Do one or more of the following problems. Each will earn you **0.5 points of Extra Credit** unless otherwise indicated. At the start of each of these problems, the name of a Python file is given in **blue**: **foo.py**. You should create and save the requested Python program source code in a file with the same name. Also add a comment at the top of each file giving your name. No starting code!

When finished, upload each **.py** file with the specified name to the Canvas **EC-1 Assignment** link. Also specify which problems you are submitted via a text comment in Canvas.

**[EC1-1]** (**exp\_add\_only.py**) Define a function **exp\_by\_adding(base,exp)** with **int** parameters **base** and **exp**, and which returns the value of **base\*\*exp**. HOWEVER: you cannot use any arithmetic operator except **+** in writing your function. No **\*\*'s** nor **\***'s are allowed! Also, you cannot use any function from any imported module.

Then define your **main()** function which reads integers **a\_base** and **an\_exp** from the user, calls **exp\_by\_adding(a\_base,an\_exp)**, and prints out the returned value. The last line of your program should be a call to **main()**.

**[EC1-2]** (**theorems.py**) Because there are only 4 different values **True**, **False** for each of two **bool** variables **A**, **B**, you can verify the following Boolean identities by substituting each of the four possible values **(A,B)** into the identity, then printing out the entire expression:

**A and (A or B) == A** (Covering Theorem 1: **cover\_1(A,B)**)

**A or (A and B) == A** (Covering Theorem 2: **cover\_2(A,B)**)

**(A and B) or (A and not B) == A** (Combining Theorem 1: **combine\_1(A,B)**)

**(A or B) and (A or not B) == A** (Combining Theorem 2: **combine\_2(A,B)**)

**not (A or B) == (not A) and (not B)** (De Morgan's Theorem 1: **morgan\_1(A,B)**)

**not (A and B) == (not A) or (not B)** (De Morgan's Theorem 2: **morgan\_2(A,B)**)

If the identities are correct, you should get an output of **True** in each of the four cases.

Do so by writing a Python program that does this. First, create Boolean functions for each of the above 6 theorems, with names and parameters as shown. Within the function body of each, return the value obtained by evaluating the given expression.  
  
Then write a **for** loop that generates each of the four possible **bool** value combinations for **(A,B)**. (We'll talk about several ways of doing this within class.) Within your loop body, print out the value returned from calling each of the above 6 functions.

**[EC1-3]** (**patterns3.py)** More fun with \* patterns... Each of these is worth **0.25 points**, except for (g) and (h), which are worth **1 point** each. Write a program that reads an **int N >= 0**, then prints out each of the following patterns of **\***. Here, you may use the \* repetition operator, if you wish. Also, each pattern must be output via a single function call to the given named function with single parameter **N**. Examples for **N==4** follow, with explanations of how the displayed diagram reflects this value of **N**:

**(a) def checkerboard(N)** => **"Checkerboard"** of **\*** and **-** of side **N:** first row starts with **\*,** then alternates **-** and **d**. Second row starts with **-**, then alternates \* and -. Thus, adjacent lines start with alternating symbols.

**\*-\*-**

**-\*-\***

**\*-\*-**

**-\*-\***

**(b) def vee(N)** => **"V"** with **N** rows**:**

**\* \***

**\* \***

**\* \***

**\***

**(c) def x(N)** => **"X"** with 2\***N-1** rows**:**

**\* \***

**\* \***

**\* \***

**\***

**\* \***

**\* \***

**\* \***

**(d)** **def solid\_diamond(N)** => 2\*N+1 lines, first with 1 star, second with 3 stars, ..., N+1 line with 2\*N+1 stars, then subsequent lines repeating the pattern, reversed; each line's stars are centered between first and Nth columns:

**\*  
 \*\*\*  
 \*\*\*\*\***

**\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*  
 \*\*\*\*\*  
 \*\*\*  
 \***

**(e)** **def hollow\_diamond(N)** => same as above, but with no interior stars (no 2 stars adjacent on same line):

**\*  
 \* \*  
 \* \*  
 \* \*  
\* \*  
 \* \*  
 \* \*  
 \* \*  
 \***

**(f) Square with number diagonals,** with outer side length of **N**, the main diagonal from upper left to lower right having **0**'s, and parallel "diagonals" above and below this main diagonal consisting of single digits **1,2,...,8,9** depending on how far each is from are from the main diagonal. The following example is for **N==9**; assume **N <= 10**.

