CSCI 141 - Fall 2019
Assignment 3

Topics: Conditionals and Loops

Reminder: You can discuss this assignment with your peers. However, the answers to the questions and programming solution MUST be your own. You cannot copy another person's code, you cannot have another person tell you what code to type, etc. If any part of this is unclear, please come see me.

1 Fibonacci Numbers and the Golden Ratio

The Fibonacci sequence is an interesting mathematical curiosity. The beginning of the sequence looks like this:

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	...
0	1	1	2	3	5	8	13	...

You might be able to spot the pattern: after the first two terms are given (0 and 1), the next term in the sequence is always the sum of the prior two terms.

The Fibonacci sequence is simple to define and not too hard to compute, but it has some surprisingly deep mathematical properties. The Fibonacci sequence and related concepts show up in patterns in nature all the time (try searching the internet for "Fibonacci in nature" for some examples). One interesting property of the sequence is that the ratio between each pair of terms in the sequence (i.e., f_n/f_{n-1}) converges to the *golden ratio*, which is often written using the Greek letter ϕ ("phi"). The value of this constant is roughly 1.61803398875, although the digits keep going forever.

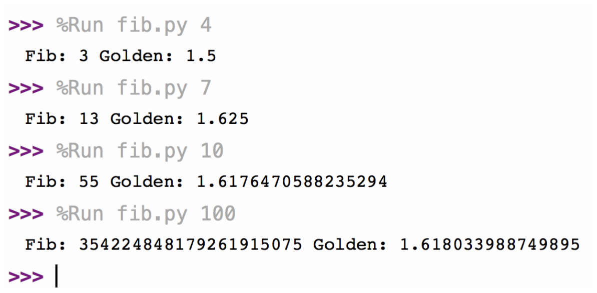
Write a program that takes a single integer **command line argument**, n, prints f_n, the nth Fibonacci number, then prints an estimate of the golden ratio computed using f_n and f_{n-1}. Notice that because the terms are numbered starting at zero, f_3 is actually the fourth number in the sequence, so be careful to get the correct one.

You may assume that n is greater than zero (you are not required to check for bad user input where $n <= 0$. When n is 1, the golden ratio calculation is $1/0$, so your program should print "infinity" for the golden ratio estimate when $n = 1$.

A couple sample outputs are given in Figure 1 below.

Testing

At this point you should know enough to test this program thoroughly. You are responsible for making sure it works for all valid inputs.



```
>>> %Run fib.py 4
 Fib: 3 Golden: 1.5
>>> %Run fib.py 7
 Fib: 13 Golden: 1.625
>>> %Run fib.py 10
 Fib: 55 Golden: 1.6176470588235294
>>> %Run fib.py 100
 Fib: 354224848179261915075 Golden: 1.618033988749895
>>> |
```

Figure 1: Sample runs of `fib.py`

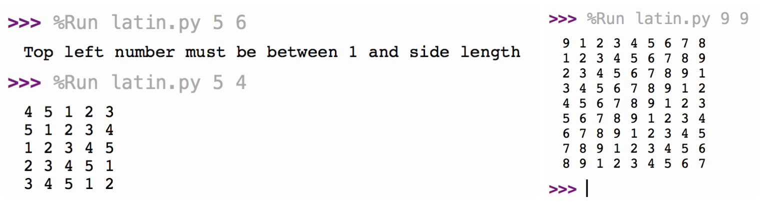
2 Latin Squares

A Latin Square is an n*n table filled with n different symbols in such a way that each symbol occurs exactly once in each row and exactly once in each column (see `http://en.wikipedia.org/wiki/Latin_square`). For example, here are two possible Latin Squares of order (i.e., side length) 4:

```
1 2 3 4        3 4 1 2
2 3 4 1        4 1 2 3
3 4 1 2        1 2 3 4
4 1 2 3        2 3 4 1
```

Your program should take two **command line arguments**. The first argument is an integer specifying the order (side length) of square. The second one is the top-left number of the square, which should be an integer between 1 and the order. If the second argument is not in this range, your program should print a message saying so and terminate. Your program will print the corresponding Latin Square.

