## statistics

- we've already done some stats using the numpy library:
    - `import numpy as np`
    - `c = np.random.random(1000)` - what does this do?
        - returns 1000 *continuous* (float) values evenly distributed over the interval `[0, 1)`
        - this is called a continuous uniform random distribution
    - `d = np.random.randint(0, 10, 1000)` - what does this do?
        - returns 1000 *discrete* (integer) values evenly distributed over the interval `[0, 10)`
        - this is called a discrete uniform random distribution
    - we also already know how to check if the values we get from these functions really are uniformly distributed - how can we visualize this?

      ```
      import matplotlib.pyplot as plt
      f, ax = plt.subplots()
      ax.hist(c, bins=30)
      ```

        - how to choose `nbins`?
            - if `nbins` is too low, you can't capture enough of the variability of your data in the plotted distribution
            - if `nbins` is too high, you capture too much of the variability, get a very noisy distribution
            - guess and test, but a rule of thumb for continuous variables is `nbins = np.sqrt(n)`
            - even easier, use `bins='auto'` to automatically calculate reasonable bin size
    - plotting the distribution of your data (with a reasonable choice of `nbins`) is really important!
        - can reveal outliers, and maybe sources of error in the data collection
        - many stats tests make assumptions about how your data are distributed, and if your data don't satisfy those assumptions, you should use a different stats test
        - so, it's good to get into the habit of plotting distribs
    - in addition to uniform distrib, the other very common continuous distribution is the normal (Gaussian) distrib

      ```
      mu, sigma = 0, 1
      s = np.random.normal(loc=mu, scale=sigma, size=1000) # s for "sample"
      f, ax = plt.subplots()
      ax.hist(x, bins=30)
      ```

    - what if your data are bimodally distributed (having 2 peaks) like this?:

      ```
      s1 = np.random.normal(loc=0, scale=1, size=1000)
      s2 = np.random.normal(loc=5, scale=0.5, size=1000)
      ```

```
# confirm we got approximately what we asked for:
s1.mean() # approx 0
s1.std() # approx 1
s2.mean() # approx 5
s2.std() # approx 0.5
bimodal = np.concatenate([s1, s2])
f, ax = plt.subplots()
ax.hist(bimodal, bins='auto')
```

      - are `s.mean()` and `s.std()` meaningful in this case? no! they're poor descriptors of this bimodal distribution, but the only way to tell is to plot and inspect the distribution

  - can also plot the distribution of discrete valued data, but to get ideal bin locations and widths, need to be explicit and specify the edges of each bin:

```
s = np.random.randint(0, 10, 1000)
s.min() # check that we got what we asked for
s.max()
f, ax = plt.subplots()
edges = np.arange(0, 11) # bin edges, 0 to 10, inclusive, steps of 1
ax.hist(s, bins=edges)
```

      - for discrete values, best to use no more than one bin per possible value, as above, otherwise you'll end up with artificial gaps between discrete values:

```
f, ax = plt.subplots()
edges = np.arange(0, 10.5, 0.5) # bin edges, 0 to 10 inclusive, 0.5 steps
ax.hist(s, bins=edges) # notice the artificial gaps
```

  - matplotlib hist vs numpy hist:

      - to calculate histograms, we've been using `ax.hist()` or `plt.hist()` from matplotlib
      - sometimes you might want to calculate a histogram without plotting it
      - use `np.histogram()`
          - returns the count in each bin, and the bin edges
          - `n, edges = np.histogram(s, bins='auto')`
          - then you can programatically do things like find what the peak value is, and where it is:
              - `n.max()`, `n.argmax()`

- `scipy.stats`

  - numpy can generate random samples from different kinds of distributions, but `scipy.stats` has a lot more stats functionality
  - `import scipy.stats as stats`
  - `stats?` shows a big list of all the stats related objects and functions in `scipy.stats`
  - instead of just asking for a random sample of numbers from a particular kind of distribution, `scipy.stats` provides "random variables" as objects, which you can then not only sample, but also call their methods:

```
rv = stats.norm() # create a continuous normal random variable object
rv.mean() # returns exactly 0.0
rv.std() # returns exactly 1.0
rv = stats.norm(loc=5, scale=0.5)
rv.mean() # returns exactly 5
rv.std() # returns exactly 0.5
s = rv.rvs(1000) # sample 1000 random values from rv
f, ax = plt.subplots()
ax.hist(s, bins=30) # similar to what we got before from np.random.normal()
```

