CSCI 121: Computer Science Fundamentals I
(Practice) Midterm Exam – Spring 2025

This is a practice exam for the in-class midterm that will be held on Monday, March 17th.

----

1. For each of the following Python scripts, write what the script would output to the console. If some script line causes an error, write *Error!* and stop running the script, similar to what Python would do.

   (a)
   ```
   a = "33"
   b = "3"
   print(b * 4)
   print(int(b) * 4)
   print(int(b) ** 2)
   print(b * 2 == a)
   print(a // b)
   print(a % b)
   ```

   (b)
   ```
   i = 20
   while i > 5:
       if i % 2 == 1:
           print(i)
       i = i - 3
   ```

   (c)
   ```
   def f(a,b):
       print(str(a) + " " + str(b))
       a = a * 10
       return a + b

   x = 1
   y = f(x,3)
   print(y)
   print(x)
   z = f(x,f(y,2))
   print(z)
   print(y)
   print(x)
   ```

   (d)
   ```
   def g(xs):
       tmp = xs[1]
       xs[1] = xs[0]
       xs[0] = tmp
       return xs
   bs = [1,2,3]
   cs = g(bs)
   print(bs)
   print(cs)
   ds = g(bs)
   print(bs)
   print(cs)
   ```

2. A friend of yours likes to track his fuel economy when they go on long car trips. For each leg of the trip, they write down the number of miles traveled and the gallons of fuel used on that leg. Each leg is a positive number of miles long and uses some fuel.

   Write a Python script that helps them compute their gas mileage on that trip. It takes a series of console inputs, the miles traveled and the gallons of fuel used for each leg of the trip.

   The input loop should end when they enter a non-negative number of miles for a leg. (It shouldn't request the gallons for that entry.)

   The script should output the gas mileage of the entire trip in miles per gallon. For example, if the result is 24.8 miles per gallon, the script should output the line

   ```
   24.8 MPG
   ```

   You can assume that they'll enter the information for one or more legs of a trip, and that they will use a positive number of total gallons.

3. The Gregorian calendar was instituted in 1582. From that year onward a *leap year* is one whose number is a multiple of 4, unless it is a century year that is not divisible by 400. This means that 1600 and 2000 were leap years, but 1900, 1800, and 1700 were not. Also, 1200 was not, because it was before 1582. Write a function `leap_year` that returns `True` if an integer year is a leap year. It should return `False` otherwise.

   Only use integer calculations in determining this function's return value.

4. Consider a *lists of lists* containing only the values `1` and `0`, and where each list is the same length. For example

   ```
   [[0,1,0,1,0],[1,0,0,0,1],[0,1,1,1,0]]
   ```

   Write a function `star_matrix` that returns a string that, when printed, yields a rectangular array of `*` and `.` characters that illustrate the row pattern of the list of lists. For example:

   ```
   >>> sm = star_matrix([[0,1,0,1,0],[1,0,0,0,1],[0,1,1,1,0]])
   >>> sm
   '.*.*.\n*...*\n.***.\n'
   >>> print(sm)
   .*.*.
   *...*
   .***.

   >>>
   ```

   The printed string has a number of rows equal to the number of lists, with each row having the same number of columns as each lists' length, and with `1` entries depicted as `*` and `0` entries depicted as `.`.

   You can assume that the list of lists has at least one list, that each of its items is a list of the same length, that that length is one or longer, and that those list entries are either `1` or `0`.

5. Write a function `has_digit` that, when given a digit from 0 to 9 and a non-negative integer returns whether that digits is one of the digits in the decimal representation of the number. For example:

   ```
   >>> has_digit(7,3567342)
   True
   >>> has_digit(5,3467342)
   False
   >>> has_digit(4,4)
   True
   >>> has_digit(3,4)
   False
   >>> has_digit(2,1211211)
   True
   ```

   You should not use any string, list, or floating point operations to do its work.