

START THIS ASSIGNMENT EARLY!

1. Reminder: The first exam will be Monday, February 13. You should review all the quizzes, labs, and homework assignments. Look at the posted solutions. You should understand all the solutions, but, more importantly, you must be able to generate all the solutions starting from scratch!

2. Programming Assignment A:

Motivation: The *logistic map* is a simple mathematical model often used to describe the self-limiting growth of biological populations.¹ Population growth depends on the rate of reproduction. A model of population growth that is even simpler than the logistic map model might assume reproduction from one generation to the next continues at a constant rate. Depending on the rate, this population will eventually die out (if the rate is less than one), remain constant (if the rate equals one), or increase forever (if the rate is greater than 1).

In the logistic map model, reproduction rate is assumed to be a function of the available resources in the environment: when the population is high, the available resources are reduced, and so the reproduction rate is reduced from its maximum value. If the population reaches its maximum theoretic capacity, then the population collapses to zero for the next generation.

The logistic map model uses the equation $f(a, x) = ax(1 - x)$ where we can think of x as the population of the current generation and $ax(1 - x)$ as the population of the next generation. Here we assume x is somewhere between 0 and 1 (where 1 is the theoretical maximum population). We will refer to this function as the logistic map function and call the value a the amplitude.

When a is between 0 and 1, the population will die out, no matter what the initial population. When a is between 1 and 2, the population will approach a constant value relatively quickly. This constant only depends on a (independent of the initial population). When a is between 2 and 3, the population will also eventually approach the same value, but first will fluctuate around that value for some time. When a is between 3 and approximately 3.57, the population will permanently oscillate between two, or more, values that again only depend on a .

Things get much more interesting for values of a above approximately 3.57. In this case the population becomes *chaotic*. The population over time seems random, not following any predictable trend. Also, slight variations in the initial population result in dramatically different results over time. This type of behavior is sometimes called the “butterfly effect.” In chaotic systems, even the tiniest change in initial conditions can alter the future in ways one cannot know *a priori*.

Functions with chaotic behavior are of interest in several areas of science. Their unpredictability and sensitivity to initial conditions make them attractive in, to name just a couple of applications,

¹http://en.wikipedia.org/wiki/Logistic_map

pseudorandom number generation and cryptography.² In this assignment, you will write two programs that allow a user to observe the behavior of the logistic map function for different initial conditions and different values of the amplitude a .

When $f(a, x) = ax(1 - x)$ is used recursively, chaotic behavior will result for some values of amplitude a . By “recursively” we mean that the function is first applied to a given starting value and then re-applied to the result. As mentioned above, you may think of the given starting value as the initial population (or the initial condition). When the argument of the function is the initial population, the function returns the population of the next generation. We can think of the initial population as the population at “time zero” and the population of the next generation as the population at “time one.” Applying the function to the population at “time one” yields the population at “time two,” and so on.

Let’s call the starting value x_0 —note the numeric subscript indicates this is the “zeroth” value, i.e., the starting point or initial condition.³ The result of applying the function f to x_0 is the value x_1 . Said mathematically, we would write $x_1 = f(a, x_0)$. Another way of saying this is $x_1 = ax_0(1 - x_0)$. Note the new numeric subscript that indicates the “next value.” If we now apply the function f to x_1 we obtain x_2 (the subscript increases by one, but in practice x_0 , x_1 , and x_2 are just some numbers). In a math class we might write $x_2 = f(a, x_1)$. We could also express this as $x_2 = ax_1(1 - x_1)$. If we apply the function f to x_2 we obtain x_3 , and so on. In general, we would say that recursive application of this function yields the set of numbers given by $x_{n+1} = ax_n(1 - x_n)$ where n is an arbitrary non-negative integer.

Step 1

With this background, write a function called `log_map()` that takes two arguments (i.e., two parameters). The function returns the value $ax(1 - x)$. Thus `log_map()` is simply the Python realization of the function $f(a, x)$ described above. The first argument corresponds to the amplitude “ a ” while the second argument corresponds to “ x ” (the fraction of the theoretical maximum population). Note that the function does not print $ax(1 - x)$! Rather, it calculates $ax(1 - x)$ and then *returns* that value using a `return` statement. This function can be written in just two lines (the `def` statement and the `return` statement).

Ensure this function works properly. The following demonstrates the proper behavior of this function.

```

1 >>> amp = 0.4
2 >>> log_map(amp, 0.5)
3 0.1
4 >>> x = log_map(amp, 0.1)
5 >>> x
6 0.036000000000000001
7 >>> x = log_map(amp, x)

```

²Pseudorandom numbers are numbers that mimic the behavior of truly random numbers and yet they are generated by a deterministic algorithm. These are important in everything from scientific modeling and simulations to gaming.

