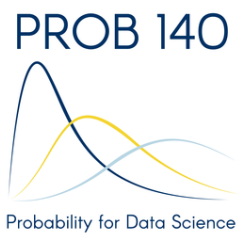


Probability for Data Science

UC Berkeley, Fall 2018

Ani Adhikari and Jim Pitman

In [91]:
```
# SETUP
from datascience import *
from prob140 import *
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import pylab
from scipy import stats
import ipywidgets as widgets
from ipywidgets import interact
from IPython.display import display
from matplotlib.ticker import FormatStrFormatter
```

In [92]:
```
def search(x_limits, cdf, u):
    """
    Runs a binary search to find the inverse cdf.
    """
    # Handle possible asymptotes.
    if cdf(x_limits[0]) > u:
        return x_limits[0]
    if cdf(x_limits[1]) < u:
        return x_limits[1]

    mid = (x_limits[0] + x_limits[1])/2
    diff = u - cdf(mid)
    if np.abs(diff) < 0.01:
        return mid
    if diff < 0:
        return search((x_limits[0], mid), cdf, u)
    return search((mid, x_limits[1]), cdf, u)

def plot_axes(cdf_table):
    values = cdf_table.column(cdf_table.num_columns - 1)
    cum = list(np.cumsum(values))
    cur_axes = plt.gca()
    cur_axes.axes.get_xaxis().set_visible(False)
    plt.yticks([0] + cum)
    plt.ylim(-0.1, 1.1)
    plt.plot([0,0], [0,1], color="k", lw=3)
    plt.xlim(-0.02, 1)
    plt.scatter([0]*(len(cum) + 1),
                [0] + cum, s=55, color="k")

def plot_discrete_cdf(cdf_table, u=None):
    """
    Plots the cdf of a discrete distribution.

    Parameters
    ----------
    cdf_table : Table
        Table of cdf values.
    u : float
        Value from (0, 1) to plot inverse cdf of.
    """
    values = cdf_table.column(0)
```

```
    values = np.append(values[0] - 2, values)

    cum = cdf_table.column(cdf_table.num_columns - 1)
    cum = np.append(0, np.cumsum(cum))

    for i in range(len(values) - 1):
        plt.plot([values[i], values[i+1]], [cum[i], cum[i]],
                 color="darkblue")
        plt.plot([values[i+1], values[i+1]], [cum[i], cum[i+1]],
                 ls="--", color="darkblue" )
    plt.scatter(values, cum, s=50, color="darkblue")

    plt.plot([values[-1], values[-1] + 2], [1,1],
             color="darkblue")

    plt.xlim(values[0], values[-1] + 2)
    plt.ylim(-0.1, 1.1)
    plt.xlabel('$x$')
    plt.ylabel('CDF at $x$')
    plt.title('Graph of CDF');

    if u != None:
        for i in range(len(values)):
            if u <= cum[i]:
                index = values[i]
                break
        height = u

        plt.plot([values[0], (index+values[0])/2], [height, height],
                 marker='>', color='red', lw=1)
        plt.plot([(index+values[0])/2, index], [height, height],
                 color='red', lw=1)
        plt.plot([index, index], [height, height/2], marker="v",
                 color="red", lw=1)
        plt.plot([index, index], [0, height/2], color="red", lw=1)

def plot_continuous_cdf(x_limits, cdf, u=None):
    """
    Plots the cdf of a continuous distribution.
    """
    x = np.linspace(*x_limits, 100)
    cdf_values = list(map(cdf, x))
    plt.plot(x, cdf_values, color="darkblue")
```

```
    plt.xlabel('$x$')
    plt.ylabel('CDF at $x$')
    plt.title('Graph of CDF');

    if not u is None:
        index = search(x_limits, cdf, u)
        height = u

        plt.plot([x_limits[0], (index+x_limits[0])/2],
                 [height, height], marker='>', color='red', lw=1)
        plt.plot([(index+x_limits[0])/2, index],
                 [height, height], color='red', lw=1)
        plt.plot([index, index], [height, height/2],
                 marker="v", color="red", lw=1)
        plt.plot([index, index], [0, height/2], color="red", lw=1)


    plt.xlim(*x_limits)

def unit_interval_to_discrete(cdf_table):
    uniform_slider = widgets.FloatSlider(
        value=0.5,min=0,max=1,step=0.02, description='u')
    @interact(u = uniform_slider)
    def plot(u):
        plot_discrete_cdf(cdf_table, u)


def unit_interval_to_continuous(x_limits, cdf):
    uniform_slider2 = widgets.FloatSlider(
        value=0.5, min=0,max=1,step=0.02, description='u')

    @interact(u = uniform_slider2)
    def plot(u):
        if (cdf(u) > x_limits[1] or cdf(u) < x_limits[0]):
            plot_continuous_cdf(x_limits, cdf)
        else:
            plot_continuous_cdf(x_limits, cdf, u)

def override_hist(*args, **kwargs):
    """
    This cleans up some unfortunate floating point precision
    bugs in the datascience library
    """
```

```
    #kwargs['edgecolor'] = 'w'
    Table.hist2(*args, **kwargs)
    ax = plt.gca()
    ticks = ax.get_xticks()
    if np.any(np.array(ticks) != np.rint(ticks)):
        ax.xaxis.set_major_formatter(FormatStrFormatter('%.2f'))

if not hasattr(Table, 'hist2'):
    Table.hist2 = Table.hist

Table.hist = override_hist
```

In [93]:
```
def plot_radial_distances():
    n = 500
    sampled_thetas = np.random.uniform(0, 2 * np.pi, n)
    sampled_radii = np.sqrt(np.random.uniform(0, 1, n))
    x = sampled_radii * np.cos(sampled_thetas)
    y = sampled_radii * np.sin(sampled_thetas)
    theta = np.linspace(0, 2 * np.pi, 100)
    uniform_slider = widgets.IntSlider(
        value=10,
        min=1,
        max=n,
        step=1,
        description='n'
    )
    @interact(i=uniform_slider)
    def plot(i):
        fig = plt.figure(figsize=(10, 5))
        ax1 = fig.add_subplot(1, 2, 1)
        ax1.plot(np.cos(theta), np.sin(theta), color='gold')
        ax1.scatter(x[:i], y[:i], color='darkblue', s=10)
        ax1.set_aspect('equal')
        ax1.set_title('Simulated Points')
        ax1.set_xticks(np.arange(-1, 1.5, 0.5))
        ax1.set_yticks(np.arange(-1, 1.5, 0.5))
        ax1.set_xlabel('x')
        ax1.set_ylabel('y')
        ax2 = fig.add_subplot(1, 2, 2)
        ax2.set_title('Empirical Histogram of Radius')
        ax2.hist(sampled_radii[:i], bins=np.linspace(0, 1, 25), density=True, color='darkblue')
        ax2.set_ylabel('Percent per Unit')
        plt.yticks(ax2.get_yticks(), ax2.get_yticks() * 100);
        ax2.set_xlabel('r')
        plt.subplots_adjust(wspace=0.5)
```

Lab Resources

- `prob140` Library Documentation (http://prob140.org/prob140/)
- Data 8 Python Reference (http://data8.org/fa18/python-reference.html)
- Prob 140 Code Reference Sheet (http://prob140.org/assets/prob140_code_reference.pdf)

- `scipy.stats` Documentation (https://docs.scipy.org/doc/scipy/reference/stats.html)

Lab 8: Simulation and the CDF

Simulation helps us understand properties of random variables. For example, earlier in the term you saw `simulate_path` for simulating Markov Chains; this was helpful for understanding transition behavior and reversibility. The `Table` method `sample` simulates drawing uniformly at random from the rows of a table; in Data 8, you used it understand the bootstrap. Simulation is also important because properties observed in simulations can lead to the development of new results.

