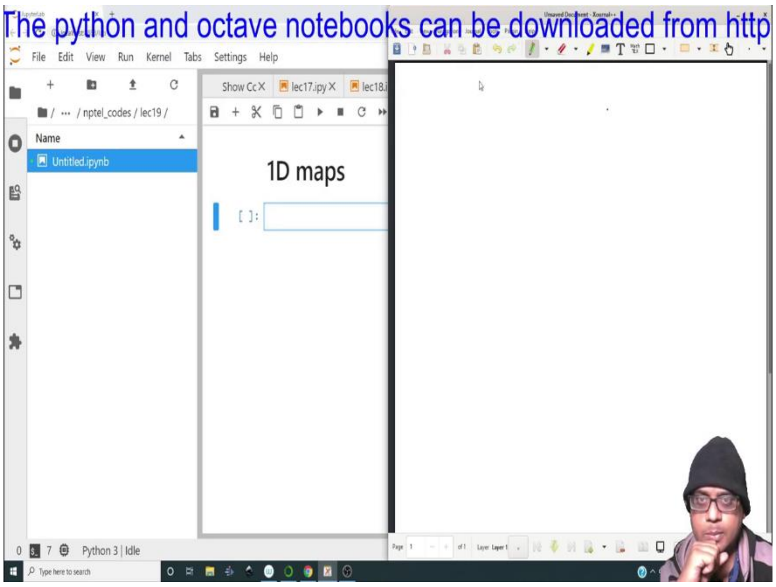
Tools in Scientific Computing
Prof. Aditya Bandopadhyay
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

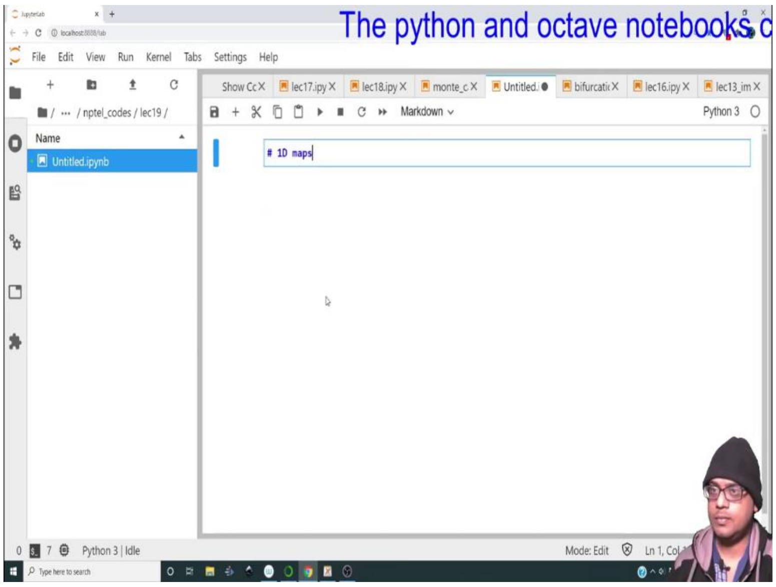
Lecture – 19
1D Maps

(Refer Slide Time: 00:34)


The python and octave notebooks can be downloaded from http
File Edit View Run Kernel Tabs Settings Help
/ ··· / nptel_codes / lec19 /
Name
Untitled.ipynb
1D maps
[ ]:
Python 3 | Idle

Hello everyone. In this lecture, we are going to look at fixed point iterations or more generally in the context of non-linear dynamics.

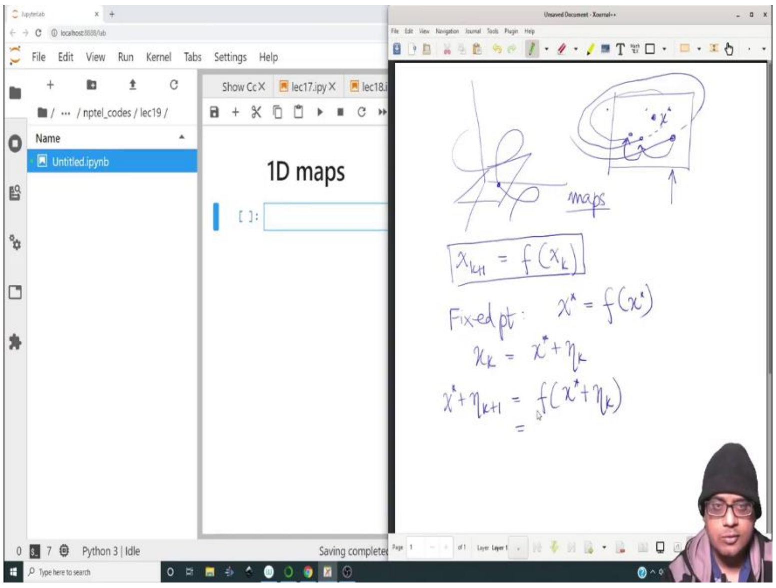
(Refer Slide Time: 00:43)


The python and octave notebooks c
File Edit View Run Kernel Tabs Settings Help
/ ··· / nptel_codes / lec19 /
Name
Untitled.ipynb
Markdown
Python 3
Mode: Edit
Python 3 | Idle

```
# 1D maps
```

We are going to look at 1D Maps. So, what are 1D maps?

(Refer Slide Time: 00:45)


File Edit View Run Kernel Tabs Settings Help
/ ··· / nptel_codes / lec19 /
Name
Untitled.ipynb
Show CcX
lec17.ipy X
Python 3 | Idle
Saving completed

1D maps

[ ]:

maps

x^*

$$\boxed{x_{k+1} = f(x_k)}$$

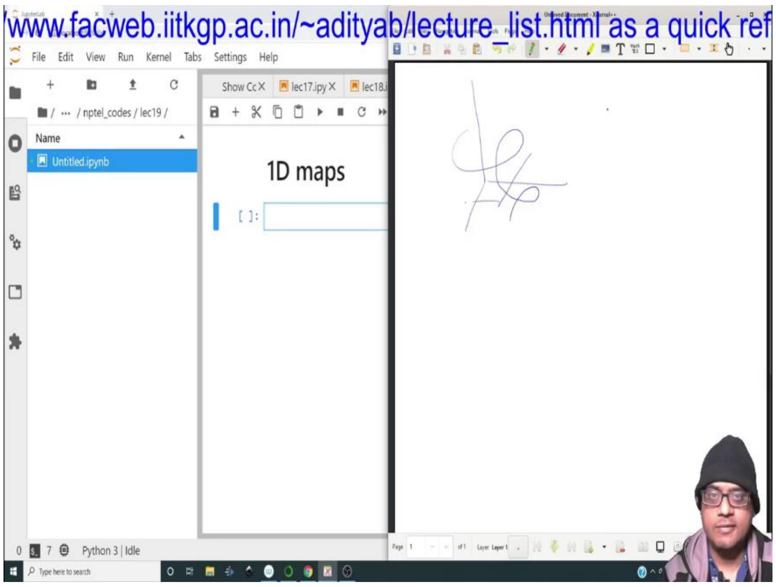
Fixed pt : $x^* = f(x^*)$

$x_k = x^* + \eta_k$

$x^* + \eta_{k+1} = f(x^* + \eta_k)$

$=$

(Refer Slide Time: 00:53)


File Edit View Run Kernel Tabs Settings Help
/ ··· / nptel_codes / lec19 /
Name
Untitled.ipynb
Show CcX
lec17.ipy X
Python 3 | Idle

www.facweb.iitkgp.ac.in/~adityab/lecture_list.html as a quick ref

1D maps

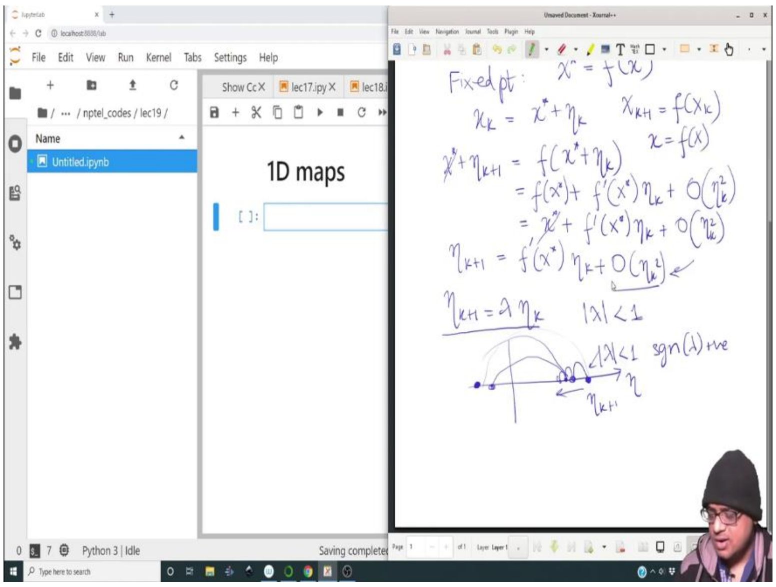
[ ]:

So, let us go back a step backward and distill the essence of a non-linear equations or in phase space right, we have an evolution of a trajectory, but through the Poincare section, we have seen that if we have a transect, it could be a time transact because of periodicity or it could be a space transact as well. You can have a reduced dynamics which you can analyze on a plane for example, in this case.

So, we can have a plane like this so, the trajectory can go like this, come back again, intersect over here so, this is 1, 2 and it can do this several times. So, we are looking at how on this particular plane, a point is mapped onto the point that point is then mapped on to this point and so on. So, such kinds of discrete observations are classified as maps alright. So, we say that x[k+1] is f(x[k]) where x[k] is then mapped to x[k +1].

So, in the context of 1-dimensional maps, these are just points on a line and we will look at the evolution of this and we have seen examples of this when we have analyzed the non-linear solvers, algebraic solver and how it led to oscillations or all these kinds of things. So, now, we will have a bit more insight and we will see how a simple map can give rise to seemingly complicated things.

(Refer Slide Time: 03:53)



Fixed pt: $x^* = f(x)$

$x_k = x^* + \eta_k \qquad x_{k+1} = f(x_k)$

$x = f(x)$

$\cancel{x^*} + \eta_{k+1} = f(x^* + \eta_k)$

$= f(x^*) + f'(x^*)\eta_k + O(\eta_k^2)$

$= \cancel{x^*} + f'(x^*)\eta_k + O(\eta_k^2)$

$\eta_{k+1} = f'(x^*)\eta_k + \underline{O(\eta_k^2)}$

$\underline{\eta_{k+1} = \lambda\eta_k} \qquad |\lambda| < 1$

$|\lambda| < 1 \quad \text{sgn}(\lambda) \text{ +ve}$

So, right away, we will observe that if we have a fixed point, then x^* if it is the fixed point is going to be $f(x^*)$. So, this is by definition a fixed point. The point is mapped onto itself; it is not moving ok. So, more generally, a map can be defined as this. Now, let us look at the behavior of a map in the vicinity of a fixed point.

So, let x[k] be written in terms of $x^* + \eta_k$ where η_k is like the correction to the fixed term alright. So, we are looking in the neighborhood of a fixed point. So, if this is the fixed point, we are looking over here. So, now x[k+1] can be written as $x^* + \eta_{k+1} = f(x^* + \eta_k)$ and we can make use of the Taylor series and write this as $f(x^*) + f'(x^*)\eta_k + \mathcal{O}(\eta_k^2)$

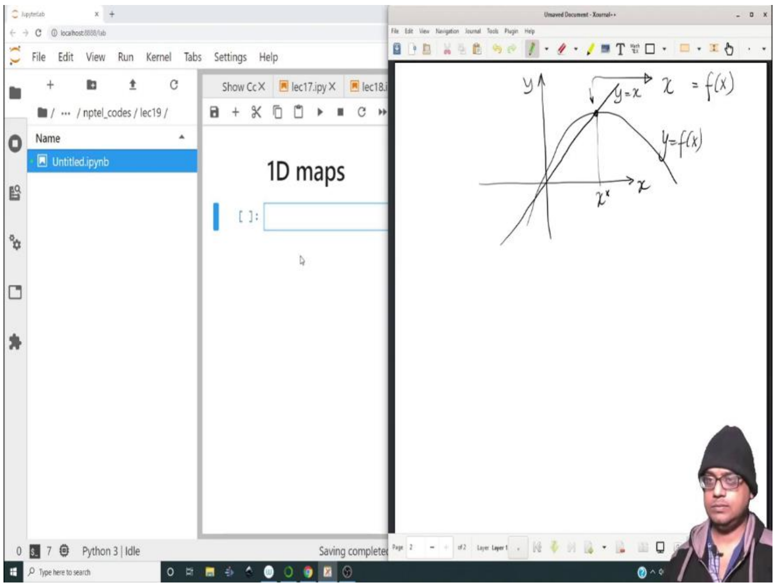
Now, because x^* is a fixed point, x^* is going to be = f(x) so, basically this is becoming f(x^*)= x^* and so, we have this. So, this cancels out and thus, we have an evolution of the correction to the fixed point as $\eta_{k+1} = f'(x^*)\eta_k + \mathcal{O}(\eta_k^2)$

Now, if this can be neglected compared to this, then the evolution is quite simple to write and it is $\eta_{k+1} = f'(x^*)\eta_k = \lambda\eta_k$. Now, the η the rather the η_k will decay if the magnitude of λ is less than 1 right. So, imagine, this is the η space, and this is η_{k+1} so, if magnitude is less than 1, then it will map on to something like this and it will map onto something like this, then this, then this. So, each time you are sort of converging towards something.

So, this is the case where $|\lambda|$, the magnitude of λ is less than 1 and the sign of lambda is positive. If the sign of lambda is negative and the magnitude remains less than 1, then this point will be mapped on to this point over here which will be then mapped on to this point, then on to this point.

So, because of the flipping of the sign for each iterate, you will have an oscillatory convergence. If lambda is = 1 and quite obviously, $\eta_{k+1} = \eta_k$ and then, you have to resort to analyzing the higher orders that you have neglected over here ok. So, now graphically how does that particular map that is x[k+1] is = f(x[k]) look or in fact, what is the solution graphically of this.

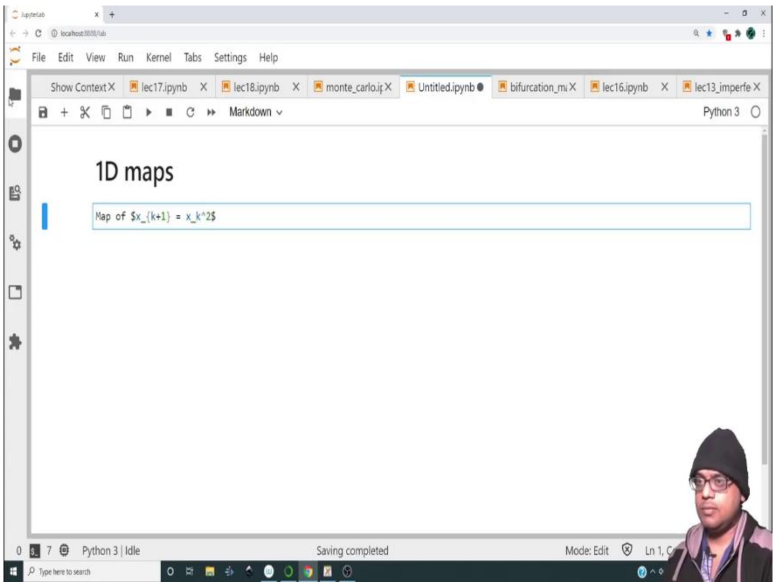
(Refer Slide Time: 06:31)


1D maps

And we have actually had a look into this, it is not that difficult to recall that on one hand you have x, on the other hand you have f(x). So, for example, let this be that particular f(x) and so, this is y = f(x) and this is x and this is y and this function is simply a straight line which goes something like this, it intersects the function over here. So, at this particular point, we have x = f(x).

So, this line is y= x. So, at this particular point, we have y =x and that is the solution, this is the fixed point so, this is the fixed point. So, now, the slope of the function f(x) at the fixed point governs what the behavior of the convergence will be. If the slope is bounded that is of the magnitude is less than 1, then we will have convergence if it is not, then we will diverge. Let us go to the computer and see how this looks like.

(Refer Slide Time: 07:53)

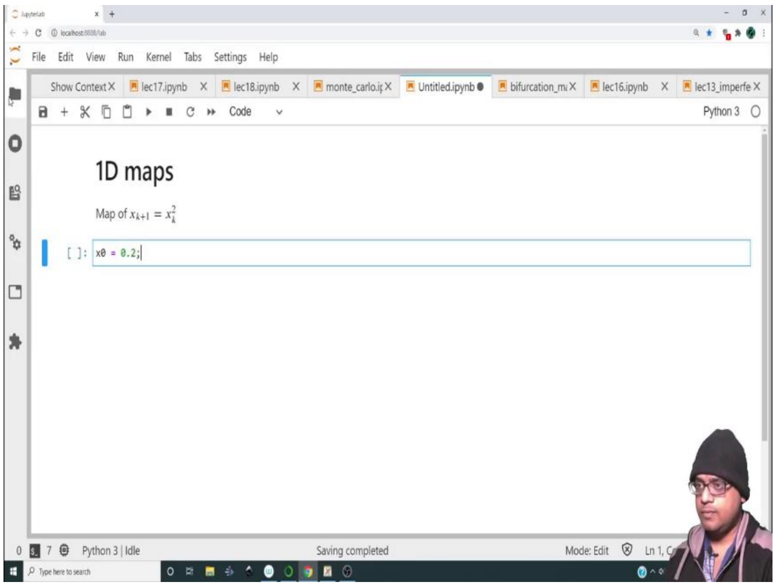


1D maps

```
Map of $x_{k+1} = x_k^2$
```

So, let us have a look into the map of, let us look at Map of $x_{k+1} = x_k^2$ ok. So, let us see how this looks like.

(Refer Slide Time: 08:08)



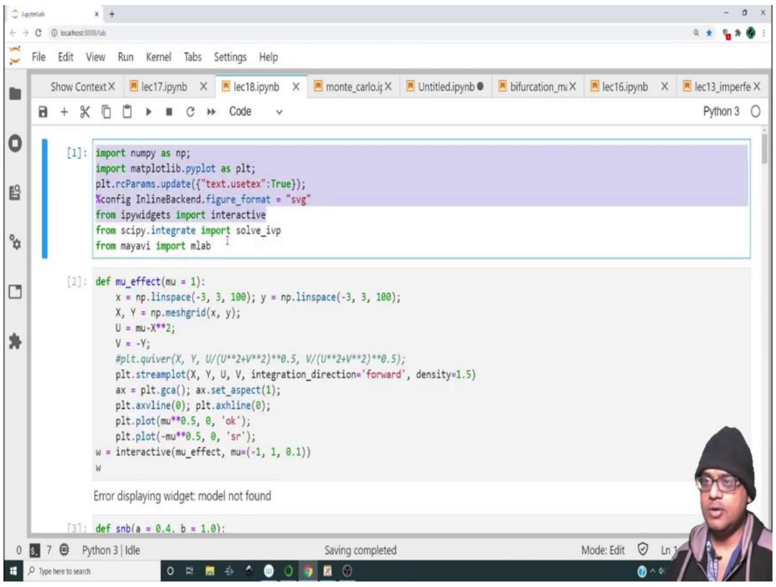
1D maps

Map of $x_{k+1} = x_k^2$

```
[ ]: x0 = 0.2;
```

So, first things first, we will need an initial value x0 = 0.2or in fact, even before starting this, let us first plot these things.

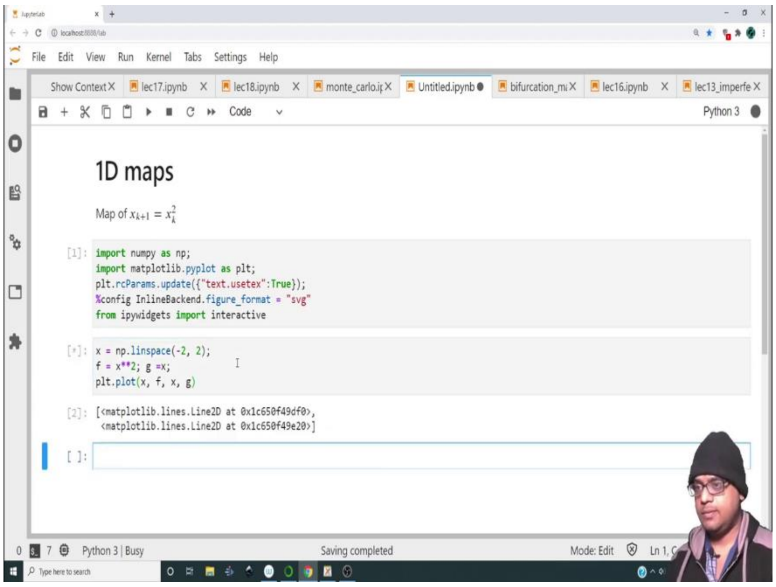
(Refer Slide Time: 08:28)



```
[1]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     from ipywidgets import interactive
     from scipy.integrate import solve_ivp
     from mayavi import mlab

[2]: def mu_effect(mu = 1):
         x = np.linspace(-3, 3, 100); y = np.linspace(-3, 3, 100);
         X, Y = np.meshgrid(x, y);
         U = mu-X**2;
         V = -Y;
         #plt.quiver(X, Y, U/(U**2+V**2)**0.5, V/(U**2+V**2)**0.5);
         plt.streamplot(X, Y, U, V, integration_direction='forward', density=1.5)
         ax = plt.gca(); ax.set_aspect(1);
         plt.axvline(0); plt.axhline(0);
         plt.plot(mu**0.5, 0, 'ok');
         plt.plot(-mu**0.5, 0, 'sr');
     w = interactive(mu_effect, mu=(-1, 1, 0.1))
     w

     Error displaying widget: model not found

[3]: def snb(a = 0.4, b = 1.0):
```

So, let x = n. So, we have to first import all the libraries alright. So, we only need this, we do not need solve_ivp and mlab over here alright.

