CSE 101: Computer Science Principles (Fall 2019)

Lab #2

Assignment Due: Saturday, September 28, 2019, by 11:59 PM

Assignment Objectives

By the end of this assignment you should be able to develop original Python functions to solve simple programming problems involving conditional statements and loops.

Getting Started

This assignment requires you to write Python code to solve several computational problems. To help you get started, we will give you a basic starter file for each problem. These files will contain *function stubs* and a few tests you can try out to see if your code seems to be correct (**note that the test cases we give you in the starter files are just examples; we will use different test inputs to grade your work!**). You need to complete (fill in the bodies of) these functions for the assignments. **Do not, under any circumstances, change the names of the functions or their parameter lists.** The automated grading system will be looking for functions with those exact names and parameter lists; if you change any function headers, the grading program will reject your solution and mark it as incorrect.

Directions

Solve each of the following problems to the best of your ability. We have provided a code skeleton for each of the programming problems. The automated grading system will execute your solution to each programming problem several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. This assignment contains 2 problems, and is worth a total of 30 points. **Note that not every problem may be worth the same number of points!**

- ⚠ Each starter file has a comment block at the top with various important information. Be sure to add your information (name, ID number, and NetID) to the first three comment lines, so that we can easily identify your work. **If you leave this information out, you may not receive credit for your work!**
- ⚠ Submit your work as a set of individual files (one per problem). **DO NOT** zip or otherwise compress your files, and **DO NOT** place all of your work in a single file. If you do so, we may not be able to grade your work and you will receive a failing grade for this assignment!
- ⚠ Every function **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).
- ⚠ Every function must explicitly ***return*** its final answer; the grading program will ignore anything that your code prints out. Along those lines, do **NOT** use `input()` anywhere within your functions (or anywhere before the `if __name__ == "__main__":` statement); your functions should get all of their input from their parameters. Programs that crash will likely receive a failing grade, so test your code thoroughly **with Python 3.7.4 or later** before submitting it.
- ⚠ Blackboard will provide information on how to submit this assignment. You **MUST** submit your completed work as directed by the indicated due date and time. We will not accept (or grade) any work that is submitted after the due date and time, or that is submitted before you have signed the course's Academic Honesty agreement.
- ⚠ **ALL** of the material that you submit (for each problem) **MUST** be your own work! You may not receive assistance from or share any materials with anyone else, except for the instructor and the (current) course TAs.

Part I: Terra Mystica (20 points)

(Place your answer to this problem in the "mystica.py" file)

In the board game *Terra Mystica*, each player has exactly 12 power units to spend performing actions in the game. Power units are distributed across three "bowls", labeled I, II, and III (the sum of units in these bowls is always exactly 12). As the game progresses, power is gained or spent by shifting units between bowls as follows:

- To spend a unit of power, move it from Bowl III to Bowl I (as long as you have at least one unit in Bowl III). If Bowl III is empty, nothing happens.
- When a player gains a unit of power:
  - If all 12 units are already in Bowl III, nothing happens. Otherwise:
  - If there is a unit in Bowl I, move it to Bowl II.
  - Otherwise, if Bowl I is empty but there is a unit in Bowl II, move one unit from Bowl II to Bowl III.
  - If Bowls I and II are both empty, then nothing happens (this means thatall 12 units are already in Bowl III).

For example, consider the following sequence of actions (Bowls are listed in order: I, then II, then III):

```
               I   II  III
Initial:       5 |  7 |  0
Gain  3  ==>   2 | 10 |  0   (move 3 units from I to II)
Gain  6  ==>   0 |  8 |  4   (move 2 power units from I to II,
                              then move the remaining 4 units from II to III)
Gain  7  ==>   0 |  1 | 11   (move 7 units from II to III)
Spend 4  ==>   4 |  1 |  7   (move 4 units from III to I)
Gain  1  ==>   3 |  2 |  7   (move 1 unit from I to II)
```

Complete the Python function `mystica()`, which takes four integer parameters. The first three parameters represent the initial number of units in Bowl I, Bowl II, and Bowl III, respectively. The fourth parameter is a non-zero value that represents the number of power units being gained or spent (a positive value indicates that power is being gained, and a negative value indicates that power is being spent). If the first three parameters do not add up to exactly 12, the function returns -1 to signal an error. Otherwise, it performs the power transfer according to the rules above and then returns an integer equal to the final value stored in Bowl I (the first bowl). ***You may assume that the total number of power units being spent will always be less than or equal to Bowl III's starting value.***

Hint: When you perform the transfer, use an `if-else` statement where one clause handles spending power units and the other handles gaining power units. If you need to gain multiple units, use a loop to process them one at a time according to these rules, in the order listed above (for example, start by processing the contents of Bowl III, then Bowl I, and finally Bowl II). Don't forget that a **negative** power change value (to spend points) **subtracts** from Bowl III but **adds** to Bowl I.

Examples:

Function Call	**Return Value**
`mystica(2, 3, 7, 4)`	`0`
`mystica(3, 4, 5, -3)`	`6`
`mystica(0, 1, 11, -10)`	`10`
`mystica(3, 6, 2, 1)`	`-1`

Part II: Abundant Numbers (10 points)

(Place your answer to this problem in the “abundant.py” file)

An *abundant number* is an integer that is less than the sum of its *perfect divisors* (a perfect divisor is a value that divides a number evenly but is strictly less than the number itself). For example, 12 is an abundant number because its perfect divisors (1, 2, 3, 4, and 6) add up to 16.

Complete the `abundant()` function, which takes a single positive integer value *n* as its argument. The function returns a `Boolean` value (either `True` or `False`) indicating whether the function parameter is an abundant number. For example, `abundant(12)` returns `True` because 12 is an abundant number, but `abundant(14)` returns `False` because 14 is not.

Hint: Use a loop and one or more helper variables to solve this problem. Start by adding up the perfect divisors of the parameter, then compare that sum to the original value.

Examples:

Function Call	**Return Value**
`abundant(12)`	`True`
`abundant(14)`	`False`
`abundant(28)`	`False`
`abundant(40)`	`True`