**Black Problem 2: Mastermind! [35 points; individual or pair]**

**Copied from: [https://www.cs.hmc.edu/twiki/bin/view/CS5/MasterMindBlack](https://www.cs.hmc.edu/twiki/bin/view/CS5/MasterMindBlack) on 3/22/2017**

Please make sure that you have read the instructions on the Assignment 8 "Black" main page so that you understand the choices available to you this week.

In this problem you will be implementing a "generalized" version of the Mastermind game! First, we describe the basic functionality of this program and then, at the bottom of this page, we describe some fun optional super extra bonus parts that you might wish to do for bonus credit.

The program begins "automatically": We just include the lines

```
if __name__ == "__main__":
    main()
```

at the bottom of the file (anywhere *after* the definition of the `main()` function itself—this is typically at the very bottom of the file), and this automatically invokes the `main()`function when the program is invoked from the command line. Those long lines before and after `name` and `main` are *pairs* of underscore symbols: There are two consecutive underscores at the beginning and two consecutive ones at the end.

To make it easy to test your different Mastermind implementations (see below), your `main()` should do nothing except to ask the user which version of the game to play, and then call either `mastermind1()` or `mastermind2()` based on the answer.

When the game begins, the program asks the player to enter:

- The number of holes per row (this is 4 in the commercial version of the game, but it can be anything in our generalized version!)
- The number of rows (this is the number of rounds in the game—it is 10 or 12 in the commercial versions of the game, but again it can be anything here!)
- The number of colors (these will actually be consecutive integers beginning at 0—in the commercial version there are 6 colors, but it can be anything here)

Once the player has specified these values, the game commences. The program chooses a random "code"—the secret selection of colors in the holes. Remember that the colors are actually integers ranging from 0 to the number of colors specified by the user, minus one.

At each round of the game, the player is asked to guess the code—that is, to guess the color for each hole. The program then "scores" the player's guess. For each guess that is the correct color in the correct hole, the player gets one "black" point. Among the remaining guesses, each guess that is the correct color but in the wrong hole gets one "white" point. The score is simply the number of black points followed by the number of white points.

After each round of play, the game board is displayed. The displayed game board shows the rows that have been guessed so far, along with their scores. Here is some sample input and output. Please follow these input/output conventions to make it easier for the grutors to evaluate your program. You may make minor cosmetic changes to this, but your input/output display should be essentially like this:

```
Welcome to Mastermind!
----------------------
How many holes per row shall we have? 3
How many rounds shall we have? 42
How many colors shall we have? 4
Enter your guess for round 0...
  Enter guess for hole 0: 0
  Enter guess for hole 1: 1
  Enter guess for hole 2: 2
===START BOARD===
Round 0   [0, 1, 2]   Score: 1 black and 1 white
====END BOARD====
Enter your guess for round 1...
  Enter guess for hole 0: 0
  Enter guess for hole 1: 0
  Enter guess for hole 2: 2
===START BOARD===
Round 0   [0, 1, 2]   Score: 1 black and 1 white
Round 1   [0, 0, 2]   Score: 1 black and 2 white
====END BOARD====
Enter your guess for round 2...
  Enter guess for hole 0: 0
  Enter guess for hole 1: 2
  Enter guess for hole 2: 0
You got it in 3 rounds!
Would you like to play again? n
Bye!
```

The game ends if the player guesses the score correctly (a win!) or the player has not succeeded at the end of the specified number of rounds. In either case, the player is asked if she/he would like to play again. If so, the process starts from the beginning, with the player being asked for the number of holes, rounds, and colors.

Your program will be evaluated both for correctness and also for quality of design and style. Approximately half of your score on this problem will be based on following the good design principles discussed in lecture and summarized below. Here are a few guidelines to follow for design and style. We'll consider these guidelines in grading your code:

- First, think about the design of your program and lay out the functions that you'll need on paper before you do any actual programming. In particular, try to identify the separate logical parts of your program. For example, you might have a part of your program that prints the contents of the board, another part that evaluates a guess, and so forth. For each part, ask yourself what arguments it will require and what results it will produce. These parts can then be translated into Python functions.
- Make sure that each of your Python functions really encapsulates one particular well-defined task. Then, write the functions one by one. For each function, include a docstring that explains the arguments that the function takes, what the function is "responsible for doing", and what result it produces. If your function is more than 10 or 12 lines of code, ask yourself if it makes sense to break it up into separate subfunctions. The answer may be "no"—but think about it.
- Think about your code carefully before you write it. There is usually more than one way to get a computational task done. Software takes a "short" time to write relative to the amount of time that it is used. It's worth programming slowly and making sure that the code that you write is clear, simple, and easy for you (or someone else) to go back and read or modify. If your code seems complicated, it probably can be rewritten in a simpler and more elegant way.
- Use comments when there is something in your code that is not self-evident. You don't need to have a comment for every line, but lines or blocks of code that do something that is not immediately obvious to the casual observer merit at least some documentation. You will find that it takes some experience to decide what is "not immediately obvious"; if in doubt, ask a professor or grutor.
- Test each function as you write it. Make sure that the function that you just wrote behaves correctly before moving on to the next function. *This will potentially save you enormous amounts of time and anguish when debugging.*

- Avoid global variables! `debug` is a notable exception, and on very rare occasions it makes sense to introduce others. In general though, it's cleaner and safer for functions to pass one another just what they need to do their jobs.
- Use descriptive function and variable names. A function name like `printBoard` is more descriptive than `pb`. Similarly, a variable name like `colors` or `numcolors` is more descriptive than `c` or `x`.