In this lab you will simulate random variables with specified distributions.

What you will learn:

- How to use `SciPy` for simulation
- How to construct and read graphs of cumulative distribution functions (cdfs)
- How being able to simulate just one distribution allows us to simulate all others

#newpage

Part 1: Simulation in SciPy

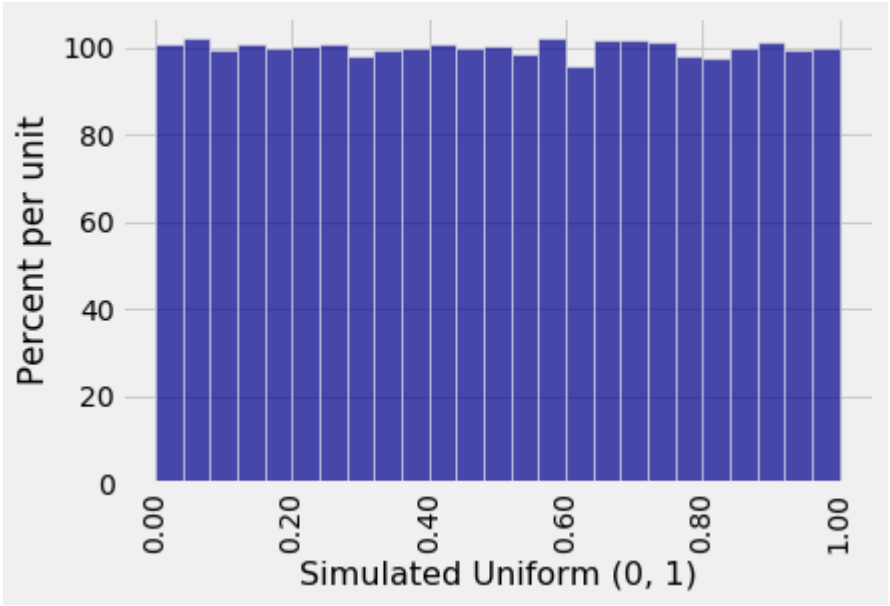
The `stats` module of `SciPy` is familiar to you by now. For any of the well known distributions, you can use `stats` to simulate values of a random variable with that distribution. The general call is `stats.distribution_name.rvs(size = n)` where `rvs` stands for "random variates" and `n` is the number of independent replications you want.

Every statistical system has conventions for how to specify the parameters of a distribution. In this lab we will tell you the specifications for a few distributions. Later you will be able to see a general pattern in the specifications.

1a) $Uniform(0, 1)$

The call is straightforward: `stats.uniform.rvs(0, 1, size=n)` . Complete the cell below to draw the histogram of 100,000 simulated values of a random variable that has the $Uniform(0, 1)$ distribution. The `hist` option `bins=25` results in 25 equal bins.

In [94]:
```
sim_uniform = stats.uniform.rvs(0, 1, size = 100000)
sim_uniform_tbl = Table().with_column(
    'Simulated Uniform (0, 1)', sim_uniform
)
sim_uniform_tbl.hist(bins=25)
```



1b) Reading the Scales of the Histogram

The unit on the horizontal axis is any unit of length; you can think of it as centimeters if you want, but we will just refer to it as the "unit". Fill in the blanks below and provide units where appropriate. Some units have been provided for you.

(i) The width of each bin is _____ units.

(ii) The height of each bar is approximately _____ per _____.

(iii) Histograms represent percents by _____, so the answers to (i) and (ii) imply that the percent of simulated values in each bin is approximately _____%.

(iv) Let the random variable U have the $Uniform(0, 1)$ distribution, and let B be any bin of the histogram. The answer to (i) implies that $P(U \in B) =$ _____%.

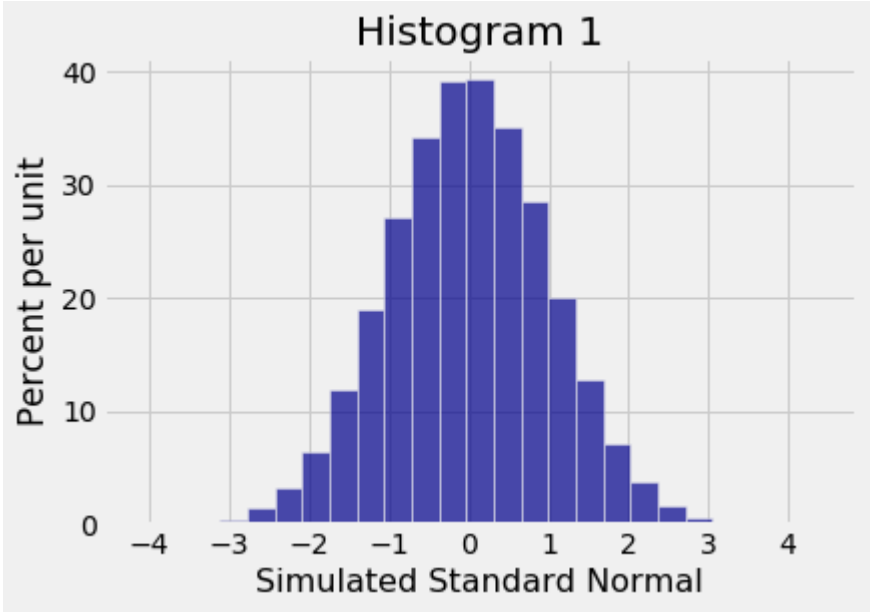
(v) If instead of `bins=25` we had used `bins=20` as the option to `hist`, then the answer to (iv) would have been _____%

(i) 0.04 (ii) 100 percent per unit (iii) area, 4 (iv) 4 (v) 5

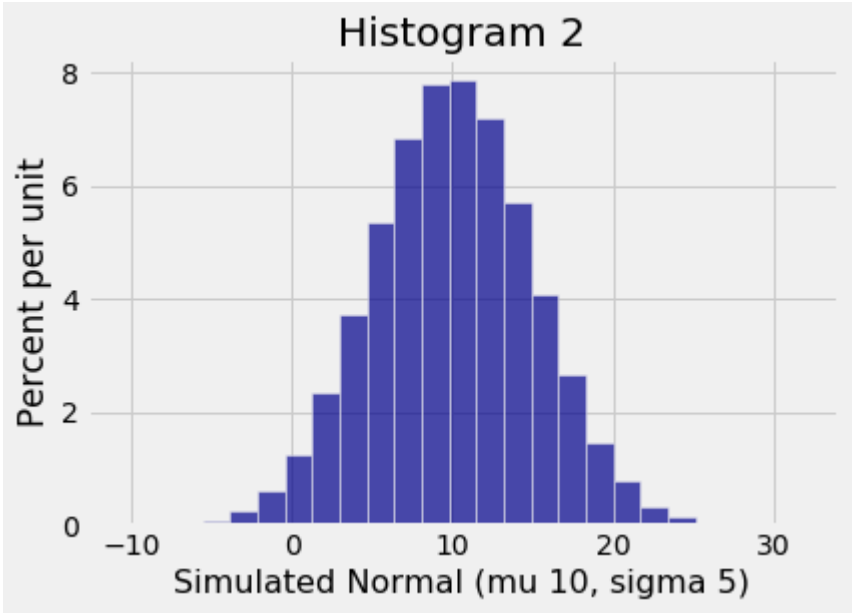
1c) Normal

To simulate normally distributed random variables, use `stats.norm.rvs(mean, sd, size=n)`. Complete the cells below to draw the empirical histograms of 100,000 simulated values of a standard normal variable and then 100,000 simulated values of a normal variable that has mean 10 and SD 5. Use the labels `Simulated Standard Normal` and `Simulated Normal (mu 10, sigma 5)` for the appropriate columns. You are welcome to use two different tables in the two cells, but use 25 bins in each histogram.

