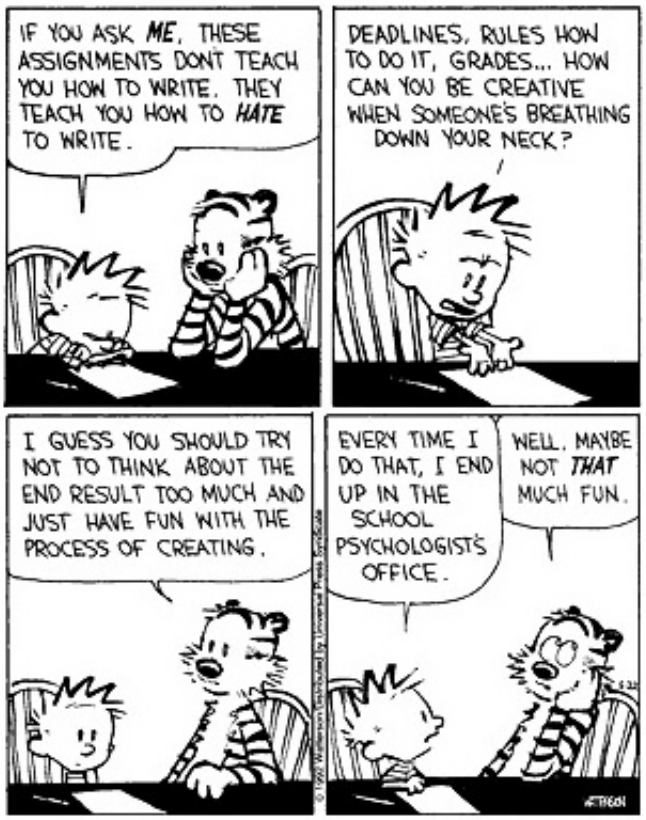
CS150 - Test Project 2
Memory
Due: Friday Dec. 9, at 6pm



A test project is an assignment that you complete on your own, without the help of others. It is a form of take-home exam. You may consult your text, your notes, your previous assignments, the notes and examples on the course web page and the Python library documentation (linked on the course web page), but use of any other source is forbidden. You may not discuss these problems with anyone aside from the course instructor. You may only ask the tutors for help with hardware problems or difficulties in retrieving your program.

You are encouraged to reuse code from your assignments or our class examples. Partial credit will be awarded, so try and get as far as you can.

Improving the program and extra credit

If you do everything that is required, the maximum number of points you can earn is 56. In order to obtain the full credit of 58, you must implement some extra features. I've provided some ideas below, but you should feel free to exercise your creativity. In addition, you may receive up to 2 points of additional extra credit (i.e. for a total of 60).

For all additions, include in the comments at the top of the file what you've added or you may not get credit.

1 The game

For this program, we will be implementing a text-based version of the game "memory" (aka concentration). See the Wikipedia page on "concentration" for more information.

When the game starts, all of the *cards* are face down. In our version, there are 16 *cards* and each of the *cards* is indicated by a number on the screen. The *cards* are laid out in four rows, each with four *cards*:

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
```

"Underneath" each of these numbers is a letter. The user does not see these, but the program will keep track of these. For example, the board above may have the following letters underneath:

```
I  D  B  F
H  B  C  D
A  A  E  E
F  H  I  C
```

The game proceeds by having the user pick two squares to look under by entering the numbers of the squares separated by a space. For example, to view 1 and 2 the user would type "1 2"; to view 7 and 15 the user would type "7 15".

The two numbers/squares that the user specified are then shown. For example, in the above case if the user entered "7 15" they would see:

```
1  2  3  4
5  6  C  8
9  10 11 12
13 14 I  16
```

If the user's selection do NOT match (like 7 and 15) then board with the cards turned over is displayed for 2 seconds and then the cards are turned back over leaving just the numbers:

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
```

The user is then prompted for another choice. If the user selects two that do match (for example 2 and 8) then those two are flipped over and they stay flipped over for the rest of the game. The user is then immediately prompted to enter an additional pair.