**def number\_diagonals(N)**=> N== 9:

**012345678**

**101234567**

**210123456**

**321012345**

**432101234**

**543210123**

**654321012**

**765432101**

**876543210**

**(g) "Nested squares"** with outer side length of **2\*N+1** and each inner square spaced as shown. Note that any two \*'s, each from a different square, are separated by at least one blank. That is, each is neither adjacent to the other on the same line nor in the same column:

**def nested\_squares(N)**=> N==4, 2\*N+1 == 9: three nested squares

**\*\*\*\*\*\*\*\*\***

**\* \***

**\* \*\*\*\*\* \***

**\* \* \* \***

**\* \* \* \* \***

**\* \* \* \***

**\* \*\*\*\*\* \***

**\* \***

**\*\*\*\*\*\*\*\*\***

**(h) "Nested right triangles"** with outer legs each having **2\*N+1** \*'s and each inner triangle spaced as shown. As in the previous problem, note that any two \*'s, each from a different triangle, are neither adjacent on the same line or in the same column:

**def nested\_triangles(N)**=> N==4, 2\*N+1 == 9: two triangles

**\***

**\*\***

**\* \***

**\* \***

**\* \* \***

**\* \*\* \***

**\* \*\*\* \***

**\* \***

**\*\*\*\*\*\*\*\*\***

N==6, 2\*N+1 == 13: three triangles

**\***

**\*\***

**\* \***

**\* \***

**\* \* \***

**\* \*\* \***

**\* \* \* \***

**\* \* \* \***

**\* \* \* \* \***

**\* \* \* \***

**\* \*\*\*\*\*\*\* \***

**\* \***

**\*\*\*\*\*\*\*\*\*\*\*\*\***

**[EC1-4]** (**gene.py**) Write a program that reads a string from the user, representing a DNA molecule as a sequence of letters **A**, **C**, **G**, and **T**, with each letter representing a different base. Then determine if it represents a potential gene, by checking it satisfies each of the following 4-part codon criteria:

1. It begins with the start codon **ATG**, where a codon is a sequence of three bases.
2. Its length is a multiple of 3.
3. It ends with any one of the stop codons **TAG**, **TAA**, or **TGA**.
4. It has no intervening stop codons, anywhere in the codon sequence between the first and last codons. That is, no intervening stop codons are allowed which start on an index evenly divisible by 3.

Do this by first defining a Boolean function **is\_valid\_DNA(seq)** that returns **True** if each character in **seq** is one of **A**, **C**, **G**, or **T**. Otherwise return **False**.

Next, define a Boolean function **is\_gene(dna)** that returns **True** when **dna** is valid DNA and such that **dna** satisfies each of the 4 codon criteria given above.

Then complete your program: use your **is\_valid\_DNA(seq)** function to validate the input string **seq**'s contents, printing **Not valid DNA** if it contains any characters other than one of **A**, **C**, **G**, and **T.** Otherwise, call your **is\_gene()** function and print **Is potential gene** if it returns **True** (it satisfies the previous 4-part codon criteria), and **Is NOT potential gene** otherwise.

Finally, if the input string is not valid DNA or violates the 4-part codon criteria, print out some kind of diagnostic information that describes details on why it's not valid. This information might describe why the contents are invalid DNA, such as which bases are illegal and at what sequence position they occur. For the 4-part codon criteria, it might be a description such as "Doesn't start with ATG" or "Length isn't a multiple of 3", and so forth.

**[EC1-5]** (**poly1.py)** Represent a polynomial **p(x)** as a list **L** of the **float** coefficients of each term. **L[0]** is the constant (**x0**) term of **p(x)**, **L[1]** is the **x1** coefficient, **L[2]** is the **x2** coefficient, and so forth. Example: **x2+3x+2** = **1.0\*x2+3.0\*x1+2.0\*x0** is represented by [**2.0, 3.0, 1.0].**

Write a program which reads two polynomials **p(x)** and **q(x)** from the user as lists of their coefficients, then prints out the list equivalent of (a) **p(x)+q(x)** and (b) **p(x)\*q(x)**.

