# Unit 7 Notebook: Introduction to Random Variables - Building Blocks for Inference

## 1. Goals of Unit 7

- Probability rules that work for any type of event
- What is a **random variable**?
- What are some types of random variable and what are their properties?
- How do we calculate probabilities involving a random variable?

## 2. General Probability Rules

**See Unit 7 slides (section 2)**

## 3. Random Variable Definitions and Main Types

**See Unit 7 slides (section 3)**

## 4. How to calculate the probability of events involving random variables.

**See Unit 7 slides (section 4)**

## 5. How to identify if a random variable "fits the definition" of a well-known random variable.

**See Unit 7 slides (section 5)**

## 6. Discrete Random Variables: Functions that Calculate Random Variable Probabilities

Fitting the Definition of a Random Variable: From the unit 7 slides (section 5), we determined that the random variable **X = number of times you flip a coin until you get a head** "fits the definition" of being a **geometric random variable** with p=0.5. Thus we say that **X** is a geometric random variable (ie. $X \sim Geom(p = 0.5)$).

Distribution Python Objects: Because **geometric random variables** are well-known and well-studied by the statistics community, the **scipy.stats** package in Python has a **geom** object which contains a series of related functions involving **geometric random variables**.

### 6.1. Probability Mass Functions

**See Unit 7 slides (section 6.1)**

.pmf() Functions: For instance, the **.pmf()** function which is associated with many **distribution scipy objects** calculates the probability that the corresponding random variable is equal to a given value (ie. $P(X = value)$).

For instance, the code below calculates $P(X = 1) = 0.5$, wehere $X$ is a geometric random variable with parameter $p = 0.5$ (ie. $X \sim Geom(p = 0.5)$).

In [1]:
```
from scipy.stats import geom
```

In [2]:
```
type(geom)
```

Out[2]:
```
scipy.stats._discrete_distns.geom_gen
```

In [3]:
```
geom.pmf(1,p=0.5)
```

Out[3]:
```
0.5
```

$P(X = 2) = 0.25$

In [4]:
```
geom.pmf(2,p=0.5)
```

Out[4]:
```
0.25
```

$P(X = 3) = 0.125$

In [5]:
```
geom.pmf(3,p=0.5)
```

Out[5]:
```
0.125
```

**These probability values match what we calculated by hand in the Unit 7 slides (section 4).**

### 6.2 Cumulative Distribution Functions

**See Unit 7 slides (section 6.2)**

.cdf() Functions: For instance, the **.cdf()** function which is associated with many **distribution scipy objects** (like **geom**) calculates the probability that the corresponding random variable is less than or equal to a given value (ie. $P(X \leq value)$).

For instance, the code below calculates $P(X \leq 2) = 0.75$, wehere $X$ is a geometric random variable with parameter $p = 0.5$ (ie. $X \sim Geom(p = 0.5)$).

In [6]:
```
geom.cdf(2,p=0.5)
```

Out[6]:
```
0.75
```

**This probability value matches what we calculated by hand in the Unit 7 slides (section 6.2).**

## 7. Examples of how to randomly generate values for a random variable

## 7.1. When we *don't know* if the random variable "fits the definition" of a *well-known* random variable.

#### For now, we will pretend that we "don't know" that our random variable X is a geometric random variable.

Experiment Keep flipping a coin until you get a head. Observe the total number of flips before stopping (including the head).

Simulation of Experiment: Let's first design a simulation that will mimic this experiment ourselves ("from scratch"). We will simulate the act of flipping a coin until we get a head. After we flip a head, we will define **X** to be the total number of flips in the experiment.

Randomly Generating a Value for a Random Variable: Thus we can think of this **X** as a **randomly generated value for the random variable X=number of times you flip a coin until you get a head** from the slides.

#### 7.1.1 Simulating Flipping a Coin

This is like sampling repeatedly from the following data frame until we get an "H".

In [7]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

In [8]:
```
coin = pd.DataFrame({'side': ['T', 'H']}, index=[0,1])
coin
```

Out[8]:

| | side |
|---|---|
| **0** | T |
| **1** | H |

#### 7.1.2 Using a "while loop" to keep "flipping" until we get a head.

Here is some code using a **"while"** loop to keep flipping our simulated coin until we get 'heads'. Rerun the cell to see how the the count 'X' varies randomly.

The **.item()** function pulls the value from the generated 1 item Series, so we can check if it equals 'H' or not.

In [9]:
```
#This will be your randomly generate value for the random variable X
X = 0

#Set flop to anything but 'H' so we can "enter" the while loop
flip='T'

#Keep executing the code in this loop WHILE the 'flip' variable is not equal t
o 'H'
while flip != 'H':

    #Draw a random sample (of size n=1) from the population of flip outcomes
    flip = coin.sample(1)['side'].item()
    print('Flip:',flip)

    #Update the random variable to be one more head
    X = X + 1
    print('Current Value of X:',X)
    print('-----------------')

print('Final Value of Randomly Generated Value for X:',X)
```

```
Flip: T
Current Value of X: 1
-----------------
Flip: H
Current Value of X: 2
-----------------
Final Value of Randomly Generated Value for X: 2
```

### 7.2. When we *know* the random variable "fits the definition" of a *well-known* random variable.

Because we know that X is a geometric random variable, we can also use the **.rvs()** function to randomly generate values for the random variable. In this case, the **.rvs()** also performs a simulation of conducting a series of "independent trials" until we get a "success" (where the probability of success in any given trial is always p).