(Refer Slide Time: 08:36)



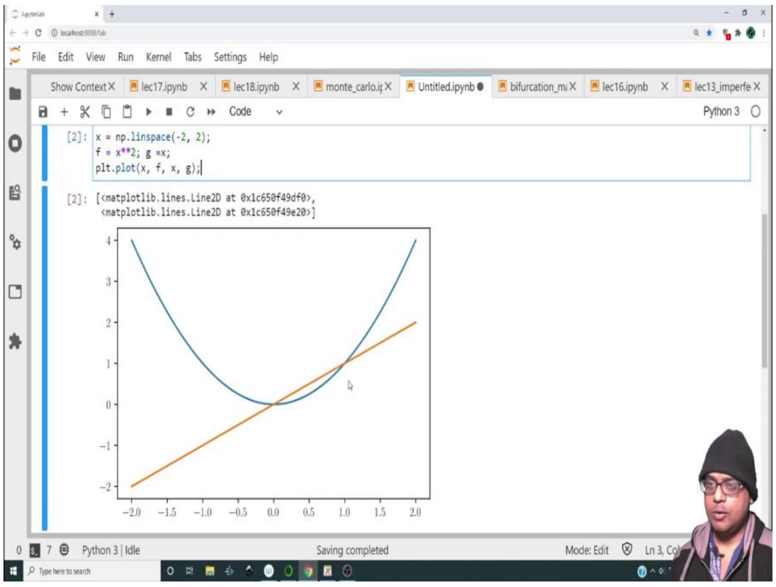
1D maps

Map of $x_{k+1} = x_k^2$

```
[1]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     from ipywidgets import interactive

[*]: x = np.linspace(-2, 2);
     f = x**2; g =x;
     plt.plot(x, f, x, g)

[2]: [<matplotlib.lines.Line2D at 0x1c650f49df0>,
      <matplotlib.lines.Line2D at 0x1c650f49e20>]

[ ]:
```

So, let me run this. Then, x = np.linspace(-2 ,2); f = x**2 and g = x, then we will do plt.plot (x ,f, x ,g) alright.

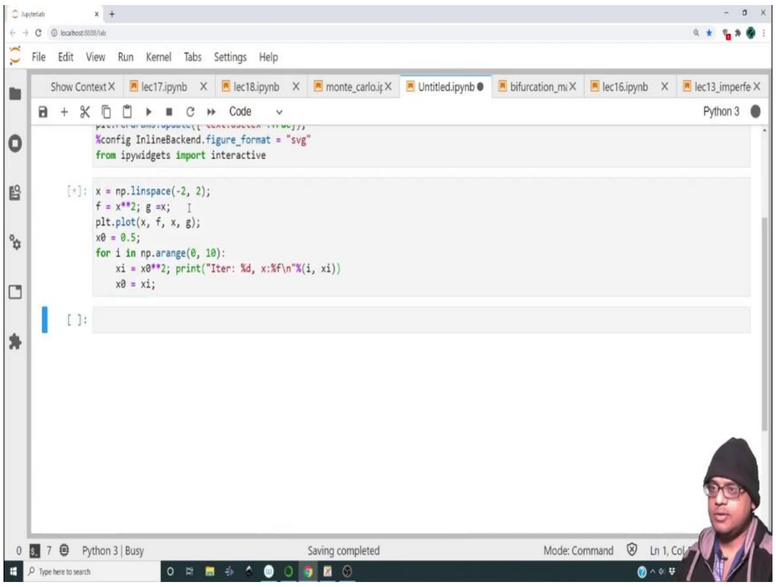
(Refer Slide Time: 09:07)



```
[2]: x = np.linspace(-2, 2);
     f = x**2; g =x;
     plt.plot(x, f, x, g);

[2]: [<matplotlib.lines.Line2D at 0x1c650f49df0>,
      <matplotlib.lines.Line2D at 0x1c650f49e20>]
```

So, this is the fixed point, obviously, x = 1 is the fixed point. So, now, what is the slope at x = 1? So, the slope at x = 1 so, this is the function so, it is going to be 2x and x = 1 the slope is greater than 1 it is 2. So, we expect that point to be an unstable point.

(Refer Slide Time: 09:49)



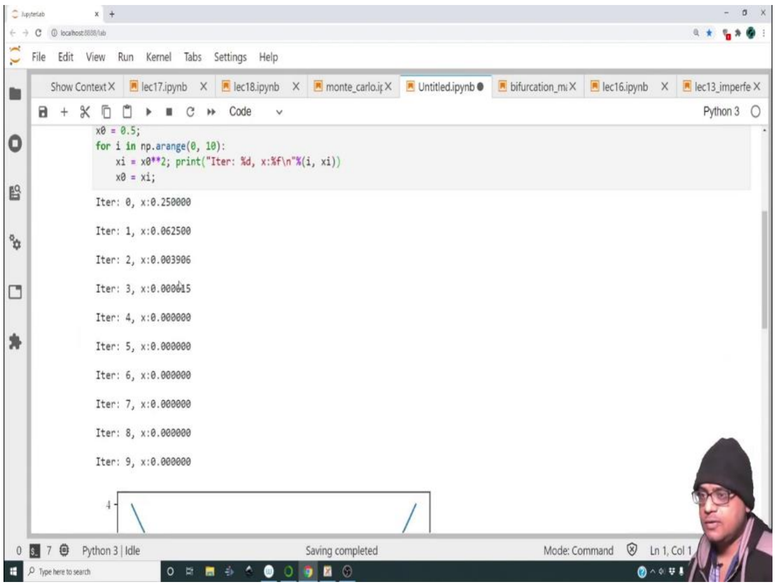
```
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive
```

```
[*]: x = np.linspace(-2, 2);
     f = x**2; g =x;
     plt.plot(x, f, x, g);
     x0 = 0.5;
     for i in np.arange(0, 10):
         xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;
```

So, now, let us perform the iterations and actually see whether an initial guess converges to that point or not. So, for that, let us in fact, reuse this code. So, we already have the plot of this. So, let x0 = say 0.5 alright, then let us first print out all the values.

So, for let us do 10 iterations for i in np.arange(0 ,10), xi= x0**2 and then, we will assign x0 as xi and just after this, we will print the value of i and so, the iteration number and we will print out the value of the present iterate or x simply. So, let it be %f and here, we will print i and we will print xi alright. So, let me run this.

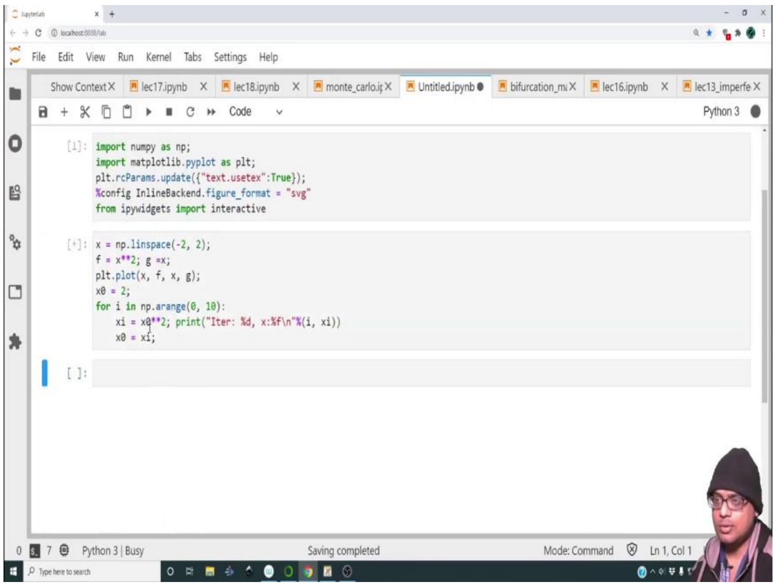
(Refer Slide Time: 11:02)



```
x0 = 0.5;
for i in np.arange(0, 10):
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

Iter: 0, x:0.250000

Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015

Iter: 4, x:0.000000

Iter: 5, x:0.000000

Iter: 6, x:0.000000

Iter: 7, x:0.000000

Iter: 8, x:0.000000

Iter: 9, x:0.000000
```

So, it starts with x = 0.25 and then, it is converging to 0, it is converging to this root so, obviously, this is also a root and over here, the slope is 0 and this attracts ok. So, immediately, we can say that this is a stable root.

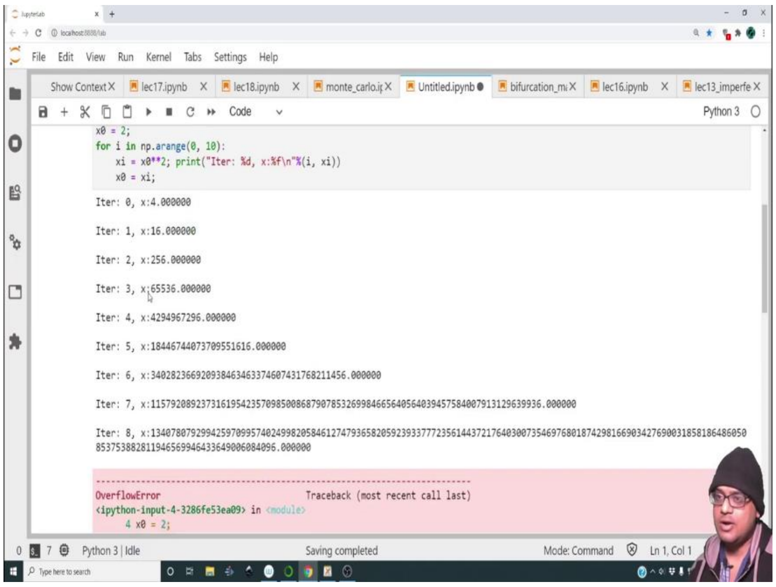
(Refer Slide Time: 11:24)



```
[1]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     from ipywidgets import interactive

[*]: x = np.linspace(-2, 2);
     f = x**2; g =x;
     plt.plot(x, f, x, g);
     x0 = 2;
     for i in np.arange(0, 10):
         xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

[ ]:
```

Let me change the initial condition to 2 alright.

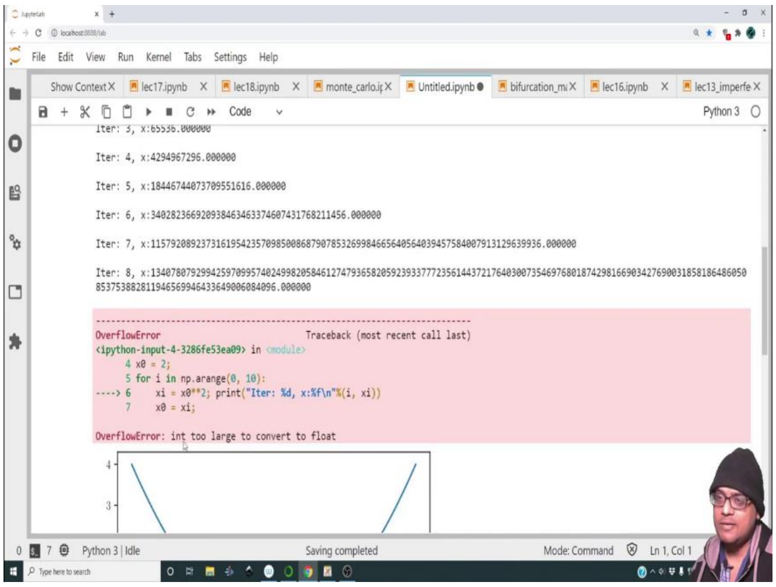
(Refer Slide Time: 11:25)



```
x0 = 2;
for i in np.arange(0, 10):
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

Iter: 0, x:4.000000

Iter: 1, x:16.000000

Iter: 2, x:256.000000

Iter: 3, x:65536.000000

Iter: 4, x:4294967296.000000

Iter: 5, x:18446744073709551616.000000

Iter: 6, x:340282366920938463463374607431768211456.000000

Iter: 7, x:115792089237316195423570985008687907853269984665640564039457584007913129639936.000000

Iter: 8, x:134078079299425970995740249982058461274793658205923933777235614437217640300735469768018742981669034276900318581864860508
53753882811946569946433649006084096.000000

---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-4-3286fe53ea09> in <module>
      4 x0 = 2;
```

When we start at 2, the iterations are growing out of bound and we have a very large value.

(Refer Slide Time: 11:31)

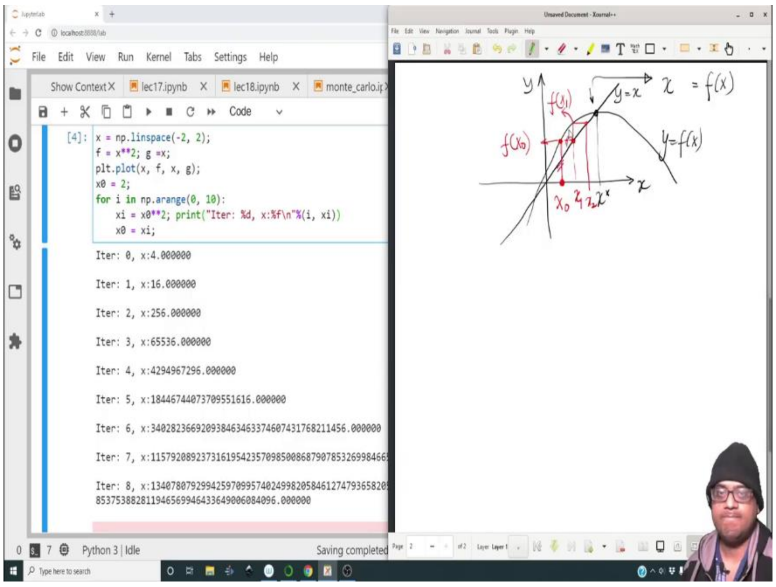


```
Iter: 3, x:65536.000000

Iter: 4, x:4294967296.000000

Iter: 5, x:18446744073709551616.000000

Iter: 6, x:340282366920938463463374607431768211456.000000

Iter: 7, x:115792089237316195423570985008687907853269984665640564039457584007913129639936.000000

Iter: 8, x:134078079299425970995740249982058461274793658205923933777235614437217640300735469768018742981669034276900318581864860508537538828119465699464336490060840 96.000000
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-4-3286fe53ea09> in <module>
      4 x0 = 2;
      5 for i in np.arange(0, 10):
----> 6     xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
      7     x0 = xi;

OverflowError: int too large to convert to float
```

And then, we have floating point error. Well, it is a representation error more like, it is an overflow error because it is getting too large for the computer to represent through it is the finite byte size that it has. So, then we would like to ideally plot what this series of iterations look like. So, before that, let us see per iteration what is really going on.

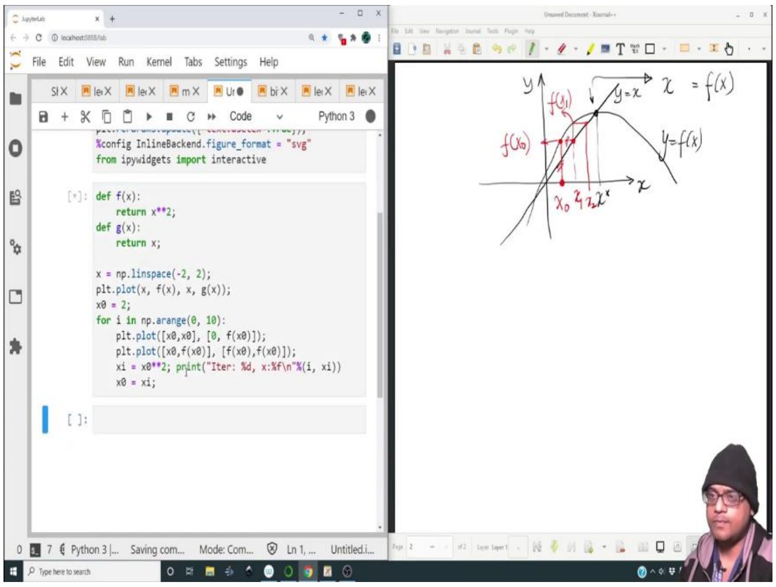
(Refer Slide Time: 12:04)



```
[4]: x = np.linspace(-2, 2);
     f = x**2; g =x;
     plt.plot(x, f, x, g);
     x0 = 2;
     for i in np.arange(0, 10):
         xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

Iter: 0, x:4.000000

Iter: 1, x:16.000000

Iter: 2, x:256.000000

Iter: 3, x:65536.000000

Iter: 4, x:4294967296.000000

Iter: 5, x:18446744073709551616.000000

Iter: 6, x:340282366920938463463374607431768211456.000000

Iter: 7, x:1157920892373161954235709850086879078532699846

Iter: 8, x:1340780792994259709957402499820584612747936582
8537538828119465699464336490060840 96.000000
```

So, suppose we have this as x0. So, the first point will be f of so, this particular point is f(x0), then we assign it to x1 so, this is assigned to this so, this becomes x1, then this becomes f(x1), this gets assigned to x2 and so on.

(Refer Slide Time: 13:04)



```
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive
```

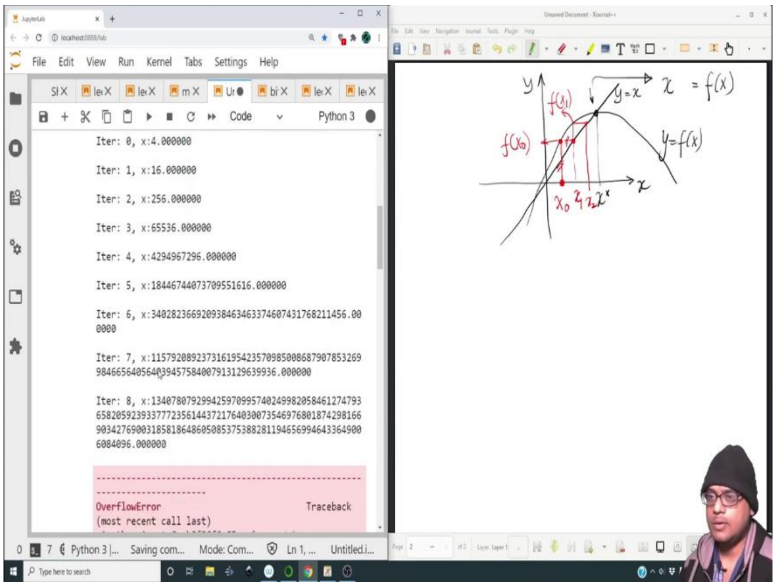
```
def f(x):
    return x**2;
def g(x):
    return x;

x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = 2;
for i in np.arange(0, 10):
    plt.plot([x0,x0], [0, f(x0)]);
    plt.plot([x0,f(x0)], [f(x0),f(x0)]);
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, let us try to achieve this particular plot on top of this. So, per new point we have to plot first, this particular line and this particular line, then so, per iterate, we have to plot the two sets of lines; one is this and one is this alright. So, alright. So, let me do it over here. So, we will first plot plt.plot so, it will be x0, f(x0) so, let me write it like this, we will convert the functions later and this we have to give a pair of x coordinates so, this also will be this and this will be 0 alright.

Then, once the new point is found, we will plot in fact, we can do it over here plt.plot the line will be going from x0, f(x0) and this second point will be f(x0) and this will be the same point so, f(x0). Now, the task is to define what f is.

So, we will go over here alright, we will declare this as a function. So, this is the function, and we will define g(x) as simply return x. So, now, this will be f(x) and this will be g(x). So, let us run this and see what happens. So, I forgot the commas alright ok.

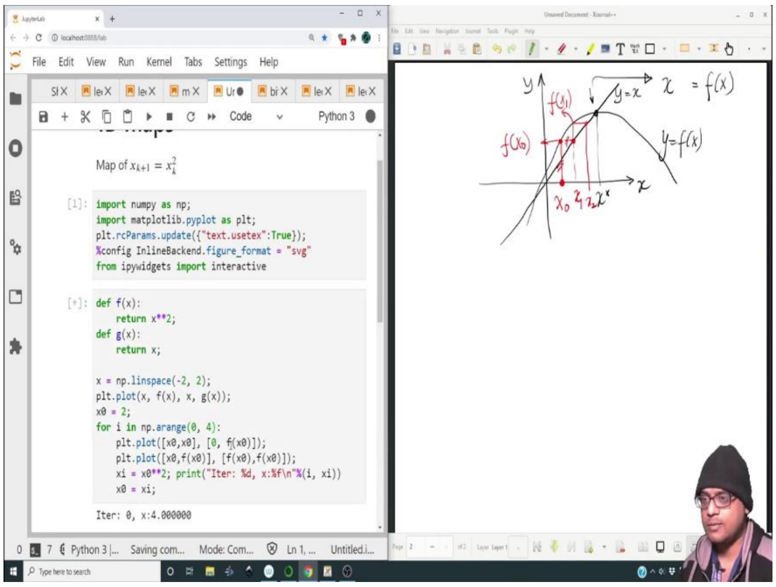
(Refer Slide Time: 14:51)



```
Iter: 0, x:4.000000

Iter: 1, x:16.000000

Iter: 2, x:256.000000

Iter: 3, x:65536.000000

Iter: 4, x:4294967296.000000

Iter: 5, x:18446744073709551616.000000

Iter: 6, x:340282366920938463463374607431768211456.00
0000

Iter: 7, x:115792089237316195423570985008687907853269
984665640564039457584007913129639936.000000