Read the two polynomials as follows. Prompt the user to enter the coefficients for **p(x)** as **L[0]**, **L[1]**, **L[2]**,... with the user entering an empty string (**ENTER**) as a no-more-data sentinel. Then do the same for **q(x)**.

**[EC1-6]** (**pascal.py)** Pascal's triangle is a series of rows of integers, with the first row having a single **int 1**, and each subsequent row having an integer P that is the sum of the two **int** values **X** and **Y** in the line above it. **Y** is immediately above the number in the previous row, and **X** is number immediately to the left of **Y**. If **X** and **Y** are blanks or non-existent, treat them as 0.

Write a program that reads **int N** from the user, then prints out the first **N** rows of Pascal's triangle.

Here is what your program should print for **N==7**: the first seven rows of Pascal's triangle, formatted as just described.

**1**

**1 1**

**1 2 1**

**1 3 3 1**

**1 4 6 4 1**

**1 5 10 10 5 1**

**1 6 15 20 15 6 1**

**[EC1-7]** (**strip\_comments.py**) Write a program that reads a string **fname** from the user, then opens the Python source code file **fname.py** with that name. Copy each of its lines into a new file named '**strip\_'** **+** **fname.py**, deleting all Python comments as you go. Recall these begin with a **#** anywhere on a line and continue to the end of the line. Your output **.py** file should still be a valid program. Be careful: you shouldn't delete comments within strings.

**[EC1-8]** (**test\_no\_dict.py**) Redo each of your five [**H8-2]** **Bool** functions that check for a particular poker hand, but **WITHOUT** using a dictionary. Thus, each of the passed arguments should be the 5-card list **hand** to check (a list of **Card** references):

**hasOnePair(hand)**

**hasTwoPairs(hand)**

**hasThreeOfAKind(hand)**

**hasFourOfAKind(hand)**

**hasFullHouse(hand)**

Your three **[H8-2]** PyTest tests for each the above functions should still pass, so I can run the tests to verify that no bugs are reported.

**[EC1-9]** (**student.py**) This is worth **1 point** of EC. Create a new class **Student.** Its instances should track the name and quiz scores taken by the student, including the number of quiz scores that are dropped and the resulting average quiz score. Use a list \_**scores** of float values as a **Student** attribute (field), with **\_scores[I-1]** stored as the score of Quiz **I**.

Here are the **Student** methods you should define:

* A constructor with a single argument, setting the name of a newly-initialized **Student**.
* Instance method **addScore(self,score)**, which adds another quiz score to the collection within the **Student**
* Instance method **getNumberQuizzes(self)** which should return the total number of quiz scores
* Instance method **calcQuizStats(self,dropNumber)**, which should calculate and set the **Student** attribute (field) **\_average**, after calculating the **float** average (mean) score after dropping the lowest **dropNumber** quiz scores. If **dropNumber** is greater than or equal to the total number of quiz scores, then calculate the average as **0.0**. This method should also set the field **\_dropNumber**
* Instance method **getAverage(self)** which should return the value of **\_average**
* Instance method **\_\_str\_\_(self)**, which should create a string representation of the student, giving the name, quiz average **\_average**, number of dropped quizzes **\_dropNumber**, and the list of each quiz score, with dropped scores marked with a **\***. Example (your output should show the same information, though your formatting may differ):   
    
  **Name: Moxie Berner  
    
  Quiz Average: 47.0**  
  **Number of Quizzes: 4  
  Dropped Quizzes: 2**  
   **Quiz Scores (\* => dropped):  
  1 - 64.0  
  2 - 30.0  
  3 - 10.5 \*  
  4 - 20.0 \***

In the **main()** method of your class, create two **Student** instances, add some quiz scores to each, calculate the quiz stats, and print out the string equivalent of each resulting instance. One of your students should have the same information as shown above, but you can choose your own information for the second.