ASTRON 3705 / PHYSICS 3704

Statistics and Data Science

Spring 2025

Instructor Information

- Lecturer: Prof. Jeffrey Newman
  - (I'll answer to Jeff or Jeffrey, Dr. Newman, Professor, Professor Newman...)
- Office: 310 Allen Hall
- Email: janewman@pitt.edu
- Phone: (412) 592-3853
- Office hours: Monday 3-3:30 or by appointment
- **Survey to help design groups has been posted on Canvas -- please complete by end of Friday!**

Reminder of our goal: testing the results from coin flips

- Consider flipping two coins. The possible things that could happen are:
  - 1st coin heads, second heads
  - 1st coin heads, second tails
  - 1st coin tails, second tails
  - 1st coin tails, second heads
- If heads and tails are equally likely, then each of these possibilities is equally likely, so we'd expect each of these possibilities to occur 1/4 of the time.
- So the probability that both coins give the same result is 1/4+1/4= 1/2 ; if we flip coins an infinite number of times, half the time we'd get this result.

Where we left off: accessing arrays

- We often only want to look at or use one or some elements of an array in a calculation (this is called 'slicing'). e.g.:

`alin[0],alin[100]`

`alin[0:10]`

* `alin[0:1]`

`alin[:]`

* `alin[10:20:2]`

`alin[90:-2]`

`alin[-1:90]`

- Python, like c (which it was developed in) and some other languages, identifies the first element of an array as element ZERO, not ONE!

- So in a 101 element array akin, the first element is `alin[0]`, the fifth would be `alin[4]`, and the last is `alin[100]` (or `alin[-1]` !!).

- As an additional complication, when slicing, Python does NOT include the last element specified, but stops at the one before it.

Variable names in Python are labels

- In Python, multiple variable names can refer to the same intrinsic object in memory.

```
a = [1,2,3,4]
b=a
b[3]=10
b
a
from copy import copy
c=copy(a)   # c = np.copy(a) would create an array
            # that is a copy of a
c[3]=5
a
```

Note: this is not how variables work in most computer languages!

Variable names in Python are **labels**

In Python, a "name" or "identifier" is like a parcel tag (or nametag) attached to an object.

```
a = 1
```

Here, an integer 1 object has a tag labelled "a".

If we reassign to "a", we just move the tag to another object:

```
a = 2
```

Now the name "a" is attached to an integer 2 object.

The original integer 1 object no longer has a tag "a". It may live on, but we can't get to it through the name "a".

If we assign one name to another, we're just attaching another nametag to an existing object:

```
b = a
```

The name "b" is just a second tag bound to the same object as "a".

from:

https://web.archive.org/web/20180411011411/http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html

For loops... and avoiding them

- Let's say:

```
x=np.linspace(0,2*np.pi,50000)
```

- and we want to plot sin(x) vs. x . In most languages, we'd do this with a for loop. In python, there are many ways to do this...
- 1) one inefficient way, by appending to a list, then converting to an array:

```
y_tmp = [] # create an empty list
for x1 in x: y_tmp.append(np.sin(x1))
y=np.array(y_tmp)
```

- 2) Another inefficient way: create an array of the right size, and use enumerate(), a built-in function in Python, to loop over its elements:

```
y=np.zeros(size(x))   # y = x*0. would also work
for i,x1 in enumerate(x): y[i]=np.sin(x1)
```

For loops... and avoiding them

- 3) Another inefficient way: use a "list comprehension" to calculate all the elements of y, then convert to an array (the `map()` function was an older way to do this)

```
y = [np.sin(x1) for x1 in x]
y=np.array(y)
```

- 4) **Much much much more efficient**: numpy functions can operate on numpy arrays all at once

```
y=np.sin(x)
```

- This is >100 times faster than the other methods! (You can test this with the `%timeit` or `%%timeit` jupyter/ipython magic command)

Plotting

- Python has powerful capabilities for plotting, whether incidental plots or publication-quality ones. Often, you'll want to plot something to diagnose if your code works, see what results are, etc. Let's do something basic:
- We created an x array and y=sin(x) array already. Now plot each one:

```
plt.plot(y)          #plots y vs. index number
plt.plot(x,'ro')     #overplots x, as red circles
```

Notes: 1) plt is actually an alias for matplotlib.pyplot

2) Arbitrary Python expressions can be used as the input to many functions. e.g., `plt.plot(2*x)` or `plt.plot(x+y)` work...