Iter: 8, x:134078079299425970995740249982058461274793
658205923933777235614437217640300735469768018742981669
034276900318581864860508537538828119465699464336490060
84096.000000

---------------------------------------------------------------------------
OverflowError                             Traceback
(most recent call last)
```

There is a ok.

(Refer Slide Time: 15:07)

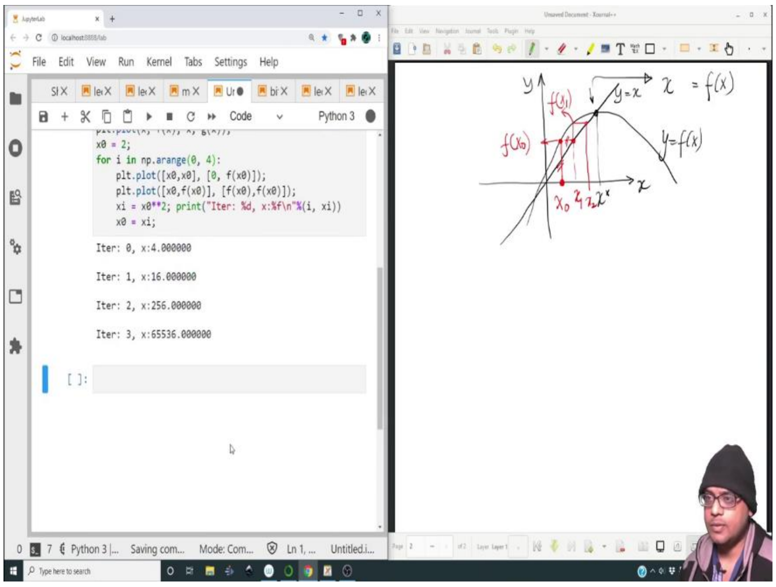


Map of $x_{k+1} = x_k^2$

```
[1]: import numpy as np;
     import matplotlib.pyplot as plt;
     plt.rcParams.update({"text.usetex":True});
     %config InlineBackend.figure_format = "svg"
     from ipywidgets import interactive

[*]: def f(x):
         return x**2;
     def g(x):
         return x;

     x = np.linspace(-2, 2);
     plt.plot(x, f(x), x, g(x));
     x0 = 2;
     for i in np.arange(0, 4):
         plt.plot([x0,x0], [0, f(x0)]);
         plt.plot([x0,f(x0)], [f(x0),f(x0)]);
         xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

Iter: 0, x:4.000000
```

We need to make lesser number of iterations or maybe only 4 alright.

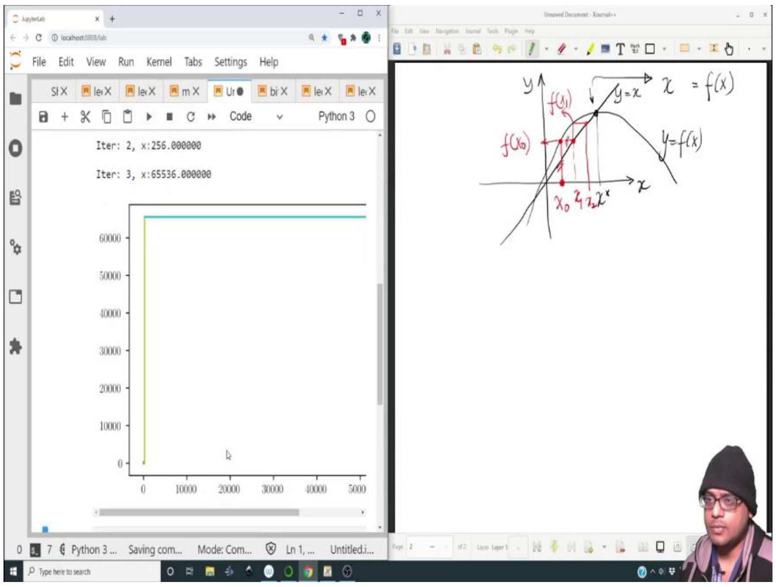
(Refer Slide Time: 15:08)



```
x0 = 2;
for i in np.arange(0, 4):
    plt.plot([x0,x0], [0, f(x0)]);
    plt.plot([x0,f(x0)], [f(x0),f(x0)]);
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

```
Iter: 0, x:4.000000

Iter: 1, x:16.000000

Iter: 2, x:256.000000

Iter: 3, x:65536.000000
```

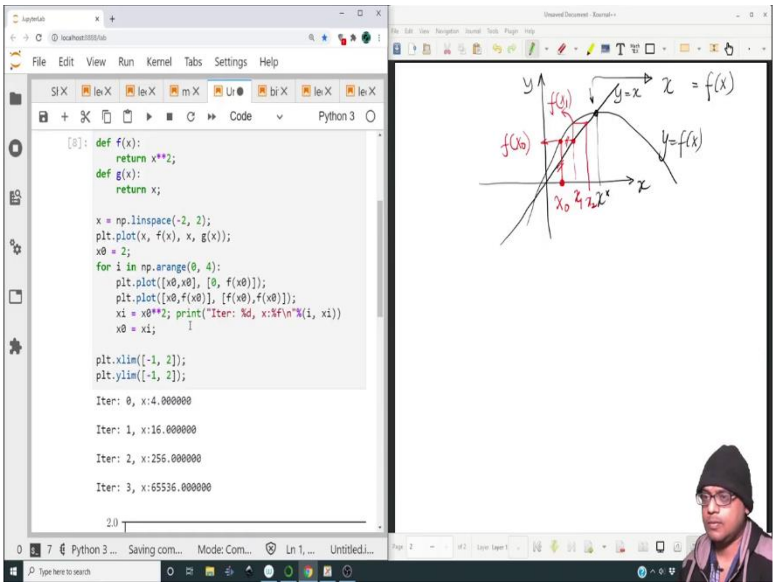
(Refer Slide Time: 15:12)



```
Iter: 2, x:256.000000

Iter: 3, x:65536.000000
```

It is flying away. So, let me focus our attention in the vicinity.

(Refer Slide Time: 15:19)

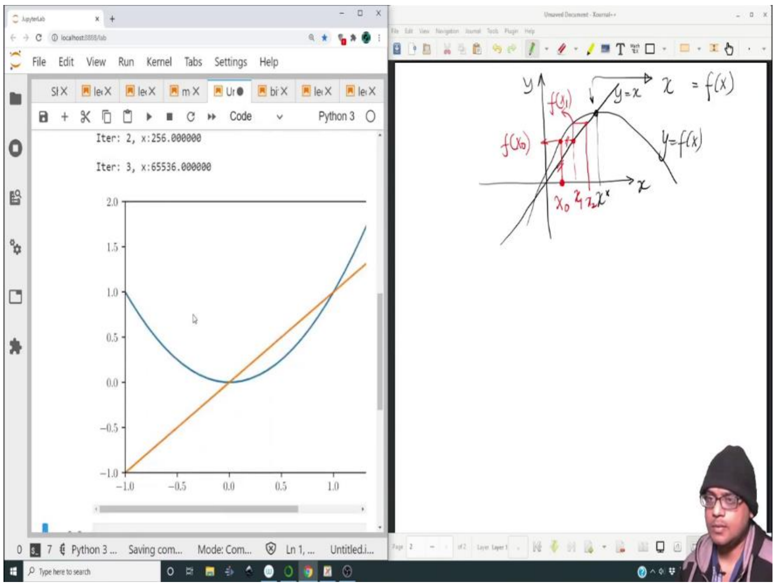


```
[8]: def f(x):
         return x**2;
     def g(x):
         return x;

     x = np.linspace(-2, 2);
     plt.plot(x, f(x), x, g(x));
     x0 = 2;
     for i in np.arange(0, 4):
         plt.plot([x0,x0], [0, f(x0)]);
         plt.plot([x0,f(x0)], [f(x0),f(x0)]);
         xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

     plt.xlim([-1, 2]);
     plt.ylim([-1, 2]);
```

```
Iter: 0, x:4.000000

Iter: 1, x:16.000000

Iter: 2, x:256.000000

Iter: 3, x:65536.000000
```

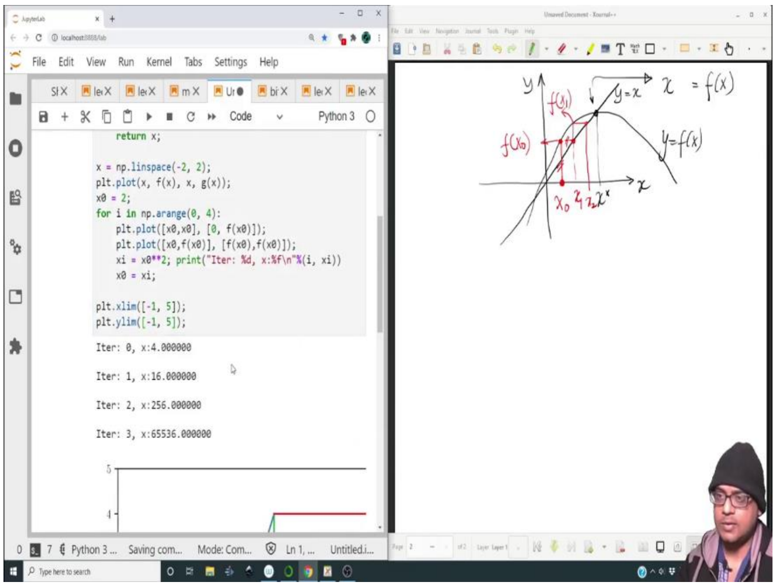
So, I need to change the plot limits.

(Refer Slide Time: 15:37)



```
Iter: 2, x:256.000000

Iter: 3, x:65536.000000
```

(Refer Slide Time: 15:45)

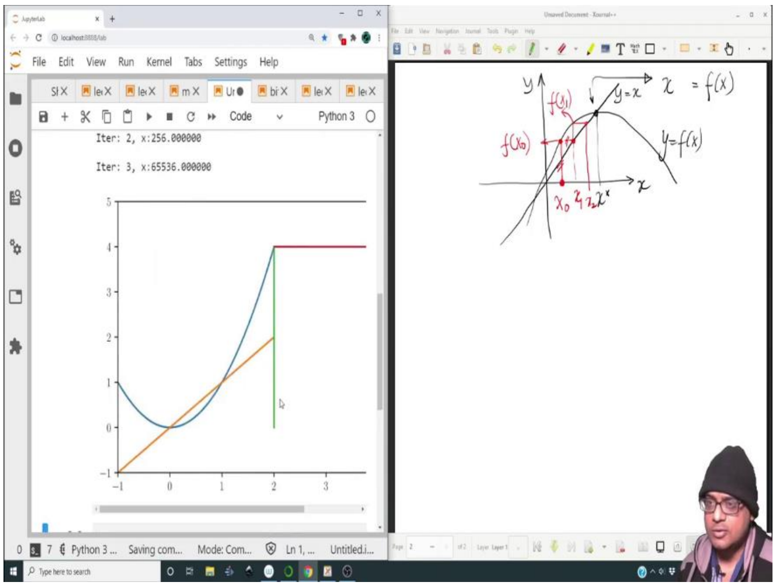


```
    return x;

x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = 2;
for i in np.arange(0, 4):
    plt.plot([x0,x0], [0, f(x0)]);
    plt.plot([x0,f(x0)], [f(x0),f(x0)]);
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 5]);
plt.ylim([-1, 5]);
```

```
Iter: 0, x:4.000000

Iter: 1, x:16.000000

Iter: 2, x:256.000000

Iter: 3, x:65536.000000
```

Sorry, I need to make it at least 5 so that we can actually see the point.

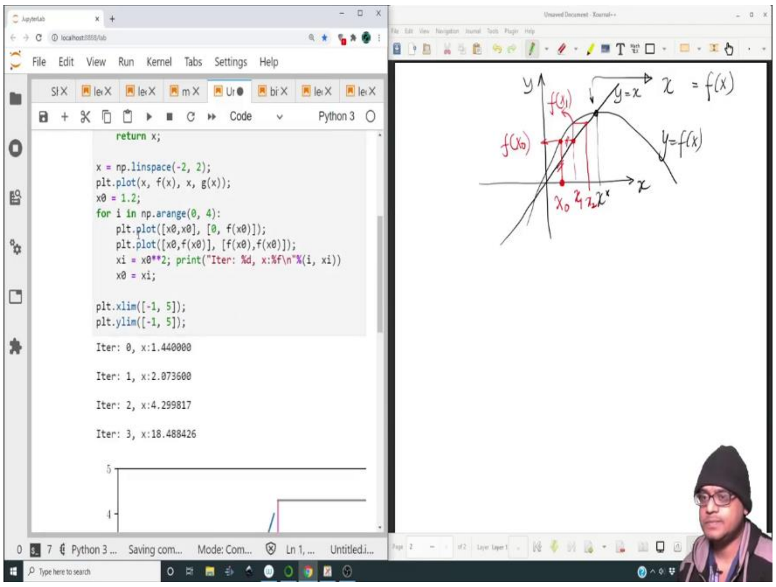
(Refer Slide Time: 15:47)



```
Iter: 2, x:256.000000

Iter: 3, x:65536.000000
```

So, it is. So, starting over here so, then, it is flying off to the infinity.

(Refer Slide Time: 16:01)

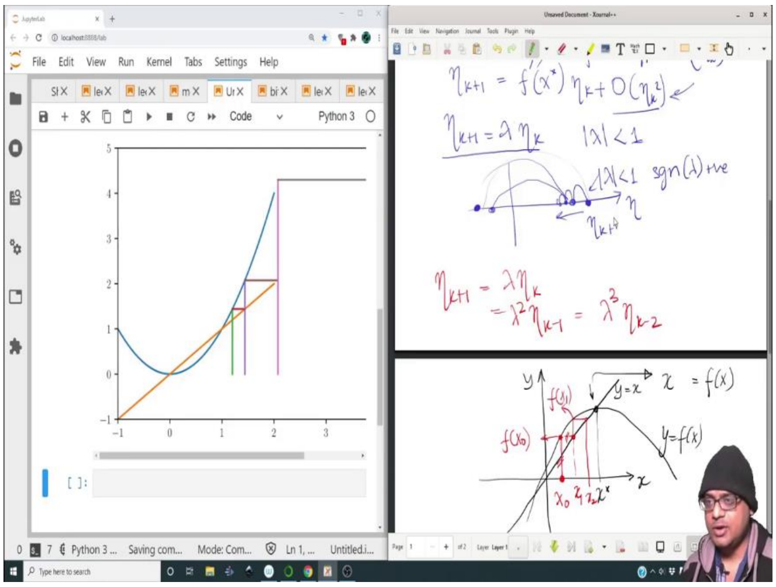


```
    return x;

x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = 1.2;
for i in np.arange(0, 4):
    plt.plot([x0,x0], [0, f(x0)]);
    plt.plot([x0,f(x0)], [f(x0),f(x0)]);
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 5]);
plt.ylim([-1, 5]);
```

```
Iter: 0, x:1.440000

Iter: 1, x:2.073600

Iter: 2, x:4.299817

Iter: 3, x:18.488426
```

In fact, let me choose 1.2 as the initial point that will be better for us to visualize.

(Refer Slide Time: 16:04)

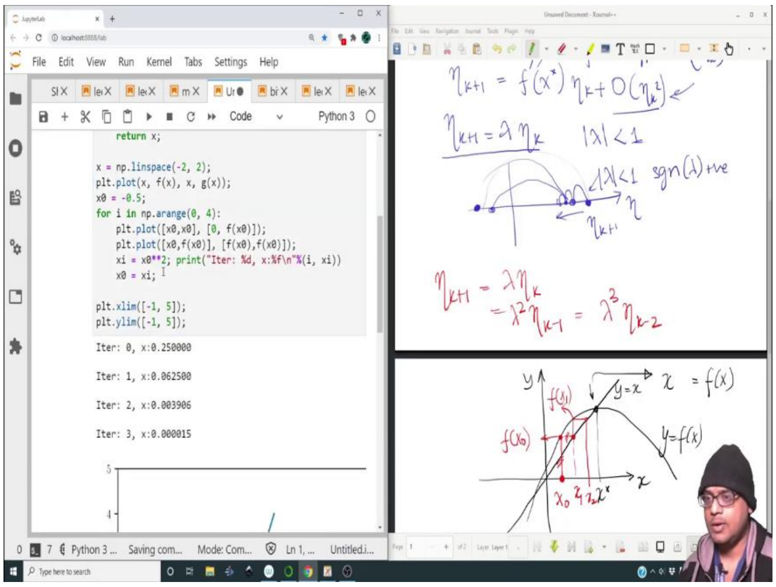


$\eta_{k+1} = f'(x^*)\,\eta_k + O(\eta_k^2)$
$\eta_{k+1} = \lambda\,\eta_k$ $|\lambda|<1$
$|\lambda|<1$ sgn(λ) +ve
η
η_{k+1}
$\eta_{k+1} = \lambda\eta_k = \lambda^2\eta_{k-1} = \lambda^3\eta_{k-2}$
y
$y=x$
$x = f(x)$
$f(x_1)$
$f(x_0)$
$y=f(x)$
x
$x_0\ x_1\ x_2\ x^*$

So, it is starting over here, then it is going to this, then this point is mapped over here, then to this and so on. So, it eventually flies off to infinity. So, because of this slope every iteration is getting a bump up so, the value of λ is greater than 1. So, after every iteration $\eta_{k+1} = \lambda\eta_k$ so, this is going to be $\lambda^2\eta_{k-1}$ and which is going to be $\lambda^3\eta_{k-2}$ and so on.

So, because of the multiplicity in λ, if λ, the magnitude of λ is greater than 1, then this will fly off to infinity as we can see over here whereas, if you take a lower value of the root, it will appear to converge.

(Refer Slide Time: 17:04)



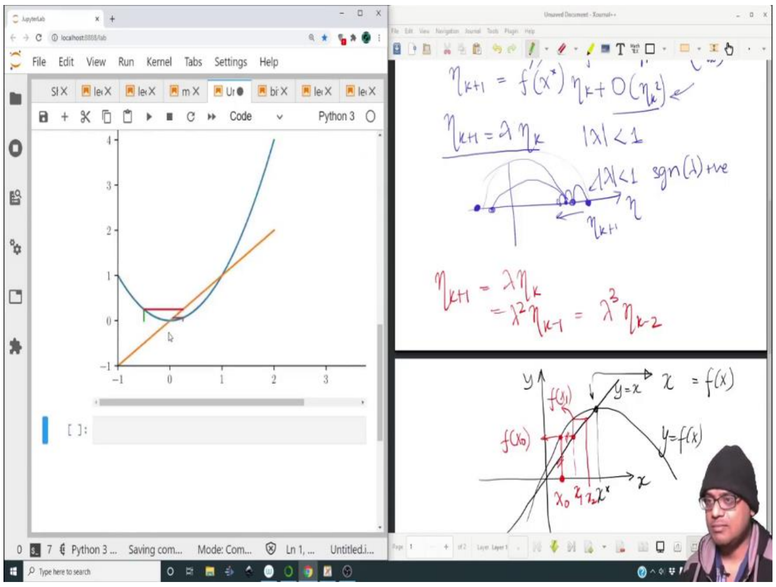
```
    return x;

x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = -0.5;
for i in np.arange(0, 4):
    plt.plot([x0,x0], [0, f(x0)]);
    plt.plot([x0,f(x0)], [f(x0),f(x0)]);
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 5]);
plt.ylim([-1, 5]);
```

```
Iter: 0, x:0.250000

Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

$\eta_{k+1} = f'(x^*)\eta_k + O(\eta_k^2)$
$\eta_{k+1} = \lambda\eta_k \quad |\lambda| < 1$
$|\lambda| < 1 \quad sgn(\lambda)$ +ve
η_{k+1}
$\eta_{k+1} = \lambda\eta_k = \lambda^2\eta_{k-1} = \lambda^3\eta_{k-2}$
$y=x$
$x = f(x)$
$f(x_0)$
$y=f(x)$
$x_0\ x_1\ x_2\ x^*$

So, if we focus on this particular fixed point, so, let me take a guess as - 0.5, we will see that it converges to the other fixed point.

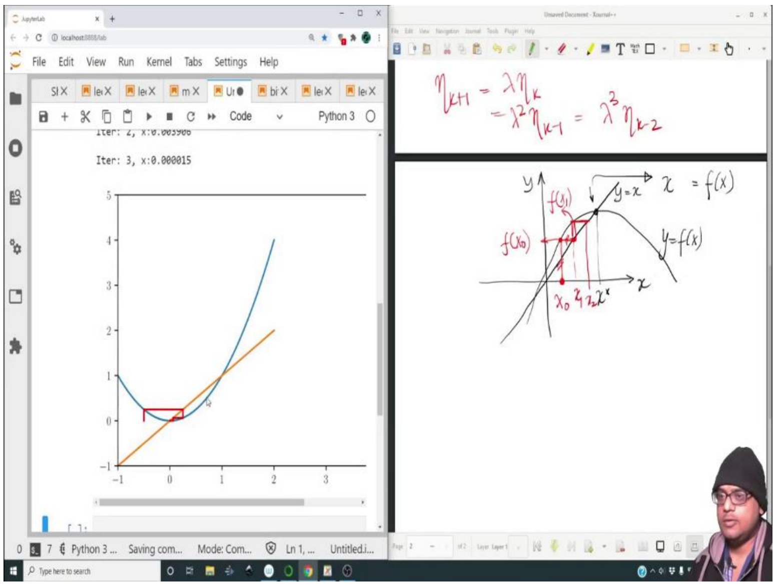
(Refer Slide Time: 17:06)



$\eta_{k+1} = f'(x^*)\eta_k + O(\eta_k^2)$
$\eta_{k+1} = \lambda\eta_k \quad |\lambda| < 1$
$|\lambda| < 1 \quad sgn(\lambda)$ +ve
$\eta_{k+1} = \lambda\eta_k = \lambda^2\eta_{k-1} = \lambda^3\eta_{k-2}$
$y=x$
$x = f(x)$
$y=f(x)$
$x_0\ x_1\ x_2\ x^*$

But the other fixed point is not that I mean it is not that interesting because it is a trivial sort of fixed point so, it is converging like this ok. So, the point that I am trying to make is it depends on what the value of the root is at the point and depending on that, we will achieve convergence or divergence.

(Refer Slide Time: 18:08)

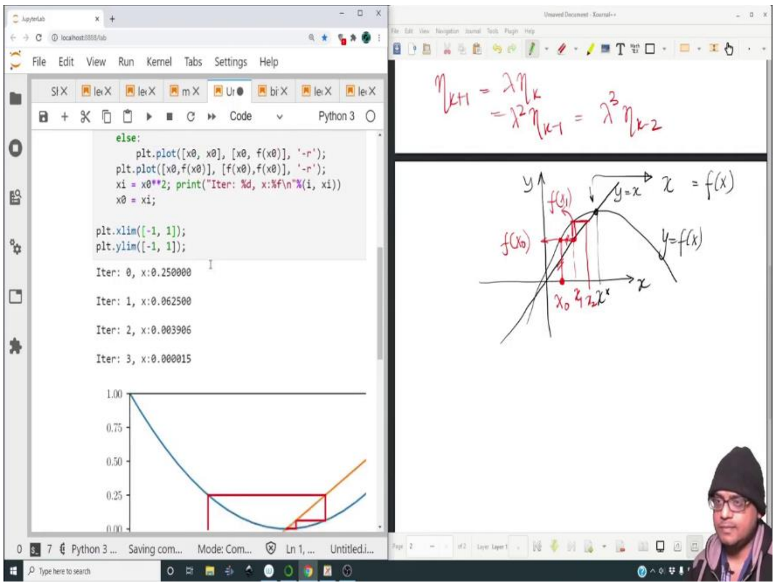


```
Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

$\eta_{k+1} = \lambda \eta_k$
$= \lambda^2 \eta_{k-1} = \lambda^3 \eta_{k-2}$
y
$f(y_1)$
$y=x$
$x = f(x)$
$f(x_0)$
$y=f(x)$
x
$x_0\ y_1\ x_2\ x^*$

So, let me just fix the plotting so that it does not plot every time from the base. So, the first point, so, let us just start plotting it like this alright. So, this is how the plot looks like.

