# python-problem-set-answer

October 20, 2020

## 1 Week 2

### 1.1 What to turn in?

(1) a module file named `python_problem_set.py`, containing all the function definitions *

(2) a Jupyter Note book file named `python-problem-set.ipynb`, containing only the test cases and explanations.

(3) the pdf version of (2) `python-problem-set.pdf`

* In case you're wondering, python allows underscores but not hyphens in module names

#### 1.1.1 Attention:

- Your function names must be precisely the same as specified in the assignment notebook, and your functions must take arguments in the same order as specified
- If you wish, you can write your function definitions into the module directly, modifying the following steps appropriately

#### 1.1.2 Step 1, solve the problems in Jupyter Notebook

when the question is asking to write a function, please:

(1) write the function in one or more cells

(2) write at least one test case in another cell(s)

for example:

[1]:
```
# this is the function definition cell
def volume_of_cylinder(h,r):
    S=3.14159*r**2
    V=S*h
    return V
```

[2]:
```
# this is a test case cell
volume_of_cylinder(2.5,3.0)
```

[2]:
```
70.685775
```

#### 1.1.3 Step 2, copy all the function definitions to python_problem_set.py

create a module named python_problem_set.py

copy the function definitions in Step 1-(1) to this module

#### 1.1.4 Step 3, remove definitions in Jupyter Notebook, import the module

As a result of step 2,

you can now remove the function defination in your Jupyter Notebook.

But remember to import the function from the module

```
from my_module import volume_of_cylinder
volume_of_cylinder(2.5,3.0)
```

### 1.2 How to create a python module?

(1) write your .py module in a seperate file (within the same folder)

(2) import your module, and import your function here

(3) run your function in this jupyter notebook

[3]:
```
# Import Statements
import python_problem_set as ps
from python_problem_set import test
import matplotlib.pyplot as plt

%matplotlib inline
```

### 1.3 Problem Group 1 - Recursion

#### 1.3.1 Problem 1.1 - Factorial

Recall that for any positive integer $n$, the product of $n$ with all positive integers less than $n$ is called $n$-**factorial** and is typically denoted $n!$. By convention, the factorial is usually extended to 0 by the definition $0! = 1$.

Define a recursive Python function

`recursive_factorial`

that returns the factorial of any non-negative integer specified as its input.

[4]:
```
test(ps.recursive_factorial, 5, 120)
```

```
Expect recursive_factorial(5) to be 120
>>> recursive_factorial(5)
120

PASS: Output matches expected output.
```

[5]:
```
test(ps.recursive_factorial, 0, 1)
```

```
Expect recursive_factorial(0) to be 1
>>> recursive_factorial(0)
1

PASS: Output matches expected output.
```

### 1.3.2 Problem 1.2 - Memoization

a. Compare the performance of the memoized Fibonnacci function to the recursive Fibonnaacci function using either the `%%timeit` magic command or the `timeit` module.

[6]:
```
def F(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return F(n - 1) + F(n - 2)

fib_memo = {}

def F_memo(n):

    if n == 1:
        return 0
    elif n == 2:
        return 1
    elif n not in fib_memo :
        fib_memo[n] = F_memo(n - 1) + F_memo(n - 2)
    return fib_memo[n]
```

[7]:
```
%%timeit -n 1000
F(10)
```

```
28.7 µs ± 1.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[8]:
```
%%timeit -n 1000
F_memo(10)
```

```
321 ns ± 83.4 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The memoized function `F_memo` took an average of `264 ns` per loop, whereas `F` took an average of `23,300 ns` per loop.*

It is clear that `F_memo` is significantly faster in this case.

**Times may fluctuate.*

b. Define a memoized recursive Python function

   recursive_factorial_memo

and compare the performance of the memoized version to the un-memoized version using either the `%%timeit` magic command or the `timeit` module.

[9]:
```
fac_memo = {}
test(ps.recursive_factorial_memo, (5, fac_memo), 120)
```

```
Expect recursive_factorial_memo(5, {}) to be 120
>>> recursive_factorial_memo(5, {})
120

PASS: Output matches expected output.
```

[10]:
```
fac_memo = {}
test(ps.recursive_factorial_memo, (0, fac_memo), 1)
```

```
Expect recursive_factorial_memo(0, {}) to be 1
>>> recursive_factorial_memo(0, {})
1

PASS: Output matches expected output.
```

[11]:
```
%%timeit -n 1000
ps.recursive_factorial(10)
```

```
4.87 µs ± 1.02 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[12]:
```
%%timeit -n 1000
ps.recursive_factorial_memo(10, fac_memo)
```

```
834 ns ± 103 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The memoized function `recursive_factorial_memo` took an average of `343 ns` per loop, whereas `recursive_factorial` took an average of `2,740 ns` per loop.*

It is clear that `recursive_factorial_memo` is significantly faster in this case.

**Times may fluctuate.*

c. Is there any difference in the performance gains in the two cases? Can you (briefly) try to explain why, or why not?

Yes, from the results of the %%timeit functions, we can see that there is a performance difference between the memoized and non-memoized functions.

