 22500]
n2 = [96, 107, 130, 89, 78, 48, 158, 62, 95]
std2 = frequency_sample_std_dev(X2, n2)
print("sample standard deviation=",std2)
```

```
sample standard deviation= 7755.100282255846
```

Other Code Sources

This technique is simple enough that it's likely someone has already written some code and made it publicly available. Using a few different search queries: "sample standard deviation python", "sample standard deviation with repeats python", and finally settling on the following webpage: Weighted standard deviation in NumPy (https://stackoverflow.com/questions/2413522/weighted-standard-deviation-in-numpy).

Having the right keywords is very important (a search for weighted standard deviation would quickly pull up what we're looking for), and we also need to make sure that the code we're using is appropriate for the use-case.

Here are a few examples from the linked StackOverflow post:

Population Average and Standard Deviation

In [117]:
```
def weighted_avg_and_std(values, weights):
    """
    Return the weighted average and standard deviation.

    values, weights -- Numpy ndarrays with the same shape.
    """
    average = np.average(values, weights=weights)
    # Fast and numerically precise:
    variance = np.average((values-average)**2, weights=weights)
    return (average, sqrt(variance))

print(weighted_avg_and_std(X, n))
```

```
(3624.0, 3073.3733909175435)
```

Sample Average and Standard Deviation

Note that this gave a "population" standard deviation rather than a "sample" standard deviation. This can be corrected as follows:

In [128]:
```
def weighted_sample_avg_std(values, weights):
    """
    Return the weighted average and weighted sample standard deviation.

    values, weights -- Numpy ndarrays with the same shape.

    Assumes that weights contains only integers (e.g. how many samples in each gr

    See also https://en.wikipedia.org/wiki/Weighted_arithmetic_mean#Frequency_wei
    """
    average = np.average(values, weights=weights)
    variance = np.average((values-average)**2, weights=weights)
    variance = variance*sum(weights)/(sum(weights)-1)
    return (average, sqrt(variance))

print(weighted_sample_avg_std(X, n))
```

```
(3624.0, 3076.451382065451)
```

The value (3076.45) now matches what we were getting previously.

Frequency Histograms

See histogram-auto-vs-manual.ipynb