Sample outputs are given in Figure 2 below.



```
>>> %Run latin.py 5 6
 Top left number must be between 1 and side length
>>> %Run latin.py 5 4
 4 5 1 2 3
 5 1 2 3 4
 1 2 3 4 5
 2 3 4 5 1
 3 4 5 1 2
```

```
>>> %Run latin.py 9 9
 9 1 2 3 4 5 6 7 8
 1 2 3 4 5 6 7 8 9
 2 3 4 5 6 7 8 9 1
 3 4 5 6 7 8 9 1 2
 4 5 6 7 8 9 1 2 3
 5 6 7 8 9 1 2 3 4
 6 7 8 9 1 2 3 4 5
 7 8 9 1 2 3 4 5 6
 8 9 1 2 3 4 5 6 7
>>> |
```

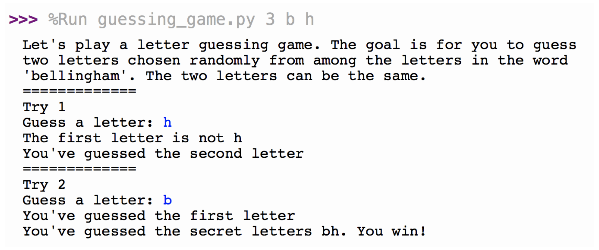
Figure 2: Sample runs of `latin.py`

2.1 Testing

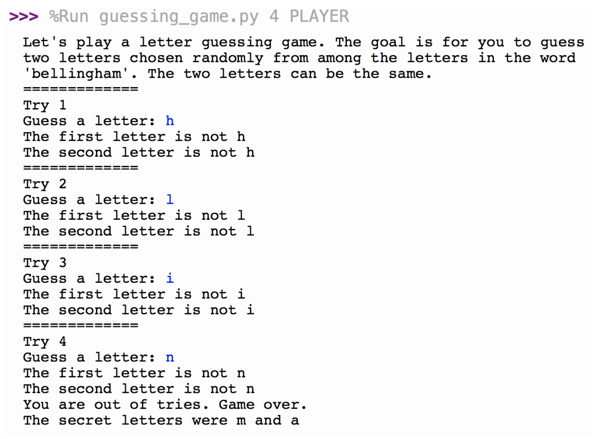
At this point you should know enough to test this program thoroughly. You are responsible for making sure it works for all valid inputs.

3 Guessing Game

You are a computer programmer working for a company, called *NostalgiaSoft*, that makes legacy (old-style, text-only) games for old people who were using computers in the early 1980s. The game you have been tasked to write is a simple guessing game. The program is run with a command line argument specifying how many guesses the player is allowed, and the player is given that many chances to guessing a secret two-character sequence. Because the game is to be marketed to alumni of Western Washington University, the letters are selected from the letters in the word **bellingham**. See the sample screen shots in Figure **??** for sample gameplay.



```
>>> %Run guessing_game.py 3 b h
  Let's play a letter guessing game. The goal is for you to guess
  two letters chosen randomly from among the letters in the word
  'bellingham'. The two letters can be the same.
  =============
  Try 1
  Guess a letter: h
  The first letter is not h
  You've guessed the second letter
  =============
  Try 2
  Guess a letter: b
  You've guessed the first letter
  You've guessed the secret letters bh. You win!
```



```
>>> %Run guessing_game.py 4 PLAYER
  Let's play a letter guessing game. The goal is for you to guess
  two letters chosen randomly from among the letters in the word
  'bellingham'. The two letters can be the same.
  =============
  Try 1
  Guess a letter: h
  The first letter is not h
  The second letter is not h
  =============
  Try 2
  Guess a letter: l
  The first letter is not l
  The second letter is not l
  =============
  Try 3
  Guess a letter: i
  The first letter is not i
  The second letter is not i
  =============
  Try 4
  Guess a letter: n
  The first letter is not n
  The second letter is not n
  You are out of tries. Game over.
  The secret letters were m and a
```

Figure 3: Two sample runs of the `guessing_game.py`. The left example is run in DEBUG mode, while the right example is in PLAYER mode.

Game Modes

To satisfy the strict guessing-game industry regulations, the game must be able to run in two modes: DEBUG mode and PLAYER mode. In DEBUG mode, the correct secret letters are given to the program via command line arguments; in PLAYER mode, the secret letters are chosen randomly by the program. Otherwise, the program's behavior is the same in both modes.

3.1 Command Line Arguments

To run in PLAYER mode, the program takes two command line arguments. To run in DEBUG mode, the program takes three command line arguments.

Guesses. In both modes, the first argument is an integer specifying the number of guesses the player is allowed. You may assume this number is nonnegative, but your program should be able to handle any number of guesses 0 or larger.

Game Mode. The second argument determines the game mode:

- If the second argument is "PLAYER", the game is in PLAYER mode and the secret letters are chosen randomly.

- If the argument is **anything else**, the game is in DEBUG mode and the program takes three arguments, the **second** and **third** of which give the two secret letters.

