15-112: Introduction to Programming and Computer Science, Spring 2020

Homework 2: Expressions and Conditionals

Due: Tuesday, January 28, 2020 by 22:00

To start this homework....

1. Create a folder named "week2"
2. Download hw2.py to that folder
3. Edit hw2.py and modify the functions as required
4. When you have completed and fully tested hw2, submit hw2.py to Autolab. For this hw, you may submit up to 20 times (which is way more than you should require), but only your last submission counts.

Some important notes:

1. After you submit to Autolab, make sure you check your score. If you arenâĂŹt sure how to do this, then ask a CA or Professor.
2. There is no partial credit on Autolab testcases. Your Autolab score is your Autolab score.
3. Read the last bullet point again. Seriously, we wonâĂŹt go back later and increase your Autolab score for any reason. Even if you worked really hard and it was only a minor error...
4. Do not use string indexing, loops, lists, list indexing, or recursion this week. The autograder will reject your submission entirely if you do.
5. Do not hardcode the test cases in your solutions.
6. The starter hw2.py file includes test functions to help you test on your own before you submit to Autolab. When you run your file, problems will be tested in order. If you wish to temporarily bypass specific tests (say, because you have not yet completed some functions), you can comment out individual test function calls at the bottom of your file in main(). However, be sure to uncomment and test everything together before you submit! Ask a CA if you need help with this.
7. Remember the course's academic integrity policy. Solving the homework yourself is your best preparation for exams and quizzes; cheating or short-cutting your learning process in order to improve your homework score will actually hurt your course grade long-term.

isEvenPositiveInt

Task 1 (3 pt) *Write the function isEvenPositiveInt(n) which, given a value n, returns True if it is even, positive, and an integer, and False otherwise.*

isPerfectSquare(n)

Task 2 (3 pts) *Write the function isPerfectSquare(n) that takes a possibly-non-int value, and returns True if it is an int that is a perfect square (that is, if there exists an integer m such that m**2 == n), and False otherwise. Do not crash on non-ints nor on negative ints.*

numberOfPoolBalls

Task 3 (3 pts) *Pool balls are arranged in rows where the first row contains 1 pool ball and each row contains 1 more pool ball than the previous row. Thus, for example, 3 rows contain 6 total pool balls (1+2+3). With this in mind, write the function numberOfPoolBalls(rows) that takes a non-negative int value, the number of rows, and returns another int value, the number of pool balls in that number of full rows. For example, numberOfPoolBalls(3) returns 6. We will not limit our analysis to a "rack" of 15 balls. Rather, our pool table can contain an unlimited number of rows. Hint: you may want to briefly read about Triangular Numbers. Also, remember not to use loops!*

getInRange

Task 4 (4 pts) *Write the function getInRange(x, bound1, bound2) which takes 3 int or float values – x, bound1, and bound2, where bound1 is not necessarily less than bound2. If x is between the two bounds, just return it unmodified. Otherwise, if x is less than the lower bound, return the lower bound, or if x is greater than the upper bound, return the upper bound. For example:*

- *getInRange(1, 3, 5) returns 3 (the lower bound, since 1 lies to the left of the range [3,5])*
- *getInRange(4, 3, 5) returns 4 (the original value, since 4 is in the range [3,5])*
- *getInRange(6, 3, 5) returns 5 (the upper bound, since 6 lies to the right of the range [3,5])*
- *getInRange(6, 5, 3) also returns 5 (the upper bound, since 6 lies to the right of the range [3,5])*

getKthDigit

Task 5 (4 pts) *Write the function getKthDigit(n, k) that takes a possibly-negative int n and a non-negative int k, and returns the kth digit of n, starting from 0, counting from the right. So:*

```
getKthDigit(789, 0) == 9
getKthDigit(789, 1) == 8
getKthDigit(789, 2) == 7
getKthDigit(789, 3) == 0
getKthDigit(-789, 0) == 9
```

setKthDigit

Task 6 (4 pts) *Write the function setKthDigit(n, k, d=0) that takes three integers – n, k, and d – where n is a possibly-negative int, k is a non-negative int, and d is a non-negative single digit (between 0 and 9 inclusive) with a default value of 0. This function returns the number n with the kth digit replaced with d. Counting starts at 0 and goes right-to-left, so the 0th digit is the rightmost digit. For example:*

```
setKthDigit(468, 0, 1) == 461
setKthDigit(468, 1, 1) == 418
setKthDigit(468, 2, 1) == 168
setKthDigit(468, 3, 1) == 1468
setKthDigit(468, 1) == 408
```

threeLinesArea and helpers

Task 7 (6 pts) *Write the function threeLinesArea(m1, b1, m2, b2, m3, b3) that takes six int or float values representing the 3 lines:*