In [95]:
```
sim_std_norm = stats.norm.rvs(0, 1, size = 100000)
sim_std_norm_tbl = Table().with_column('Simulated Standard Normal', sim_std_norm)
sim_std_norm_tbl.hist(bins = 25)
plt.xticks(np.arange(-4, 4.1))
plt.title('Histogram 1');
```



In [96]:
```
sim_norm = stats.norm.rvs(10, 5, size = 100000)
sim_norm_tbl = Table().with_column('Simulated Normal (mu 10, sigma 5)', sim_norm)
sim_norm_tbl.hist(bins = 25)
plt.title('Histogram 2');
```



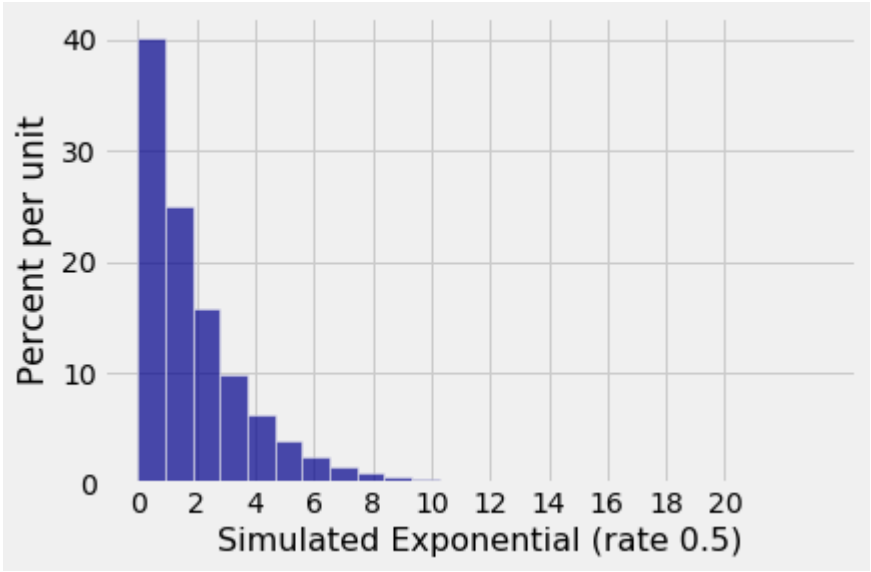
Compare the numbers on the horizontal axes of the two histograms, and fill in the blank.

The value **-1** on the horizontal axis of Histogram 1 is the same as the value 5 on the horizontal axis of Histogram 2 expressed in **standard** units.

1d) Exponential

The exponential distribution has two common parametrizations. One is the rate λ, which is what we use in Prob 140. The other is the mean $\frac{1}{\lambda}$. The mean is called the `scale` parameter in `stats`. Run the cell below to simulate 100,000 values of a random variable that has the exponential distribution with rate 0.5.

In [97]:
```
sim_expon = stats.expon.rvs(scale=1/0.5, size=100000)
sim_expon_tbl = Table().with_column(
    'Simulated Exponential (rate 0.5)', sim_expon
)
sim_expon_tbl.hist(bins=25)
plt.xticks(np.arange(0, 21, 2));
```



Find the average of the simulated values and check that it is consistent with the rate.

In [98]:
```
np.mean(sim_expon) # Yes it is consistent with 1/0.5 = 2
```

Out[98]:
```
1.9971716336478547
```

#newpage

Part 2. The Idea

How are all these random numbers generated? In the rest of the lab we will develop the method that underlies all the simulations above, by considering examples of increasing complexity.

Our starting point is a distribution on just four values.

Suppose X has the distribution `dist_X` below.

In [99]:
```
vals_X = make_array(-2, 1, 4, 7)
probs_X = make_array(0.3, 0.1, 0.2,0.4)

dist_X = Table().values(vals_X).probability(probs_X)
dist_X
```

Out[99]:

Value	Probability
-2	0.3
1	0.1
4	0.2
7	0.4

Our goal is to simulate one value of X. That is, we want to come up with a process that returns one of the four possible values of X with the right probabilities.

2a) A Vertical Unit Interval

The graphic below shows the probabilities in `dist_X` stacked vertically as in Lab 4. From the bottom to the top, therefore, you have the unit interval.

Now imagine throwing a dart at the unit interval. That is, let U be a random variable that has the $Uniform$ distribution on $(0, 1)$, and suppose you mark the value of U on the unit interval shown in the graph.

In [100]:
```
plot_axes(dist_X)
```

Find the following probabilities and see how they are related to the distribution of X.

(i) $P(U \leq 0.3)$

(ii) $P(0.3 < U \leq 0.4)$

(iii) $P(0.4 < U \leq 0.6)$

(iv) $P(0.6 < U \leq 1)$

i. 0.3

ii. 0.1

iii. 0.2

iv. 0.4

2b) Idea for Simulating X

Starting with a $Uniform(0, 1)$ random variable U, propose a method of generating a value of X.

Your method should take U as its input and return one of the four possible values as output, in such a way that for each $i = -2, 1, 4, 7$, the chance of returning the value i is $P(X = i)$.

Just describe your method in words. No formula or code is needed.

Generate a value of U. Find out which of the four intervals does it belong to : $(U \leq 0.3)$, $(0.3 < U \leq 0.4)$, $(0.4 < U \leq 0.6)$ or $(0.6 < U \leq 1)$. Based on that X will take its value. So for example if we find U falls in the interval $(0.3 < U \leq 0.4)$ then we will generate X as 1 with probability 0.1

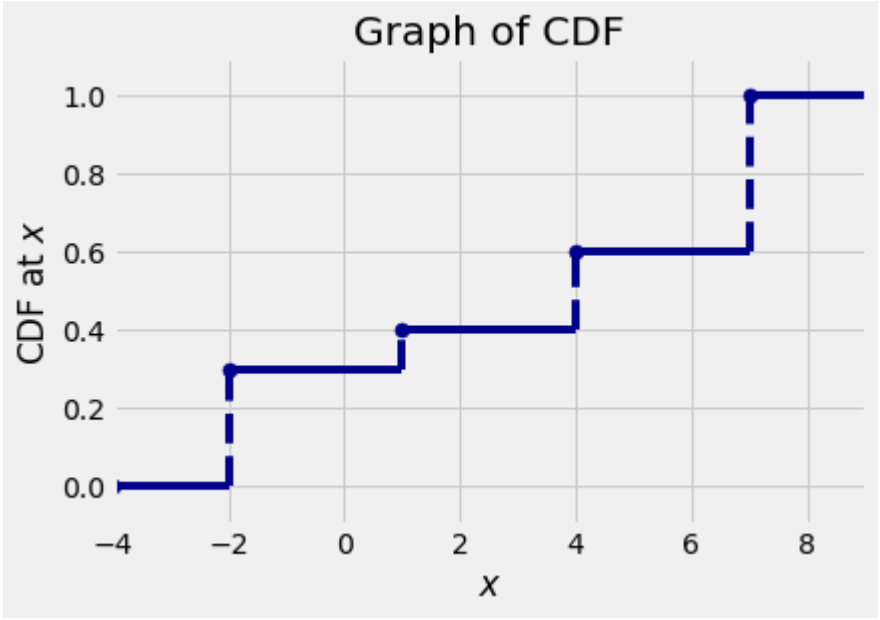
#newpage

Part 3. Visualizing the Idea

The method `plot_discrete_cdf` takes a distribution as its argument and plots a graph of the cdf.