The reason why we are getting faster results is becasue we are saving previously computed results for each input. Since the functions we are memoizing are one-to-one, we are able to simply return the previously calculated and saved result. Thus, in order to compute that same result again, we simply have to lookup the result in a cached dictionary.

### 1.3.3 Problem 1.3 - Binomial Coefficients

Define two python functions

`binomial_factorial`

and

`binomial_recursive`

that both take in integers $n$ and $k$ and compute the corresponding binomial coefficient, but for `binomial_factorial`, use your recursive factorial function defined Problem 1.1 to explicitly define the binomial coefficient, and for `binomial_recursive` directly use the recurrence relation above. Make sure that the output of these functions is of type `int`!

[13]:
```
test(ps.binomial_factorial, (10, 7), 120)
```

```
Expect binomial_factorial(10, 7) to be 120
>>> binomial_factorial(10, 7)
120

PASS: Output matches expected output.
```

[14]:
```
test(ps.binomial_factorial, (10, 0), 1)
```

```
Expect binomial_factorial(10, 0) to be 1
>>> binomial_factorial(10, 0)
1

PASS: Output matches expected output.
```

[15]:
```
test(ps.binomial_factorial, (10, 10), 1)
```

```
Expect binomial_factorial(10, 10) to be 1
>>> binomial_factorial(10, 10)
1

PASS: Output matches expected output.
```

[16]:
```
test(ps.binomial_recursive, (10, 7), 120)
```

```
Expect binomial_recursive(10, 7) to be 120
>>> binomial_recursive(10, 7)
120

PASS: Output matches expected output.
```

[17]:
```
test(ps.binomial_recursive, (10, 0), 1)
```

```
Expect binomial_recursive(10, 0) to be 1
>>> binomial_recursive(10, 0)
```

```
1

PASS: Output matches expected output.
```

[18]:
```
test(ps.binomial_recursive, (10, 10), 1)
```

```
Expect binomial_recursive(10, 10) to be 1
>>> binomial_recursive(10, 10)
1

PASS: Output matches expected output.
```

### 1.3.4 Problem 1.4 - The Logistic Map

Your recursive function might be too slow to run

In that case, use the memoization method from problem 1.2 to accelerate it

Define a python function

`logistic`

that takes the parameters $n$ and $r$ as well as $x_0$, the inital value in the sequence, as inputs and recursively computes $x_n$.

[19]:
```
# Test case where the population will eventually die, when r is between 0 and 1
r = 0.1
test(ps.logistic, (10, r, 0.5), 2.4309016841715593e-11)
```

```
Expect logistic(10, 0.1, 0.5) to be 2.4309016841715593e-11
>>> logistic(10, 0.1, 0.5)
2.4309016841715593e-11

PASS: Output matches expected output.
```

[20]:
```
# Test case where population will quickly approach (r - 1) / r, when r is␣
 ↪between 1 and 2
r = 1.5
print(f"(r - 1) / r = {(r - 1) / r}", "\n")
test(ps.logistic, (10, r, 0.5), 0.3333973075663317)
```

```
(r - 1) / r = 0.3333333333333333

Expect logistic(10, 1.5, 0.5) to be 0.3333973075663317
>>> logistic(10, 1.5, 0.5)
0.3333973075663317

PASS: Output matches expected output.
```

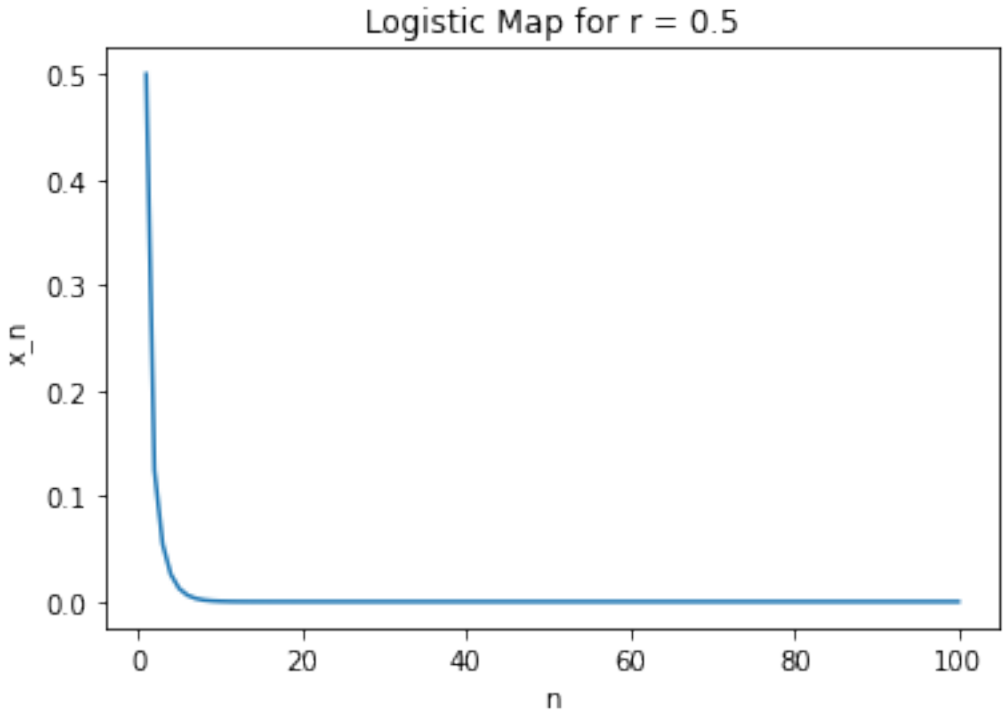
Make five distinct plots $x_n$ as a function of $n$ for $x_0 = 0.5$ and $r = 0.5, 1.5, 2.5, 3.3, 3.5$ up to $n = 100$. You should see qualitatively different behaviors for these various values of $r$ – describe them.