Plotting

- In the previous example we just gave `plt.plot()` an array of values, and it assumed x=[0,1,2,3,...]
- To plot y vs. x, do (for instance):

```
plt.clf()    # Clear the plot before doing a new one.
plt.plot(x,y,'b-')
```

- Note that you list X first, then Y!
- Do `?plt.plot` to see a list of the different plot styles.
- You can specify color ('r', 'g', 'b', etc. for red/green/blue) then style ('-' for lines, '--' for dashed, ':' for dotted line, '.' for points, 'o' for circles, etc.):
  - e.g.: `plt.plot(x,y,'k--')` will plot a black dashed line

If you're looking for something more to do:

- Try to generate $y=x^{3/2}$ using all the different alternative methods with/without for loops. Confirm that you are getting consistent results. Use `%%timeit` (with a large number of array elements in x, e.g. 100k) to compare execution speed.

- Generate plots of the curves for Gaussian distributions of mean 0 and different sigmas (I expect you to be able to find the formulae you need online). Overplot them, using different colors / line styles/ symbols. Use the `label` keyword in the calls to plt.plot(), add a legend with `plt.legend()`, and add axis titles with `plt.xlabel()` and `plt.ylabel()`. Search for information on the functions to find the syntax you need!

- A couple more examples are in the notebook

Back to our goal: testing the results from coin flips

- Consider flipping two coins. The possible things that could happen are:
  - 1st coin heads, second heads
  - 1st coin heads, second tails
  - 1st coin tails, second tails
  - 1st coin tails, second heads
- If heads and tails are equally likely, then each of these possibilities is equally likely, so we'd expect each of these possibilities to occur 1/4 of the time.
- So the probability that both coins give the same result is 1/4+1/4= 1/2 ; if we flip coins an infinite number of times, half the time we'd get this result.

Generating random numbers

- So, first we need to generate random numbers!
- In Python, we can use the `numpy.random.rand()` function to generate random numbers, evenly distributed between 0 & 1.
- The syntax is:

```
import numpy.random as random
randnum=random.rand(ndim1 , ndim2 , ndim3...)
```

where `ndim1, ndim2`, etc. are the size of the 1st, 2nd, etc. dimensions of the array (unlike with e.g. `np.zeros`, this is not provided as a tuple; `random.random_sample()` can do the same thing but use tuples to specify dimensions).

Making heads & tails from random #s

- We want to have tails 50% of the time and heads 50% of the time.
- Since `random.rand()` generates random #s evenly distributed between 0 & 1, we can call the result 'tails' if the result is 0-0.5, and 'heads' if it is 0.5-1.
- How do you do a logical test like that in Python?
  - With an IF statement, just like in other languages. Try this, repeatedly (hit shift-enter in a code box containing this code) to rerun it:

```
if random.rand(1) < 0.5:
    print("tails")  #note: indented by 4 spaces, no less!
else:
    print ("heads")
```

- Note: if you really wanted to do this in one line, you could do:

```
result = ("tails" if random.rand(1) < 0.5  else "heads") ; print(result)
```

Let's break that down:

```
if random.rand(1) < 0.5:
  print("tails")  #note: indented by 4 spaces, no less!
else:
  print("heads")
```

- In Python, the logical part of the if statement is followed by a colon. The set of expressions to be executed if it is true (or false) are all indented by 4 spaces to indicate that they are executed together; in other languages you might use braces ({ }) or BEGIN...END statements for that.
- I.e.: the syntax is:

```
if (some expression):
   (what to do if that expression is true)
```

- What is evaluated as 'True' in Python? Try:

```
if 1: print("1: True")     # if statements with no else
if 0: print("0: True")     # can be done in 1 line easily
if 2: print("2: True")
if 0.4: print("0.4: True")
```

The else part is optional...

```
if random.rand(1) < 0.5:
   print("tails")  #note: indented by 4 spaces, no less!
else:
   print("heads")
```

- The underlying pattern is:

```
if (some expression):
 (what to do if that expression is true)
else:
 (what to do if it's not true)
```

- The command following else is run if the expression is not true.
  - But what is going on with random.rand(1) < 0.5 anyway?
- This is a *relational* expression in Python: it tests whether the expression on the left is less than (<) the expression on the right.