³In practice x_0 is just some number, perhaps 0.201, but for now we want to leave this completely general.

```

8 >>> x = log_map(amp, x)
9 >>> x = log_map(amp, x)
10 >>> x
11 0.002178231484034866
12 >>> # Reset amplitude and x.
13 >>> amp = 3.95
14 >>> log_map(amp, 0.5)
15 0.9875
16 >>> x = log_map(amp, 0.9875)
17 >>> x
18 0.04875781249999983
19 >>> x = log_map(amp, x)
20 >>> x = log_map(amp, x)
21 >>> x = log_map(amp, x)
22 >>> x
23 0.9547350446277946

```

In line 1 the amplitude `amp` is set to 0.4. The amplitude is held constant in lines 1 through 11. The logistic map function is called with this amplitude and an x value of 0.5 in line 2. This yields 0.1 as shown in line 3. The value of 0.1 is fed back to the logistic map function in line 4 and the value of x is set equal to the result which is shown in line 6. In lines 7 through 9, the value of x is fed back into the logistic map function and the return value is assigned back to x . In line 11 we see the current value of x . If we kept going, the value of x would asymptotically approach 0.

Lines 13 through 23 mirror lines 1 through 11 except the amplitude has been reset to 3.95. In line 15 we see the result of feeding 0.5 into the logistic map function is 0.9875, i.e., the value of x nearly doubles. Then, as shown in lines 16 through 18, using 0.9875 for the x value yields approximately 0.04876, i.e., the next value of x has dropped by more than a factor of 20. Continuing this for three more cycles yields approximately 0.9547. Since this function is chaotic, there is no way to predict the next value other than to actually calculate it!

Step 2

Write a function called `show_list()` which takes two arguments, a list and an integer. This function displays each item of the list it is passed. The elements should be shown one per line and should be preceded by an index that starts at the value of the second argument and is incremented by one for each element.

The following demonstrates the correct behavior of this function:

```

1 >>> x = ["baa", "baa", "black", "sheep"]
2 >>> show_list(x, 1)
3 1 baa
4 2 baa

```

```

5 3 black
6 4 sheep
7 >>> show_list(x, 0)
8 0 baa
9 1 baa
10 2 black
11 3 sheep
12 >>> show_list([6.2, 4.3, 10, -7, "boo!"], 7)
13 7 6.2
14 8 4.3
15 9 10
16 10 -7
17 11 boo!

```

Step 3

Write a function called `make_generations()` that takes three arguments. This function uses `log_map()` to create a list of values (which we can think of as the values of successive generations). The first argument of `make_generations()` is the amplitude, the second is the initial value, and the third is the number of generations beyond the initial value. The function does the following:

1. Creates a list that contains the initial value as its single element.
2. Uses a `for`-loop and `log_map()` to determine the population of each successive generation. The value of each generation should be appended to the list created in the second step. This continues for however many generations are specified by the third argument to the function.
3. Returns the list.

The following demonstrates the correct behavior of this function (note: this assumes the `log_map()` function has been written and is working properly).

```

1 >>> make_generations(0.4, 0.5, 3)
2 [0.5, 0.1, 0.036000000000000001, 0.013881600000000004]
3 >>> make_generations(3.95, 0.5, 3)
4 [0.5, 0.9875, 0.04875781249999983, 0.183202928469848]

```

In both these calls to `make_generations()`, the initial value is 0.5 and the number of generations beyond the initial value is 3 (hence the list that is returned has four values—the initial value and the three successive generations). In the first call the amplitude is 0.4 and in the second it is 3.95.

Step 4

We now want to use the functions you wrote for steps 1, 2, and 3 to implement a program that calculates and displays the values of the logistic map function using an amplitude and initial value that the user specifies. The user also specifies the number of generations.

You must write a `main()` function that does the following:

1. Prompts the user for the amplitude, initial value, and number of generations.
2. Calls `make_generations()` to obtain a list of populations.
3. Calls `show_list()` to show the elements of the list produced by step 2. The values should be displayed starting from 0.

The last statement in your program must be a call to `main()`.