- note that each time you sample a random value, you get different values out:

```
ax.hist(rv.rvs(1000), bins=30) # each call adds a new sampling to the plot
ax.hist(rv.rvs(1000), bins=30)
ax.hist(rv.rvs(1000), bins=30)
```

- the benefit of using a random variable object is that it provides an exact representation of a particular type of distribution
- to access it analytically as a function of x, call the `.pdf()` method
  - `rv.pdf(x)` - PDF = probability density function, or more typically, just "distribution"
  - probability always has to sum to 1, so area under the curve == 1
  - let's plot the exact representation of the normal distribution over top of the normalized histogram of our 1000 sampled values from that distribution:

```
f, ax = plt.subplots()
ax.hist(s, bins=30, normed=True) # plot a normalized distrib, area == 1
x = np.arange(3, 7, 0.01) # evenly spaced x values from 3 to 7
y = rv.pdf(x) # exact distribution
ax.plot(x, y)
ax.set_xlabel('x')
ax.set_ylabel('probability')
ax.set_title('mu=5, sigma=0.5, n=1000')
f.canvas.set_window_title('sampled and exact distributions')
```

- stats tests:
  - you've collected a bunch of data, presumably sampled from some natural process
  - you plot the distribution of your data, and see that it's roughly normally distributed
  - how can you check if the mean of your data is significantly different from, say, 0?
    - do a stats test, which gives you p-value (probability) of null hypothesis
    - if p-value < some threshold (at least 0.05), null hypothesis is false, mean of your data is significantly different from 0
    - in this case, use a "1-sample t-test", `stats.ttest_1samp()`

```
rv = stats.norm(loc=2, scale=10) # mean is 2, std is 10
s = rv.rvs(50) # acquire small amount of data
f, ax = plt.subplots()
ax.hist(s, bins='auto') # does it look normal? barely
t, p = stats.ttest_1samp(s, 0) # p > 0.05, can't reject null hypothesis
```

    - having higher n, i.e. more data, gives you more statistical power, i.e. better able to detect a weak effect:

```
s = rv.rvs(500) # acquire more data from same source
f, ax = plt.subplots()
ax.hist(s, bins='auto') # does it look normal? yes
t, p = stats.ttest_1samp(s,  0) # p < 0.05, can reject null hypothesis
```

    - or, having a stronger effect allows you to get away with less data:

```
rv = stats.norm( loc=4, scale=5) # 2x the mean, 1/2 the std
s = rv.rvs(50) # acquire small amount of data
f, ax = plt.subplots()
ax.hist(s, bins='auto') # does it look normal? barely
t, p = stats.ttest_1samp(s,  0) # p < 0.05, can reject null hypothesis
```

- if you have two samples of data, e.g. control vs. treatment, are they significantly different?
- do a 2-sample t-test `stats.ttest_ind()`, safest is called "Welch's", which doesn't assume the two samples have equal variance (std squared)

```
s1 = stats.norm.rvs( loc=2, scale=2, size=200) # control
s2 = stats.norm.rvs( loc=3, scale=1, size=200) # treatment
f, ax = plt.subplots()
ax.hist(s1, bins='auto')
ax.hist(s2, bins='auto')
t, p = stats.ttest_ind(s1, s2,  equal_var=False) # Welch's
# p < 0.05, reject null hypothesis, samples are significantly different
```

- t-test is a kind of "parametric" test, assumes data come from some distribution that can be described by some parameters, in this case mean and std of normal distrib
- there are also "non-parametric" tests, which assume nothing about the underlying distributions
- this makes them safe in their assumptions, but gives them less statistical power
- a commonly used one is Mann-Whitney U test
    - `u, p = stats.mannwhitneyu(s1, s2)` still gives p < 0.05, but is higher than for Welch's t-test
- to test if a sample comes from a given type of distribution, use the "Kolmogorov-Smirnov" test, whose null hypothesis says the sample comes from the distribution:

```
mu, sigma = bimodal.mean(), bimodal.std()  # blindly assume it's normal
d, p = stats.kstest(bimodal,  'norm', args=(mu, sigma))
# p = 0.0, reject null, not normal
s = stats.norm.rvs( loc=-2, scale=2, size=200)
d, p = stats.kstest(s,  'norm', args=(-2, 2))
# p > 0.05, can't reject null, likely normal
```