Other "relational" tests in Python:

- <=: less than or equal to
- <: less than
- ==: equal to
- !=: not equal to
- >=: greater than or equal to
- >: greater than
- We can combine these with standard Boolean logic (`and`, `or`, `not`). e.g.:

```
a=4
if ( (a < 5) and not(a == 7) ): print(a)
```

Back to coins

- We simulated one coin flip with:

```
if random.rand(1) < 0.5:
   print("tails")  #note: indented by 4 spaces, no less!
else:
   print("heads")
```

- Now, let's do 1000 flips. Remember the syntax:

```
randnum=random.rand(ndim1 , ndim2 , ndim3...):
```

```
Nsims=1000
print( (random.rand(Nsims) > 0.5).sum(), 'heads, for a fraction of ',(random.rand(Nsims) > 0.5).sum()/Nsims)
```

- **What happened here?**
- (Note: we could have gotten the same results as `(...).sum()` by using `np.sum(random.rand(Nsims) > 0.5)` but it's good to get used to using object methods.

What was the problem?

- The random # generator gets reinitialized whenever you run random.rand() !
- Write a couple of lines of code that will print out the total & fraction of heads for the same sample of 'coin flips' instead of two different samples. A reminder: our old code is:

```
Nsims=1000
print( (random.rand(Nsims) > 0.5).sum(), \
'heads, for a fraction of ', \
(random.rand(Nsims) > 0.5).sum()/Nsims )
```

- Once it's working, increase Nsims to 10000, 100_000, 1_000_000, 10_000_000 and see what happens when you execute the same command repeatedly (with shift-enter).

Now, we'll move the goalposts...

- What I'd like us to do is test how the frequency of two coins coming up the same behaves, as the number of simulations increases.
- We could do this by repeatedly executing similar lines of code, but it's very likely we'd make a mistake.
- So let's make our very own Python function!
- A function will take some input, and provide an output (possibly just an indication of successful completion); they can be almost arbitrarily complicated.

Creating a function

- We want our function to have only one input - the number of tests to do - and one output - the fraction of times we get the same result on both coins.

- The first line tells Python we are defining a function, and what it's inputs are:

```
def sim2coins(ntests):
```

Creating a function

```
def sim2coins(ntests):
```

- We might add a comment telling us what the routine does:

```
# simulate ntests tosses of 2 coins
```

- Python ignores anything following a # (unless it's within a string variable). All other lines of code in the function should be indented by 4 spaces.
- Now, let's simulate the first coin, with 0=tails, 1=heads:

```
coin1=random.rand(ntests) > 0.5
```

- and the second:

```
coin2=random.rand(ntests) > 0.5
```

- Next, we set what the function will return - it could be any sort of Python variable, array, etc. In Python, returning something is optional.

```
return np.sum( coin1 == coin2 )/ntests
```

To recap:

```
def sim2coins(ntests):
# simulate ntests tosses of 2 coins
coin1=random.rand(ntests) > 0.5
coin2=random.rand(ntests) > 0.5
return np.sum( coin1 == coin2 )/ntests
```

To test this out, in a Python code box do:

```
print( sim2coins(1000) )
```

and see if it works! Also try:

```
a=sim2coins(1000)
```

Putting the function in a file

- 1) Open up a new file in vscode: e.g., hit the new file button next to VSCODE at the top left of the window.
- In general, Python scripts/programs should be put in files ending in .py , so give your file a filename accordingly: **sim2coins.py**
- 2) At the top of the file, put the lines:

```
import numpy as np
# we need to do the imports necessary
# for the module within the module's file
import numpy.random as random
```

- Then copy the text of `sim2coins`() to the new file and save it.
- You can then load the function into memory with

```
%run sim2coins.py
```

- This will treat it as a python 'script': a set of commands executed just like they were at a command prompt.