The following demonstrates the correct behavior of this program where the amplitude is 0.4, the initial population is 0.5, and there are eight generations beyond the initial value. As you can see, the population simply dies out.

```

1 Enter the amplitude: 0.4
2 Enter the initial value: 0.5
3 Enter the number of generations: 8
4 0 0.5
5 1 0.1
6 2 0.036000000000000001
7 3 0.0138816000000000004
8 4 0.005475560472576002
9 5 0.002178231484034866
10 6 0.0008693947166547302
11 7 0.0003474555477925532
12 8 0.00013893392897394456

```

In the following, the initial population is still 0.5 and eight generations are shown beyond the initial value. However, here the amplitude is 3.95 and hence the population is chaotic.

```

1 Enter the amplitude: 3.95
2 Enter the initial value: 0.5
3 Enter the number of generations: 8
4 0 0.5
5 1 0.9875
6 2 0.04875781249999983
7 3 0.183202928469848
8 4 0.591076481106183
9 5 0.9547350446277946

```

```

10 6 0.17070335479006285
11 7 0.5591766918412491
12 8 0.9736675706137671

```

Place your program in a file called `hw4a.py`. At the start of the file put comment lines that give your name, the date, the class (CptS 111), and the assignment (HW 4A). *Also*, be sure to list anybody with whom you collaborated on the assignment. (Collaboration is encouraged, but you must acknowledge it!)

2. Programming Assignment B:

Step 1

You know that we can subtract `ints` and `floats`. In the expression $x - y$, x is known as the minuend and y is the subtrahend.

In Python we cannot directly subtract one list from another. However, let's assume we want to do just that: we are given a list of minuends and a corresponding list of subtrahends and we want to create a new list that contains the element-by-element subtraction of the two lists.

Write a function called `subtract_lists()` that takes two arguments. The first argument is a list of minuends. The second argument is a list of subtrahends. It is assumed that all the elements in both lists are numeric quantities and that the lists are of equal length. This function does the following:

1. Initializes an empty list.
2. Uses a `for`-loop to subtract an element in the second list from the corresponding element in the first list. The result of this subtraction is appended to the list created in step 1.
3. Returns the resulting list.

The following demonstrates the proper behavior of this function:

```

1 >>> x = [5, 6, 7]
2 >>> y = [10.5, 20.7, -15.34]
3 >>> subtract_lists(x, y)
4 [-5.5, -14.7, 22.34]
5 >>> subtract_lists([1.2, -3.2], [53.5, 10.0 + 3 * 17.1])
6 [-52.3, -64.5]

```

Step 2

We now want to write a program that builds on the program you wrote for Programming Assignment A. This program starts exactly as program `hw4a.py` did: The user is prompted for the amplitude,

initial population, and number of generations. The values for the desired number of generations are shown.

In this new version of the program we now add code to accomplish the following:

1. The user is prompted for a new initial value.
2. Using the original amplitude but the new initial value, a new list of populations is generated. (The number of generations is the same as was originally requested.) This is accomplished by calling `make_generations()` with the new initial value.
3. The new list of populations is shown to the user using the `show_list()` function.
4. The difference between the populations obtained using the original initial value and the new initial value is found using `subtract_lists()`.
5. The list of differences is shown to the user. This is preceded by a blank line and text which says what the values represent (see the example output shown below).

The following demonstrates the correct behavior of this function. Here the amplitude is 0.4 and the two starting populations are 0.5 and 0.7.

```

1 Enter the amplitude: 0.4
2 Enter the initial value: 0.5
3 Enter the number of generations: 4
4 0 0.5
5 1 0.1
6 2 0.036000000000000001
7 3 0.0138816000000000004
8 4 0.005475560472576002
9 Enter new initial value: 0.7
10 0 0.7
11 1 0.084
12 2 0.0307776000000000006
13 3 0.011932135735296003
14 4 0.00471590394883619
15
16 Difference between generations:
17 0 -0.19999999999999996
18 1 0.016
19 2 0.00522240000000000055
20 3 0.0019494642647040015
21 4 0.000759656523739812

```

Notice how the magnitude of the difference between these two sets of populations gets progressively smaller. It doesn't matter that one sequence started at 0.5 and the other started at 0.7—both ultimately approach zero.

Here is another example where the initial values are again 0.5 and 0.7 but now the amplitude is 3.95:

```

1 Enter the amplitude: 3.95
2 Enter the initial value: 0.5
3 Enter the number of generations: 4
4 0 0.5
5 1 0.9875
6 2 0.04875781249999983
7 3 0.183202928469848
8 4 0.591076481106183
9 Enter new initial value: .7
10 0 0.7
11 1 0.8295000000000001
12 2 0.5586475124999998
13 3 0.9739138536463714
14 4 0.10035235432208041
15
16 Difference between generations:
17 0 -0.19999999999999996
18 1 0.15799999999999992
19 2 -0.5098897
20 3 -0.7907109251765234
21 4 0.49072412678410265

```

Notice in this case the difference between the two sets of generations does not appear to follow any type of pattern. And, in fact, it does *not* follow a pattern because of the chaotic behavior of the logistic map function for this amplitude.

Don't forget that the last statement in the program should be a call to `main()`.

Put this program in a file called `hw4b.py`. Be sure to include comment lines that give your name, the date, the class, the assignment name (HW 4B). Also list any people with whom you collaborated.