1) **Collision avoidance.** For warm-up exercise 12 you were asked to revisit the lab 4 house animation. This was the provided program: lab4.py. Download and test it. Notice that the stars are annoyingly drawn not only in the sky but also in “front” of (on top of) the house. For warm-up 12 you were provided with alternate (optional) code, lab4v2.py, that avoided generating stars that collided with the house. Download and test it. Find the part of the code in lab4v2.py that achieves this “star-house-collision avoidance” and explain how it works to a TA or instructor. (No need to explain the details of the insideHouse() method itself. Just how the method is used.)

_student: Get check 1_

2) Note that you could use a similar collision avoidance technique when avoiding putting the words in your project 4 word cloud in the same location. Let’s practice this technique by generating 6 distinct random integers between 1 and 10. (I.e., no repeats – “collisions” – allowed). How? Read on...

a. Start by generating 6 random integers between 1 and 10, not checking for repeats for now, and adding them each to a list. Print the list to check that your code is working. Notice that if you run your program repeatedly, you will often get repeated numbers among the six.

_student: Get check 2_

b. Now add a while loop, a la check 1 above, so that when the randomly generated value has previously been added your list of numbers, it tries again until it finds one that is new. Only after you’re sure you have a new number (not already in the list) should you append it to the list.

_student: Get check 3_

[BTW, for project 4, do not worry at all about avoiding overlap/collisions among the words in your word cloud until you already have a word cloud of the most frequent words in your text, randomly placed, without regard to overlap, with the size of each word proportional to its frequency of occurrence. Until that is completely working, don’t write any code to address the “collision-avoidance” aspect of the problem.]

3) **Rolling two dice.** In class we created a program that simulated the rolling of a single die, and computed the probability of the die showing each of the possible die values 1 through 6. Save a new copy of that program and modify it so that it simulates the rolling of two dice, computing the probability of rolling each of the possible values from 2 through 12. Which of these possible values has the lowest probability of being rolled? And the highest? Can you explain your results?

_student: Get check 4_
The Black Jack dealer, cont’d. In class we created a top-down design for a program that simulated the dealer in Blackjack for the purpose of determining how likely the dealer is to bust.

In coding it up, we left it at around the “second-level” of our top-down design. Here it is fleshed out a little further, to around “level 3” of the top-down design: `blackjackDealerSim.py`. Note that I’ve now added code for `simGames()` as well as the outline/pseudocode for the `dealerBusts()` function, the final level of our top-down design. Read all the code and comments carefully. Then fill in the code based on the provided comments/pseudocode. (Literally translate each un-coded comment into a line of python code, paying attention to the indentation of the comments, which indicate the indentation level of that line of code.) Test your program for input values of 1000 hands per starting card, then 10000 hands, 100000 hands, and 1 million hands.

Get check 5

The Black Jack dealer, revisited. Our black jack dealer simulation tells us how likely the dealer is to bust, assuming we have no prior information about the dealer’s hand. But when you play Blackjack in real life, the dealer starts with one card showing, so you actually have an initial piece of information about the dealer’s hand. Thus, what we really want to know is how likely the dealer is to bust given the dealer’s initial visible card. For this checkpoint you will complete this task of generating a list of dealer bust rates, one for each possible initial showing card.

A copy of the program we wrote in class with some incomplete modifications toward this new goal is found in the file `blackjackDealerSim2.py`. (New/modified lines throughout are marked with ###.) Read the comments/changes in each function carefully – notice, e.g., the new parameter that has been added to `dealerBusts()` – and complete the necessary modifications.
Test your program for input values of 1000 hands per starting card, then 10000 hands, 100000 hands, and 1 million hands.

Get check 6

5) Monte Carlo Simulation. The Monte Carlo method is a way to approximate a value using repeated random sampling. We did this, for example, when we approximated the probability of rolling a 7 when tossing two dice in check 1 above. We will use the Monte Carlo method to estimate the value of the world’s most famous transcendental number: \(\pi\).

For this checkpoint you will simulate the act of throwing thousands of darts onto a square (2x2) dart board. (No graphical output required.) As the “darts” are being “thrown” onto uniform-randomly generated points in the square, we will count how many of the darts “land” inside the circle that is inscribed by our 2x2 square. Since we know a 2x2 square has an area of exactly 4, we can use the percentage of darts that land inside the circle to estimate the area of the circle!

As an example, if 50% of the darts end up inside the circle, it would mean that the area of the circle is roughly 50% of 4, which is 2. (If you draw yourself a picture of the circle inscribed by our square, you should be able to visually see whether this is lower or higher than the actual answer.)

Note that this circle, since it’s inscribed in a 2x2 square, has a radius of 1. Now consider the formula for the area of a circle (look it up online if you don’t remember it). Can you see why approximating the area of this circle is the same as approximating \(\pi\)?

Have your program prompt the user for how many dart throws s/he wants to simulate. Use a loop to simulate that number of dart throws, keeping track of how many darts “land” inside the circle. (See implementation tips below.) Then output the approximate area of the circle, which is also the approximate value of \(\pi\).

Implementation tips:
- A framework for the code is provided for you in pi.py.
- You should assume the 2x2 square dartboard goes from \(x = -1\) to \(x = 1\), and from \(y = -1\) to \(y = 1\). Each dart’s randomly generated landing point \((x, y)\) needs to be within this square of points, and all points in the square must have uniform probability of being “hit.” With a little thought, this can be achieved by calls to the random() function. (Recall last night’s warm-up exercise... for this checkpoint we’re asking: how would you generate a random floating point number between -1 and 1?)
- To check if your randomly generated point \((x, y)\) is inside the circle of radius 1, you might recall that the formula for a circle is \(x^2 + y^2 = r^2\). Hence if \(x^2 + y^2\) is \(<= 1\), that means the dart landed inside the circle.
- Use an accumulator variable to count the number of darts that land in the circle.
- See how many dart-throws you need to simulate to get close to the correct value of \(\pi\)!

Get check 7

6) Bonuses. If you complete these please email/show your code to Prof Chung so she can appreciate it too! (Also send/show her any old lab bonuses you completed that you took pride in.)
a. Monte Carlo visualization: Give the above Monte Carlo simulation a graphical display by drawing a small dart board and displaying each of the points where the darts land (use one color to draw points that land inside the circle and another for points that land outside the circle). Display the approximate area of the circle underneath the dartboard after the simulation completes. Make sure your output mentions the name of the special number being approximated.

b. Blackjack simulation: If the dealer starts with a 5, how many more cards on average will they deal to themselves before she busts? Calculate for each starting card, the average number of cards the dealer deals before the dealer busts. As always, there are multiple ways to do this, but one python tip that might be useful: a function can return two values, delimited by commas. A call to such a function would look something like: var1, var2 = myfunction().

Get bonus checks