Putting the function in a file

- One file can actually contain many functions or procedures - e.g. subroutines that are used as part of a main program.
- The end of each function is indicated by the end of the indented region.
- The file can also contain python commands, just as you'd enter them at the command line.

Making new python modules

- If you want to make new python modules (e.g. `scipy.random` is a module, containing python functions like `rand()`), just put all the functions that go together in a file named YOUR_MODULE_NAME_HERE.py located in your python search path.

- To add the directory we just created to this path, you'll need to set up an environment variable. If you use bash or zsh (the previous and current default shells for Macs), the way to do this is to add the following at the end of your .bash_profile file:

```
PYTHONPATH="${PYTHONPATH}:/Users/USERNAME/python"
export PYTHONPATH
```

Making new python modules

- In CSH/TCSH, add the following lines at the end of your .cshrc or .tcshrc file:

```
setenv PYTHONPATH ${PYTHONPATH}:/Users/USERNAME/python
```

- For Windows, it's messier...

- After you've done this, you can do things like:

```
import sim2coins as s2c
print(s2c.sim2coins(1000))
```

- **Try it out!**

Time to move on to the lecture 3 notebook!

Suppose we make a change...

- Let's see, instead, how often coin1 is 0 (=tails) and coin2 is 1 (=heads).
- **Edit your sim2coins.py file to return the fraction of cases where that happened.** We have to do this a little differently: evaluating `(coin1==False and coin2==True)` will generate an error.
- Instead, we need to use `np.logical_and()`, which applies the `and` operation to *each element* of two arrays, element by element. So change the last bit of the function:

```
tails_and_heads = np.logical_and(coin1==False, coin2==True)
return ???
```

- Save the file
- Then, at your Python prompt do:

```
print(s2c.sim2coins(1000))
```

- and see what you get!

Why didn't that work?

- Python will automatically compile a module the first time you import it. However, to save time it won't automatically recompile a routine after that. We have to force it to.

- We can do that with the commands:

```
from importlib import reload
reload(s2c)
```

  - (Note:If you try doing `reload(sim2coins)` you'll find it doesn't work!)

- After recompiling, run it again...

How do our results depend on the number of trials?

- Let's try:

```
nsims_list=np.array([100,500,1000,5000,1E4,5E4,1E5,1E6])
result=nsims_list*0.
for i,nsims in enumerate(nsims_list):
    result[i]=s2c.sim2coins(nsims)
```

- Now, plot the fraction of successes as a function of the number of simulations.
- It would be helpful to use a logarithmic x axis: you can do this by:

```
plt.semilogx(x,y[ plot symbols, optional keywords, etc.)
```