[21]:
```
r = 0.5
# With r between 0 and 1, the population will eventually die, independent of␣
 ↪the initial population.
n_val = list(range(1,101))
x_n_val = list(ps.logistic_gen(100, r, 0.5))

plt.xlabel('n')
plt.ylabel('x_n')
plt.title('Logistic Map for r = 0.5')

plt.plot(n_val, x_n_val)
```

[21]:
```
[<matplotlib.lines.Line2D at 0x11318f0a0>]
```

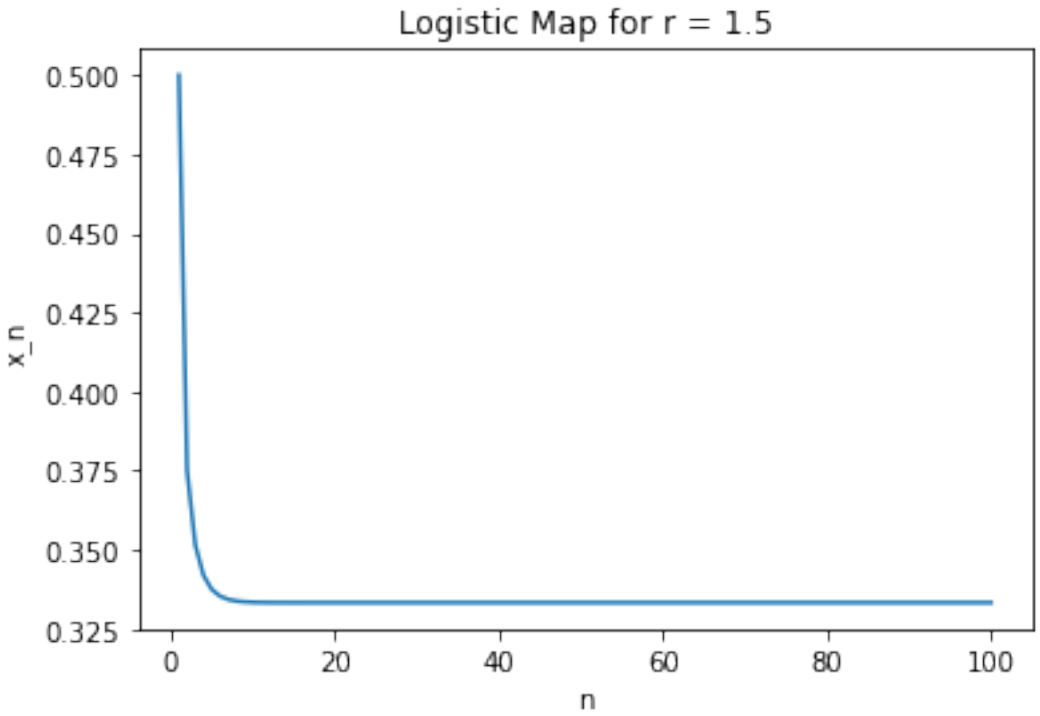



[22]:
```
r = 1.5
# With r between 1 and 2, the population will quickly approach the value (r -␣
 ↪1) / r, independent of the initial population.
n_val = list(range(1,101))
x_n_val = list(ps.logistic_gen(100, r, 0.5))

plt.xlabel('n')
plt.ylabel('x_n')
plt.title('Logistic Map for r = 1.5')
```

```
plt.plot(n_val, x_n_val)
```

[22]: [<matplotlib.lines.Line2D at 0x113266a30>]