Run the cell below to get a graph of the cdf of the random variable X in Part 1.

In [101]:
```
plot_discrete_cdf(dist_X)
```



3a) Reading the Graph

Let F_X be the cdf of X. What is the definition of $F_X(2.57)$? Does the graph show the correct value for $F_X(2.57)$?

$F_X(2.57) = P(X \leq 2.57)$. Yes the graph shows the correct value of $F_X(2.57)$ which is 0.4

3b) Jumps

At what points x does the graph have a jump? For each point x at which there is a jump, find the size of the jump in terms of the distribution of X.

At -2, 1, 4, and 7 the graph has jumps. Size of the jump at $x = -2$ is 0.3 which is equal to $P(X = -2)$. Size of the jump at $x = 1$ is 0.1 which is equal to $P(X = 1)$. Size of the jump at $x = 4$ is 0.2 which is equal to $P(X = 4)$. Size of the jump at $x = 7$ is 0.4 which is equal to $P(X = 7)$

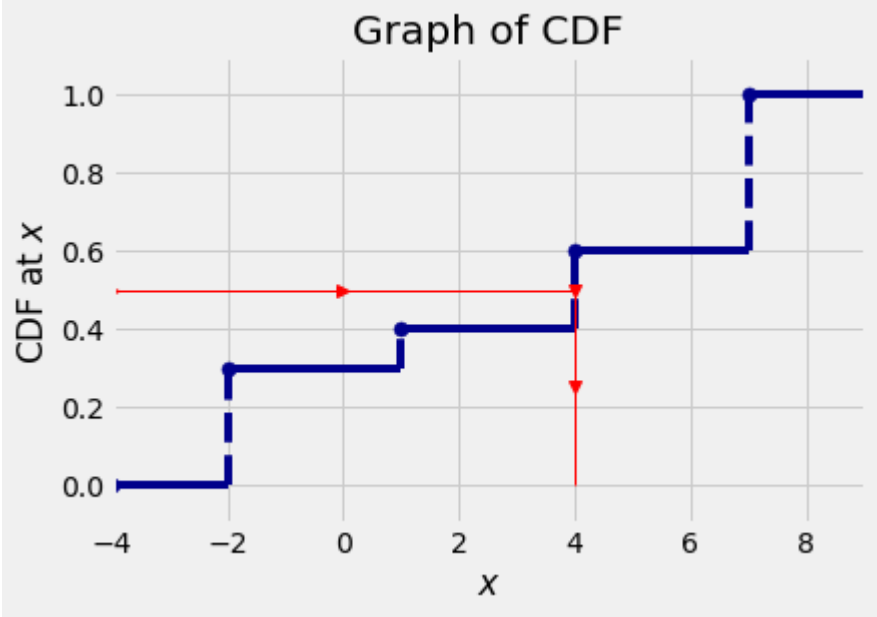
3c) From the Unit Interval to Values of X

The function `unit_interval_to_discrete` takes a distribution as its argument and displays an animation of a method that takes a number on the unit interval and returns one value of a random variable that has the given distribution.

Run the cell below. Move the slider around and see how the returned value changes **depending on the starting value** in the unit interval. How is the method that is being displayed related to the one you proposed in Part 1?

In [102]:
```
unit_interval_to_discrete(dist_X)
```

u 0.50

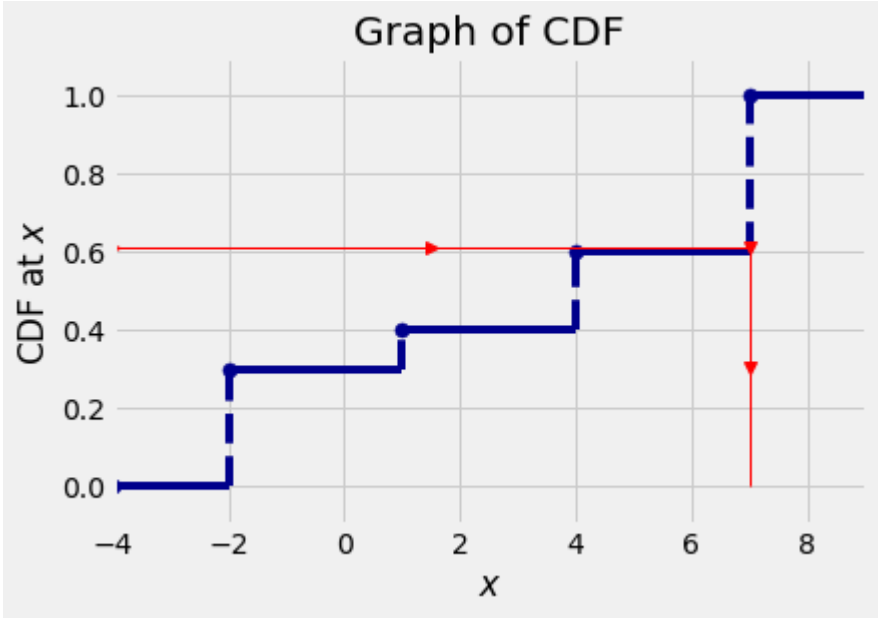


3d) A Random Starting Point

The method `plot_discrete_cdf` that you used earlier also takes a second argument which is a number between 0 and 1.

Complete the cell below so that the second argument is picked uniformly at random from (0, 1). Run the cell a few times. How is it related to the method you proposed in Part 1 for generating a value of X?

In [103]:
```
plot_discrete_cdf(dist_X, stats.uniform.rvs(0, 1, size = 1))
```



It's very closely related to the method proposed in Part 1. In Part 1, we decided the value of X based on whichever interval the value of U was in. Here, similar to part 1, we apply a function to uniformly randomly generated number (between 0 and 1) so that it maps to a value of X.

#newpage

Part 4. Extension to Continuous Distributions

Now suppose you want to generate a random variable that has a specified continuous distribution. Let's start with the exponential (λ) distribution.

4a) [ON PAPER] Exponential CDF

Let T have the exponential distribution with rate λ. Let F_T be the cdf of T. Write the formula for F_T. Remember that the cdf is a function on the entire number line $(-\infty, \infty)$; make sure you specify the function on the whole line.

4b) Plotting the Exponential CDF

As a numerical example, let T be a random variable that has the exponential distribution with rate $\lambda = 0.5$, or equivalently, with expectation 2. Define a function `expon_mean2_cdf` that takes a numerical argument x and returns $F_T(x)$. Use `np.exp(y)` for e^y.

Make sure that for **all** numerical values of x your function returns the value you specified in **4a**.