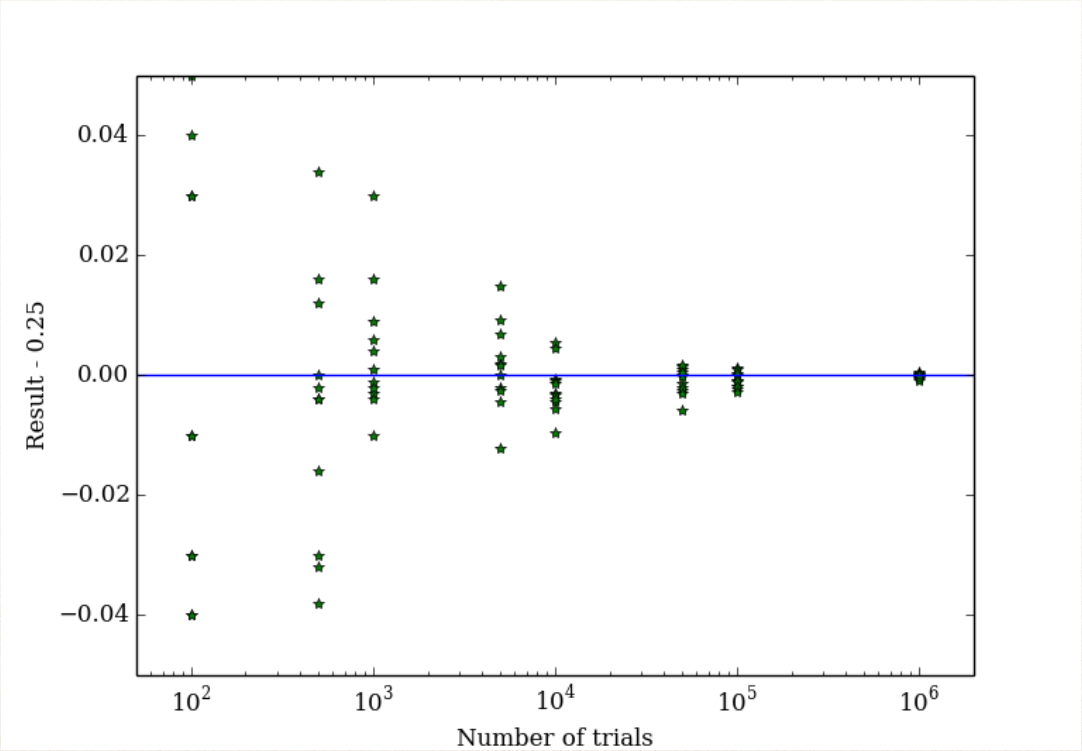
- Or, for an existing plot, you can convert to a logarithmic x axis with:

```
plt.xscale('log')
```

How do our results depend on the number of trials?

- **1) Plot the result - 0.25 (i.e., deviation from the expected probability) as a function of nsims**
  - Plot the points as green stars (look at the help on `plt.plot()`).
  - Use a logarithmic x axis.
  - Use a y axis range from -0.05 to +0.05 (look at the help on `plt.ylim()`)
- **2) By adding another, outer for loop, repeat this 20 times, overplotting all the results.**
  - If you put all the plot commands in the same code box, all the plots will be shown on the same axes, as we want.
- **3) Overplot the line y=0 to help guide the eye.**

Results from doing this 20 times:



As nsims gets larger, our results get closer & closer to 0.25

This convergent value defines the probability in the frequentist view

- As `nsims` gets larger, our results stay closer & closer to 0.25 .
- So the more trials we do, the fraction of times that coin1 is 0 and coin2 is 1 gets closer and closer to 1/4.
- In the *frequentist* definition, the *probability* of the event is the fraction of times it will occur if we repeat the experiment an infinite number of times - so that probability is 1/4, just like we all would have expected.
- This is not the only possible definition of probability - as we'll find out later.

More Monte Carlos: Simulating 6-sided dice

- An ordinary die has 6 sides, numbered from 1 to 6. How do we get that from a random number between 0 and 1?
- We want to convert the random number to an integer value. There are three ways to do this conversion:
  - `np.floor` (the largest integer below a #)
  - `np.round` (by default, rounds to nearest integer)
  - `np.ceil` (the smallest integer above a #)

- Note that the results of these routines will have data type float, not int !
- We want a result from 1 to 6...

3 ways to do this

```
nsims=1000
```

- Floor:

```
rolls_f=np.floor(random.rand(nsims)*6) + 1
```

- Round:

```
rolls_r=np.round(random.rand(nsims)*6 + 0.5)
```

- Ceil:

```
rolls_c = np.ceil(  ???  )
```

- All 3 of these should give equivalent results! If you think about what numbers you get out if the random # is 0.01 or 0.99, you should be able to convince yourself of this.