The game ends when the user identifies all of the matching pairs. When the game ends, the text "You win!" is displayed along with the finished board and the amount of time it took the player to solve the game. The user should then be prompted to enter their name and the top five fastest scores are printed. For example:

```
You win!
I  D  B  F
H  B  C  D
A  A  E  E
F  H  I  C

It took you: 8.4567864069 seconds

Enter your name: Dave

Top five scores
---------------
Dave K     8.45678964069
Ada L      15.89640693665
Bill G     29.33058905602
Julia R    50.11800122261
Mark Z     101.221760273
```

These top five scores will be stored in a file and should be updated each time the game is played.

When run, your program should start playing the game automatically. If your module is imported, however, the game will not start playing.

Demo

To help you understand how the game works, I've made a demo version available. From:

```
http://www.cs.middlebury.edu/~dkauchak/classes/cs150/assignments/TP2/
```

download both of the `.pyc` files. In `Terminal` (or windows command-line) navigate to the directory where you saved these files and type:

```
python memory.pyc
```

2 Implementation

As always, I strongly suggest an incremental approach to developing this program. Before you start programming, develop a design that describes what functions you will need, the parameters each function should take and how each function will work. You won't turn this in, but it will save you a lot of time and grief if you think through the design of the program first.

As you start to code, work incrementally: pick one function and get it working before moving on.

Read through this entire section before you start the program since I give some hints about implementing certain parts.

2.1 Representation

As you're thinking about how you will write your program, think about what information you need to store and update as the game progresses and how you are going to represent this information. For example, for hangman we had a number of pieces of information:

- The word the user was trying to guess
- The current version of the guessed word
- The letters that had been guessed so far
- The number of guesses that the user had left

What information do we need to keep track of for this game? As you think about this, think about how you will store this data (as a string, a list, a set, a dictionary, etc.). You should choose an option that is convenient and efficient. You will be graded based on your choices (for example, if you try and do it all with strings, you will not get full credit).

2.2 Game flow

Besides the representation, also think about the general flow of the game. There will be some sort of loop where you repeat some steps over and over as long as the game is in session. Think about what will happen during one "turn":

- the current version of the board should be displayed
- get the guess from the user
- check whether the guess is a match or not and act accordingly

This is a rough skeleton of the flow of the game. Flush out the details a bit more before you start coding and it will make your life much easier down the road.

2.3 One route

There are many ways of implementing this and getting it working. Here is one approach:

1. Pick your board representation and figure out a way to generate a random board configuration. For each number, the board will need to keep track of which letters are associated with each letter.

   Hint: the `random` class has a function called `shuffle` that takes any list and randomly shuffles all of the elements in the list. For example, try shuffling ['A', 'A', 'B', 'B']?

2. Decide how you're going to keep track of which numbers on the board have been correctly guessed. Write some code to display the board. The board should be printed out as 4 lines. If a square/number has not been correctly guessed, then the number will be printed. If the square/number has been correctly guessed, then the letter will be printed.

   To check that you have this working correctly, manually add numbers that have been "correctly guessed" and make sure that the board is displayed correctly by your code.

3. Get the input from the user and then regardless of whether they match or not, show the updated version of the board (i.e. with the letters flipped over the selected numbers), then revert back to the original board after the 2 second pause.

   Warning: For this program, it is important that you either run the program from the command-line OR run it in debug mode. If you run the program with the green arrow you will not be able to get the user interface to work properly.

   Hints:

   - In the `time` module there is a function called `sleep` that takes the number of seconds as a parameter and the program stops executing for that number of seconds.
   - You cannot "delete" text that you've already printed on the screen. However, one way to make it look like the text has been deleted is to print out a number of blank lines (e.g. 100). This causes anything that was on the screen to disappear. If you do this every time before you display the board, it will appear to the user like the board is simply being updated.
   - There are many ways of updating the board to reflect the current guess and then undoing them if the guess is not right. Think about the best approach for your implementation.