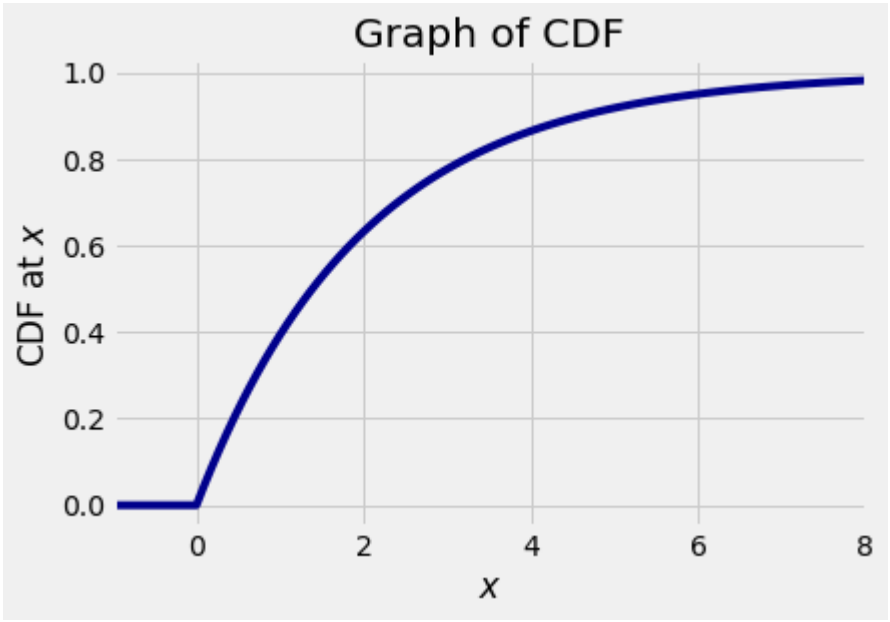
In [104]:
```
# don't use "lambda" as that means something else in Python
lamb = 0.5

def expon_mean2_cdf(x):
    if (x > 0):
        return 1 - np.exp(-lamb * x)
    else:
        return 0
```

The function `plot_continuous_cdf` plots the cdf of a continuous variable. The first two arguments:

- an interval (a, b) over which to draw the cdf
- the name of a cdf function that takes a numerical input and returns the value of the cdf at that input

Run the cell below to check that your function `expon_mean2_cdf` looks good.

In [105]:
```
plot_continuous_cdf((-1, 8), expon_mean2_cdf)
```



4c) Idea for Simulating an Exponential Random Variable

Suppose you are given one $Uniform(0, 1)$ random number and are asked to simulate T. Based on Part 3 of the lab, propose a method for doing this by using the graph above.

You don't have to prove that the method works. We'll do a formal proof in lecture. Just propose the method.

We apply a function that is inverse of cdf to the U to get value of X

4d) Visualizing the Idea

The animation in the cell below is analogous to the one in Part 3. Its arguments are:

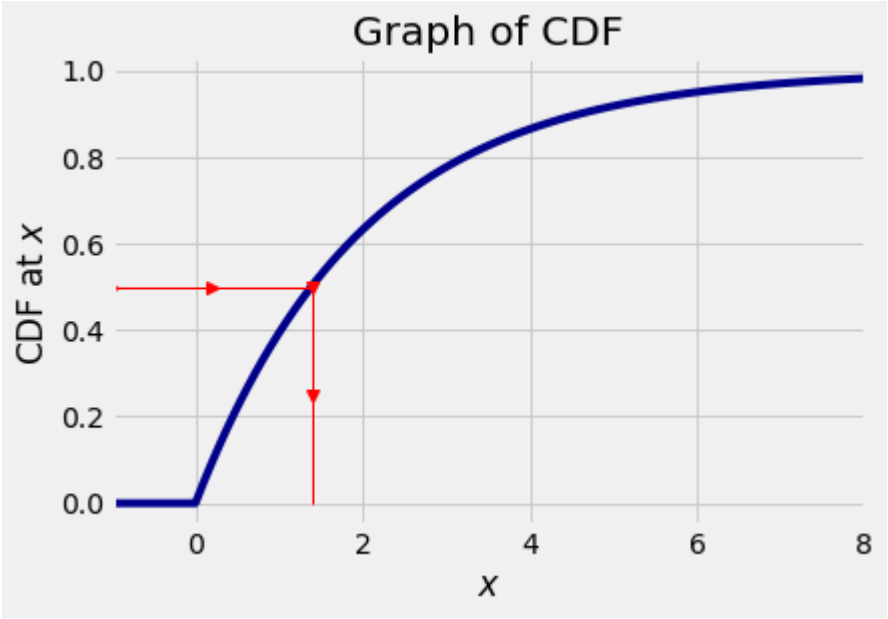
- a plotting interval
- the name of a continuous cdf function

The output demonstrates a method for picking a number on the positive real line starting with a value on the unit interval that forms the vertical axis.

Run the cell and move the slider around to see how the returned value changes depending on the starting value on the vertical axis.

In [106]:
```
unit_interval_to_continuous((-1, 8), expon_mean2_cdf)
```

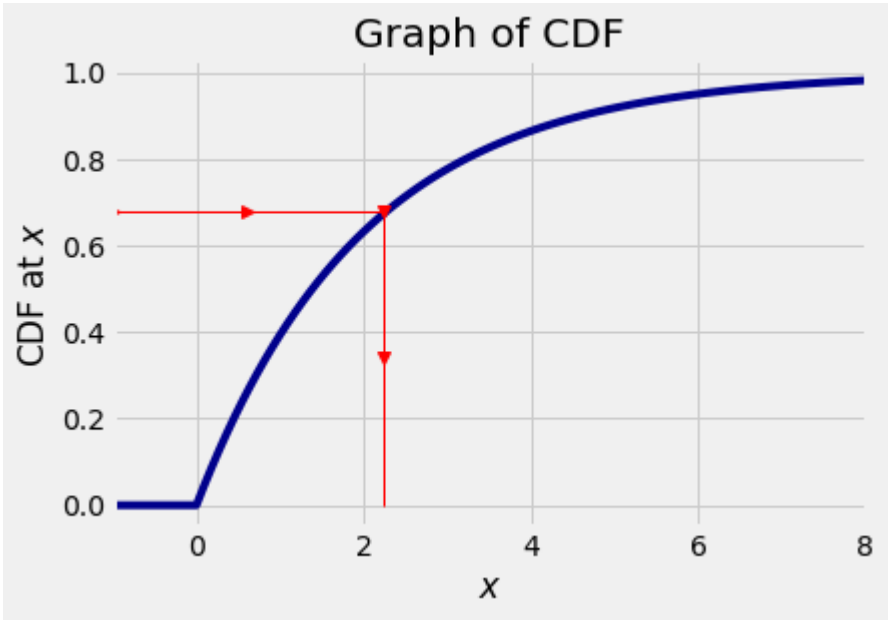
u 0.50



The method `plot_continuous_cdf` takes an optional third argument that is a number between 0 and 1.

Complete the cell below so that the third argument is a random number picked uniformly from (0, 1). Run the cell a few times. How is the output related to the method you proposed in **4c** for generating a value of T?