Let's test it:

- How often do we 'roll' a 1, 2, 3, etc.?
  - For an ordinary die, each possibility is equally likely, and will occur one-sixth of the time.
  - In Python, the `plt.hist()` function allows us to check this easily.
- **Bring up the help on `plt.hist()` using ?.**
- To apply it:

```
plt.hist(rolls_f)
```

Adjusting the histogram

- By default, `plt.hist()` uses 10 bins ranging from the minimum to maximum of the array.
- We can use the `bins` keyword to set the number of bins, and the `range = (min,max)` keyword to set the minimum and maximum to use (note that `(min,max)` is a tuple).
- **1) Using these keywords, adjust your plot to use 6 bins, centered at 1, 2, 3, 4, 5 and 6.**

Simulating 10 dice: multidimensional arrays

- A numpy array need not have only one dimension. E.g.:

```
img = zeros( (200,200) )
```

will create a 200 x 200 array, with zeros everywhere.

- `np.zeros()` and similar routines can take a tuple of dimension sizes as input, for arbitrary numbers of dimensions.

Simulating 10 dice

- Let's start with:

```
rolls_f=np.floor(random.rand(nsims)*6) + 1
```

- Remember that `random.rand()` can create arrays of arbitrary dimensions:

```
randnum=random.rand(ndim1 , ndim2 , ndim3...)
```

so:

```
rolls=np.floor(random.rand(nsims,10)*6 ) + 1
```

would be equivalent to doing `nsims` different simulations of rolling a total of 10 dice.

Simulating 10 dice: the slow way

```
nsims = int(2E4)
rolls=np.floor(random.rand(nsims,10)*6 ) + 1
```

- is equivalent to doing `nsims` simulations of rolling 10 dice.
- If you want to find the total of the 10 rolls for each simulation, you could do:

```
total_roll=np.zeros(nsims)
for i in np.arange(nsims):
    total_roll[i]=np.sum(rolls[i,:])
```

- Then you can do `plt.hist(total_roll)` to see the results of your simulation... **adjust the number of bins and range as necessary to show all the values in the array** (you may find `np.min()` and `np.max()` helpful).

We can do this more efficiently...

- A trick using `np.sum`:
- `np.sum(array, axis = dim)` will return the total of `array` over the `dim`th dimension (where `dim` could be 0, 1, 2,etc.). If no axis is provided, it sums the entire array.
- So you could replace the lines:

```
total_roll=np.zeros(nsims)
for i in arange(nsims):
    total_roll[i]=np.sum(rolls[i,:])
```

with one short, and much quicker (by a factor of hundreds), line:

```
total_roll=np.sum(rolls,axis=1)
```

Let's try simulating 2,5,10,100 dice...

- We don't need to actually do these simulations separately - we can just simulate once, and take different subsets. If we do:

```
nsims=int(2E4)
rolls=np.floor(random.rand(nsims,100)*6 ) + 1
```

We could get the results for 5 rolls with:

```
for i in arange(nsims):
    total_roll[i]=np.sum(rolls[i,0:5])
```

