CSE 101: Introduction to Computational and Algorithmic Thinking

Stony Brook University

Homework Assignment #3

Spring 2018

Assignment Due: EXTENDED TO March 11, 2018 by 11:59 pm

Assignment Objectives

This homework assignment will give you practices on if-statements, lists and while-loops.

Getting Started

Visit Piazza and download the "bare bones" file `homework3.py` onto your computer, as well as `homework3_driver.py`. Open `homework3.py` in PyCharm and fill in the following information at the top:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)
4. the course number (CSE 101)
5. the assignment name and number (Homework #3)

Do not, under any circumstances, change the names of the functions or their argument lists. The grading software will be looking for exactly those functions provided in `homework3.py`.

Submit your final `homework3.py` file to Blackboard by the due date and time. Late work will not be graded.

Code that crashes and cannot be graded will earn no credit. It is your responsibility to test your code by running it through `homework3.py` and by creating your own test cases.

Part I: Virtual Pet Frog (20 points)

You are now the proud owner of a virtual pet – a cute, green frog that leads a simple, yet somehow busy life. You will write some code that updates the frog's mood as he (or she) engages in certain activities. Write a function `frog()` that takes the following arguments, in this order:

- `mood`: a positive integer that indicates the starting mood of the frog. A higher value indicates a happier mood.
- `actions`: a list of strings containing some combination of `'eat'`, `'work'`, `'play'` and `'read'`

Throughout the day the frog eats, works, plays and reads books. These actions affect the frog in different ways:

- If the action is `'play'`, then add 3 to the frog's current mood.

- If the action is `'eat'` and the frog's current mood is at least 50% of his starting mood, then add 1 to the frog's current mood.
- If the action is `'eat'` and the frog's current mood is less than 50% of his starting mood, then subtract 2 from the frog's current mood.
- If the action is `'read'` and the frog's current mood is at least 75% of his starting mood, then subtract 3 from the frog's current mood.
- If the action is `'read'` and the frog's current mood is less than 75% of his starting mood, then subtract 4 from the frog's current mood.
- If the action is `'work'`, then subtract 5 from the frog's current mood.

Regardless of the frog's mood, any action reduces his mood by 1.

If at any time the frog's mood becomes zero or negative, the function stops performing actions and returns 0. Otherwise, the function processes the entire list of actions and returns the frog's final mood.

Examples:

Function Call	Return Value
`frog(44, ['eat', 'play')`	`46`
`frog(16, ['play', 'eat', 'work')`	`12`
`frog(27, ['play', 'eat', 'read', 'play', 'read', 'read', 'work', 'read', 'eat', 'eat', 'work', 'work', 'work')`	`0`
`frog(41, ['play', 'work', 'eat', 'play')`	`39`
`frog(23, ['work', 'play', 'eat', 'read', 'work', 'read', 'play')`	`6`
`frog(50, ['read', 'eat', 'eat', 'read', 'work', 'read', 'work')`	`25`
`frog(10, ['eat', 'read', 'work', 'eat', 'read', 'read', 'read', 'work')`	`0`
`frog(38, ['play', 'eat', 'work', 'play', 'eat', 'work', 'read', 'play', 'play', 'work', 'read', 'work')`	`13`
`frog(7, ['work', 'play', 'play', 'read', 'work')`	`0`
`frog(24, ['work', 'play', 'eat')`	`20`

Part II: A Pac-Man Puzzler (20 points)

Pac-Man is a video game character who normally eats dots, power pellets, and the occasional ghost. Recently, he's fallen on hard times and has been reduced to eating letters of the alphabet. Pac-Man can eat most letters of the alphabet, but he is unable to digest any of the characters in the word "GHOST" (uppercase or lowercase). When he reaches one of these characters in a list, he loses his appetite and stops eating.

Complete the `pacman(line)` function, which traces Pac-Man's progress through a list of uppercase and lowercase letters (with no spaces, digits, or symbols) and returns the final state of the list. The initial list might be empty. Use underscores (_) to represent consumed characters and a less-than sign (<) to represent Pac-Man's final position (either at the very beginning or end of the list, or right before the character that stopped him). The returned value also includes any uneaten letters of the list.

For example, consider the list `['b', 'a', 't', 'c', 'h']`. Pac-Man can eat the `'b'` and the `'a'`, but stops when he reaches `'t'` (because it is one of the letters in "GHOST"). Thus, the final list will be `['_', '<', 't', 'c', 'h']`.

If Pac-Man stops eating in the middle of the input, you will need to replace the last character that he successfully consumed with Pac-Man himself. If he eats all the characters in the list, append him to the list. If the first character in the list is from "GHOST", simply insert him at the front of the list.

Hint #1: Use a while-loop to have Pac-Man eat as many characters as possible before stopping.

Hint #2: You can use the `insert` method to insert an item into a list. Take a look at the documentation at https://docs.python.org/3.6/tutorial/datastructures.html.

