|  |  |
| --- | --- |
| **Testing** | **Evidence** |
| I created version one, where a random number is generated and a single guess is made. The program decides whether the two are the same or not. I tested a correct guess, an incorrect guess, and a guess on either side of the random number. |  |
| In version two, the program loops forever until the user correctly guesses the random number. To test this version, I got the guess right on the first try, tried some boundaries around the random number, and some completely wrong numbers. To end the loop, I always had to finish with getting it right. |  |
| In version three I added the hints about whether the number is higher or lower than the guess. I started off testing with getting the guess right on the first try. Then I tried a variety of higher numbers, a variety of lower numbers, a variety of higher numbers, a mix, and boundary values around the number. |  |
| I started creating version four, which adds in a guess counter and a limit to the number of guesses. I started this out hard coded as 10.  I changed the while loop to accommodate for this, and added an extra if/else statement so that the “number is lower/higher” would not print if this was the user’s last guess.  However, this meant that the guess\_num > 0 comparison was being made twice, and therefore wasn’t really necessary in the while loop.  So, I changed the while loop to an infinite loop (while True) with two ways to break it – if the guess is correct or if the user runs out of guesses. |  |
| Testing version four. I tried getting it right on the first guess, getting it wrong a few times, getting it wrong nine times, and then running out of guesses. They all worked as expected. |  |
| In version 5, I changed around the code a little with the number of guesses and added input to create two different modes – easy mode and hard mode. Easy mode allows the user ten guesses, and hard mode allows only four. For input, I prompted the user to input either E or H, referring to Easy or Hard. I used .strip() and .lower() in case they accidentally added spaces, or used e or E, etc. I tested this by using a variety of lower case letters, upper case, and spaces. I made sure that when I typed e it would allow ten guesses, and h would only allow four. |  |
| In version 6 I added error handling so that if unexpected or invalid values were inputted my program would be able to not break and instead prompt the user to re-enter until their input is valid.  To do this I added two functions, one for each time the user inputs. mode\_input only uses an if/elif/else statement because it is dealing with strings. guess\_input, however, requires a try and except because int input can cause the program to crash. That function also needed an if/else to make sure the integer was in range.  There were quite a few things that I needed to test to make sure this was working. I needed to test invalid (hello, 37 for mode input, enter, etc.), boundary – both invalid and valid (0, 1, 2, 99, 100, 101), and expected values. I also needed to make sure that re-entering input if it was invalid wasn’t counted as a guess, and that easy and hard were still allowing the correct amount of guesses.  I tested all this, and everything seemed to be working. |  |
| In version 7 I wanted to make the game a bit more playable. I made the output more clear instruction-wise, added in a “-----------------” divider before the game started.  For testing I commented out the testing print statement played the game without worrying about trying different values. Through doing this, I discovered that when I lost the game I wanted to know what the number actually was. So I added a print statement to the program that told the user what the random number was if they lost. |  |
| Here is a screen recording of me running through my testing plan with the final version of the code.  (If it doesn’t load, it is in the screenshots folder. It is the only video.) |  |