or with:

```
total_roll_5=np.sum(rolls[:,0:5],axis=1)
```

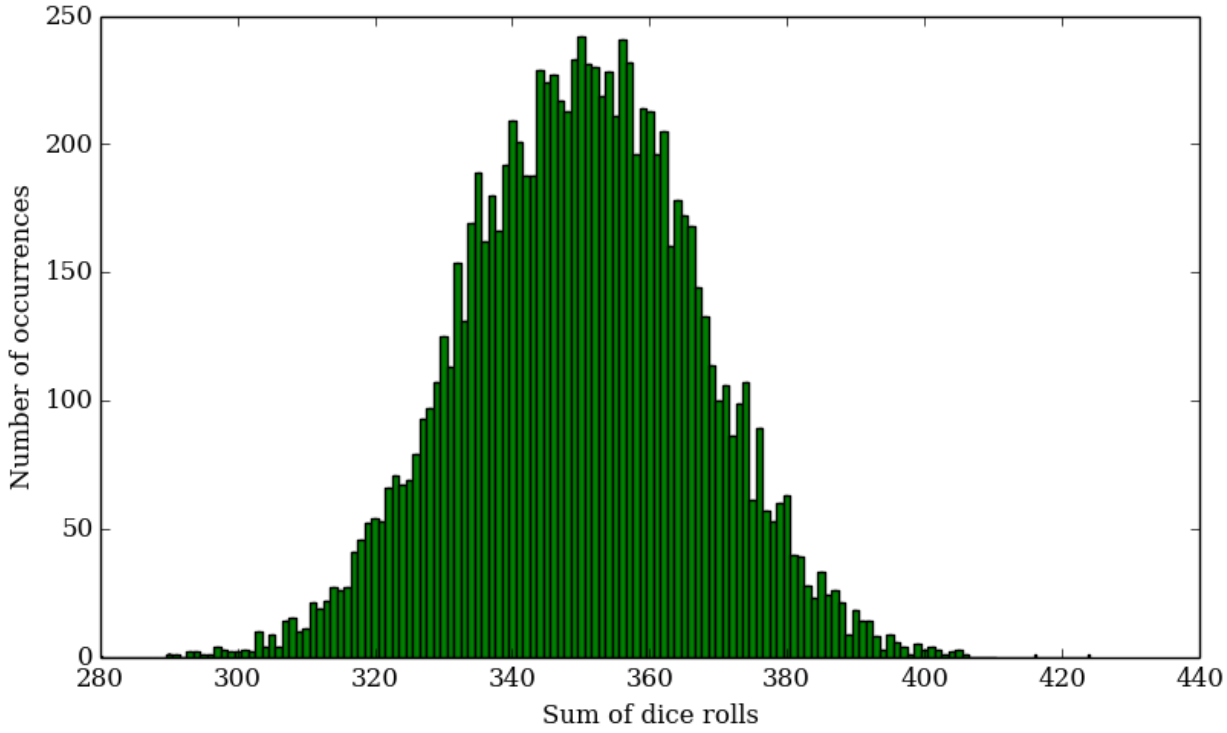
- or we could just proceed directly to plotting each case all in one line:

```
plt.hist(np.sum(rolls[:,0:5],axis=1) )
```

- **Plot up histograms for 2,5,10, & 100 rolls.**

Results

This sort of curve hopefully looks familiar - it is very close to a Gaussian distribution. We'll talk about them more later.



The Central Limit Theorem states that (with modest assumptions) the sum (or average) of a large number of random variables will approach a Gaussian distribution. Here we see it in action, starting with a totally flat, coarse-grained distribution.

Putting output in a file

- We often want to put output in a permanent file, not to the screen.
- The matplotlib routine `plt.savefig(filename)` does this.
- Depending on the filename given, savefig will write the current plot as a png, pdf, ps, eps or svg file.
- E.g.: `plt.savefig('spam.pdf')` will write the output in a PDF file named "spam.pdf"

Viewing the file

- On a Mac, you can view the file by navigating to it in the Finder (the face icon on the left side of your Dock), or else at a terminal prompt issue the command:

```
open spam.pdf
```

- Anything you can do at a prompt, you can do in jupyter/ipython (including notebooks) by putting a '!' first:

```
!open spam.pdf
```

- On a linux machine, use `gv` instead of `open` .

Homework

If we get to this slide on Wednesday, homework is due next Friday, Jan. 24, via Canvas.

Problem 1 (of 1):

1A) Make a module containing a new function, `simncoins()`, that takes as input the # of coins to flip per simulation, ***ncoins,*** and the total number of simulations ***nsims***; and returns back an array containing the total number of heads from flipping *ncoins* coins, with one element for each of the *nsims* simulations (so the array contains the results of *nsims* sets of coin flips).

Not necessary, but a good idea in general: You can test your code by calculating for only 2 coins, see how often you get 2 heads (or 1 head, or 0 heads), and compare to your expectation.

Homework

1B) Consider flipping 2, 4, 8, 16, 32, 64, or 128 coins. Make separate plots with histograms of the total number of heads that you get (produced from 50,000 trials, i.e. nsims=50_000), for each number of coins.

To do this, you will need to save your plots; I would prefer PNG format (remember to provide your code as well though!)

I will set up an assignment in Canvas for you to turn in your results.