CS 116 Winter 2013

Assignment 05
Due at 10:00 am on Wednesday, February 27

- All solutions are to be in Python.
- Read and follow the instructions in the Python Style Guide for CS116 available on the course Web page.
- You are encouraged to use helper functions in your solutions as needed. Include them in the same file as your solution but make helper functions separate functions from the main function. Do **not** make them local functions. You do not need to provide examples and tests for these helper functions, but as usual you must provide contracts and purpose statements.
- While you may use global constants in your solutions, do not use global variables for anything other than testing.
- Download the testing module from the course Web page. Include "*import check"* in each solution file.
- You may import and use the math module if needed for any questions. Do not import any other modules.
- Be sure to use strings that exactly match those specified on the assignment and the interface. Changing them in any way may result in the loss of correctness marks.
- Do **not** use Python iteration (loops) or lists. Any repetition should be implemented using recursion.
- Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include reference to the parameter names of your functions.
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources. Do not email or share your code with any of your fellow students.
- Read the course Web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the style guide. Specifically, your solutions should be placed in files a05qY.py, where Y is a value from 1 to 4.
- Download the interface file from the course Web page.
- For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe, including the definition of constants and helper functions where appropriate.
- Read each question carefully for restrictions before posting on piazza. Test data for all questions will always meet the stated assumptions for consumed values.
- Assignments will not be accepted through email. Course staff will not debug code emailed to them.
- Late assignments for any reason are not accepted

Coverage: Module 5

1. We studied Fibonacci numbers in Module 03. Fibonacci numbers are defined as $F_n = F_{n-1} + F_{n-2}$ with base cases $F_0 = 0$ and $F_1 = 1$. In Python, recursive implementations of Fib(n) may become limited by the depth of the recursion as n gets large. Therefore, approximating the n^{th} Fibonacci number can be a useful tool, particularly when an approximation algorithm does not involve computation of $F_{n-1} + F_{n-2}$. For this question you will implement two algorithms in python that predict the n^{th} Fibonacci number in a Fibonacci number sequence.

   a) Implement an accumulatively recursive version of Fibonacci numbers, called `acc_fib` which uses the technique of `fib3` from Module 03, slide 31. Your function should take in a positive integer, `n` and produce the n^{th} Fibonacci number. You will need to use an accumulative helper function for `acc_fib`. As this is a recursive function, you do not need to test for values of `n` greater than 900.
   Note: You must use the algorithm as given in `fib3` for this implementation.

   b) Related to Fibonacci numbers is a figure called the ***Golden Ratio*** which can help to generate the n^{th} Fibonacci number excluding the need of F_{n-1} and F_{n-2}. The golden ratio can be computed as follows:

$$\frac{1 \pm \sqrt{5}}{2}$$

Assignment 05
Due at 10:00 am on Wednesday, February 27

We let the positive generated value be represented by φ *(Phi)* and the negative value represented by *τ (Tau)*.

$$\varphi = \frac{1+\sqrt{5}}{2} \quad , \quad \tau = \frac{1-\sqrt{5}}{2} \tag{1.1}$$

These two numbers are used to approximate the n^{th} Fibonacci number in a Fibonacci number sequence according to the formula:

$$F_n = (\varphi^n - \tau^n) / \sqrt{5} \tag{1.2}$$

Write a python function `approx_fib` that takes in a positive integer `n`, and produces an *integer* approximation to the n^{th} Fibonacci number based on (1.2). You must compute the values of φ and *τ* as given in (1.1). The final value produced by the calculation in (1.2) will be a float, however you must use integer type casting in order to produce a final integer output.

Testing: The approximation begins to get inaccurate for Fibonacci numbers greater than or equal to a particular integer `m`. In your testing of `approx_fib`, try to find the value of `m` such that `approx_fib` begins to lose accuracy in approximating the n^{th} Fibonacci number. Assuming that `acc_fib` is correct from part(a), you can compare your values generated by `acc_fib` to those generated by `approx_fib`. Note: You will not be marked for this comparative testing, or for finding the correct value of `m`. However, you are expected to test each part(a) and part(b) separately.
For example:

```
acc_fib(3) => 2
approx_fib(10) => 55
```

2. Write a Python function `trip_cost` which calculates the total cost for an entire elementary school to go on a school trip to the Museum. The function consumes the number of tickets for children in different age groups, the number of adult tickets, the number of teachers that are members and the time of day that the trip is taking place.