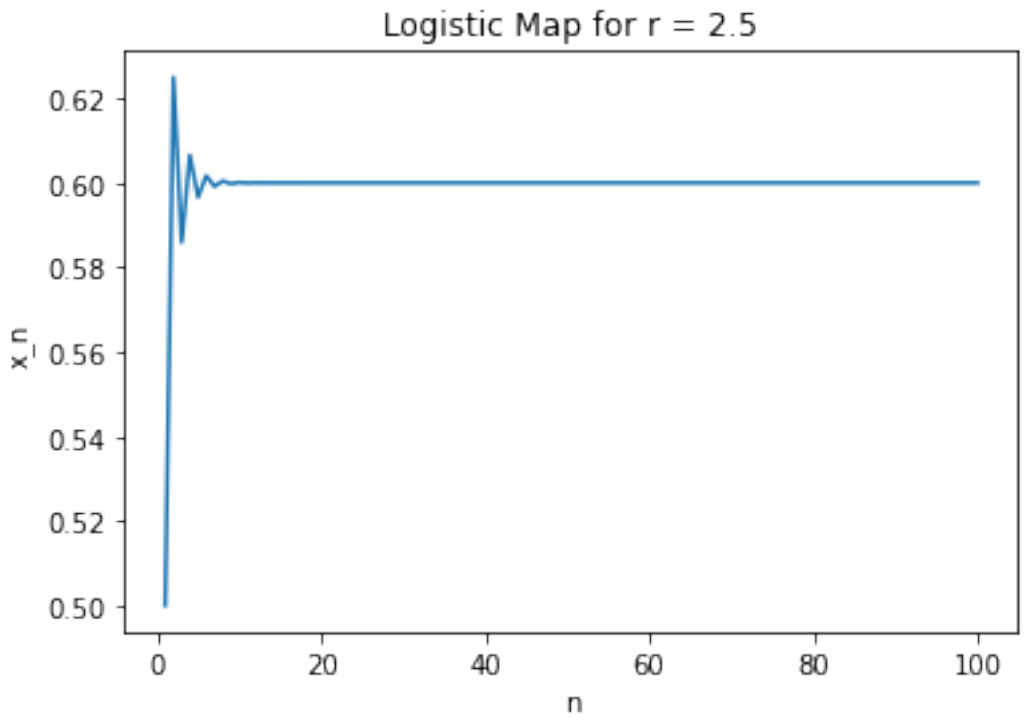



[23]:
```
r = 2.5
# With r between 2 and 3, the population will also eventually approach the same␣
 ↪value
# (r - 1) / r, but first will fluctuate around that value for some time.
n_val = list(range(1,101))
x_n_val = list(ps.logistic_gen(100, r, 0.5))

plt.xlabel('n')
plt.ylabel('x_n')
plt.title('Logistic Map for r = 2.5')

plt.plot(n_val, x_n_val)
```

[23]: [<matplotlib.lines.Line2D at 0x1132c9fd0>]

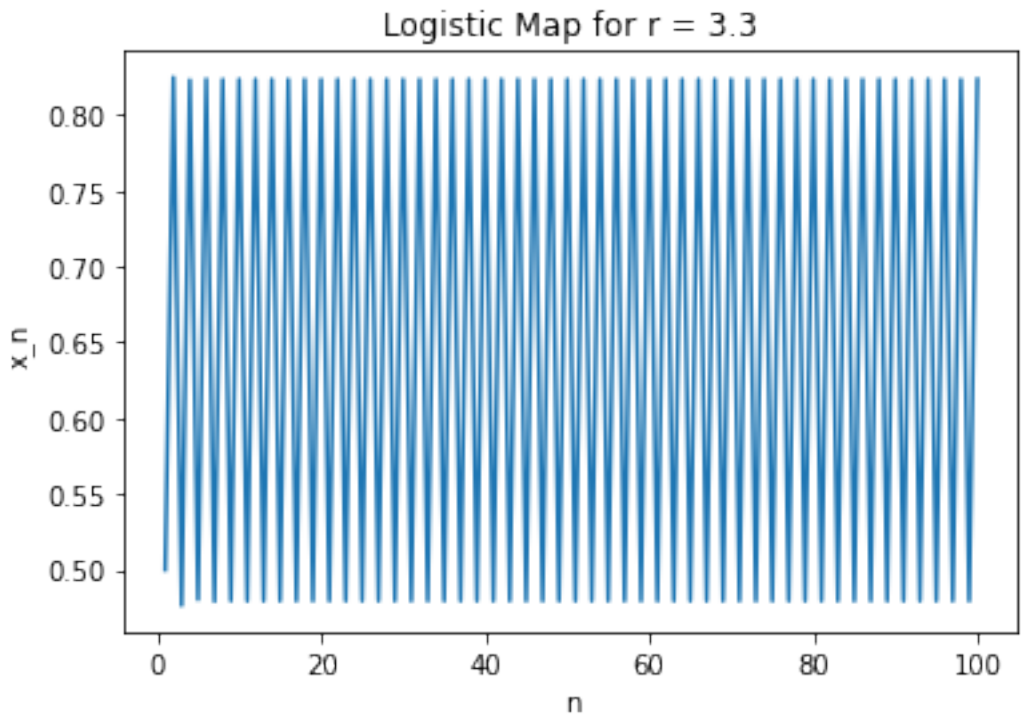



[24]:
```
r = 3.3
# With r between 3 and 1 + √6  3.44949, from almost all initial conditions the␣
 ↪population will approach permanent oscillations between two values. These␣
 ↪two values are dependent on r.
n_val = list(range(1,101))
x_n_val = list(ps.logistic_gen(100, r, 0.5))

plt.xlabel('n')
plt.ylabel('x_n')
plt.title('Logistic Map for r = 3.3')

plt.plot(n_val, x_n_val)
```