Examples:

Function Call	Return Value
`pacman(['D'])`	`['_', '<']`
`pacman(['g', 'y', 'o', 'R', 'C', 'l', 's', 'U', 'm', 'q'])`	`['<', 'g', 'y', 'o', 'R', 'C', 'l', 's', 'U', 'm', 'q']`
`pacman(['m', 'j]')`	`['_', '_', '<']`
`pacman(['H'])`	`['<', 'H']`
`pacman(['s', 'c', 'P', 'u', 'U', 'I', 'T', 'z', 'R'])`	`['<', 's', 'c', 'P', 'u', 'U', 'I', 'T', 'z', 'R']`
`pacman(['j', 'F', 'Q', 'K', 't', 'i'])`	`['_', '_', '_', '<', 't', 'i']`
`pacman(['m', 'K', 'x', 'J', 's'])`	`['_', '_', '_', '<', 's']`
`pacman(['h', 'i', 'o', 'J', 'x', 'e', 'u', 's'])`	`['<', 'h', 'i', 'o', 'J', 'x', 'e', 'u', 's']`
`pacman(['a', 'B', 'K', 'm', 'n', 'X', 'j'])`	`['_', '_', '_', '_', '_', '_', '_', '<']`
`pacman(['u', 'I', 'a', 'U', 'P', 'c', 'k', 'M', 'R', 'x'])`	`['_', '_', '_', '_', '_', '_', '_', '_', '_', '_', '<']`

Part III: Matching Brackets (20 points)

Write a function `brackets(expr)` that takes a string consisting only of the symbols `(`, `)`, `{`, `}`, `[` and `]` and analyzes the string to make sure that the brackets are all "balanced" (matched) properly. For example the strings `'{{[()]()()}}'` and `'(()[{[]}(([])())])'` contain balanced brackets, whereas `'{{[()][()()}}'`, `'{{[()]})'` and `'({[()]}'` are unbalanced.

We refer to `(`, `{` and `[` as "left brackets", and `)`, `}` and `]` as "right brackets".

The basic algorithm to implement works like this:

```
input: a string S
output: a list L
for each character in S
```

```
if the character is a left bracket
    append the character to L
else if the character is a right bracket, but L is empty
    return the string 'error'
else if the character is a right bracket and
             the rightmost element of L is a matching left bracket
    delete the rightmost element of L (because we have a match)
else we must have a mismatch, so return L
```

Note that if the original string is properly balanced, the return value should be an empty list.

Hint: You can use the `pop` method to remove an item from a list. Take a look at the documentation at https://docs.python.org/3.6/tutorial/datastructures.html.

Examples:

Function Call	**Return Value**
`brackets('(((({})))))')`	`[]`
`brackets('(([[({}{)(())]]))')}`	`['(', '(', '[', '[', '(', '{']`
`brackets('[][{[]}{}]{[]()}')`	`[]`
`brackets('()[]')`	`[]`
`brackets('[]{}[]')`	`[]`
`brackets('())[({{}[{{{[]}}}}]})]')`	`'error'`
`brackets('(([][()[[]]([])][]]))')`	`['(', '(']`
`brackets('{}[]')`	`[]`
`brackets('[(){}][()]{}[{}]')`	`[]`
`brackets('[{([()])}]')`	`[]`
`brackets('([{}([])])')`	`[]`
`brackets('[]{)[({([])}){}]}')`	`['{']`
`brackets('{[()]]}[]')`	`['{']`
`brackets('(()[])[]{}{}[}][]')`	`['[']`
`brackets('[](()([(){][}){}]))')`	`['(', '(', '[', '(']`
`brackets('([]{}{}[{((([[]]))))}])')`	`[]`
`brackets('{[{}]{[]}}{{()}}[]')`	`[]`
`brackets('{}(({[[[[]]]]}))')`	`[]`
`brackets('}[[([][{}])]]')`	`'error'`
`brackets('{}[[{]]')`	`['[', '[', '{']`

Part IV: It's Hailing! (20 points)

The **hailstone sequence** is defined as the integer sequence that results from manipulating a positive integer value n as follows:

- If n is even, divide it by 2 (using integer division)
- If n is odd, multiply it by 3 and then add 1

Repeat this process until you reach 1.

For example, starting with $n = 5$, we get the sequence $5, 16, 8, 4, 2, 1$.

If $n = 6$, we get the sequence $6, 3, 10, 5, 16, 8, 4, 2, 1$.

If $n = 7$, we get $7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

As far as anyone can tell, this process will eventually reach 1 for any starting value, although mathematicians have been unable to formally prove this property as of yet.

For this part of the assignment you will actually write two functions, although only the second one will actually be tested:

- `hail_length(n)` returns the total number of values in the hailstone sequence generated from the function argument (including the initial value and the final 1). For example, `hail_length(5)` would return the value 6.
- `siblings(length, maximum)`, which uses `hail_length()` internally to find all of the integers from 1 through the `maximum` (inclusive) that generate a hailstone sequence of the specified `length`. These values are added to a list which is returned by `siblings()`. For example, `siblings(6,35)` would return the list `[5, 32]`. `siblings(10,100)` would return `[12, 13, 80, 84, 85]`. The values in the list need not be sorted, as in these examples.

