Do as many of the following as we have time for in class.   
  
The zipped folder **lab\_3\_starting.zip** contains the code for all of the following. It's posted next to this handout within the Canvas assignment for this Lab. Download it, unzip to a folder, then open the folder as a PyCharm project before continuing below.

Some of the following problems ask you to upload a requested .**py** files below to Canvas within the **Lab 3 Assignment** link. Do so for **calcpi.py**, **find\_min.py**, and **modules\_1.py** (**L3-1**, **L3-4**, **L3-5**). Do any others for your own programming practice.

**[L3-1]** (**calcpi.py**) An interesting way of estimating **math.pi** uses a random number generator. Consider the circle of radius 1, inscribed inside a square of side 2. Assume the origin (0,0) is at the center of both the unit circle and square. See the picture to the right…  
  
Do the following calculation for some large value of **N**. Repeat the following **N** times: generate a random point **(x,y)** (use the **random.uniform(-1.0,1.0)** function in Python's **random** module), and count (**in\_circle**) how many times this point falls in the unit circle (**(x,y)**'s distance from the origin **(0,0)** **<= 1**).

Since the area of the square is **4.0==2.0\*2.0**, and the area of the circle is **math.pi\*1.0\*1.0 == math.pi**, we'd expect this to happen about **math.pi/4.0** of the time.

Finally, compute **4\*in\_circle/N**, and see how close it is to **math.pi** for large values of **N**.

Here is pseudocode on how to implement this calculation of an approximate value for pi. "Pseudocode" is structured English that gives the algorithmic structure of a Python program, resembling actual Python. It's often useful to write pseudocode first, getting the algorithm correct then translating it into working Python code. The starting code for this problem has each of the following steps as comments:

1: Read the value of **N** from the user

2: Set up an accumulator counter named **in\_circle**, initialized to **0**

3: Repeat **N** times => **for count in range(N)**:

3.1: Generate a random **x** in the range **(-1.0,1.0)**

3.2: Generate a random **y** in the same range

3.3: if **(x,y)** is within **1.0** of the origin **(0,0)**:

3.3.1: Add **1** to **in\_circle**

4. (not in the above loop body's 3.1, 3.2, 3.3) Calculate and print new pi estimate **pi\_estimate = 4\*in\_circle/N,** along with its absolute value difference from **math.pi.**

Translate the above into a working Python program **calcpi.py**. **When you are finished, submit your working .py source file to Canvas.** Note you'll need to import two different modules at the top of your code: **math** and **random**.

As HTT3 suggests: **start small, get something small working, then add to it.**

One strategy might be:

1. Implement step 1: read an **int** N from the user.
2. Implement everything except for step 3 (the for loop); when run, it should print out **0.0**.
3. Next, get the step 3 **for**-loop working. Set it up to iterate **N** times, adding **1** to **in\_circle** if randomly-generated**(x,y)** is within **1.0** of **(0.0,0.0)**. (We'll discuss how to write this condition **if** statement in class.)  
     
   Remember that **distance((x,y),(0,0)) = math.sqrt((x-0)\*\*2+(y-0)\*\*2)**. However, for now just set **x = y = 0.0** within the loop. When run, it should print out **4.0** since **in\_circle==N** because every point **(0,0)** is inside the unit circle.
4. Finally, change the loop body to implement 3.1 and 3.2, so **x =** "random **float** **>= -1.0**, **<= 1.0**". Do the same for **y**.

**[L3-2]** (**debug\_1.py**) Together in class, we will run the PyCharm debugger on the provided code. We will follow the posted PDF on "PyCharm Debugging".

**[L3-3]** (**debug\_2.py**) Same as previous, but with a different program.

Extra files: (**debug\_3.py**, **debug\_4.py**, **debug\_5.py**) A collection of "mystery" programs. If you have time, use the PyCharm debugger to discover what they do.

**[L3-4]** (**find\_min.py**) The provided code reads how many integers for the user to input, then reads them, one by one. While doing so, it keeps track of the smallest entered **int** so far. When done, it prints out the smallest integer of all those that were entered. Note the use of an accumulator variable **min\_so\_far** that tracks the current minimum at any point during execution.  
  
However: there's a subtle bug in this code. Find it. Add a comment at the top of your code that describes the bug, then **submit the resulting .py source file to Canvas**. Can you fix this bug? It's surprisingly tricky...

**[L3-5]** (**modules\_1.py**) The provided code shows the beginning of a program that imports and uses the **math** module. Follow the instructions given in the comment at the start of the file and **submit your final .py source file to Canvas**.

**[L3-6]** (**modules\_2.py**) The provided code shows the beginning of a program that imports and uses the **random** module. Follow the instructions given in the comment at the start of the file.

Extra file: (**birthday.py**):

Here's a program which computes the probability of no two people in a group of **N** sharing the same birthday. First it calculates and prints the theoretical probability (**365\*364\*...\*365-N+1) / 365\*\*N**

Then it builds a list of **N** birthdays, with birthdays chosen as random **int** Julian dates from **1..365**.

It does so 10000 times, counting how many of these lists have no duplicate birthdays. Finally, it prints out the fraction of the time this occurs: **count\_of\_unique\_birthday\_lists / 10000**.

As **N** gets large, the theoretical and experimental probabilities get closer...