(Refer Slide Time: 18:13)



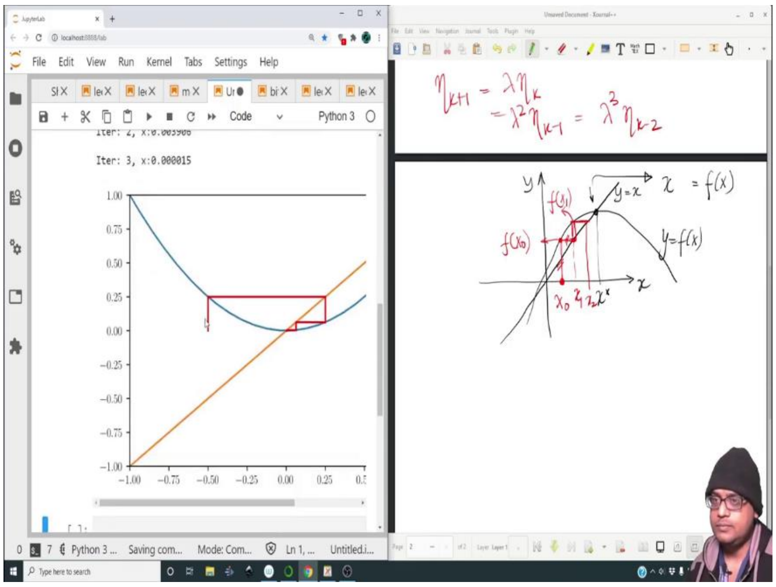
```
        else:
            plt.plot([x0, x0], [x0, f(x0)], '-r');
        plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
        xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
        x0 = xi;

plt.xlim([-1, 1]);
plt.ylim([-1, 1]);
```

```
Iter: 0, x:0.250000

Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

$\eta_{k+1} = \lambda \eta_k$
$= \lambda^2 \eta_{k-1} = \lambda^3 \eta_{k-2}$
y
$f(y_1)$
$y=x$
$x = f(x)$
$f(x_0)$
$y=f(x)$
x
$x_0\ y_1\ x_2\ x^*$

Let me reduce the scale, let me make it something like this.

(Refer Slide Time: 18:16)

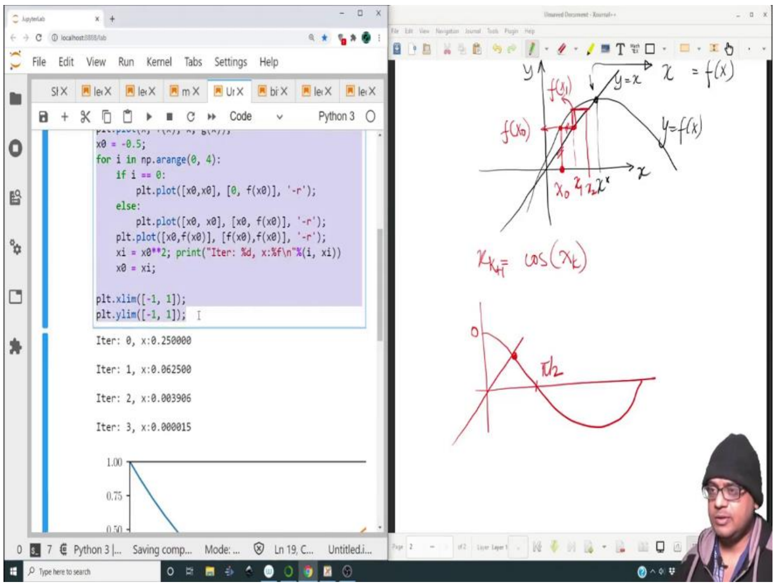


```
Iter: 3, x:0.000015
```

$\eta_{k+1} = \lambda \eta_k$
$= \lambda^2 \eta_{k-1} = \lambda^3 \eta_{k-2}$
y
$y=x$
$x = f(x)$
$f(x_1)$
$f(x_0)$
$y=f(x)$
x
$x_0\ x_1\ x_2\ x^*$

So, we start off with a guess over here, when it is mapped onto this value, then it is mapped onto this, this, and this and then it converges. So, it is pretty fast convergence, and the divergence is pretty wild because of the fact that the root is larger than 1.

(Refer Slide Time: 18:44)



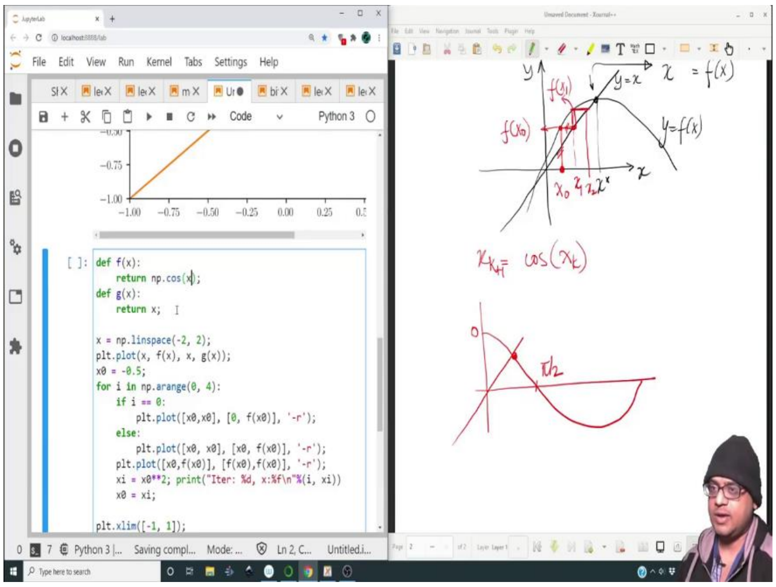
```
x0 = -0.5;
for i in np.arange(0, 4):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 1]);
plt.ylim([-1, 1]);
```

```
Iter: 0, x:0.250000

Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

Now, let us look at the other case where the root does converge. So, let us take a case where x_k or $x_{k+1} = \cos(x_k)$. So, let us look at what graphically the root should be. So, this is the plot of cos and this is the straight line. So, there is a root somewhere over here.

So, this is going to be $\pi/2$. So, there is a root between this fixed point between 0 and $\pi/2$ alright.

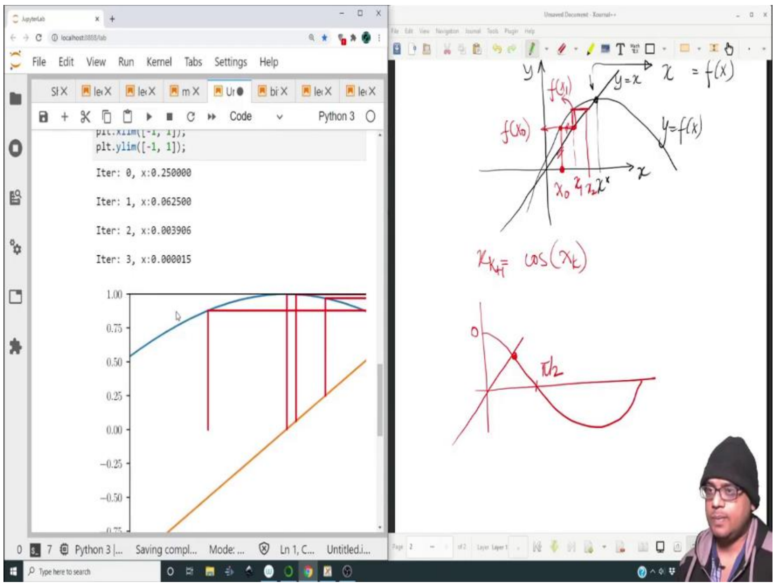
(Refer Slide Time: 19:15)



```
def f(x):
    return np.cos(x);
def g(x):
    return x;

x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = -0.5;
for i in np.arange(0, 4):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 1]);
```

So, let me copy this snippet, let me paste it over here and let me simply make this as np.cos(x). So, let me run this.

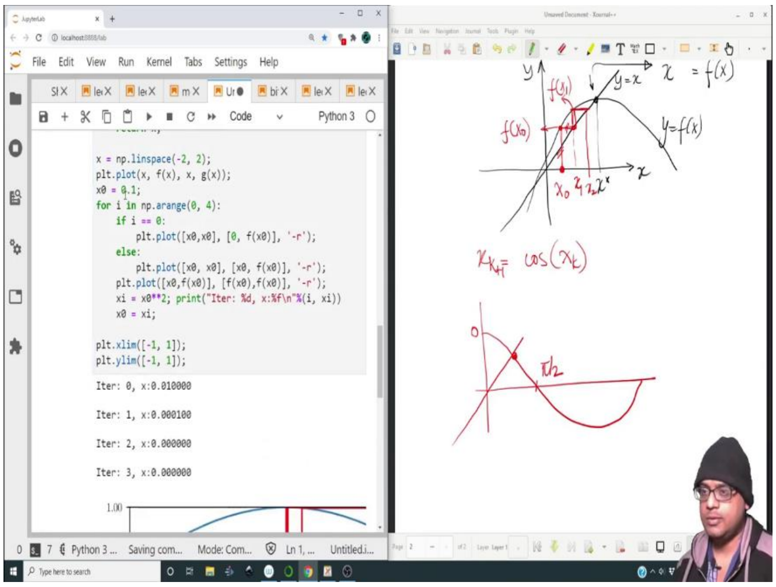
(Refer Slide Time: 19:20)



```
plt.xlim([-1, 1]);
plt.ylim([-1, 1]);
```

```
Iter: 0, x:0.250000

Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

(Refer Slide Time: 19:29)

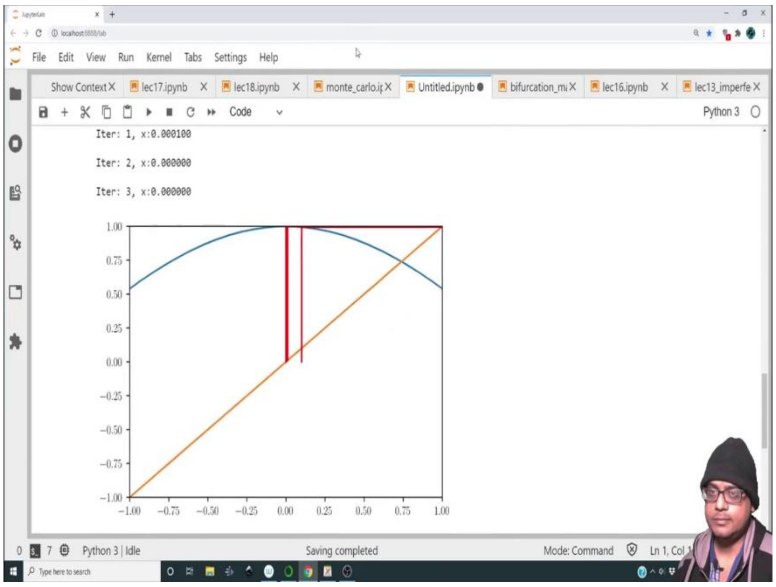


```
x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = 0.1;
for i in np.arange(0, 4):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 1]);
plt.ylim([-1, 1]);
```

```
Iter: 0, x:0.010000

Iter: 1, x:0.000100

Iter: 2, x:0.000000

Iter: 3, x:0.000000
```

So, let me choose an initial point something like 0.1.

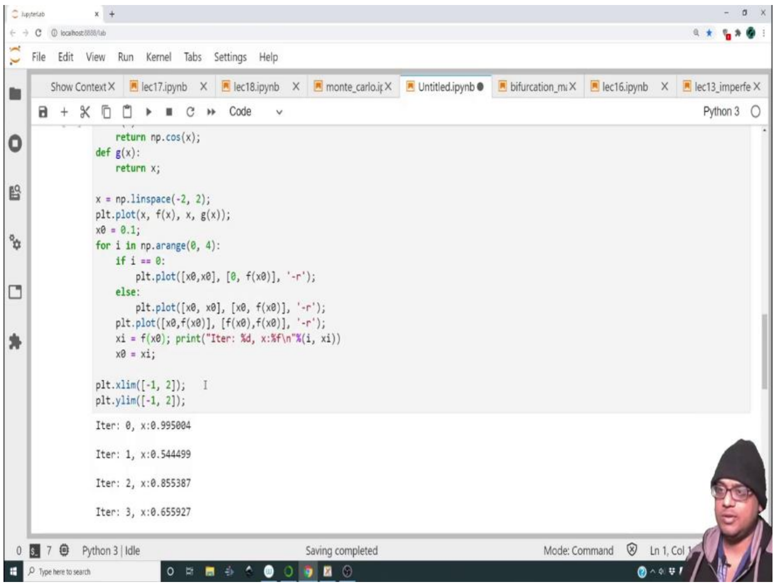
(Refer Slide Time: 19:31)



```
Iter: 1, x:0.000100

Iter: 2, x:0.000000

Iter: 3, x:0.000000
```

So, what is the slope of the root so that is also critical to see?

(Refer Slide Time: 19:55)

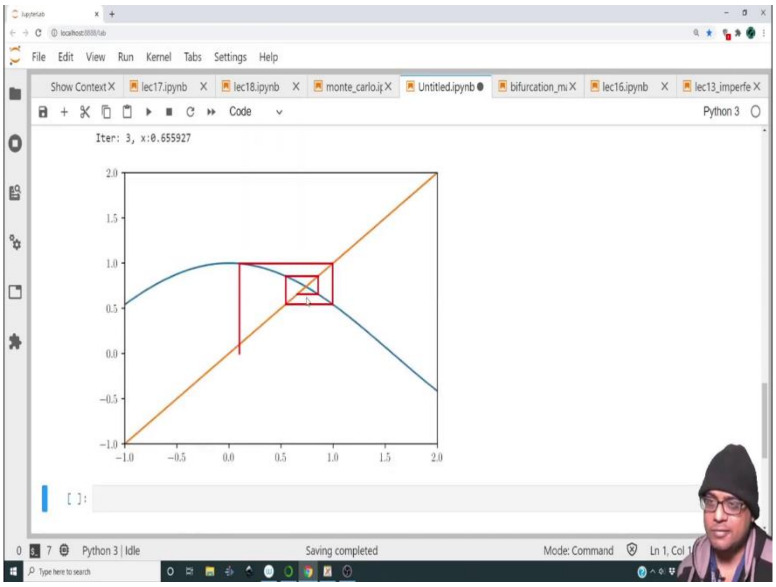


```
    return np.cos(x);
def g(x):
    return x;

x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = 0.1;
for i in np.arange(0, 4):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 2]);
plt.ylim([-1, 2]);
```

```
Iter: 0, x:0.995004

Iter: 1, x:0.544499

Iter: 2, x:0.855387

Iter: 3, x:0.655927
```

Well, let us expand the axis a bit. So, my bad this should have been changed and xi should be f(x0), it should be simply this alright.

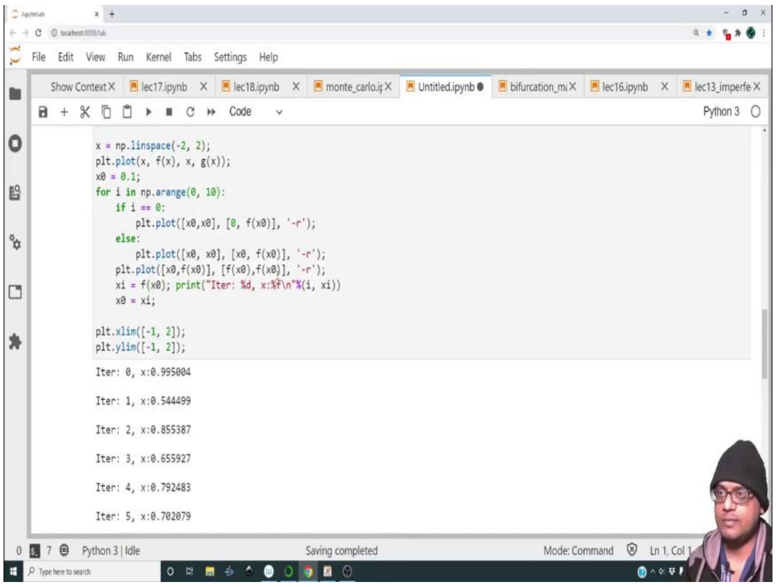
(Refer Slide Time: 20:24)



```
Iter: 3, x:0.655927
```

So, we do have convergence happening.

(Refer Slide Time: 20:37)

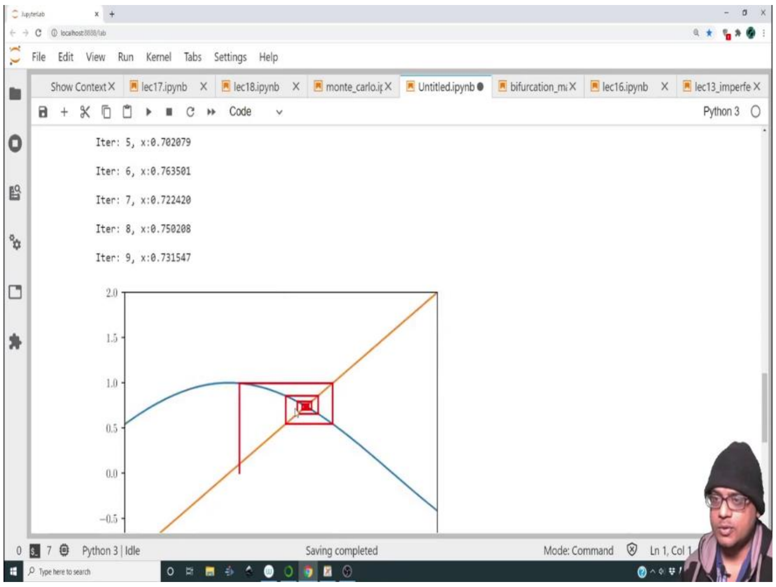


```python
x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));
x0 = 0.1;
for i in np.arange(0, 10):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 2]);
plt.ylim([-1, 2]);
```

```
Iter: 0, x:0.995004

Iter: 1, x:0.544499

Iter: 2, x:0.855387

Iter: 3, x:0.655927

Iter: 4, x:0.792483

Iter: 5, x:0.702079
```

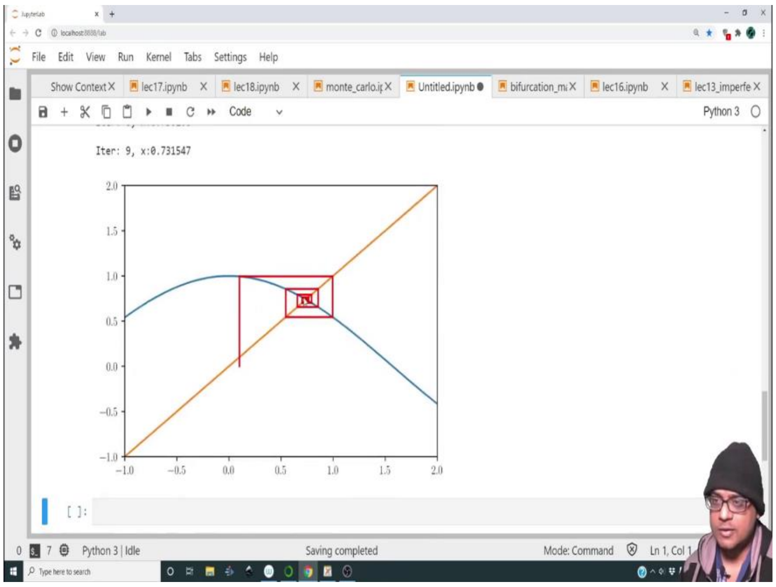
And let me increase the number of iterations to highlight how that convergence takes place. So, let me take 10 iterations.

(Refer Slide Time: 20:38)