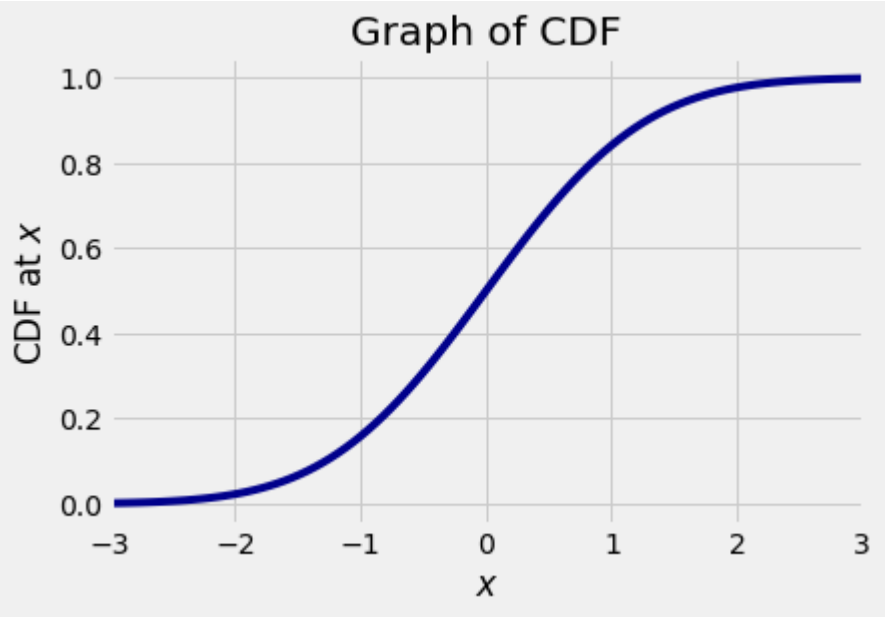
In [107]:
```
plot_continuous_cdf((-1, 8), expon_mean2_cdf, stats.uniform.rvs(0, 1, size = 1))
```



4e) Simulating a Standard Normal Random Variable

Complete the cell below so that the random value pointed to by the downwards arrow has the standard normal distribution. Remember from Section 14.3 (http://prob140.org/textbook/chapters/Chapter_14/03_Central_Limit_Theorem) of the textbook that the `stats` module already has a function that takes a numerical input and returns the value of the standard normal cdf at the input. You don't need to define a new one.

In [108]:
```
plot_continuous_cdf((-3, 3), stats.norm.cdf)
```



4f) [ON PAPER] The General Method

Let F be any continuous increasing cdf. That is, suppose F has no jumps and no flat bits.

Suppose you are trying to create a random variable X that has cdf F, and suppose that all you have is F and a number picked uniformly on $(0, 1)$.

(i) **Fill in the blank:** Let U be a uniform $(0, 1)$ random variable. To construct a random variable $X = g(U)$ so that X has the cdf F, take $g =$ _____.

(ii) **Fill in the blank:** Let U be a uniform $(0, 1)$ random variable. For the function g defined by

$$g(u) = \_\_\_\_\_, \quad 0 < u < 1$$

the random variable $X = g(U)$ has the exponential (λ) distribution.

[Note: If F is a discrete cdf then the function g is complicated to write out formally, so we're not asking you to do that. The practical description of the method of simulation is in Parts 1 and 2.]

#newpage

Part 5. Empirical Verification that the Method Works

a) The Initial Values

Create a table that is called `sim` for simulation and consists of one column called `Uniform` that contains the values of 100,000 i.i.d. $Uniform(0, 1)$ random variables.

In [109]:
```
N = 100000
u = stats.uniform.rvs(0, 1, size = 100000)
sim = Table().with_column("Uniform", u)
sim
```

Out[109]:

Uniform
0.721201
0.956558
0.574768
0.696073
0.485887
0.0779093
0.328952
0.0740608
0.278105
0.9805

... (99990 rows omitted)

b) Tranformation to Exponential

Use **4f** and the values in the column `Uniform` to create an array of values that have the exponential distribution with rate 0.5. This is what is going on "under the hood" in `stats.expon.rvs` .

Do not simulate new random numbers, as you will lose the connection with the values in `Uniform` . Use `np.log(y)` for $\log(y)$.

Augment `sim` with a column containing the new array.

In [110]:
```
def uniform_to_exponential_mean2(u):
    return -(2 * np.log(1-u))

exponential_mean2 = sim.apply(uniform_to_exponential_mean2, 'Uniform')
sim = sim.with_column('Sim. Exponential (rate 0.5)', exponential_mean2)
sim
```

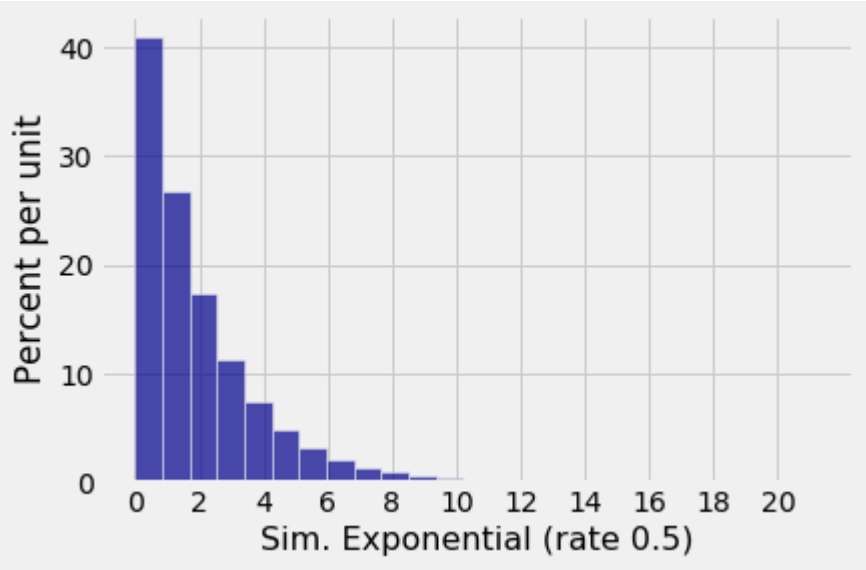
Out[110]:

Uniform	Sim. Exponential (rate 0.5)
0.721201	2.55453
0.956558	6.27267
0.574768	1.71024
0.696073	2.38193
0.485887	1.33063
0.0779093	0.162223
0.328952	0.797828
0.0740608	0.153893
0.278105	0.651751
0.9805	7.87471

... (99990 rows omitted)

Run the cell below and compare with the histogram in 1(d) to confirm that your calculation is correct.

In [111]:
```
sim.hist('Sim. Exponential (rate 0.5)', bins=25)
plt.xticks(np.arange(0, 21, 2));
```



5c) Transformation to Standard Normal

Let Z be standard normal. As you know, `stats.norm.cdf(z)` evaluates to $\Phi(z)$, the value of the standard normal cdf at z.

In Section 14.5 (http://prob140.org/textbook/chapters/Chapter_14/05_Confidence_Intervals) you also saw that the "percent point function" `stats.norm.ppf` is such that if `stats.norm.cdf(z)` evaluates to p, then `stats.norm.ppf(p)` evaluates to z.

That is, the percent point function at p is the value z such that $\Phi(z) = p$.

Use **4f** and the values in the column `Uniform` to create an array of values that have the standard normal distribution. Augment `sim` with a column containing the new array.

Please be patient. The code might take a while to run.

