Randomness and Reproducibility

As we learned in the beginning of this week, the concept of randomness is a cornerstone for statistical inference when drawing samples from larger populations.

In this tutorial, we are going to cover the following:

- Randomness and its uses in python.
- Utilizing python seeds to reproduce analysis.
- Generating random variables from a probability distribution.
- Random sampling from a population.

What is Randomness?

In the beginning of this week's lectures, we touched on the significance of randomness when it comes to performing statistical inference on population samples. If we have complete randomness, our estimates of means, proportions, and totals are ubiased. This means our estimates are equal to the population values on average.

In Python, we refer to randomness as the ability to generate data, strings, or, more generally, numbers at random.

However, when conducting analysis it is important to consider reproducibility. If we are creating random data, how can we enable reproducible analysis?

We do this by utilizing pseudo-random number generators (PRNGs). PRNGs start with a random number, known as the seed, and then use an algorithm to generate a psuedo-random sequence based on it.

This means that we can replicate the output of a random number generator in python simply by knowing which seed was used.

We can showcase this by using the functions in the python library ***random***.

Setting a Seed and Generating Random Numbers

```
In [22]: import random
```

```
In [25]: random.seed(1234)

         random.random()
```

```
Out[25]: 0.9664535356921388
```

```
In [30]: random.seed(1234)

         random.random()
```

```
Out[30]: 0.9664535356921388
```

Random Numbers from Real-Valued Distributions

Uniform

```
In [34]: random.uniform(25,50)
```

```
Out[34]: 48.4817249340941
```

```
In [39]: unifNumbers = [random.uniform(0,1) for _ in range(1000)]
```

In [40]:
```
unifNumbers
```

```
Out[40]: [0.23788198954553252,
          0.760782064096187,
          0.5451979628358582,
          0.43542427937193884,
          0.13391215823727187,
          0.31798334545694995,
          0.1384662575556831,
          0.8052689019764517,
          0.33091378468188515,
          0.17408067158027618,
          0.20924753778922134,
          0.2597532013462456,
          0.3608174225378401,
          0.8055003693807395,
          0.6374736865961922,
          0.2844211325609758,
          0.9793307379875307,
          0.5338735535084044,
          0.0806373235735206,
          0.39550468816698325,
          0.3975027456035918,
          0.2252087011381787,
          0.8396728631824533,
          0.35872302835435266,
          0.25564989570132823,
          0.1361596776723818,
          0.13677599605328916,
          0.965484270832286,
          0.8594233061813019,
          0.45697878504608436,
          0.43557667171906733,
          0.39343253492529473,
          0.39185351936711377,
          0.06978900219801909,
          0.30761283428998143,
          0.6030641842819198,
          0.05462321609862608,
          0.04604763607678508,
          0.6983096303231778,
          0.06800493213575587,
          0.7178099174727228,
          0.23023028117334798,
          0.18628794772279178,
          0.6964716063790292,
          0.07652950369557432,
          0.7513138484550492,
          0.7535527075874838,
          0.5834206717505125,
          0.36678107309503116,
          0.140133861198243,
          0.44638213194534193,
          0.09842156587991124,
          0.7665995540076791,
          0.44125311904173403,
          0.4770863739013008,
          0.4467073510492393,
          0.5174147240852541,
          0.1388031568384971,
          0.5230558481933215,
          0.9739525394354624,
          0.8018641913949679,
          0.2063660315131386,
          0.952349586002711,
          0.5006441978577517,
          0.23627112518373916,
          0.8851569743829092,
          0.12766827900589794,
          0.3557017979213829,
          0.0015246047590677936,
          0.8563510346044559,
          0.04548370816604941,
          0.14869265002371224,
          0.8261458864863193,
          0.9214398763721203,
          0.7592939873531079,
          0.21444549996428308,
          0.7570802277403016,
          0.31218758114238643,
          0.3382642920015092,
```