[24]:
```
[<matplotlib.lines.Line2D at 0x113324880>]
```

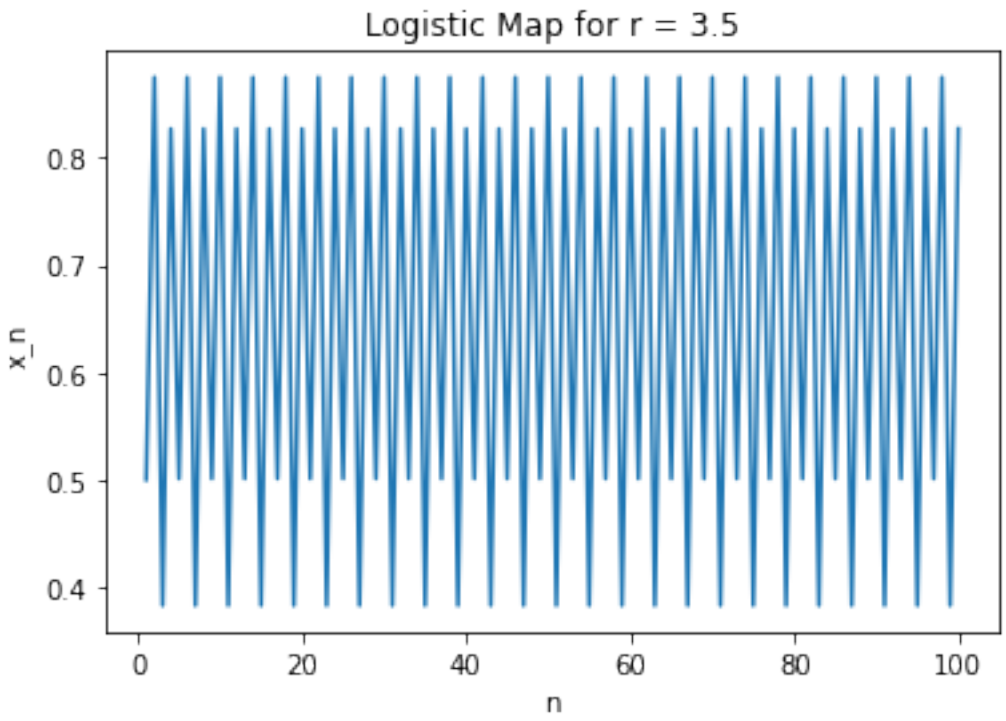



[25]:
```
r = 3.5
# With r between 3.44949 and 3.54409 (approximately), from almost all initial␣
 ↪conditions the population will approach permanent oscillations among four␣
 ↪values.
n_val = list(range(1,101))
x_n_val = list(ps.logistic_gen(100, r, 0.5))

plt.xlabel('n')
plt.ylabel('x_n')
plt.title('Logistic Map for r = 3.5')

plt.plot(n_val, x_n_val)
```

[25]:
```
[<matplotlib.lines.Line2D at 0x11548f9a0>]
```

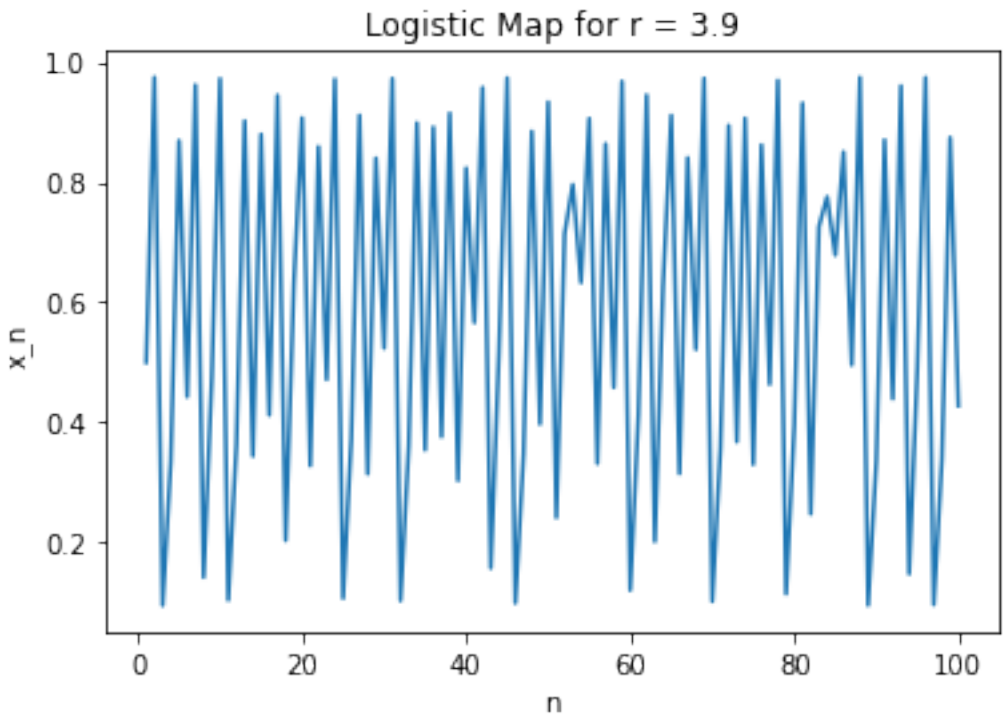



[26]:
```
r = 3.9
# Most values of r beyond 3.56995 exhibit chaotic behaviour.
n_val = list(range(1,101))
x_n_val = list(ps.logistic_gen(100, r, 0.5))

plt.xlabel('n')
plt.ylabel('x_n')
plt.title('Logistic Map for r = 3.9')

plt.plot(n_val, x_n_val)
```

[26]:
```
[<matplotlib.lines.Line2D at 0x1154e3c70>]
```




## 1.4 Problem Group 2 - Searching for stuff

### 1.4.1 Problem 2.1 - Linear Search

Do an internet search for "**linear search**" and read up on it. Convince yourself that you know how it works, and write a function

`linear_search`

that takes a list `L` of positive integers and a number `n` that may or may not be in the list as its input and outputs the list index of the number `n` by performing a linear search. If the number isn't in the list, the function should print `The specific number is not in the list.` and should return nothing.

[27]:
```
sample_list = [2, 5, 8, 8, 10, 20, 2020]
test(ps.linear_search, (sample_list, 10), 4)
```

```
Expect linear_search([2, 5, 8, 8, 10, 20, 2020], 10) to be 4
>>> linear_search([2, 5, 8, 8, 10, 20, 2020], 10)
4

PASS: Output matches expected output.
```

[28]:
```
test(ps.linear_search, (sample_list, 8), 2)
```

```
Expect linear_search([2, 5, 8, 8, 10, 20, 2020], 8) to be 2
>>> linear_search([2, 5, 8, 8, 10, 20, 2020], 8)
2

PASS: Output matches expected output.
```

[29]:
```
test(ps.linear_search, (sample_list, 50), None)
```

```
Expect linear_search([2, 5, 8, 8, 10, 20, 2020], 50) to be None
>>> linear_search([2, 5, 8, 8, 10, 20, 2020], 50)
The specific number is not in the list.
None

PASS: Output matches expected output.
```

### 1.4.2 Problem 2.2 - Bisection(Binary) Search

Do an internet search for "**bisection search**" (aka **binary search**) and read up on it. Convince yourself that you know how it works, and write a function

`bisection_search`

that takes a sorted list `L` of positive integers **in ascending order** and a number `n` that may or may not be in the list as its input and outputs the list index of that number by performing a bisection search. If the number isn't in the list, the function should print `The specific number is not in the list.` and should return nothing.

[30]:
```
sample_list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,␣
↪61, 67, 71, 73, 79, 83, 89, 97]
test(ps.bisection_search, (sample_list, 79), 21)
```

```
Expect bisection_search([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97], 79) to be 21
>>> bisection_search([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97], 79)
21

PASS: Output matches expected output.
```

[31]:
```
test(ps.bisection_search, (sample_list, 2), 0)
```

```
Expect bisection_search([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97], 2) to be 0
>>> bisection_search([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97], 2)
0

PASS: Output matches expected output.
```

[32]:
```
test(ps.bisection_search, (sample_list, 50), None)
```

```
Expect bisection_search([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97], 50) to be None
>>> bisection_search([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97], 50)
The specific number is not in the list.
None

PASS: Output matches expected output.
```

### 1.4.3 Problem 2.3 - Bisection Root Finding

Do an internet search for "Bisection Root Finding" and read up on it. Convince yourself that you know how it works, and write a function

`bisection_root`

taking the following inputs:

- A real-valued function `f` of a single real variable
- A left-hand value `x_left`
- A right-hand value `x_right`
- A tolerance `epsilon`

Provided `f` has a single root at some $x_0$ in the open interval $(x_{\text{left}}, x_{\text{right}})$ and is either an increasing or decreasing function on that interval, `bisection_root` should output the location of the root with an accuracy of $\epsilon$.

[33]:
```
def poly(x):
    # Should have roots x = -2, 1, 5.
    return (x**3 - 4 * x**2 - 7 * x + 10)

def golden(x):
    # The positive root should be the golden ratio
    return (x**2 - x - 1)
```

[34]:
```
test(ps.bisection_root, (poly, 0, 2, 0.000001), 1)
```

```
Expect bisection_root(<function poly at 0x115513280>, 0, 2, 1e-06) to be 1
>>> bisection_root(<function poly at 0x115513280>, 0, 2, 1e-06)
Found root in 1 iterations.
1.0

PASS: Output matches expected output.
```

[35]:
```
test(ps.bisection_root, (poly, -4, -1, 0.000001), -2.0000000298023224)
```

```
Expect bisection_root(<function poly at 0x115513280>, -4, -1, 1e-06) to be
-2.0000000298023224
```

```
>>> bisection_root(<function poly at 0x115513280>, -4, -1, 1e-06)
Found root in 25 iterations.
-2.0000000298023224

PASS: Output matches expected output.
```

[36]:
```
test(ps.bisection_root, (poly, 6, 10, 0.000001), None)
```

```
Expect bisection_root(<function poly at 0x115513280>, 6, 10, 1e-06) to be None
>>> bisection_root(<function poly at 0x115513280>, 6, 10, 1e-06)
The function does not pass through zero for the given interval.
None

PASS: Output matches expected output.
```

[37]:
```
test(ps.bisection_root, (golden, 0, 2, 0.000001), 1.6180343627929688)
```

```
Expect bisection_root(<function golden at 0x1155131f0>, 0, 2, 1e-06) to be
1.6180343627929688
>>> bisection_root(<function golden at 0x1155131f0>, 0, 2, 1e-06)
Found root in 18 iterations.
1.6180343627929688

PASS: Output matches expected output.
```

### 1.4.4 Problem 2.4 - A Physical Application: Projectile Range Maximization

Nancy is at the top of a hill located at the origin $(x, y, z) = (0, 0, 0)$. She throws a baseball in the $x$-$y$ plane so that its initial velocity is $(v\cos\theta, v\sin\theta, 0)$ where $v > 0$ is its initial speed. The hill slopes down linearly away from the origin at an angle $\phi$ below the $x$-axis. The **range** of the basebell is the distance it travels in the $x$-direction before impacting the hill.

- Determine an expression for the range $r$ of the baseball in terms of $v, \phi, \theta$.
- For a given $v$ and $\phi$, determine the angle $\theta$ at which the range is maximized. You should find that it's independent of $v$.
- Find a way use your bisection root finding function to computationally determine the maximum range of the baseball for $\phi = 0, \pi/8, \pi/4, 3\pi/8$, and check your numerical results against the exact results.

**(1)** Since we only care about the range in the x-direction, we can simplify this to a 2-D kinematics problem.

Should I be assuming that $y$ is up, not $z$?

The velocity in the x-direction is $v\cos\theta$, and it is thrown down a hill with a downward slope of angle $\phi$. The ball also has an upward velocity of $v\sin\theta$.

The range that the ball travels in the x-direction is:

$$r = \frac{v^2}{g \cos^2 \phi} \left(\sin\left(2\theta + \phi\right) + \sin \phi\right)$$

**(2)** To maximize $r$ with respect to $\theta$, we want to maximize $\sin\left(2\theta + \phi\right)$.

In other words, find $\theta$ when $\sin\left(2\theta + \phi\right) = 1$.

$$2\theta + \phi = \frac{\pi}{2}$$

$$\theta = \frac{\pi}{4} - \frac{\phi}{2}$$

**(3)** Solve:

$$\frac{dr}{d\theta} = \frac{d}{d\theta} \frac{v^2}{g \cos^2 \phi} \left(\sin\left(2\theta + \phi\right) + \sin \phi\right) = 0$$

$$\frac{4v^2}{g} \frac{\cos\left(2\theta + \phi\right)}{1 + \cos\left(2\phi\right)} = 0$$

$$\cos\left(2\theta + \phi\right) = 0$$

[38]:
```
import math
def dr(phi):
    return lambda theta: math.cos(2 * theta + phi)
```

[39]:
```
phi = 0
test(ps.bisection_root, (dr(phi), 0, math.pi/2, 0.000001), 0.7853981633974483)

print(f"Expect {math.pi / 4 - phi / 2}")
```

```
Expect bisection_root(<function dr.<locals>.<lambda> at 0x11551b160>, 0,
1.5707963267948966, 1e-06) to be 0.7853981633974483
>>> bisection_root(<function dr.<locals>.<lambda> at 0x11551b160>, 0,
1.5707963267948966, 1e-06)
Found root in 1 iterations.
0.7853981633974483

PASS: Output matches expected output.
Expect 0.7853981633974483
```

[40]:
```
phi = math.pi / 8
test(ps.bisection_root, (dr(phi), 0, math.pi/2, 0.000001), 0.5890486225480862)

print(f"Expect {math.pi / 4 - phi / 2}")
```

```
Expect bisection_root(<function dr.<locals>.<lambda> at 0x11551baf0>, 0,
1.5707963267948966, 1e-06) to be 0.5890486225480862
>>> bisection_root(<function dr.<locals>.<lambda> at 0x11551baf0>, 0,
1.5707963267948966, 1e-06)
Found root in 3 iterations.
0.5890486225480862

PASS: Output matches expected output.
Expect 0.5890486225480862
```

[41]:
```
phi = math.pi / 4
test(ps.bisection_root, (dr(phi), 0, math.pi/2, 0.000001), 0.39269908169872414)

print(f"Expect {math.pi / 4 - phi / 2}")
```

```
Expect bisection_root(<function dr.<locals>.<lambda> at 0x1155271f0>, 0,
1.5707963267948966, 1e-06) to be 0.39269908169872414
>>> bisection_root(<function dr.<locals>.<lambda> at 0x1155271f0>, 0,
1.5707963267948966, 1e-06)
Found root in 2 iterations.
0.39269908169872414

PASS: Output matches expected output.
Expect 0.39269908169872414
```

[42]:
```
phi = 3 * math.pi / 8
test(ps.bisection_root, (dr(phi), 0, math.pi/2, 0.000001), 0.19634954084936207)

print(f"Expect {math.pi / 4 - phi / 2}")
```

```
Expect bisection_root(<function dr.<locals>.<lambda> at 0x115527670>, 0,
1.5707963267948966, 1e-06) to be 0.19634954084936207
>>> bisection_root(<function dr.<locals>.<lambda> at 0x115527670>, 0,
1.5707963267948966, 1e-06)
Found root in 3 iterations.
0.19634954084936207

PASS: Output matches expected output.
Expect 0.19634954084936207
```

## 1.5 Problem Group 3 - Fun with primes

### 1.5.1 Problem 3.1 - Sieve of Eratosthenes

Do an internet search for "Sieve of Eratosthenes" and read up on it. Convince yourself that you understand how it works, then write a Python function

`sieve_Eratosthenes`

that takes an integer $n \geq 2$ as its input and outputs a Python list all prime numbers less than

or equal to $n$. Make sure to test your function for some low $n$ for which the answer can easily be computed by hand.

[43]:
```
primes_less_than_100 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,␣
↪53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
test(ps.sieve_Eratosthenes, 100, primes_less_than_100)
```

```
Expect sieve_Eratosthenes(100) to be [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
>>> sieve_Eratosthenes(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97]

PASS: Output matches expected output.
```

### 1.5.2 Problem 3.2 - Prime Factorization

Write a Python function

`prime_factors`

with takes an integer $n \geq 2$ as its input and outputs a Python list with the prime factors of $n$ as its output. If a certain prime factor appears more than once, say $k$ times, in the prime factorization of $n$, then it should appear that many times in the list returned by your function. For example, running your function on the number 250 should yield the list `[2, 5, 5, 5]`.

[44]:
```
prime_factors_of_250 = [2, 5, 5, 5]
test(ps.prime_factors, 250, prime_factors_of_250)
```

```
Expect prime_factors(250) to be [2, 5, 5, 5]
>>> prime_factors(250)
[2, 5, 5, 5]

PASS: Output matches expected output.
```

[45]:
```
prime_factors_of_600851475143 = [71, 839, 1471, 6857]
test(ps.prime_factors, 600851475143, prime_factors_of_600851475143)
```

```
Expect prime_factors(600851475143) to be [71, 839, 1471, 6857]
>>> prime_factors(600851475143)
[71, 839, 1471, 6857]

PASS: Output matches expected output.
```

## 1.6 Problem Group 4 - Weighty problem

Work with your partner to solve this problem. Once you have a working solution, try to make it as efficient possible.

Group codename: (Replace this text with the codename for your group. This can be any reasonably appropriate word with lowercase alphabet characters. E.g., batman)

A variant of a problem posed in the 16th century:

A merchant had a forty pound measuring weight that broke into $n$ pieces as the result of a fall. When the pieces were subsequently weighed, it was found that the weight of each piece was a whole number of pounds and that the $n$ pieces could be used to weigh every integral weight between 1 and 40 pounds. What is the minimum value of $n$ and what were the weights of the pieces?

(Remember that the merchant had a weighing scale and weights could be added to either side of the scale.

This problem is known as "The Weight Problem of Bachet de Meziriac"

It is important to note that an integral weight can be weighed by using a two sided scale, and a weight can be measured by placing pieces on both sides. So for some set of pieces, the weights we are able to measure is any combination of adding or subtracting the pieces, or not including them at all.

For example, to calculate a total weight of `2`, using pieces `1` and `3`, we take `3 - 1 = 2`.

[46]:
```
# O(1) Solution
pieces = [1, 3, 9, 27]
```

Solve by induction.

In order to weigh every integral weight between 1 and 40 pounds, the first piece must be `1`.

Consider a set of weights `a`, `b`, `c`. Let `n = a + b + c` be the sum of weights we currently have. The weight `n` is the maximum weight we can weigh with weights `a`, `b`, `c`. Assume that we can measure any weight `j` in the range `[1, n]` by some combination of `a`, `b`, and `c`.

Let `p = 2n + 1`, be the next weight in the set. I claim that with the set of weights `a`, `b`, `c`, `p`, I can measure every weight less than or equal to `a + b + c + p`.

Any weight in range `[n + 1, p - 1]` can be measured by the combination `p - j` where `j` some weight in the range `[1, n]`.

Any weight in range `[p + 1, n + p]` can be measured by the combination `p + j`.

Therefore, by adding weight `p`, we can measure any weight in the range `[1, a + b + c + p]`, completing induction.

[47]:
```
test(ps.weight_set, (40), [1, 3, 9, 27])
```

```
Expect weight_set(40) to be [1, 3, 9, 27]
>>> weight_set(40)
[1, 3, 9, 27]

PASS: Output matches expected output.
```

It just so happens that the solution by induction produces a set that sums to 40.

[48]:
```
pieces = ps.weight_set(40)
if sum(pieces) == 40:
```

```
    print("The solution works, and the sum of all pieces is the total weight of␣
 ↪40 pounds.")
    print(f"The weight broke into {len(pieces)} pieces.")
else:
    print("The sum of the pieces does not add to the total weight of 40 pounds.
 ↪")
```

```
The solution works, and the sum of all pieces is the total weight of 40 pounds.
The weight broke into 4 pieces.
```