   If there are more than 30 children in any age group, only the first 30 children pay for admission – but those in excess of 30 are admitted free. This does not apply to adults. Some teachers may have membership cards for which they receive a 40% discount off their ticket price. The time of day can be ("`Day`" or "`After_3`") and represents when the school is visiting the Museum. "`Day`" prices are as listed below. All tickets purchased at the "`After_3`" time of day receive a 50% discount, however in this case the teacher membership discount does not apply. You may assume schools in Ontario pay 13% tax on all tickets for children and adults 12 and up ☺. So this must be calculated in `trip_cost` after all other discounts have been applied.
   Age groups and corresponding ticket values are as follows:

Assignment 05
Due at 10:00 am on Wednesday, February 27

`Day` Prices	**Group**	**Age Range**	**Cost**
	Group_A	Ages 6 years and under	$5.00
	Group_B	Ages 7 to 11	$8.00
	Group_C	Ages 12 to 17	$10.00 *
	Adult_Tickets	Ages 18 and up	$12.00 *
Note:	Adult members may have membership 40% discount card		
`After_3`	Discount 50% on each full price ticket		
			*13% sales tax applies after all other discounts have been applied

For example, as listed in the first case below, a group containing 40 children in `group_a`, 55 children in `group_b`, 29 children in `group_c`, and 33 `adults` (13 of whom are `members`) who attend the museum `After_3`, will have total cost of $582.59.
For example,
`trip_cost(group_a, group_b, group_c, adults, members, time)`
`trip_cost` (40, 55, 29, 33, 13, "`After_3`") => 582.59
`trip_cost` (20, 15, 29, 35, 05, "`Day`") => 995.18
`trip_cost` (40, 40, 40, 40, 0, "`Day`") => 1271.40

3. Have you ever wondered, how many rubik's cubes it would take to reach the top of the Mathematics and Computers Building (MC) if you stacked them all one on top of another? Well, now is your chance to find out☺. Our estimated height of MC is 24.5 meters. A rubik's cube is a cube measuring approximately .057 meters on each side. You will write two versions of a function `get_numrc` that consumes one float greater than zero, `height`, representing the height of a building in meters, and produces the total number of rubik's cubes needed to reach that height, if they were stacked one on top of another.

   You will test `get_numrc` with varying heights. The output must be the smallest integer **X**, such that (total height of **X** rubik's cubes) >= `height`. For example, if your calculations indicate that 193.2 rubik's cubes are needed for a building of height 11m, then the function must produce 194.
   Your function must take into consideration a compression factor that increases for rubik's cubes closer to the bottom. For instance, if there is one rubik's cube on top of another rubik's cube, it will be compressed by a factor of (1-1/100000). If any given rubik's cube has n rubik's cubes on top of it, it's height will be compressed by $(1-1/100000)^n$ times it original height.

   a) Write a recursive function, `get_numrc_a`, which calculates the number of rubik's cubes needed to reach the top of a building with height = `height`, and a compression factor for each rubik's cube ($(1-1/100000)^n$). As this is a recursive function, you do not need to test with heights greater than 50m.
   b) The total number of rubik's cubes needed, `total` can also be calculated by the formula in (3.1).

$$\texttt{total} = \log(1-\texttt{height}*(1-\texttt{comp})/\texttt{h\_rc})/\log(\texttt{comp}) \quad (3.1)$$

where `h_rc` is the height of a rubik's cube, `comp` is the compression factor of (1-1/100000), and `height` is the height of any given building. Write a non-recursive function `get_numrc_b` which has the same input and output as part(a). Note: that `height` and `h_rc` must be in the same measurement units for this formula to work correctly. You may use `math.log`, and assume all logarithms are natural logarithms (*base e*). The tests for part(a) and part(b) should produce the same integer result. However, in part(b) you may test with heights up to 1000m.

For example:

```
get_numrc_a(42.5) =>  749
get_numrc_b(20) => 352
get_numrc_b(400) => 7276
```

4. Consider a dice battle game called *What-Are-the-Odds?* It is a team battle game where team_ab plays against team_cd and two players are on a team. Each player rolls a die and adds their roll value to their teammate's value. The first team_ab, will roll values represented by `a` and `b`. The second team_cd, will roll values represented by `c` and `d`. Let the sum of the dice values for team_ab be represented by **AB**, and the sum of the dice values for team_cd be represented by **CD**.

   Rules of battle are as follows and apply to either team.
   - Die values will be integers from 1 to 6, inclusive
   - If only one of **AB** and **CD** are odd: The team with the odd number wins.
   - If **AB** and **CD** are both odd: The largest odd number wins. If there is a tie, the individual rolls of `a`, `b`, `c`, and `d` must be examined. The team with the largest odd number roll wins.
   - If exactly one of **AB** or **CD** is an even number <= 6: The team with the even number loses.
   - If **AB** and **CD** are both even and <=6: The team with **AB** or **CD** that is divisible by the largest odd number wins.
   - If **AB** and **CD** are both even and >6: The team with **AB** or **CD** that is divisible by the largest odd number wins.
   - A tie will occur in all situations in which a winner is not determined.

Write a python function, `win_odds`, that consumes 4 dice values, `a`, `b`, `c`, or `d` (integers 1-6), and produces the winning team represented by a string ("`AB`", "`CD`", or "`ABCD`" when there is a tie). Some examples follow:

```
win_odds(2, 3, 6, 6) => "AB"
win_odds(2, 3, 5, 6) => "CD"
win_odds(2, 2, 1, 3) => "ABCD"
win_odds(1, 1, 1, 2) => "CD"
win_odds(3, 5, 5, 5) => "CD"
win_odds(2, 4, 2, 2) => "AB"
win_odds(1, 5, 1, 3) => "AB"
win_odds(6, 6, 4, 4) => "AB"
```