In [112]:
```
def norm_ppf(k):
    """Optional helper function."""
    return 0

standard_normal = sim.apply(stats.norm.ppf, 'Uniform')
sim = sim.with_column('Sim. Standard Normal', standard_normal)
sim
```

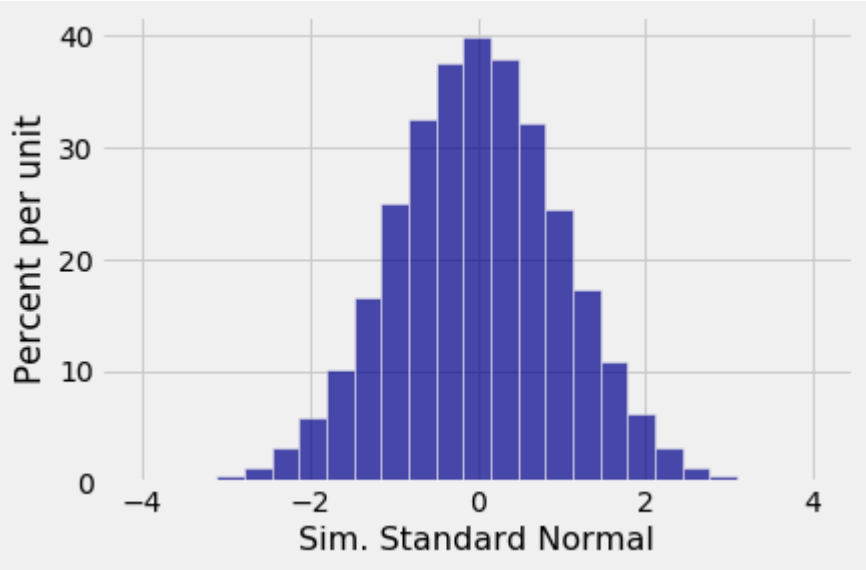
Out[112]:

Uniform	Sim. Exponential (rate 0.5)	Sim. Standard Normal
0.721201	2.55453	0.586413
0.956558	6.27267	1.71207
0.574768	1.71024	0.188527
0.696073	2.38193	0.513138
0.485887	1.33063	-0.035383
0.0779093	0.162223	-1.41928
0.328952	0.797828	-0.44281
0.0740608	0.153893	-1.4462
0.278105	0.651751	-0.58848
0.9805	7.87471	2.06419

... (99990 rows omitted)

Run the cell below to check that your calculation is correct.

In [113]:
```
sim.hist('Sim. Standard Normal', bins=25)
```



5d) Another Normal

Now augment `sim` with a column whose contents are the values in `Sim. Standard Normal` transformed so that they have the normal distribution with $\mu = 10$ and $\sigma = 5$. You don't need `apply` for this one.

In [114]:
```
z = sim.column('Sim. Standard Normal')
sim = sim.with_column('Sim. Normal (Mu=10, Sigma=5)', (z * 5) + 10)
sim
```

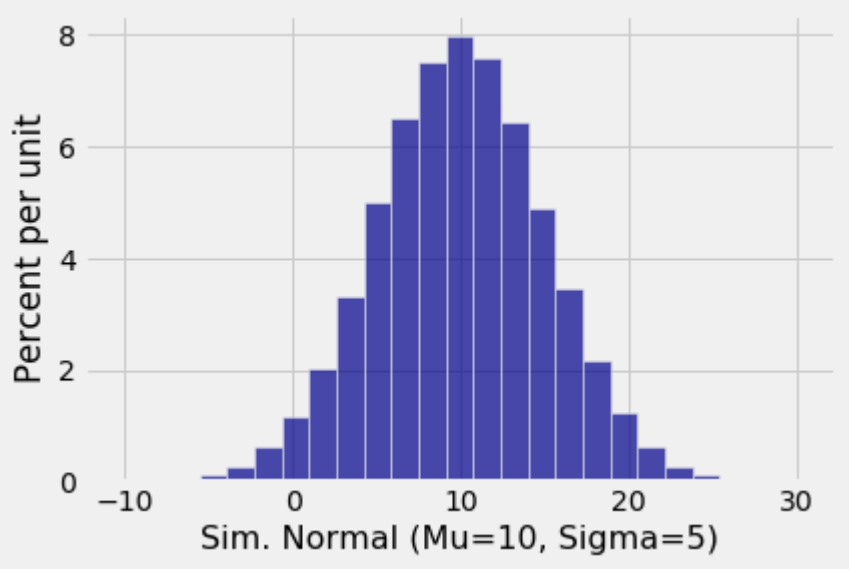
Out[114]:

Uniform	Sim. Exponential (rate 0.5)	Sim. Standard Normal	Sim. Normal (Mu=10, Sigma=5)
0.721201	2.55453	0.586413	12.9321
0.956558	6.27267	1.71207	18.5604
0.574768	1.71024	0.188527	10.9426
0.696073	2.38193	0.513138	12.5657
0.485887	1.33063	-0.035383	9.82308
0.0779093	0.162223	-1.41928	2.90362
0.328952	0.797828	-0.44281	7.78595
0.0740608	0.153893	-1.4462	2.76901
0.278105	0.651751	-0.58848	7.0576
0.9805	7.87471	2.06419	20.321

... (99990 rows omitted)

Run the cell below to confirm that your calculations are correct.

In [115]:
```
sim.hist('Sim. Normal (Mu=10, Sigma=5)', bins=25)
```



At this point, go back and look through Part 5. Notice that the only time you generated random numbers was when you simulated 100,000 uniform (0, 1) values. All the other variables were deterministic transformations of the uniform variable.

#newpage

Part 6. Radial Distance

You can apply the general method developed above to simulate values of any continuous random variable. Here is an example.

Consider a point (X, Y) picked uniformly on the unit disc $\{(x, y) : x^2 + y^2 \leq 1\}$. That's the disc with radius 1 centered at the origin (0, 0).

Let R be the distance between the point (X, Y) and the center $(0, 0)$.

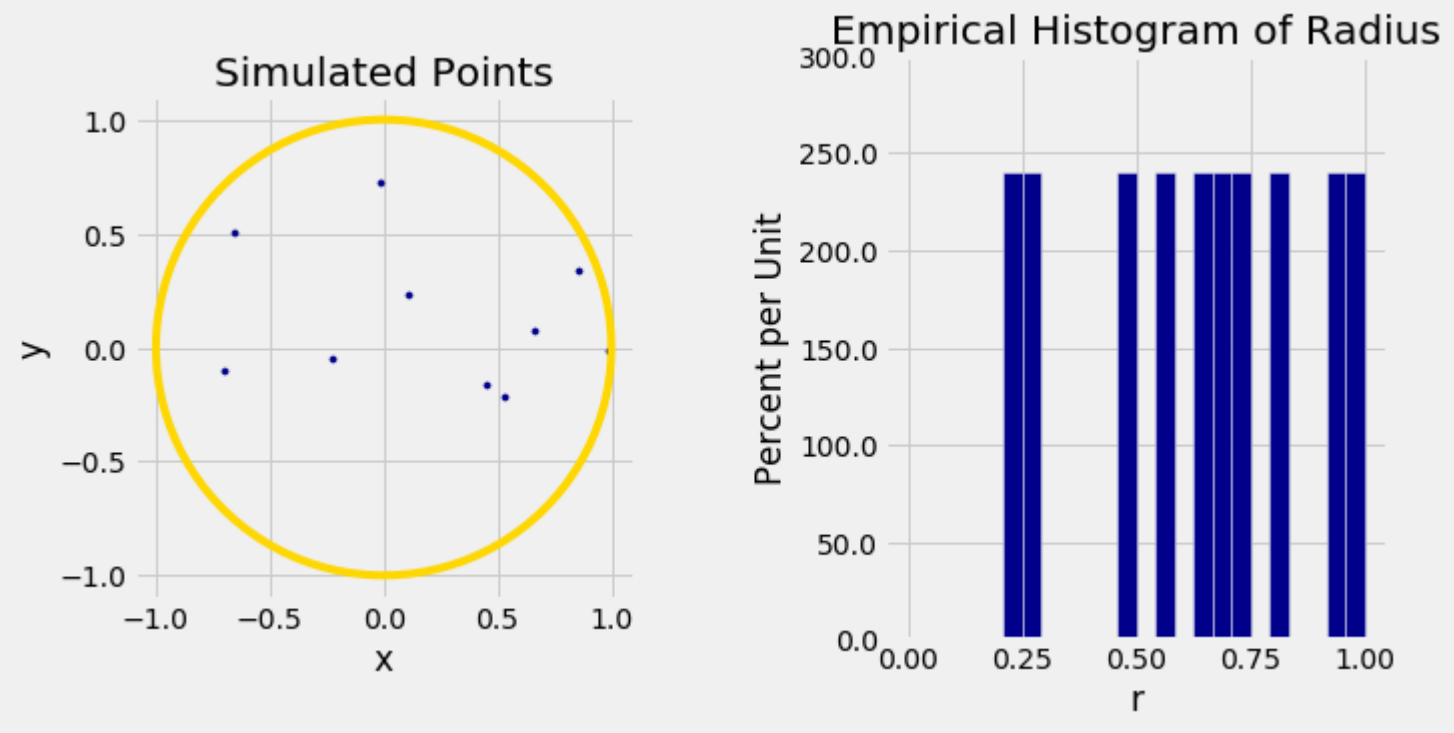
The point (X, Y) is random, so the radial distance R is random as well and has a density.