4. Update the functionality to differentiate when the pairs match vs. when the pairs don't match.

5. Add checking to make sure that the user enters valid numbers. *You may assume that they enter two integers separated by a space.* You must however check to see if they are valid board spaces and whether or not they have already been selected. As long as either of these cases applies, then you should prompt the user to enter another selection.

6. Add functionality to check if the user has won and print "You win!" when the game is won. Print the time it took the user to solve the problem. At this point you should have a working version of the memory game.

Hint: During testing, it may be easier if you make your game easier (for example, just a 2 by 2 board).

7. Add functionality to keep track of the top five scores. As always, try and break this down into different functional components and test each one individually.

    - You will need to store this data permanently from execution to execution. To do this, the data will need to be stored in a file.
    - You can hard-code the filename where you will be storing this data as a constant (e.g. memory.scores.txt).
    - Think about how you are going to store the data in the file. It's up to you, because you will be reading and writing the data on the file.
    - You may assume that the scores/times are unique (but not necessarily the names). This will likely be true and will make your life easier.
    - You will not need to invert a dictionary for this problem. If you find yourself trying to do this, you're probably making it more complicated than it needs to be
    - In the `os.path` module there is a function called `exists` that returns `True` if a file exists already or `False` otherwise. This can be useful for getting things started when there are no scores.

Extra Credit

You may add any extra credit functionality that you like that does not make the game simpler. Below are some possible suggestions. For any extra credit options that you add, make sure to include them in the comments at the top of the program.

- Check to make sure that the user actually enters numbers and that the user enters two separate numbers. The `isdigit` method in the string class may be useful.
- Allow the user to specify different board sizes. The number of elements needs to be even, but besides that you can be flexible.
- Allow the user to specify different difficulty levels. You can change the difficulty by: decreasing the pause time, changing the board size, allowing for more repetitions of each letter (instead of just 2).
- Include a better introduction to the game
- Improve the board display: better identification of which cards were just flipped over, better identification of which numbers are still available, etc.
- Turtle graphics?
- Make the game multiplayer and then keep track of each player's score.

When you're done

Make sure that your program is properly commented:

- You should have a docstring comment for each module at the top of the file
- You should have comments after the module docstring stating your name, course (including section number), assignment number and the date.
- Each function should have an appropriate *docstring*
- Other miscellaneous comments to make things clear

In addition, make sure that you've used good ***style***.

Submission procedure

For this assignment you will have one or more .py files. If you just have a single file, then submit that file directly.

To submit multiple files, create a directory with your name followed by the lab number. For example, my folder would be called `davidkauchakTP2`. Put all the files inside this directory and the create a .zip file from this directory. On the Macs, right-click on the directory and select "Compress...". If you're working on Windows, right-click on the file and select "Send to" then select "Compressed (zipped) Folder" (or if you don't have this option, use http://www.winzip.com/). You will then see a file with a .zip extension be created.

Submit this .zip file digitally at the usual location:

http://www.cs.middlebury.edu/~dkauchak/classes/cs150/submission/

Enter the relevant information and upload your file. If you have problems with this, please let me know. This link can also be found in the "Resources" section of the course web page at the bottom.

Grading

		points
style/comments		20
	if/while/for	
	variable naming	
	comments/docstrings	
	formatting	
	parameters	
	additional functions	
	representation choices	
	misc	
functionality		
	board displays correctly	4
	board is randomly initialized	4
	loops until game completed	3
	properly handles match	4
	properly handles non-match	4
	handles improper user input	4
	times game	3
	displays top 5 after winning	4
	updates top 5 after game	3
	top 5 persists from game to game	3
Required extra add-ons		2
extra credit		2
total		58 (+2)