```
Iter: 5, x:0.702079

Iter: 6, x:0.763501

Iter: 7, x:0.722420

Iter: 8, x:0.750208

Iter: 9, x:0.731547
```

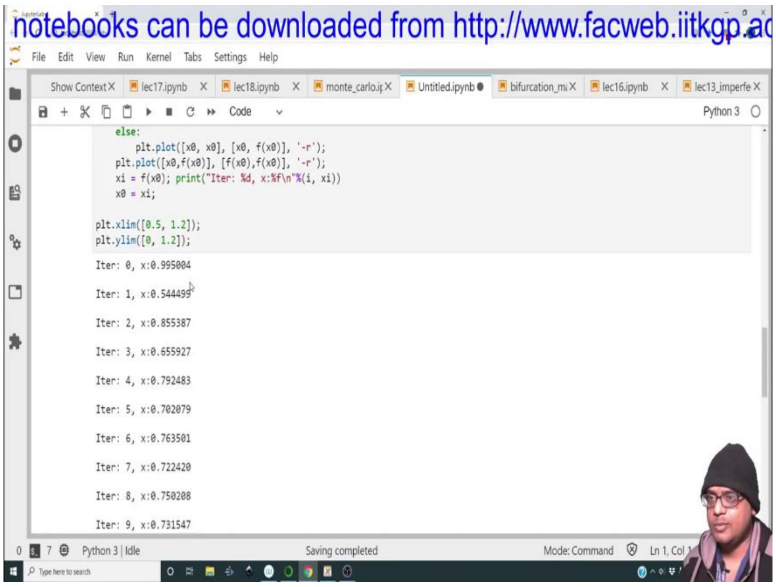
(Refer Slide Time: 20:40)



```
Iter: 9, x:0.731547
```

Boom, so, we do have a convergence and it appears to be somewhere around 0.7.

(Refer Slide Time: 20:51)



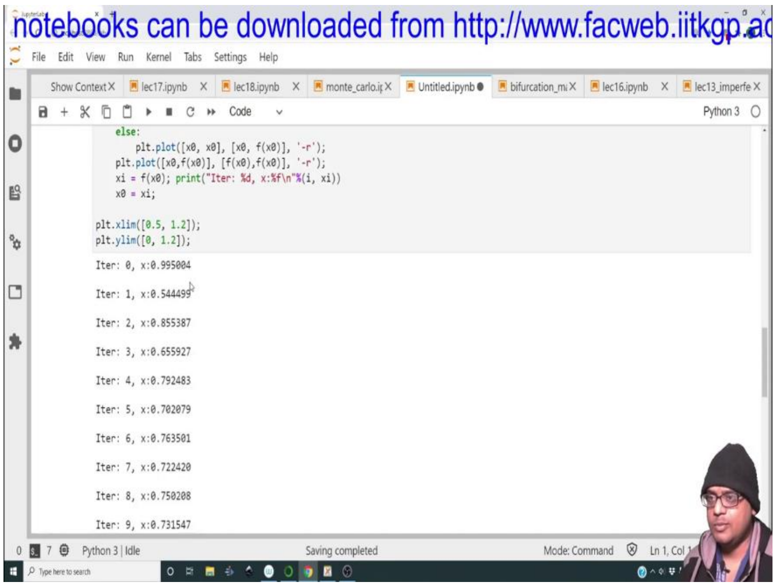
notebooks can be downloaded from http://www.facweb.iitkgp.ac

```
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([0.5, 1.2]);
plt.ylim([0, 1.2]);
```

```
Iter: 0, x:0.995004

Iter: 1, x:0.544499

Iter: 2, x:0.855387

Iter: 3, x:0.655927

Iter: 4, x:0.792483

Iter: 5, x:0.702079

Iter: 6, x:0.763501

Iter: 7, x:0.722420

Iter: 8, x:0.750208

Iter: 9, x:0.731547
```

So, let me reduce the scale to 0.5 to 1.2 so that we have a better look at what is really transpiring and such kinds of plots are called as cobwebs and we have already discussed about this.

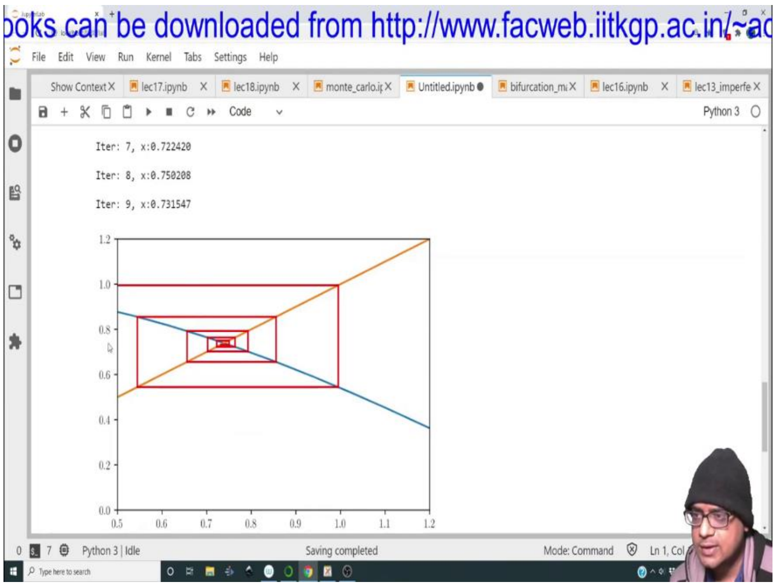
(Refer Slide Time: 21:07)



```
        else:
            plt.plot([x0, x0], [x0, f(x0)], '-r');
        plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
        xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
        x0 = xi;

    plt.xlim([0.5, 1.2]);
    plt.ylim([0, 1.2]);
```

```
Iter: 0, x:0.995004

Iter: 1, x:0.544499

Iter: 2, x:0.855387

Iter: 3, x:0.655927

Iter: 4, x:0.792483

Iter: 5, x:0.702079

Iter: 6, x:0.763501

Iter: 7, x:0.722420

Iter: 8, x:0.750208

Iter: 9, x:0.731547
```

(Refer Slide Time: 21:08)

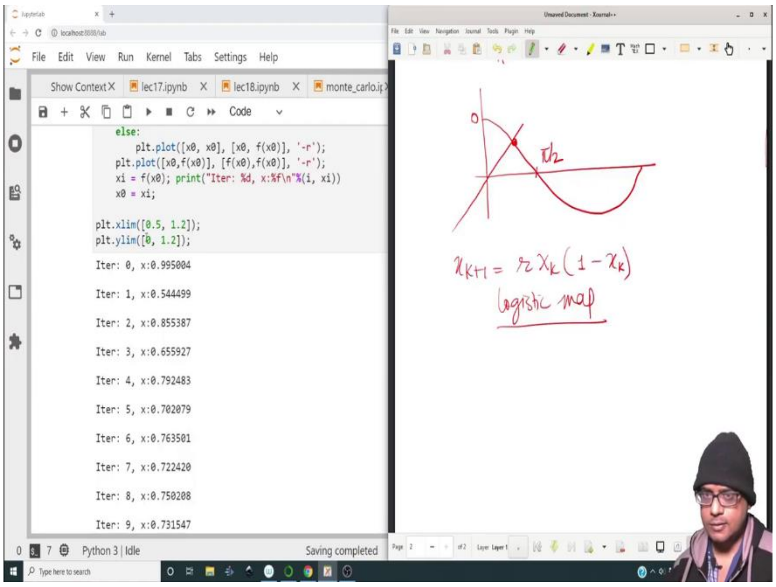


```
Iter: 7, x:0.722420

Iter: 8, x:0.750208

Iter: 9, x:0.731547
```

So, 0 to 1.2 because they resemble the cobwebs of a spider web; so, those are called as cobwebs alright. So, it starts somewhere over here, then it goes on to this, then it maps onto this and so on. So, each iterate it appears to converge towards this. So, at this point, the slope of cosine you will find that the magnitude is less than 1.

So, this is how roots can converge depending on the local or the slope at the fixed point and graphically, it is rather easy to figure out what the fixed point should be, but numerically for the fixed point so, for the points to converge to the fixed point, you have to ensure that the root is bounded to 1 alright.

(Refer Slide Time: 21:56)

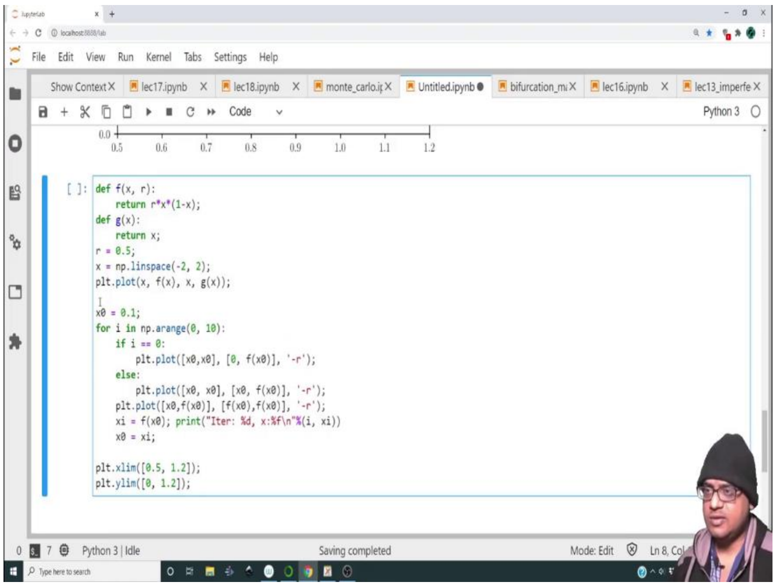


```
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([0.5, 1.2]);
plt.ylim([0, 1.2]);
```

```
Iter: 0, x:0.995004

Iter: 1, x:0.544499

Iter: 2, x:0.855387

Iter: 3, x:0.655927

Iter: 4, x:0.792483

Iter: 5, x:0.702079

Iter: 6, x:0.763501

Iter: 7, x:0.722420

Iter: 8, x:0.750208

Iter: 9, x:0.731547
```

So, now let us proceed to a very famous map that is $x_{k+1} = rx_k(1 - x_k)$. So, this particular map is also called as the logistic map. So, now, let us encode this and see what happens.

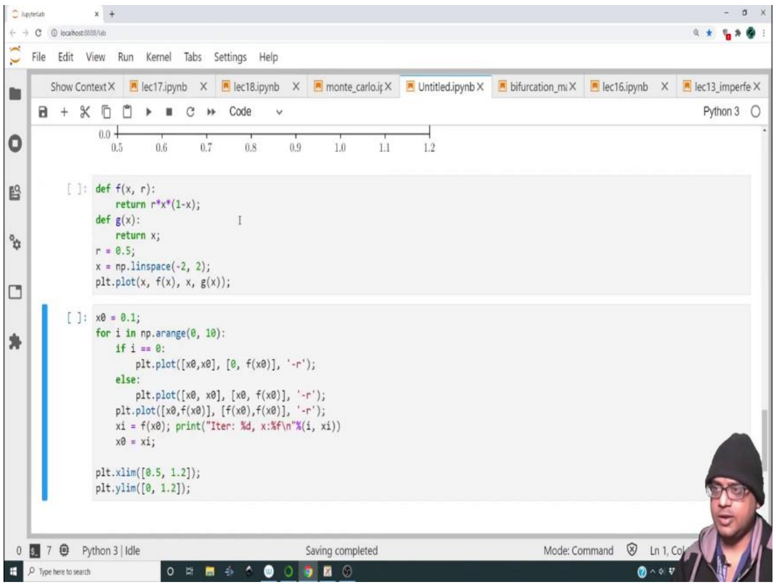
(Refer Slide Time: 22:26)



```
def f(x, r):
    return r*x*(1-x);
def g(x):
    return x;
r = 0.5;
x = np.linspace(-2, 2);
plt.plot(x, f(x), x, g(x));

x0 = 0.1;
for i in np.arange(0, 10):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([0.5, 1.2]);
plt.ylim([0, 1.2]);
```

So, first of all, I mean based on the value of r even before solving this so, let me just encode it. So, let me split the cell over here to show you how the plot looks like.

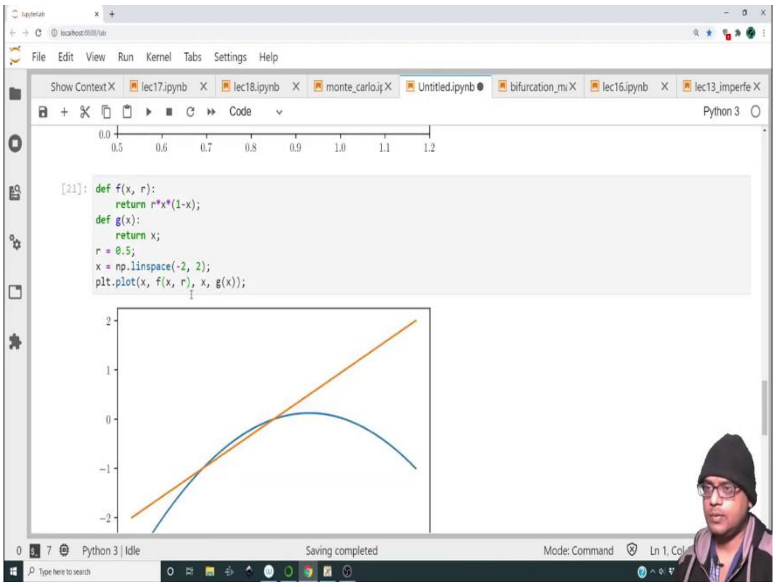
(Refer Slide Time: 22:47)



```
[ ]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 0.5;
     x = np.linspace(-2, 2);
     plt.plot(x, f(x), x, g(x));
```

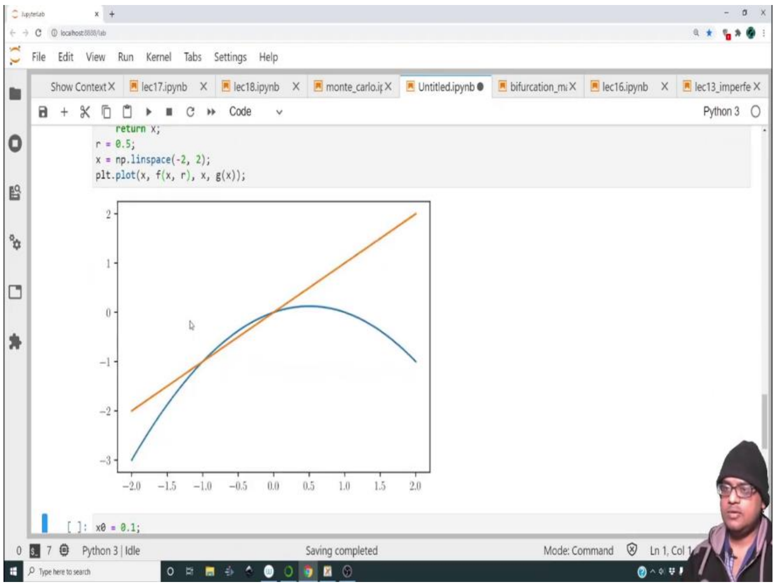
```
[ ]: x0 = 0.1;
     for i in np.arange(0, 10):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0)], '-r');
         plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
         xi = f(x0); print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

     plt.xlim([0.5, 1.2]);
     plt.ylim([0, 1.2]);
```

(Refer Slide Time: 22:54)



```
[21]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 0.5;
      x = np.linspace(-2, 2);
      plt.plot(x, f(x, r), x, g(x));
```

(Refer Slide Time: 22:55)

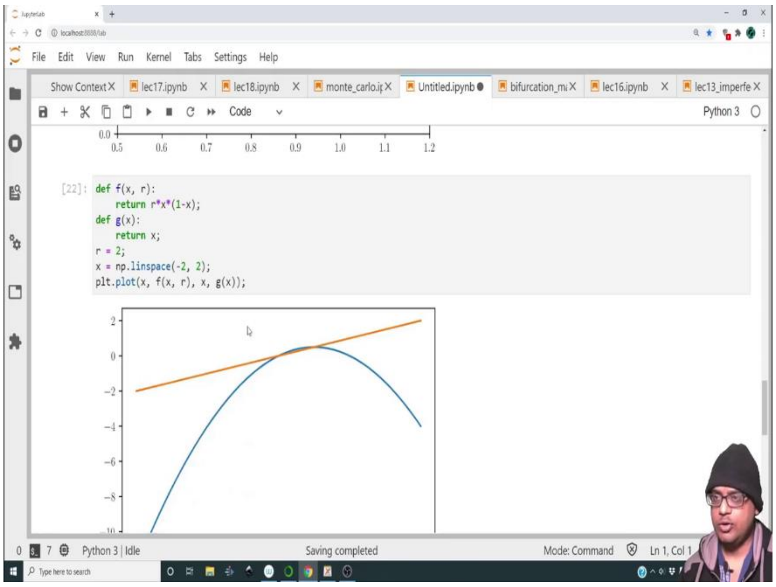


```
    return x;
r = 0.5;
x = np.linspace(-2, 2);
plt.plot(x, f(x, r), x, g(x));
```

```
[ ]: x0 = 0.1;
```

So, this is how the plot looks like and there is obviously, a fixed point over here and a fixed point at 0.

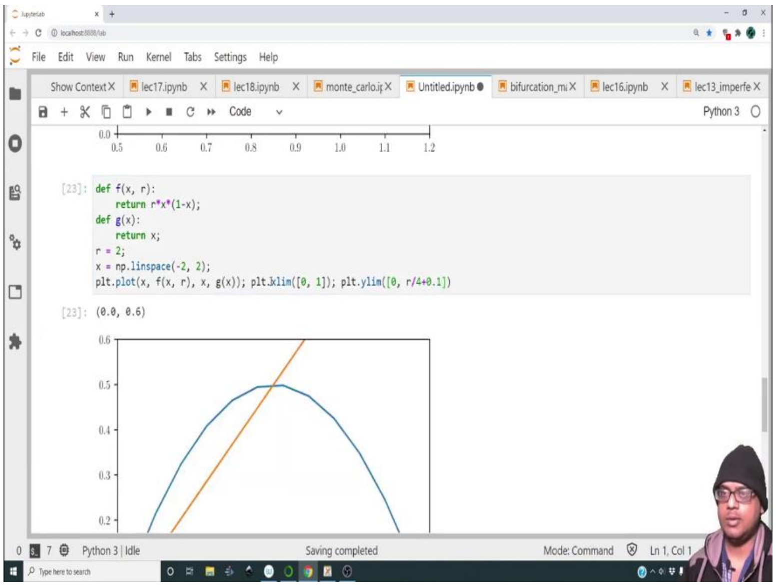
(Refer Slide Time: 23:05)



```
[22]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2;
      x = np.linspace(-2, 2);
      plt.plot(x, f(x, r), x, g(x));
```

If we increase the value, then let me make it 2. So, there is a fixed point over here.

(Refer Slide Time: 23:12)

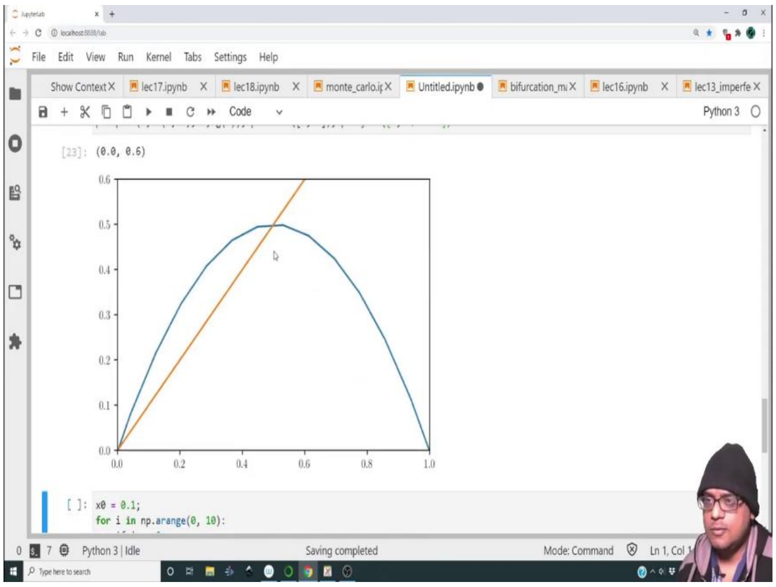


```
[23]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2;
      x = np.linspace(-2, 2);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
```

```
[23]: (0.0, 0.6)
```

So, let us focus on 0 to 1 and so, what is the maximum going to be? You said x = half so, it is going to be r/ 4. So, let us choose this to r/4 + 0.1 alright.

(Refer Slide Time: 23:38)

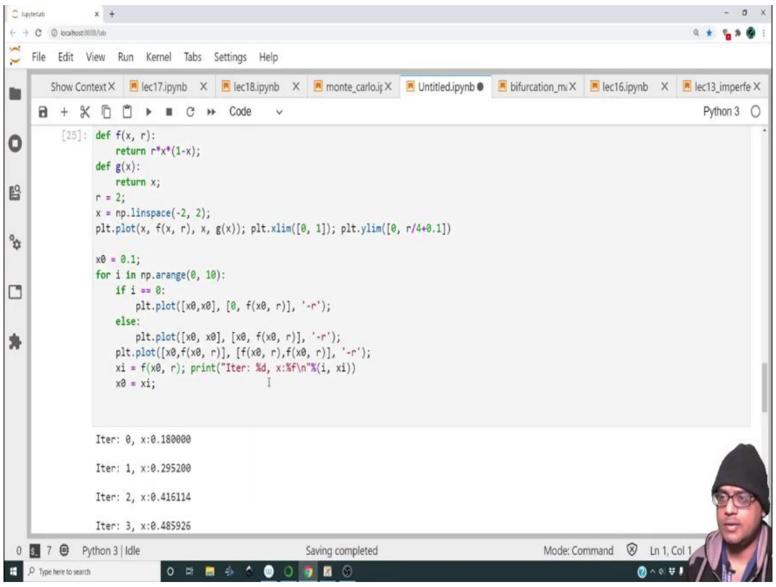