*y = m1*x + b1 y = m2*x + b2 y = m3*x + b3*

First find where each pair of lines intersects, then return the area of the triangle formed by connecting these three points of intersection. If no such triangle exists (if any two of the lines are parallel), return 0.

To do this, you must write three helper functions:

- ***lineIntersection(m1, b1, m2, b2)*** *to find where two lines intersect (which you will call three times)*
  - *This function takes four int or float values representing two lines and returns the x value of the point of intersection of the two lines. If the lines are parallel, or identical, the function should return None.*
- ***distance(x1, y1, x2, y2)*** *to find the distance between two points (again called three times)*

  - *This function takes four int or float values representing two points and returns the distance between those points.*

- ***triangleArea(s1, s2, s3)*** *to find the area of a triangle given its side lengths (which you will call once).*
  - *This function takes three int or float values representing side lengths of a triangle, and returns the area of that triangle. To do this, you may wish to to use Heron's Formula.*

You may write other helper functions if you think they would be useful, but you must at least write these three exactly as described, and then you must use them appropriately in your solution. Once you have written and tested your helper functions, then move on to writing your threeLinesArea function, which of course should use your helper functions. That's the whole point of helper functions. They help!

Note that helper functions help in several ways. First, they are logically ***simpler****; they break down your logic into smaller chunks that are easier to reason over. Second, they are independently* ***testable****, so you can more easily isolate and fix bugs. And third, they are* ***reusable****, so you can use them as helper functions for other functions in the future. All good things!*

colorBlender

Task 8 (8 pts) *This problem implements a color blender, inspired by this tool. In particular, we will use it with integer RGB values (it also does hex values and RGB% values, but we will not use those modes). Note that RGB values contain 3 integers, each between 0 and 255, representing the amount of red, green, and blue respectively in the given color, where 255 is "entirely on" and 0 is "entirely off".*

For example, consider this case. Here, we are combining crimson (rgb(220, 20, 60)) and mint (rgb(189, 252, 201)), using 3 midpoints, to produce this palette (using our own numbering convention for the colors, starting from 0, as the tool does not number them):

```
color0: rgb(220,  20,  60)
color1: rgb(212,  78,  95)
color2: rgb(205, 136, 131)
color3: rgb(197, 194, 166)
color4: rgb(189, 252, 201)
```

There are 5 colors in the palette because the first color is crimson, the last color is mint, and the middle 3 colors are equally spaced between them.

So we could ask: if we start with crimson and go to mint, with 3 midpoints, what is color #1? The answer then would be rgb(212, 78, 95).

One last step: we need to represent these RGB values as a single integer. To do that, we'll use the first 3 digits for red, the next 3 for green, the last 3 for blue, all in base 10 (decimal, as you are accustomed to). Hence, we'll represent crimson as the integer 220020060, and mint as the integer 189252201.

With all that in mind, write the function colorBlender(rgb1, rgb2, midpoints, n), which takes two integers representing colors encoded as just described, a non-negative integer number of midpoints, and a non-negative integer n, and returns the nth color in the palette that the tool creates between those two colors with that many midpoints. If n is out of range (too small or too large), return None.

For example, following the case above: colorBlender(220020060, 189252201, 3, 1) returns 212078095

***Hint:** RGB values must be ints, not floats. When calculating midpoint colors, you can mostly use the built-in round function. However, the built-in round function has one major flaw: it varies in whether it chooses to round .5 up or down (ugh!). You can fix this by doing an extra check for whether a number is <number>.5 and choosing to always round up in that case.*