.rvs() Function: The **.rvs()** function randomly generates a series of values for a corresponding random variable.

Remember that for our coin flip random variable **X**, this $p = 0.5$ (where "success" is flipping a head).

#### One Randomly Generated Value for X

In [10]:
```
X=geom.rvs(p=0.5, size=1)
X
```

Out[10]:
```
array([1])
```

#### Ten Randomly Generated Values for X

In [11]:
```
X=geom.rvs(p=0.5, size=10)
X
```

Out[11]:
```
array([1, 2, 3, 4, 1, 1, 3, 1, 3, 1])
```

## 8. Continuous Random Variables: Functions that Calculate Random Variable Probabilities

### 8.1. Why do we not use probability mass functions (ie. P(Y=value)) for continuous random variables?

#### See Unit 7 slides (section 8.1).

### 8.2. Cumulative Density Functions (cdf) and Probability Density Functions (pdf)

#### See Unit 7 slides (section 8.2).

### 8.3. Properties of Cumulative Density Functions (cdf) and Probability Density Functions (pdf)

#### See Unit 7 slides (section 8.3

### 8.4. Calculating the probabilities of events involving random variables using pdf and cdf curves

#### See Unit 7 slides (section 8.4).

### 8.5 Calculating the probabilites of events involving well-known random variables in Python.

Suppose that after collecting data on the Youtube watching habits on a large sample of adults, researchers decided that the random variable X = the number of hours a randomly selected adult spends watching Youtube each week closely "fits the definition" of another well-known random variable called a truncated normal random variable.

A **truncated normal random variable** has four parameters that are associated with it:

- μ = mean of the random variable (had it not been truncated)
- σ = standard deviation of the random variable (had it not been truncated)
- a = lower bound of the random variable
- b = upper bound of the random variable

Suppose that the researchers specifically know the parameters associated with our X truncated random variable are μ=0, σ=2, a=0, and b=20.

We can now import the **truncnorm** object from **scipy.stats** to use a series of functions related to **truncated normal random variables**.

In [12]:
```
from scipy.stats import truncnorm
```

Most functions involving a **random variable** object (like **truncnorm**, **geom**, and others) require us to specify the corresponding random variable parameters inside of that function as well as potentially other information.

For a **truncated normal random variable** these parameters are:

- loc = mean of the random variable (had it not been truncated)
- scale = standard deviation of the random variable (had it not been truncated)
- a = lower bound of the random variable
- b = upper bound of the random variable

Thus we can use the **.cdf()** function for **truncnorm** below to calculate $P(X \leq 2) = 0.683$.

In [13]:
```
a, b, loc, scale = 0.0, 20, 0, 2
truncnorm.cdf(2, a=a, b=b, loc=loc, scale=scale)
```

Out[13]:
```
0.6826894921370859
```

Because the **.cdf()** only calculates areas to going to the left (ie. $P(X \leq value)$), we need to use the relationship of $P(X > value) = 1 - P(X \leq value)$.

$P(X \leq 2) = 1 - P(X < 2) = 1 - 0.683 = 0.317$.

In [14]:
```
a, b, loc, scale = 0.0, 20, 0, 2
1-truncnorm.cdf(2, a=a, b=b, loc=loc, scale=scale)
```

Out[14]:
```
0.31731050786291415
```

# 9. Calculating Summary Statistics of a Random Variable

**See Unit 7 slides (section 9).**

## 9.1 Calculating a Summary Statistic of a Random Variable - "by hand"

**See Unit 7 slides (section 9.1).**

## 9.2 Calculating a Summary Statistic of a Random Variable - in Python

**Example: X=number of flips until getting a head**

Remember $X \sim Geom(p = 0.5)$.

Using ScyPy functions, we compute the mean, the median, the standard deviation, and the proportion less than 2 hours for this population.

In [15]:
```
params = ['mean', 'median', 'std', 'prop']
pop = [geom.mean(p=0.5),
      geom.median(p=0.5),
      geom.std(p=0.5),
      geom.cdf(2, p=0.5)]
pd.DataFrame({'population': pop}, index=params)
```

Out[15]:

| | population |
|---|---|
| **mean** | 2.000000 |
| **median** | 1.000000 |
| **std** | 1.414214 |
| **prop** | 0.750000 |

**How does this compare with what we calculated by hand in the Unit 7 slides (section 9)?**

# 10. Coding: while loops

**See section 7.1 of this Jupyter notebook.**

---

STAT 207, Victoria Ellison and Douglas Simpson, University of Illinois at Urbana-Champaign

In [ ]: