- go over DataFrame exercise from class 09
- go over homework4
- go over project requirements: https://scipycourse2020.github.io/project

statistics

- we've already done some stats using numpy:
  - get 1000 *continuous* (float) values evenly distributed over the interval `[0, 1)`, this is called a continuous uniform random distribution:

```
import numpy as np
c = np.random.random(1000)
```

  - every time you ask for a sample, you get a different set of values, thanks to your computer's random number generator. If you want to get the same sample every time you run your code, set the "seed" of the random number generator to some constant value, e.g. `np.random.seed(0)` before you start random sampling. This is useful for generating identical results each time you run your code:

```
np.random.seed(0) # set fixed seed
np.random.random(10) # get some values
np.random.seed(0) # restore fixed seed
np.random.random(10) # get identical values
# don't restore fixed seed
np.random.random(10) # get different values
```

  - get 1000 *discrete* (integer) values evenly distributed over the interval `[0, 10)`, this is called a discrete uniform random distribution:

```
d = np.random.randint(0, 10, 1000)
```

  - we know how to check how these values are distributed, by visualizing them:

```
import matplotlib.pyplot as plt
def cf():
    plt.close('all')
f, ax = plt.subplots()
ax.hist(c, bins=30)
```

    - for discrete distributions, best to use as many bins as you have possible values:

```
d.min() # check that we got what we asked for
d.max()
f, ax = plt.subplots()
edges = np.arange(0, 10+1) # +1 for the right bin edge
ax.hist(d, bins=edges)
```

- otherwise, you'll end up with artificial gaps between discrete values:

```
f, ax = plt.subplots()
edges2 = np.arange(0, 10+0.5, 0.5) # 0 to 10 inclusive, 0.5 steps
ax.hist(d, bins=edges2) # notice the artificial gaps
```

- plotting the distribution of your data is 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
  - 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(s, bins=30)
```

- generate bimodally distributed (having 2 peaks) data by combining two normal distributions

```
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]) # combine both into a single array
f, ax = plt.subplots()
ax.hist(bimodal, bins=30)
```

  - are `bimodal.mean()` and `bimodal.std()` meaningful in this case? no, they're poor descriptors of this bimodal distribution, best way to tell is to plot and inspect the distribution
- matplotlib hist vs numpy hist:
  - to plot histograms, we've been using `ax.hist()` or `plt.hist()` from matplotlib

    - sometimes you might want to calculate a histogram without plotting it
    - `np.histogram()` returns the count in each bin, and the bin edges
    - `n, edges = np.histogram(bimodal, bins=30)`
    - 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
s.mean() # approx 5
s.std() # approx 0.5
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 variable, 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, density=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 PDFs')
```

    - can also calculate and plot the cumulative distribution function (CDF), which describes the fraction of data that fall below `x`, as a function of `x`
    - can construct a CDF by taking the PDF and accumulating the bin counts as you move from left to right
    - use `rv.cdf(x)`, and plot the CDF with `ax.hist(s, bins, cumulative=True)`
    - here's the equivalent CDF for the same sample and random variable:

```
f, ax = plt.subplots()
# plot normalized CDF:
ax.hist(s, bins=30, density=True, cumulative=True) # max val == 1
x = np.arange(3, 7, 0.01) # evenly spaced x values from 3 to 7
y = rv.cdf(x) # exact cumulative 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 CDFs')
```

- 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 most 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(a, popmean)` where `a` is the sample of observations and `popmean` is the population mean you want to test it against.

```
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, cannot 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 10x the amount of 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 kind is called "Welch's", which doesn't assume the two samples have equal variance (standard deviation squared)

```
s1 = stats.norm.rvs(loc=0, scale=1, size=1000) # control
s2 = stats.norm.rvs(loc=0.5, scale=1, size=500) # treatment
f, ax = plt.subplots()
ax.hist(s1, bins='auto')
ax.hist(s2, bins='auto')
t, tp = stats.ttest_ind(s1, s2, equal_var=False) # Welch's
# tp << 0.05, reject null hypothesis, samples are significantly different
```

- t-test is a "parametric" test, assumes data come from some distribution that can be described by a small set of 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
- one common non-parametric test is the Kolmogorov-Smirnov (2-sample) test, which only assumes that your two data sets are drawn from a continuous distribution:

```
d, kp = stats.ks_2samp(s1, s2)
# kp << 0.05, but higher than tp
```

- another common non-parametric test is Mann-Whitney U test, which doesn't even assume continuous distributions (allows for discrete distributions as well), but you pay for that with less statistical power:

```
u, up = stats.mannwhitneyu(s1, s2)
# up << 0.05, but higher than tp
```

- nice comparison of KS vs. MWU: https://www.quora.com/What-are-the-differences-between-the-Kolmogorov-Smirnov-test-and-the-Mann-Whitney-U-test/answer/Ted-Wrigley - or see `KS_vs_MWU.txt`
- visually checking distributions for normality is important, but you can also *quantitatively* test for normality with tests, e.g. Kolmogorov-Smirnov 1-sample test, `stats.kstest()`
- null hypothesis says that the sample comes from the specified theoretical 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
```

- for more advanced R-like statistical modelling, see the `statsmodels` package (http://statsmodels.org)

stats exercises

1. Load in some example data from `stats.csv`. What's the easiest way to load such data?
2. The data should have two columns: `control` and `treatment`. Extract the columns as two separate 1D arrays. Remove any NaN values. Hint: DataFrames have a `.dropna()` method.
3. Plot the distributions of both the control and treatment data in the same figure. Label them with a legend. Do they both look normal? Do they look significantly different? Are their means and standard deviations different? Save the figure to a `stats.png` file.
4. Choose an appropriate stats test to see if the two distributions are significantly different. Do the results differ if you choose an inappropriate test?
5. Use the KS test separately on each of the two distributions to check for normality.