You should assume that the program is run with two arguments, the second of which is "PLAYER", or three arguments, the second and third of which are each length-1 strings containing letters from the word "bellingham". Your program is **not** required to verify that this is true before proceeding.

3.2 Pseudocode

Your manager has provided you with the following pseudocode that outlines the program's behavior:

1. The program prints a brief blurb that explains the game to the user.
2. If the game is in PLAYER mode, the program randomly chooses two secret letters from among the letters of the word `bellingham`. Because the letters are chosen independently, both the first and second secret letters might be the same. If the game is in DEBUG mode, the secret letters are the ones supplied in the second and third command line arguments.
3. Next, the guessing phase of the program begins. While the player still has guesses remaining (the initial number of guesses is given by the first command line argument), the game prompts the user to guess a letter.
4. For each of the two secret letters that has not already been correctly identified in a prior guess, the program prints a message stating whether the guess was correct or not. Notice that once a letter has been guessed correctly, output for subsequent guesses should not mention that letter. The user may guess the letters in any order (i.e., they do not need to guess the first letter correctly before guessing the second letter).
5. If the player has successfully guessed both letters, the program prints "You win" and terminates right away, even if the player has guesses remaining.
6. If the player has not exhausted their number of guesses, go back to step 3 (prompt for another guess).
7. If the player runs out of guesses before guessing both letters correctly, the program prints a message that tells the player they've run out of guesses and reveals the correct secret letters.

Write your program in a file called `guessing_game.py`. This game can be implemented many different ways. Declare and use as many variables as you need to keep track of guesses and secret letters. The logic for a sample "you lose" game play is shown below.

Program is run as: `python3 guessing_game.py 4 PLAYER`

Number of Guesses is 4

Program is in PLAYER mode, so secret letters are randomly chosen. Secret letters are: bh

User Guess 1: b

Game Response: You have guessed the first letter. The second letter is not b.

User Guess 2: g

Game Response: The second letter is not g.

User Guess 3: l

Game Response: The second letter is not l.

User Guess 4: e

Game Response : You are out of tries. Game over. The secret letters were b and h.

3.3 Testing

The advantage of implementing DEBUG mode is that it makes testing a lot easier–you can choose the secret letters carefully to test the various possible outcomes. In particular, you should be sure to test all combinations of guessing patterns and secret letter scenarios. Here's a non-comprehensive list of things to consider, just to get you started:

- The user guesses one letter correctly, then the other before they run out of guesses.
- The user guesses one letter correctly, but not the other before running out of guesses.
- The user guesses neither letter correctly before running out of guesses.
- The user is given zero, one, or many guesses
- The user guesses the letters in or out of order (first then second, second then first)
- The letters are the same and the user guesses them, or runs out of tries.
- Finally, test that PLAYER mode works too - make sure that the secret letters are generated randomly, all letters in `bellingham` are possible, and it's possible for the two letters to be the same.

Submission

Make sure your programs are thoroughly tested, check over the rubric, and upload `fib.py`, `latin.py`, and `guessing_game.py` to Canvas. Fill out the A3 Hours quiz on Canvas with an estimate of how many hours you spent on this assignment.

Rubric - 54 points total

Style Points (6 points)

Author, date, and program description given in a comment at the top of the file	3 points
Code is commented adequately and variables are appropriately named	3 points

`fib.py` (10 points)

n is supplied as the first command line argument	1 point
Correct Fibonacci number is printed	5 points
Correct Golden ratio estimate is printed	3 points
Golden ratio estimate is "infinity" when $n = 1$	1 points

`latin.py` (10 points)

Command line arguments provide inputs	1 point
Square is the correct size	2 points
Square has each value once per row and once per column	5 points
Program prints a message and terminates if the top left number is not in the valid range	2 points

`guessing_game.py` (28 points)

Reads the number of guesses from the first command line argument	1
Determines the game mode based on the second command line argument	1
In PLAYER mode, two random characters are chosen from the letters in `bellingham`	4
In DEBUG mode, the two secret characters are taken from the second and third command line arguments	2
The user is given the correct number of guesses	4
The program specifies which (if any) of the secret letters have been guessed correctly after each guess	4
Letters that have been guessed correctly are not mentioned in the output for later guesses.	4
Once both letters are guessed, the program tells the player they've won and terminates immediately even if there are guesses remaining.	4
If the player loses, the answer is revealed.	4

4 Challenge Problem

This challenge problem is worth two points of extra credit: Write a program that takes a single non-negative decimal integer as a command line argument, then prints the binary representation of the number with no leading zeros.

Submit your Challenge Problem solution in a file named `binary.py`.