For the sake of debugging, you should have the following line at the top of your code:

```
debug = True # debug is True if debugging info is to be
displayed, and False otherwise
```

Throughout your code, you should have conditional statements of the form:

```
if debug:
    print blah, blah, blah
```

that print out key information about the inner workings of your program.

**In particular**, if `debug` is `True`, your program should print out the computer's chosen secret random code (the sequence of colors that you are trying to guess). This will allow you to ensure that the behavior of your program is correct. Later, of course, you can set `debug` to `False` to deactivate the printing of this information.

For example, in the game shown above, here's what things might look like in debug mode:

```
% python mastermind.py
Welcome to Mastermind!
----------------------
How many holes per row shall we have? 3
How many rounds shall we have? 42
How many colors shall we have? 4
DEBUG MODE:  THE CODE IS [0, 2, 0]
Enter your guess for round 0...
  Enter guess for hole 0: 0
  Enter guess for hole 1: 1
  Enter guess for hole 2: 2
DEBUG MODE:  MATCH AT HOLE 0
===START BOARD===
Round 0   [0, 1, 2]   Score: 1 black and 1 white
====END BOARD====

...
```

## Part One: Insecure Program

The first version of your game should be called `mastermind1()`. Internally, it should use `eval(input())` **not** int(input()) to read the user's guesses. As we saw in class, this allows the user to enter Python expressions and cheat at the game. *Do not* validate the input; if there are four colors and the user guesses 10, that's fine (it will simply show up as an incorrect guess).

To make cheating possible, you'll find that it's necessary for the function that reads guesses to know the secret code, even though that's not actually necessary for proper operation of the function. For the purposes of this part; we'll accept that minor glitch.

Your program should play the Mastermind game correctly when the user types legitimate guesses. However, it should also be possible for a knowledgeable user to win the game in one round by cheating. **In a comment at the top of** `mastermind1()`, clearly explain how to win by cheating.

## Part Two: Secure Program

The second version of your game should be called `mastermind2()`. For the most part, it should be the same as `mastermind1()` except that it will need to call a few revised functions. In this version, you must *fully* validate everything the user types. That means that you should ensure that every input is in fact an integer, you should check those integers to make sure they are sensible (e.g., the number of colors shouldn't be -1 or absurdly large), and illegal guesses should be detected. You will find the `try...except`construct discussed in class to be useful in validating input. If the user enters bad values, you should print a complaint and ask for the value again.

## Hints

In addition to the observations about this problem that we made in class, here are a few things that will be of use to you:

- To generate a random number, include the line

  ```
  from random import *
  ```

  at the top of your file. Then, the function `randint(x, y)` can be used to generate a random integer in the range from x to y inclusive.

- All Python types can be cast as strings. This is useful when printing! For example, if `mylist = [1,2,3]` then we can print it as part of a string as follows:

  ```
  print("My list is: " + str(mylist))
  ```

  The same can be done with integers, floating-point numbers, etc.

- Recall that to construct a list incrementally, we can use the `append` method. For example:
- `In [1]: mylist = []`
- 
- `In [2]: mylist.append(1)`
- 
- `In [3]: mylist.append(2)`
- 
- `In [4]: mylist`
- `Out[4]: [1, 2]`
- Remember that a function can return several items by placing those items in a list and returning the list!

## Optional Bonus Features

**Totally Optional Super Fun Bonus Parts!** If you want more entertainment, here are several fun optional bonus parts for you to consider:

- **[Up to 8 bonus points]** Use `turtle` or another drawing tool to visualize the board. That is, the user should see a graphical display of the board, the guesses, and the scoring rather than a textual representation.
- **[Up to 15 bonus points]** Allow the player to specify two options: Option 1 is that the computer chooses the code and the player tries to guess it (what you just did above) and option 2 is that the human player chooses the code (and keeps it secret from the computer) and the computer tries to guess the code! For full bonus credit on this, your computer player must be "good". See the Mathworld web page for some references on good strategies. This problem has been studied by mathematicians and computer scientists and there is a rich body of published research on it!

If you add these bonus features, please include a comment at the top of your file *clearly documenting* what you did and how to use your program.

## Submit

Be sure to submit your code with `debug` set to `True` so that we can see the debugging information (and particularly the secret code) when testing your program!