Examples:

Function Call	**Return Value**
`siblings(1, 968)`	`[1]`
`siblings(15, 853)`	`[11, 68, 69, 70, 75, 384, 416, 424, 426, 452, 453, 454]`
`siblings(20, 107)`	`[9, 56, 58, 60, 61]`
`siblings(13, 825)`	`[17, 96, 104, 106, 113, 640, 672, 680, 682]`
`siblings(8, 700)`	`[3, 20, 21, 128]`
`siblings(15, 181)`	`[11, 68, 69, 70, 75]`
`siblings(10, 748)`	`[12, 13, 80, 84, 85, 512]`
`siblings(6, 205)`	`[5, 32]`
`siblings(9, 113)`	`[6, 40, 42]`
`siblings(7, 152)`	`[10, 64]`

Part V: Vampire Hunters (20 points)

An intrepid band of vampire hunters has just arrived at a town where a coven of vampires is preying on the townspeople. Your job is to write a function `vampire_hunt(humans, vampires, hunters)` that will simulate how the populations of townspeople and vampires will change over time as the vampire hunters set to work.

The town is populated by `humans` people, is threatened by `vampires` scary vampires, and is protected by `hunters` vampire hunters. Each vampire can convert one person a day into a new vampire. (Luckily, the vampire hunters are all immune to vampire bites.) Each vampire hunter can destroy one vampire per day. Since we don't know how many days it might take to reduce one of the populations to 0 (i.e., `humans` or `vampires`), we cant use a for-loop to simulate the fight. Instead, we will use a while-loop, which will run until one of the two

populations reaches 0.

The bulk of your program will take place in a while-loop. As long as each population (`humans` and `vampires`) is greater than 0, your program should:

1. Determine how many vampires are destroyed that day (the vampire hunters always strike first, during daylight hours), and update the vampire population appropriately. This counter may reach 0, but it must never be allowed to become negative.
2. If there are any vampires remaining, they attack the remaining townspeople that night. Each vampire converts one person into a new vampire. Update the vampire population and human population accordingly. Like the previous step, the human population can reach 0, but it should never become negative.

Sample Simulation #1

```
Starting values: humans = 84, vampires = 6, hunters = 2.
Day #1:
    Humans: 84  Vampires: 6.
    Hunters destroyed 2 vampires.
    Vampires converted 4 people into vampires.
Day #2:
    Humans: 80  Vampires: 8.
    Hunters destroyed 2 vampires.
    Vampires converted 6 people into vampires.
Day #3:
    Humans: 74  Vampires: 12.
    Hunters destroyed 2 vampires.
    Vampires converted 10 people into vampires.
Day #4:
    Humans: 64  Vampires: 20.
    Hunters destroyed 2 vampires.
    Vampires converted 18 people into vampires.
Day #5:
    Humans: 46  Vampires: 36.
    Hunters destroyed 2 vampires.
    Vampires converted 34 people into vampires.
Day #6:
    Humans: 12  Vampires: 68.
    Hunters destroyed 2 vampires.
    Vampires converted 12 people into vampires.

Function will return [0, 78]
```

Sample Simulation #2:

```
Starting values: humans = 29, vampires = 11, hunters = 7.
Day #1:
    Humans: 29  Vampires: 11.
    Hunters destroyed 7 vampires.
```

```
    Vampires converted 4 people into vampires.
Day #2:
    Humans: 25  Vampires: 8.
    Hunters destroyed 7 vampires.
    Vampires converted 1 people into vampires.
Day #3:
    Humans: 24  Vampires: 2.
    Hunters destroyed 2 vampires.

Function will return [24, 0]
```

Examples:

Function Call	Return Value
`vampire_hunt(51, 2, 8)`	`[51, 0]`
`vampire_hunt(33, 2, 2)`	`[33, 0]`
`vampire_hunt(67, 11, 5)`	`[0, 48]`
`vampire_hunt(83, 11, 8)`	`[80, 0]`
`vampire_hunt(59, 15, 4)`	`[0, 62]`
`vampire_hunt(26, 2, 3)`	`[26, 0]`
`vampire_hunt(50, 3, 4)`	`[50, 0]`
`vampire_hunt(84, 11, 2)`	`[0, 87]`
`vampire_hunt(20, 9, 4)`	`[0, 13]`
`vampire_hunt(61, 12, 7)`	`[53, 0]`

How to Submit Your Work for Grading

To submit your `.py` file for grading:

1. Login to Blackboard and locate the course account for CSE 101.
2. Click on "Assignments" in the left-hand menu and find the link for this assignment.
3. Click on the link for this assignment.
4. Click the "Browse My Computer" button and locate the `.py` file you wish to submit. Submit only that one `.py` file.
5. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the above directions again. We will grade only your last submission.