6a) Visualization

Run the cell below. The figure on the left shows simulated i.i.d. copies of the point. On the right you have the empirical histogram of the simulated distances. Move the slider to increase the number of simulations.

In [116]:
```
plot_radial_distances()
```

n 10



6b) [ON PAPER] The CDF and Its Uses

(i) Find the possible values of R, and then find F_R, the cdf of R.

[Remember that $F_R(r)$ is the probability of an event. Draw the unit circle, and then shade the event that has probability $F_R(r)$.]

(ii) Find f_R, the density of R. Sketch (by hand, on your paper), a graph of the density.

(iii) Find a function g such that if U has the $Uniform(0, 1)$ distribution then the random variable $g(U)$ has the same distribution as R.

6c) Simulation

Augment the table `sim` from Part 5 with a column `Sim. Radial Distance` that contains 100,000 simulated values of R based on the simulated uniform variables in the `Uniform` column.

In [117]:
```
sim = sim.drop('Sim. Radial Distance', 'Sim. Radial Distance ')
```

In [118]:
```
r = sim.column('Uniform')
sim = sim.with_column('Sim. Radial Distance', np.sqrt(r))

sim
```

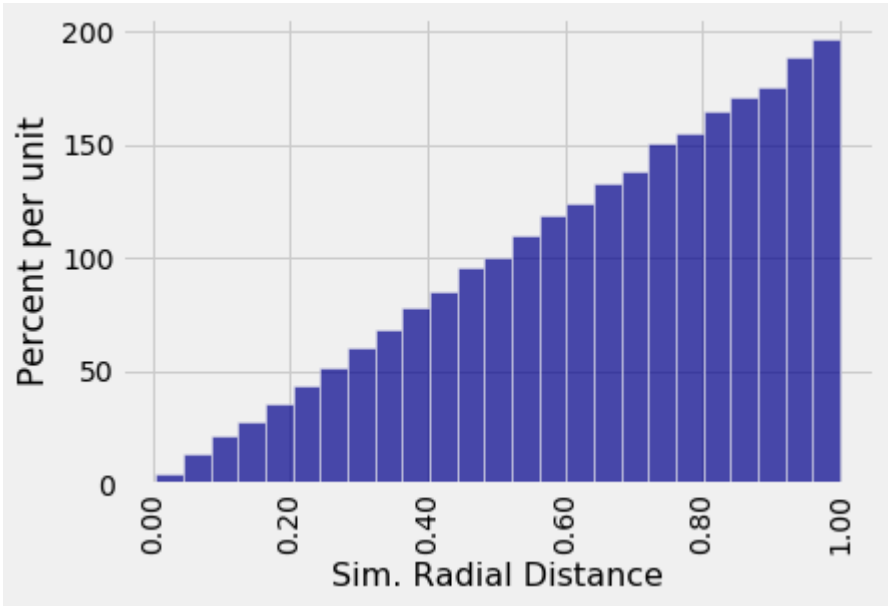
Out[118]:

Uniform	Sim. Exponential (rate 0.5)	Sim. Standard Normal	Sim. Normal (Mu=10, Sigma=5)	Sim. Radial Distance
0.721201	2.55453	0.586413	12.9321	0.849236
0.956558	6.27267	1.71207	18.5604	0.978038
0.574768	1.71024	0.188527	10.9426	0.758135
0.696073	2.38193	0.513138	12.5657	0.83431
0.485887	1.33063	-0.035383	9.82308	0.697056
0.0779093	0.162223	-1.41928	2.90362	0.279122
0.328952	0.797828	-0.44281	7.78595	0.573543
0.0740608	0.153893	-1.4462	2.76901	0.272141
0.278105	0.651751	-0.58848	7.0576	0.527357
0.9805	7.87471	2.06419	20.321	0.990202

... (99990 rows omitted)

Finally, run the cell below and compare with what you saw in **6a**.

In [119]:
```
sim.hist('Sim. Radial Distance', bins=25)
```



Conclusion

You have learned that:

- To simulate a random variable with a desired distribution, what you need is a uniform random number generator and the cdf of the desired distribution. You can then use the method of this lab to simulate the value.
- The only random numbers a statistical system needs are uniform on (0, 1). Random numbers from all other distributions follow by the method you have developed in this lab.
- Discrete cdfs consist of jumps and flat parts, at places that you have identified.
- If you have a standard normal variable you can easily transform it to become any other specified normal.

Since uniform $(0, 1)$ random numbers are central to all simulations, their quality is very important for the accuracy and reliability of simulations. Testing and assessing uniform random number generators is serious business, because random number generators don't really produce random numbers. They follow deterministic processes that produce results that have properties that resemble those of random numbers. That is why they are called Pseudo Random Number Generators or PRNGs. Python uses the Mersenne Twister (https://docs.python.org/3.6/library/random.html), one of the most tested and reliable PRNGs. SciPy uses the Mersenne Twister for

RandomState (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.RandomState.html#numpy.random.RandomState) and draws from a large number of discrete and continuous distributions. Take a look at the list on the RandomState page and see how many you can recognize.

Submission Instructions

1. **Save your notebook using File > Save and Checkpoint.**
2. Run the cell below to generate a pdf file.
3. Download the pdf file and confirm that none of your work is missing or cut off.
4. Submit the assignment to Lab08a and Lab08b on Gradescope.

Your written submission (Lab08a) should include:

- Part 4
- Part 6

Your code submission (Lab08b) should include:

- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

Logistics

1. Examine the generated pdf before uploading to make sure that it contains all of your work.
2. When submitting to Gradescope, select the pages of your upload corresponding to each question.
3. If you encounter any difficulties when submitting or exporting your assignment, please make a private Piazza post **before the deadline**.

In [120]:
```
import gsExport
gsExport.generateSubmission("Lab08.ipynb")
```

```
Processing Lab08.ipynb
Generated notebook and autograded
Attempting to compile LaTeX
Finished generating PDF
```

Download this and submit to gradescope! (Lab08_submission.pdf)

In [ ]:
```

```