Normal

In [44]:
```
mu = 0

sigma = 1

random.normalvariate(mu, sigma)
```

Out[44]: 0.30852342022571383

In [47]:
```
mu = 5

sigma = 2

random.normalvariate(mu, sigma)
```

Out[47]: 4.983951444189755

In [49]:
```
mu = 0

sigma = 1

[random.normalvariate(mu, sigma) for _ in range(10000)]
```

```
Out[49]: [0.31719356185477876,
          1.5322305782142647,
          -1.064561617549296,
          0.6719572942183434,
          -1.5888425522142458,
          1.0079456179631017,
          0.319288984742672,
          -0.04631902244344771,
          0.3080500195313596,
          0.3831024866835417,
          0.7560887769963058,
          -0.9628618215341885,
          -0.013026515123845969,
          0.46235258959346276,
          0.09226149649418314,
          -0.5054244226049383,
          0.1607466030869456,
          0.2613889412582743,
          0.4546794564190649,
          -0.9394085659422335,
          -0.6828237462376194,
          -0.3601769303612483,
          1.3980139773043236,
          -2.274409195996234,
          2.5725416827853342,
          1.074892408065199,
          0.29031159536310464,
          1.1276259916031734,
          -0.8864713979932093,
          2.522648053868628,
          0.3988471776536287,
          -0.8193066916133336,
          1.2751941668554705,
          1.4790204898497652,
          -0.3709300876848959,
          -0.021879272164415563,
          -0.30560053170321566,
          -0.6720521996295945,
          -0.0060033168500770555,
          0.35411891722637795,
          0.7979360804667908,
          -0.48359730523517697,
          -0.48788548546255295,
          0.15448708372041858,
          -2.7976468731293345,
          0.44533596145335114,
          0.12542095138695503,
          -1.102394345226662,
          0.0140603693198146,
          0.7698469980388079,
          0.11918241992780275,
          0.489597576731724,
          -0.6281919193858125,
          0.4763839742736879,
          0.4099258605443432,
          1.194334696275126,
          -0.4187101734604863,
          -0.21909491656022842,
          0.4590934472587962,
          -0.4851853185283378,
          -1.9635241140213628,
          2.145692476994688,
          -0.8171861055250083,
          0.517850231203265,
          -0.42155398200079464,
          -1.2713025883891427,
          0.18006382930831236,
          -0.19554577431068737,
          -1.0342587692689686,
          -0.25756872478943277,
          0.16706908928494987,
          -0.1795719590928659,
          1.0526185177881506,
          -0.9730332193295291,
          -0.5036850619359083,
          -0.7485505822861623,
          0.5780140262465802,
          0.32287771035007246,
          -1.2060353443002094,
```

Random Sampling from a Population

From lecture, we know that **Simple Random Sampling (SRS)** has the following properties:

- Start with known list of *N* population units, and randomly select *n* units from the list
- Every unit has **equal probability of selection = *n/N***
- All possible samples of size *n* are equaly likely
- Estimates of means, proportions, and totals based on SRS are **UNBIASED** (meaning they are equal to the population values on average)

In [10]:
```
import random
import numpy as np
```

In [50]:
```
mu = 0

sigma = 1

Population = [random.normalvariate(mu, sigma) for _ in range(10000)]
```

In [56]:
```
SampleA = random.sample(Population, 500)
SampleB = random.sample(Population, 500)
```

In [57]:
```
np.mean(SampleA)
```

Out[57]: -0.09255958393489437

In [58]:
```
np.std(SampleA)
```

Out[58]: 0.9766605050917198

In [59]:
```
np.mean(SampleB)
```

Out[59]: -0.017574156683834715

In [61]:
```
np.std(SampleB)
```

Out[61]: 1.013465105914656

In [69]:
```
means = [np.mean(random.sample(Population, 1000)) for _ in range(100)]

np.mean(means)
```

Out[69]: 0.0031415824668387876

In [76]:
```
standarddevs = [np.std (random.sample(Population, 1000)) for _ in range (100)]
np.mean(standarddevs)
```

Out[76]: 1.0017229717328646