```
[23]: (0.0, 0.6)
```

```
[ ]: x0 = 0.1;
     for i in np.arange(0, 10):
```

So, this is how, this is the fixed point and now, the slope of the fixed point appears to be somewhat close to 0. So, we expect this fixed point to be a stable fixed point, it is going to be an attracting fixed point. So, let us see.

(Refer Slide Time: 23:54)

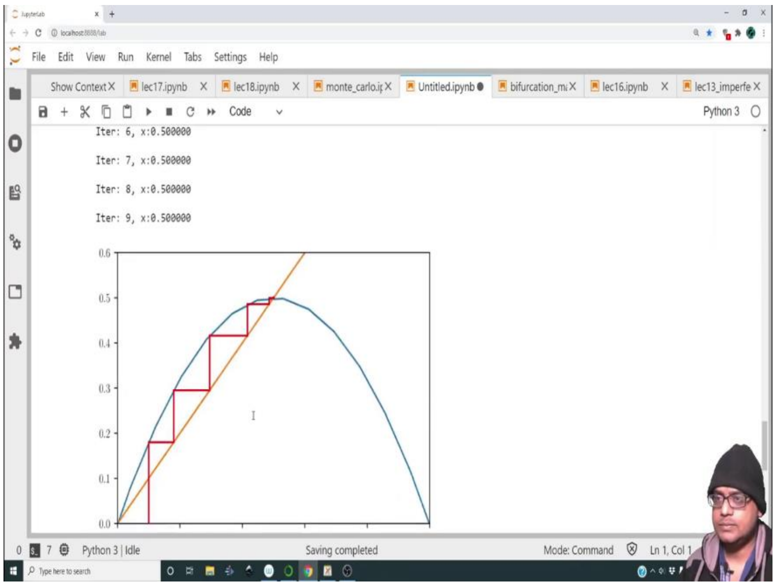


```
def f(x, r):
    return r*x*(1-x);
def g(x):
    return x;
r = 2;
x = np.linspace(-2, 2);
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

x0 = 0.1;
for i in np.arange(0, 10):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

```
Iter: 0, x:0.180000

Iter: 1, x:0.295200

Iter: 2, x:0.416114

Iter: 3, x:0.485926
```

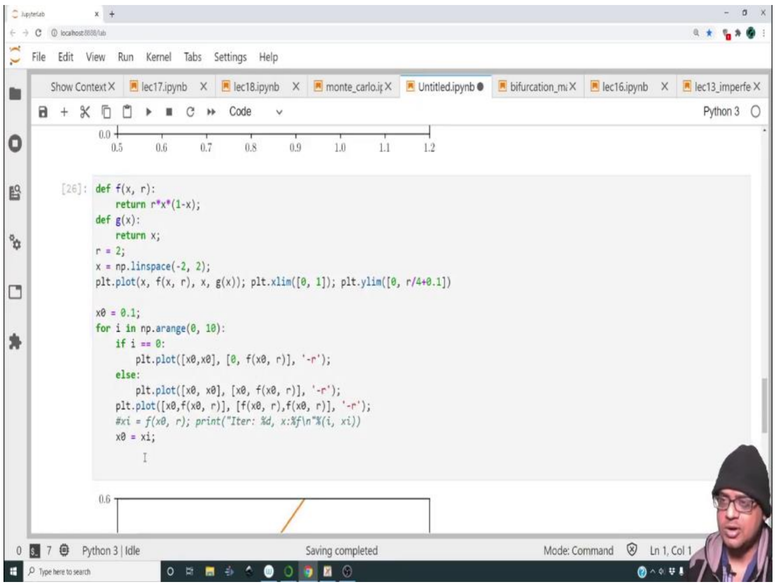
So, let me now merge this bit of code along with this and we will see whether or not that fixed point is indeed attracting or not. Let me remove this, we have to pass r as the additional argument because in this problem, r appears as a parameter to the map and we will see that depending on the value of r, we will have varying behaviors.

(Refer Slide Time: 24:27)



```
Iter: 6, x:0.500000

Iter: 7, x:0.500000

Iter: 8, x:0.500000

Iter: 9, x:0.500000
```

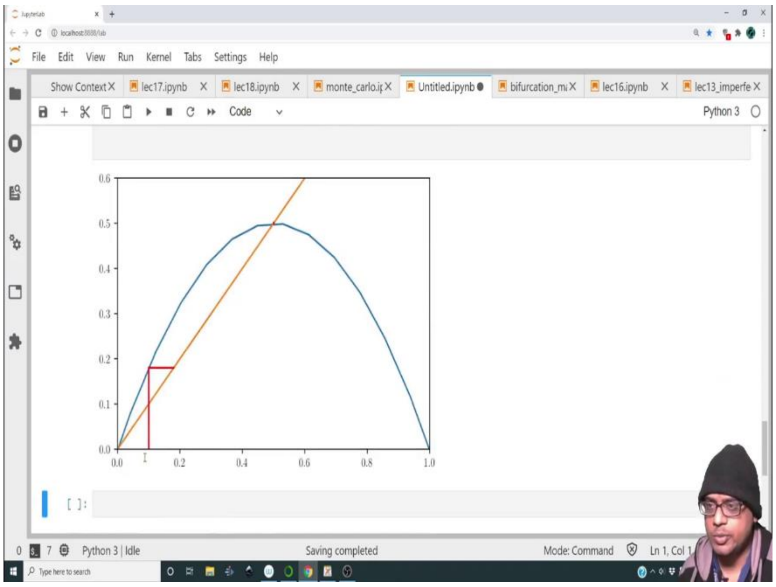
(Refer Slide Time: 24:40)



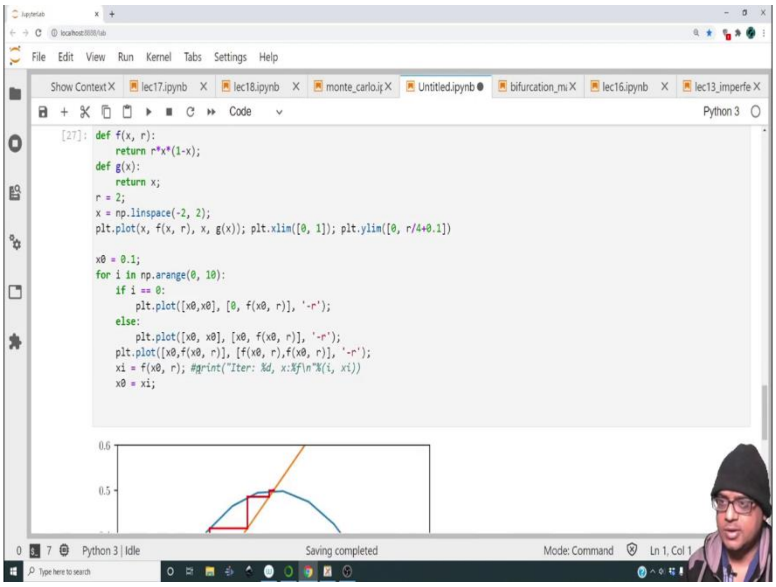
```
[26]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2;
      x = np.linspace(-2, 2);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

      x0 = 0.1;
      for i in np.arange(0, 10):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          #xi = f(x0, r); print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

In fact, let me turn off the printing of the values. If at all you want to play around with this, you can turn on printing the values and then, it will be clearer to you how that iteration actually takes place.

(Refer Slide Time: 24:44)



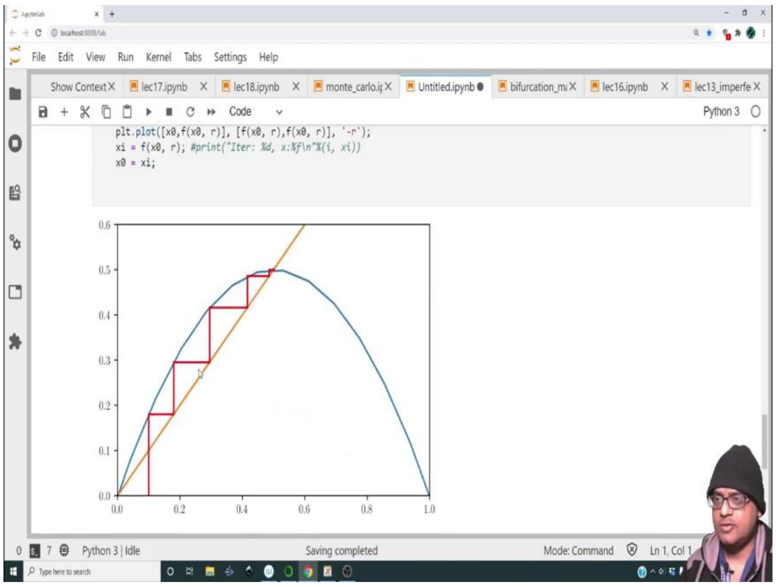
(Refer Slide Time: 24:48)



```
[27]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2;
      x = np.linspace(-2, 2);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

      x0 = 0.1;
      for i in np.arange(0, 10):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

But for me, I find it more illuminating, I find it more illuminating if the values are indeed shown through a figure.

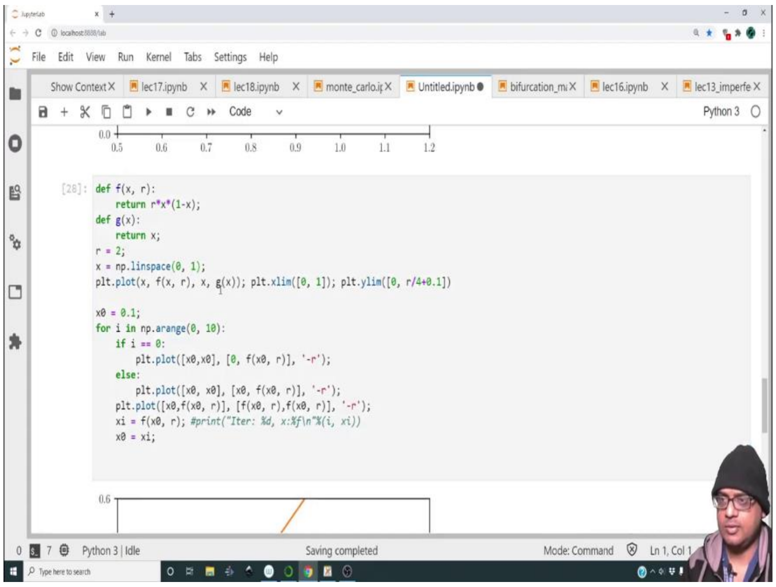
(Refer Slide Time: 24:51)



```
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, this is the guess, and it gets mapped onto this, then it gets mapped onto this, it gets mapped onto this and so on until it reaches the root.

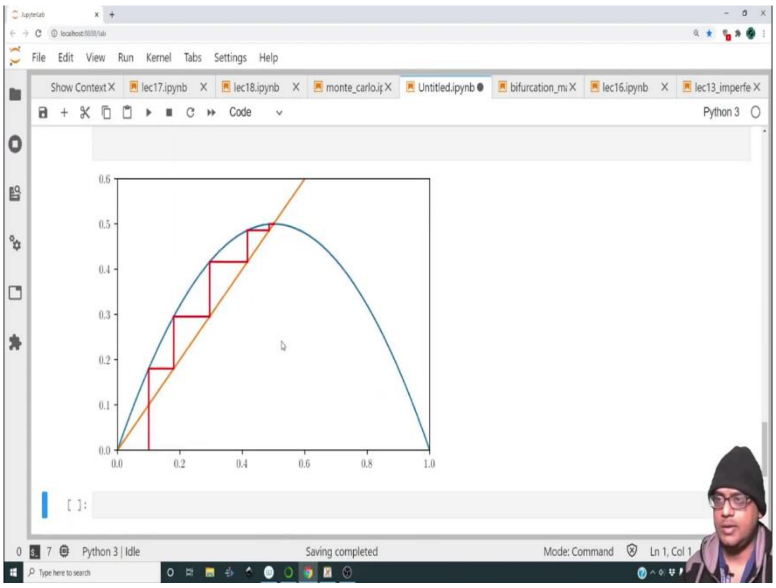
(Refer Slide Time: 25:09)



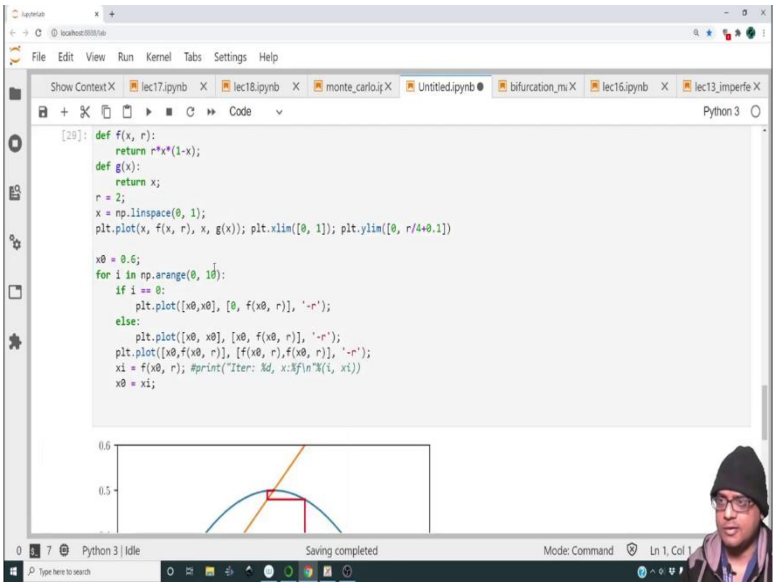
```
[28]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2;
      x = np.linspace(0, 1);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

      x0 = 0.1;
      for i in np.arange(0, 10):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

Let us increase the resolution of the curves simply by changing the range alright great.

(Refer Slide Time: 25:13)



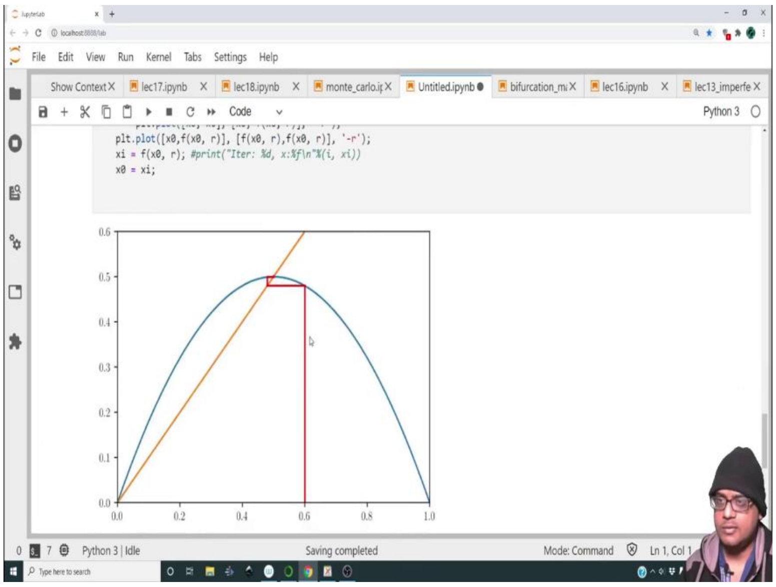
(Refer Slide Time: 25:25)



```
[29]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2;
      x = np.linspace(0, 1);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

      x0 = 0.6;
      for i in np.arange(0, 10):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

So, and similarly, if we choose a value somewhere over at 0.6 suppose, we should be still getting that kind of a convergence.

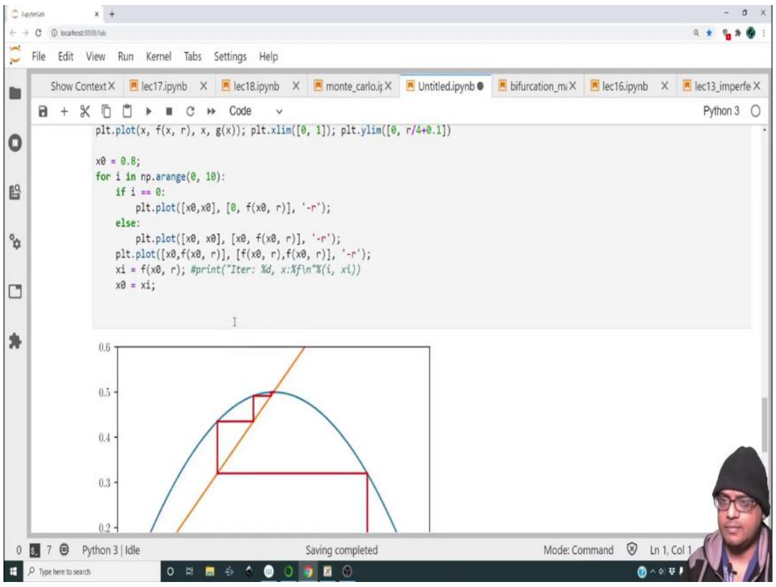
(Refer Slide Time: 25:26)



```
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

So, there you see it converges.

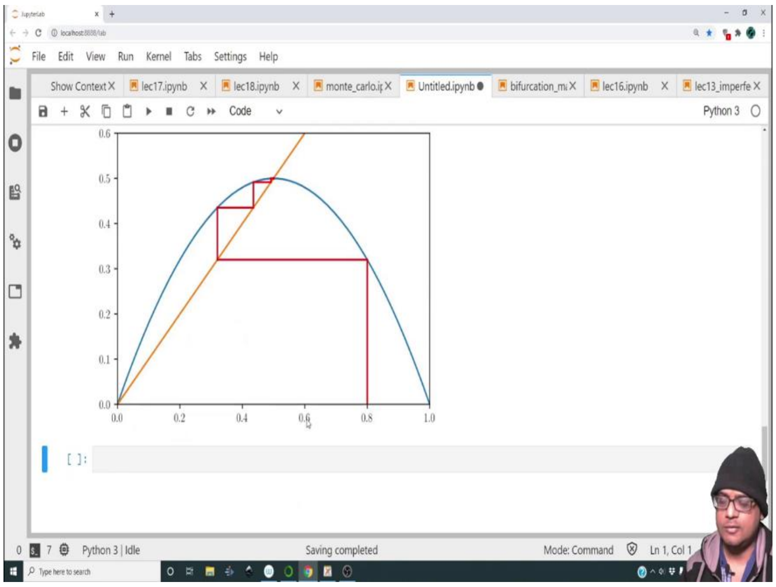
(Refer Slide Time: 25:33)



```
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

x0 = 0.8;
for i in np.arange(0, 10):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

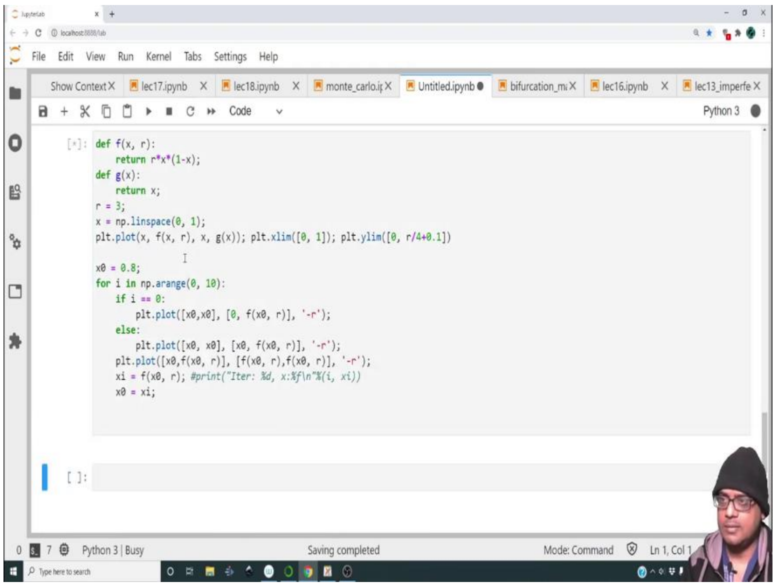
Let me make it 0.8 and again it converges.

(Refer Slide Time: 25:35)



So, regardless of where the initial guess is, you will see that everything converges to the fixed point.

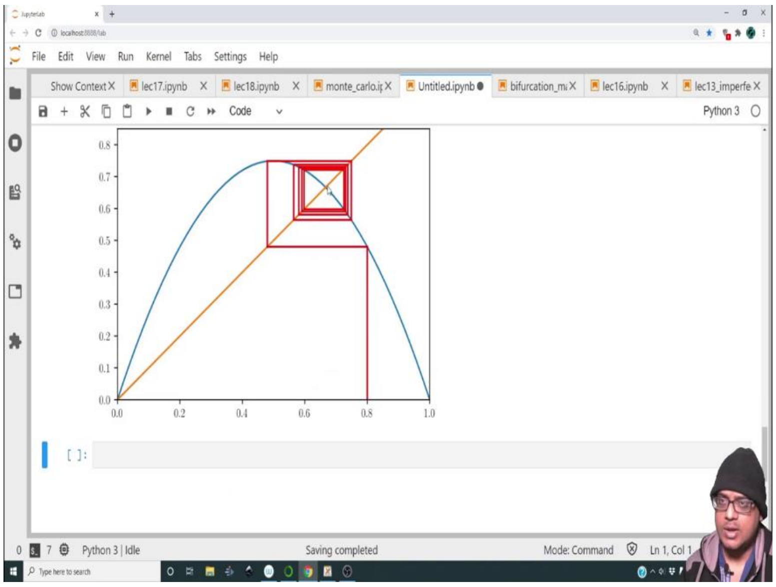
(Refer Slide Time: 25:47)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 3;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

     x0 = 0.8;
     for i in np.arange(0, 10):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;
```

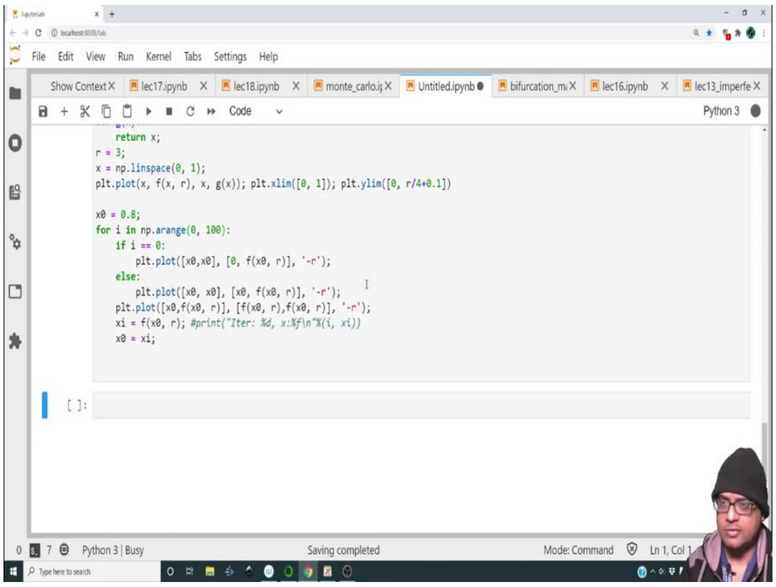
Let me increase the value of r, let me make it 3. Now, what happens look at this?

(Refer Slide Time: 25:48)



So, when r = 3, something quite I mean it seems to converge, but it may take a huge amount of iterations to converge ok, it still not converged.

(Refer Slide Time: 26:06)

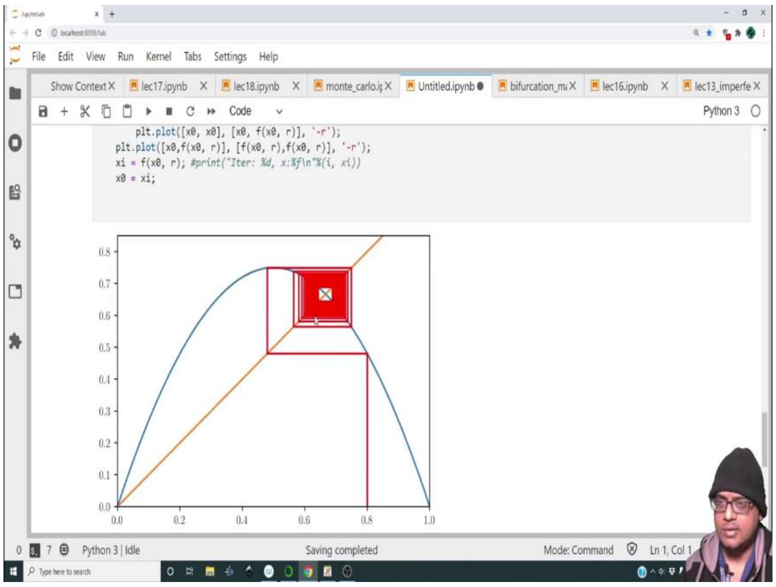


```
    return x;
r = 3;
x = np.linspace(0, 1);
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])

x0 = 0.8;
for i in np.arange(0, 100):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

```
[ ]:
```

Let me take 100 iterations.

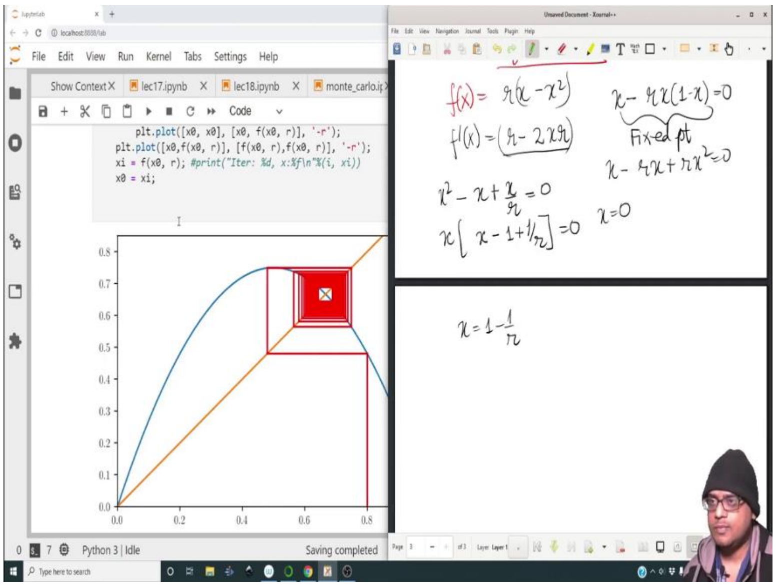
(Refer Slide Time: 26:08)



```
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

It is still not converged. So, why is this happening? Let us go back and look at the equation.

(Refer Slide Time: 26:15)



```
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

$f(x) = r(x - x^2)$

$f'(x) = \underline{(r - 2xr)}$

$\underbrace{x - rx(1-x)}_{\text{Fixed pt}} = 0$

$x - rx + rx^2 = 0$

$x^2 - x + \frac{x}{r} = 0$

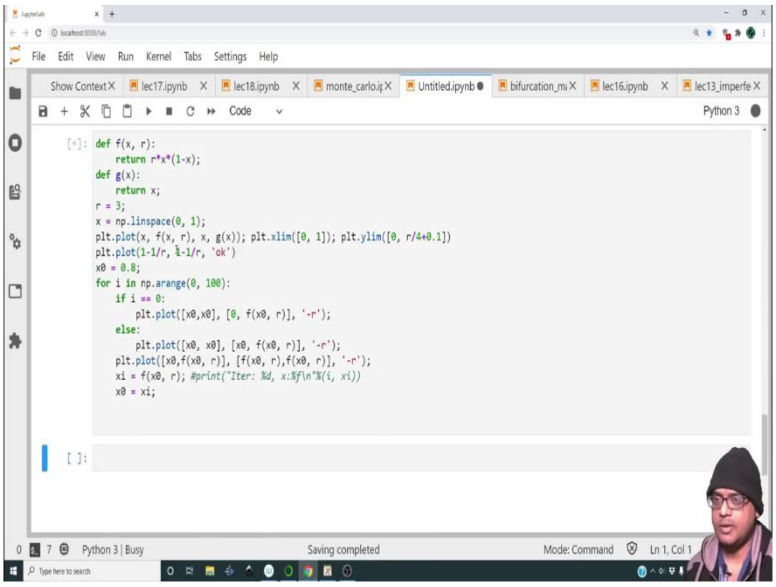
$x\left[x - 1 + 1/r\right] = 0$ $x = 0$

$x = 1 - \frac{1}{r}$

So, the $f(x) = r(x – x^2)$. So, what is $f`(x)$? It is going to be $r - 2xr$ ok. So, when. So, $x - rx(1– x) = 0$ so, the solution of this would give you the fixed point and this is the slope at the fixed point.

So, we have to substitute the slope over here. So, let us expand this. So, it is $x - rx + rx^2 = 0$ so, we can write it as a $x^2 - x + x/r = 0$ so, this is what x common. So, this is $x - 1 + 1/r = 0$. So, $x = 0$ is a root right and $x = 1 - 1/r$ is a root.

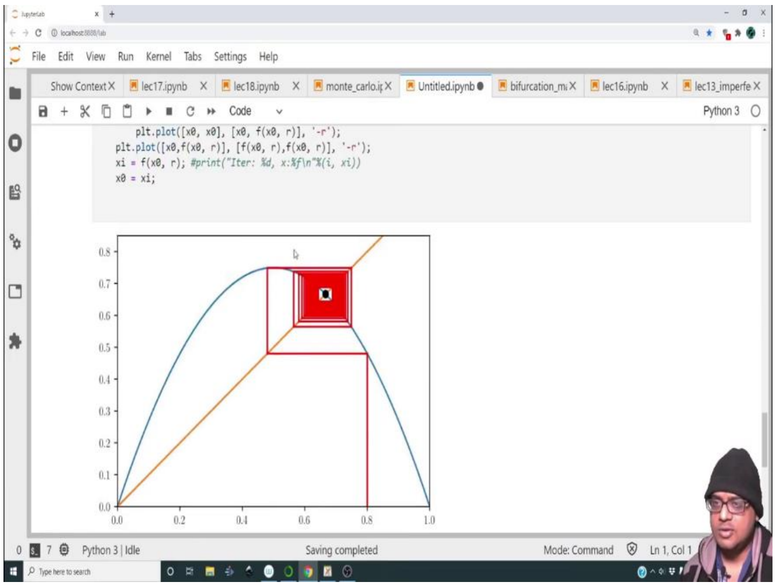
(Refer Slide Time: 27:45)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 3;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')
     x0 = 0.8;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

[ ]:
```

So, let us superpose that root on top of this. So, plt.plot(1 - 1/r, 1 – 1/r) let us mark it with a black marker.

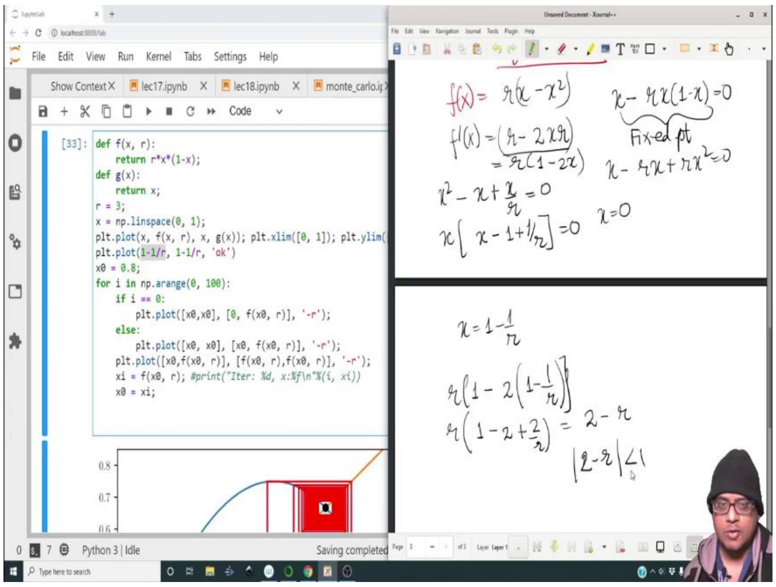
(Refer Slide Time: 28:07)



```
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, this point is the root.

(Refer Slide Time: 28:18)



```
[33]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3;
      x = np.linspace(0, 1);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim(
      plt.plot(1-1/r, 1-1/r, 'ok')
      x0 = 0.8;
      for i in np.arange(0, 100):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

$f(x) = r(x - x^2)$

$f'(x) = (r - 2xr)$
$= r(1-2x)$

$x - rx(1-x) = 0$
Fixed pt

$x - rx + rx^2 = 0$

$x^2 - x + \frac{x}{r} = 0$

$x\left[x - 1 + 1/r\right] = 0 \quad x = 0$

$x = 1 - \frac{1}{r}$

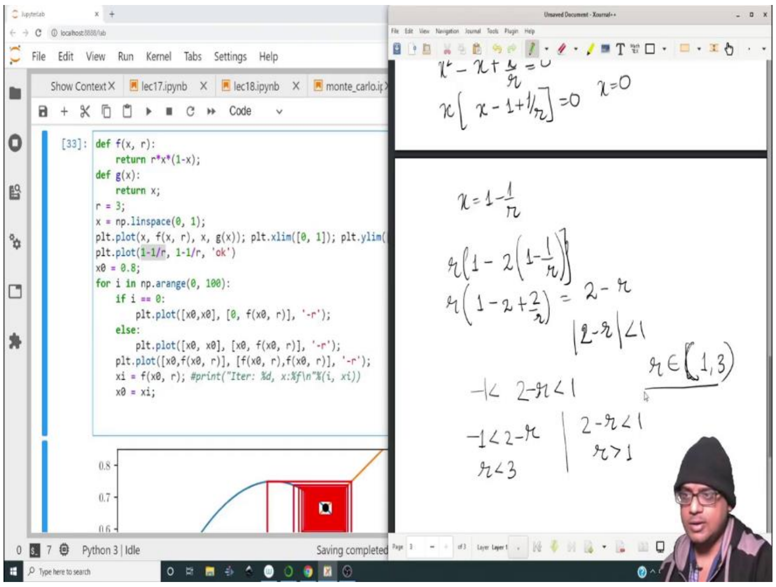
$r\left[1 - 2\left(1 - \frac{1}{r}\right)\right]$

$r\left(1 - 2 + \frac{2}{r}\right) = 2 - r$

$|2 - r| < 1$

Now, we are interested in finding out what the value of the slope is at the root. So, the value of the slope will be r*(1- 2*(1-1/r)), this is the root so, I have just taken r common, this is 1 - 2x. So, what is this? This is going to be r*(1 - 2 + 2/r). So, this is going to be 2 - r alright. So, when r so, 2 - r mod of this less than 1.

(Refer Slide Time: 29:02)



```
[33]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3;
      x = np.linspace(0, 1);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim(
      plt.plot(1-1/r, 1-1/r, 'ok')
      x0 = 0.8;
      for i in np.arange(0, 100):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

$x^2 - x + \frac{x}{r} = 0$

$x\left[x - 1 + 1/r\right] = 0 \quad x = 0$

$x = 1 - \frac{1}{r}$

$r\left[1 - 2\left(1 - \frac{1}{r}\right)\right]$

$r\left(1 - 2 + \frac{2}{r}\right) = 2 - r$

$|2 - r| < 1$

$r \in (1, 3)$

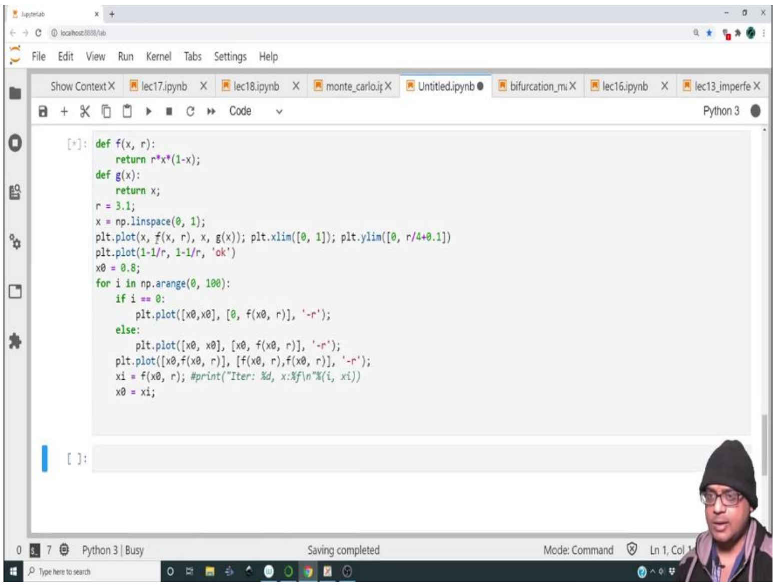
$-1 < 2 - r < 1$

$-1 < 2 - r \quad | \quad 2 - r < 1$

$r < 3 \quad | \quad r > 1$

So, if 2 - r is less than 1, then it means 2 - r should be less than 1 and it should be greater than -1. So, the first inequality says that 1 is this and this implies r should be less than 3 and this the second inequality implies that r should be greater than 1. So, if r lies in the open set 1 to 3, then we have a stable fixed point.

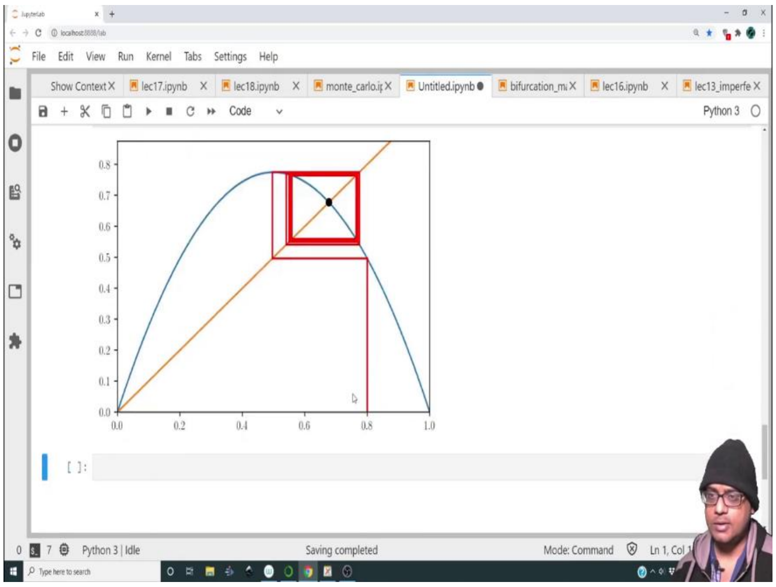
(Refer Slide Time: 29:45)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 3.1;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')
     x0 = 0.8;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

[ ]:
```

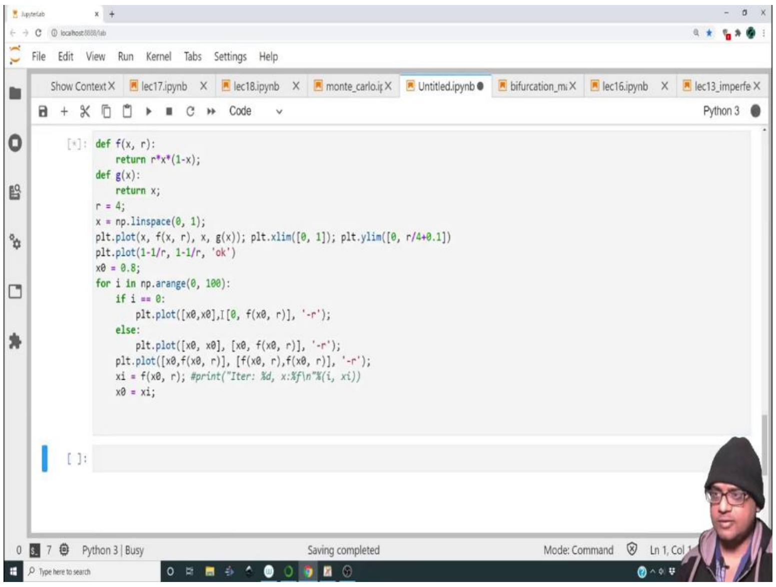
So, let me make it 3 point 1, let us see what happens.

(Refer Slide Time: 29:47)



Well, it just keeps around right, it keeps on going and most likely it will diverge after so many iterations, I have not kept the number of iterations very high.

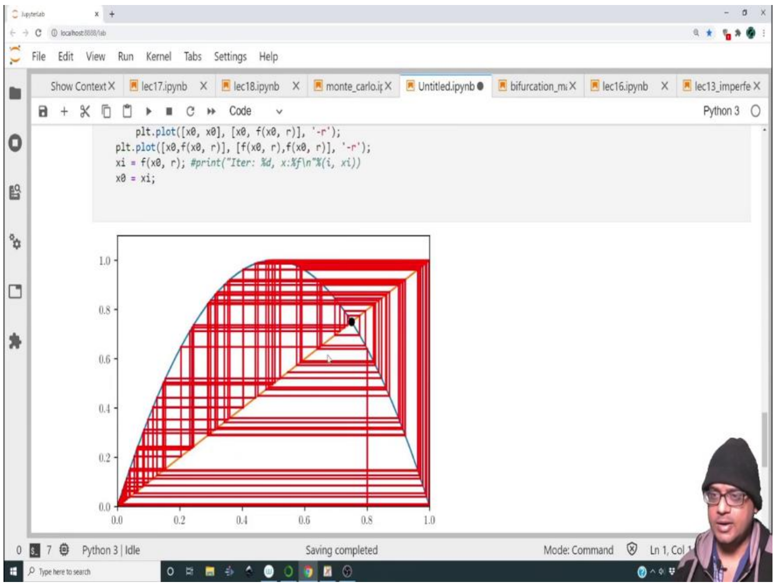
(Refer Slide Time: 30:03)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 4;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')
     x0 = 0.8;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0],I[0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;
```

But let me increase this, let me make it 4, let us it will be clear.

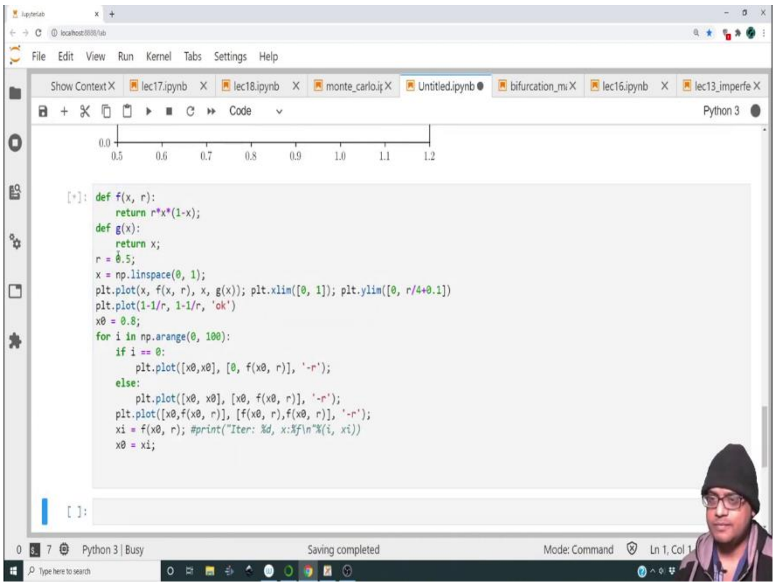
(Refer Slide Time: 30:03)



```
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, at 4, it is going all over the place and really, I mean there is something which is quite bizarre to see, you do not expect this kind of behavior from well, you do not expect in the sense that for the uninitiated, you do not you will feel that things will be very well behaved, but it seems to go all over the place, it is not converging to the fixed point alright.

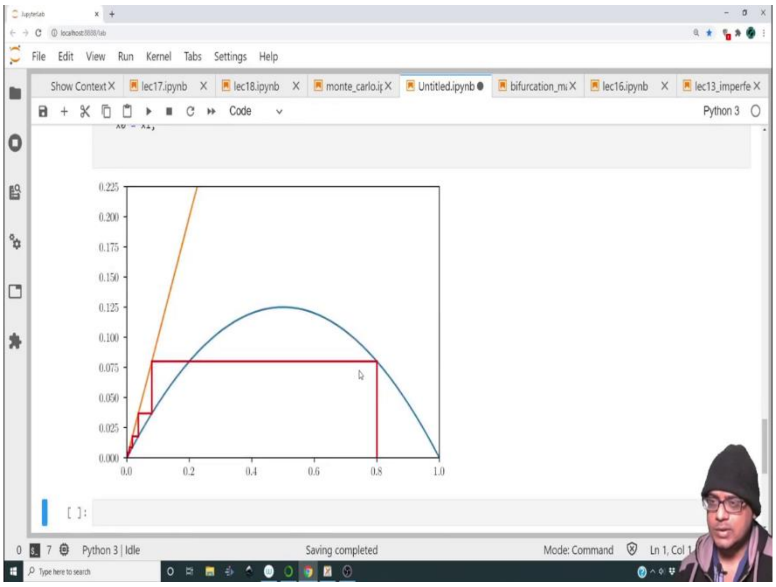
(Refer Slide Time: 30:31)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 0.5;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')
     x0 = 0.8;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;
```

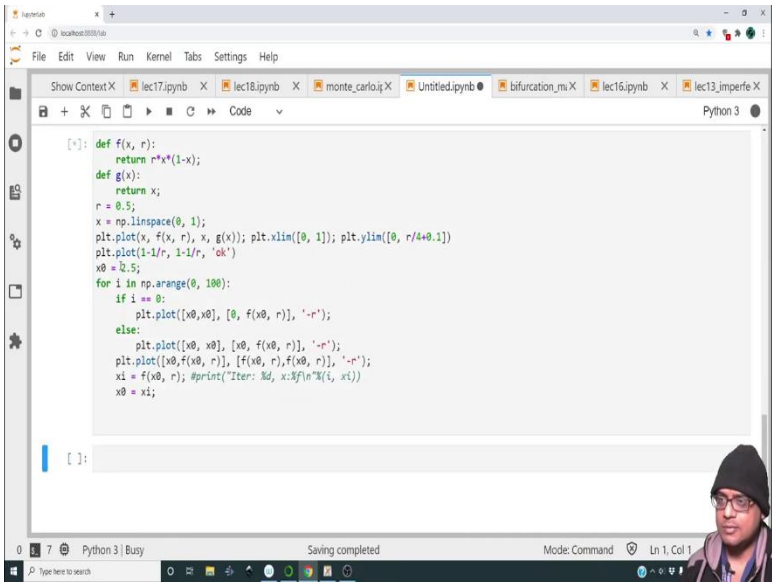
What happens at r = 0.5?

(Refer Slide Time: 30:33)



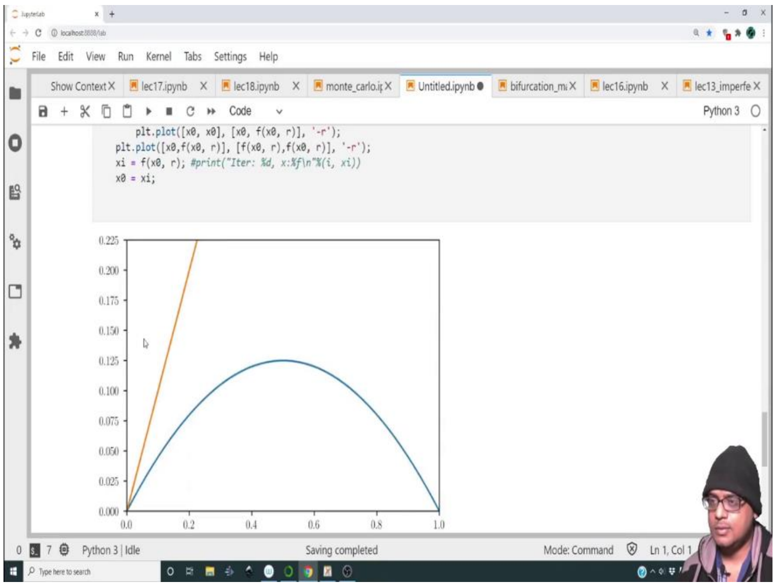
And the same thing, I mean it goes and it gets attracted towards the other root that is x = 0 ok, there is no intersection point.

(Refer Slide Time: 30:47)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 0.5;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')
     x0 = 2.5;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

[ ]:
```

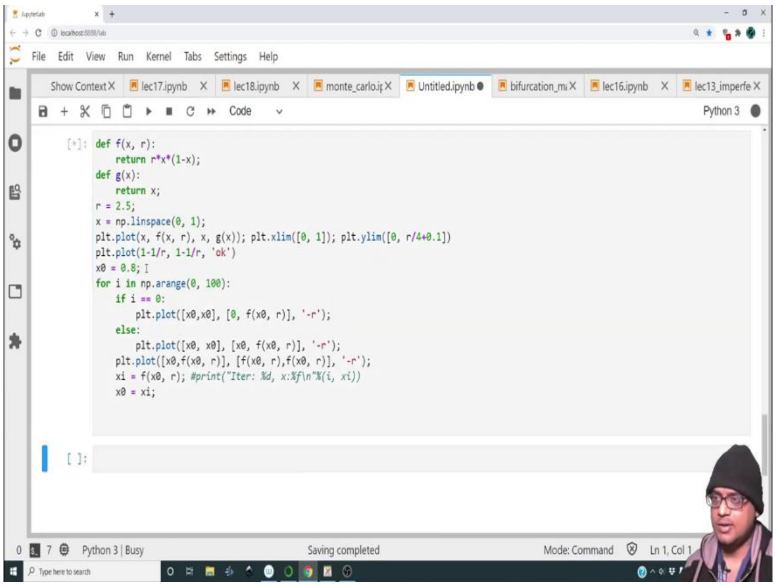
(Refer Slide Time: 30:49)



```
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, when things are between.

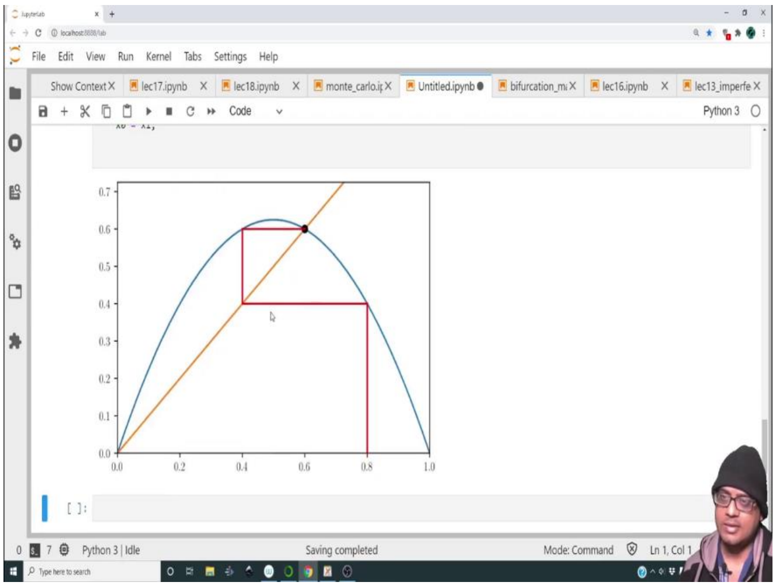
(Refer Slide Time: 30:54)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 2.5;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')
     x0 = 0.8;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;

[ ]:
```

So, when things are when the r is between 1 and 3, we expect the fixed point to be stable like this.

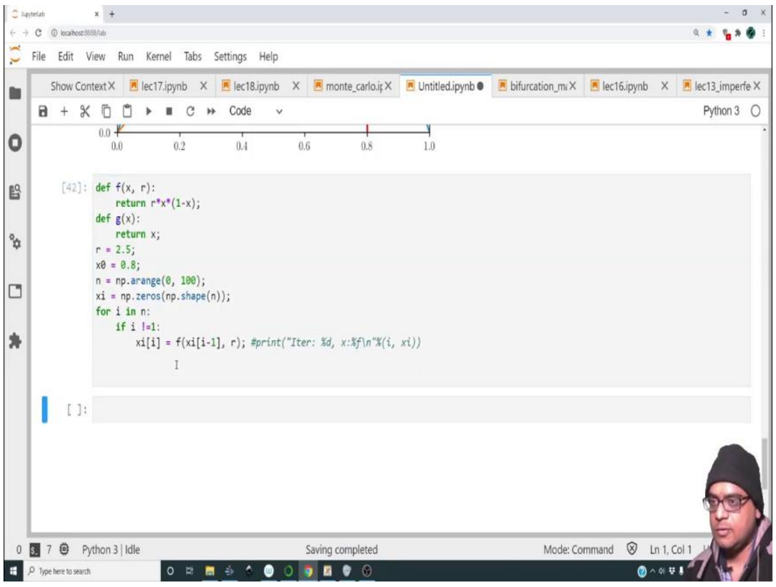
(Refer Slide Time: 31:02)



But when r exceeds 3 that is when weird things begin to happen and let us now take a peek into why this kind of behavior actually occurs so that we can once we look at the time series or not the time series, but the iterate series that per iteration how does the root vary, we will be in a position to explain why there is a certain periodicity in the way the roots or the iterates appear once that is clear.

We can then move on to the bifurcation diagram where ultimately we will see how that period doubling that we have seen in the case of the van der Pol oscillator that kind of a period doubling also occurs in such kinds of iterated maps. So, let us see.

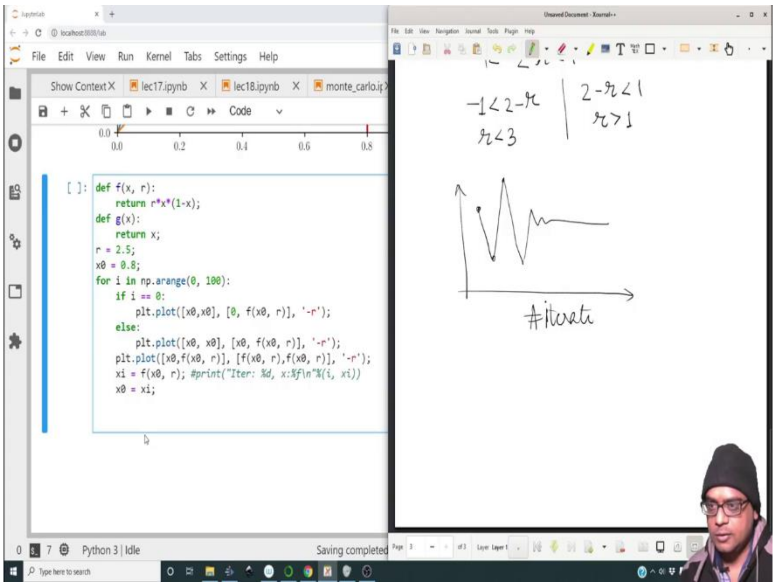
(Refer Slide Time: 32:11)



```
[42]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2.5;
      x0 = 0.8;
      n = np.arange(0, 100);
      xi = np.zeros(np.shape(n));
      for i in n:
          if i !=1:
              xi[i] = f(xi[i-1], r); #print("Iter: %d, x:%f\n"%(i, xi))
```

Let us now plot the iterates how they look like. So, let me grab hold out this snippet and instead of plotting the cobwebs, we are now interested in plotting the.

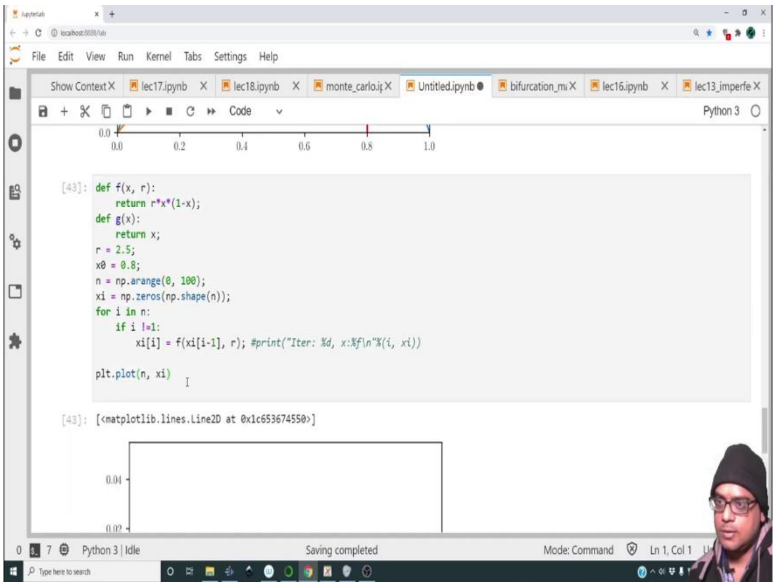
(Refer Slide Time: 32:42)



```
[ ]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 2.5;
     x0 = 0.8;
     for i in np.arange(0, 100):
         if i == 0:
             plt.plot([x0,x0], [0, f(x0, r)], '-r');
         else:
             plt.plot([x0, x0], [x0, f(x0, r)], '-r');
         plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
         xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
         x0 = xi;
```

So, we are now interested in plotting the time series right. So, for that we need to plot the following thing let me show you. So, on the x axis, we will have the iteration number and on the y axis, we will have the actual value of the x, the present x so, it will look something like this or if it converges, it will look something like this. So, let us have a look; let us have a look.

(Refer Slide Time: 35:07)

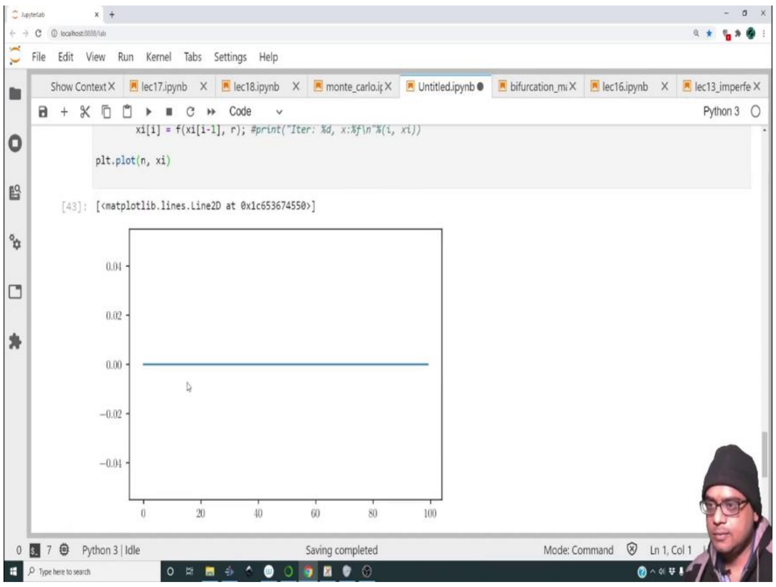


```
[43]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2.5;
      x0 = 0.8;
      n = np.arange(0, 100);
      xi = np.zeros(np.shape(n));
      for i in n:
          if i !=1:
              xi[i] = f(xi[i-1], r); #print("Iter: %d, x:%f\n"%(i, xi))

      plt.plot(n, xi)

[43]: [<matplotlib.lines.Line2D at 0x1c653674550>]
```

So, for that we need to create. So, let us create first the n array. So, let this be this. So, for i in n right and we need to store the value of iterate so, this will be xi = np. let me initialize it by zeros so, np.shape of n. So, in case we change n, we already have a soft code method of defining what xi will be right.

So, once that happens, we do not need all this plotting anymore so, we can get rid of these lines, we can also get rid of this alright and so, here, I can write xi[i] will be xi[i-1] that is it, that is pretty much the code and we have to not do this for i = 1, for i != 1 we have to do this alright. So, let me run this. So, we have forgot this, if i != 1 alright. So, it runs and so, at the end, let us plot. So, we will do plt.plot and this will be a plot of n versus xi alright. So, well.

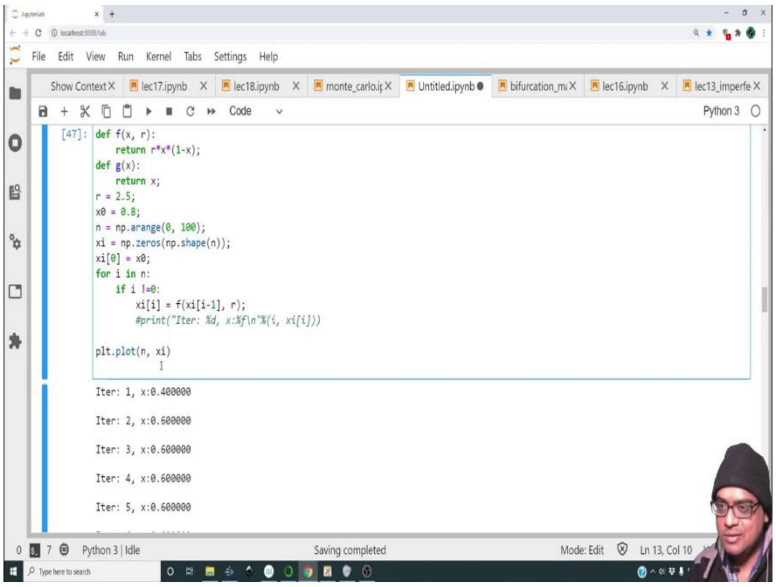
(Refer Slide Time: 35:16)



```
        xi[i] = f(xi[i-1], r); #print("Iter: %d, x:%f\n"%(i, xi))

plt.plot(n, xi)
```

```
[43]: [<matplotlib.lines.Line2D at 0x1c653674550>]
```

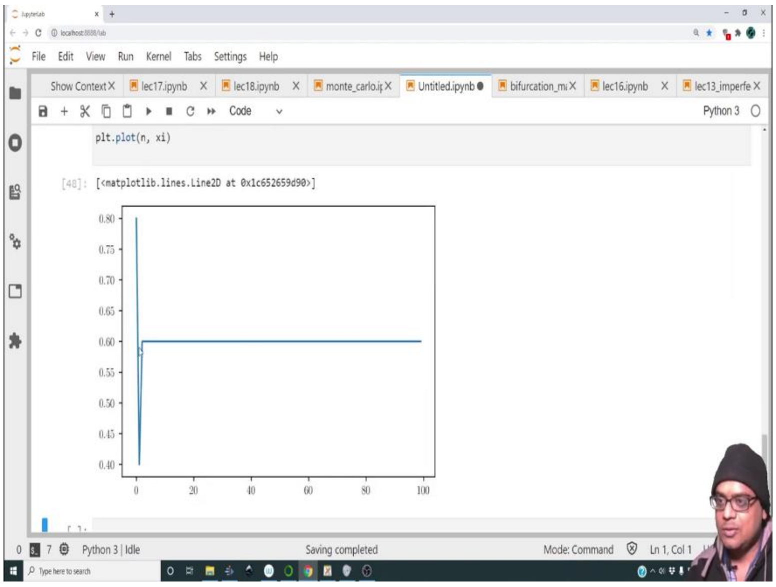
(Refer Slide Time: 35:35)


File Edit View Run Kernel Tabs Settings Help
Show Context lec17.ipynb lec18.ipynb monte_carlo.i Untitled.ipynb bifurcation_m lec16.ipynb lec13_imperfe
Code
Python 3
Python 3 | Idle
Saving completed
Mode: Edit
Ln 13, Col 10

```
[47]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 2.5;
      x0 = 0.8;
      n = np.arange(0, 100);
      xi = np.zeros(np.shape(n));
      xi[0] = x0;
      for i in n:
          if i !=0:
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)

      Iter: 1, x:0.400000

      Iter: 2, x:0.600000

      Iter: 3, x:0.600000

      Iter: 4, x:0.600000

      Iter: 5, x:0.600000
```

This has to be 0 ok. What seems to be the issue? We have not given the initial value. So, xi[0] = x0 ok. So, we have not passed on the initial value and we are trying to iterate so obviously, x = 0 was the initial value and so, if I do not write this line, it means that x0 will be 0 because I have initialized everything to 0, it was simply iterating zeros and 0 is a fixed point ok; 0 is a fixed point. So, simply we are getting zeros ok. So, now that we have this in place, we can suppress this and let us see ok.

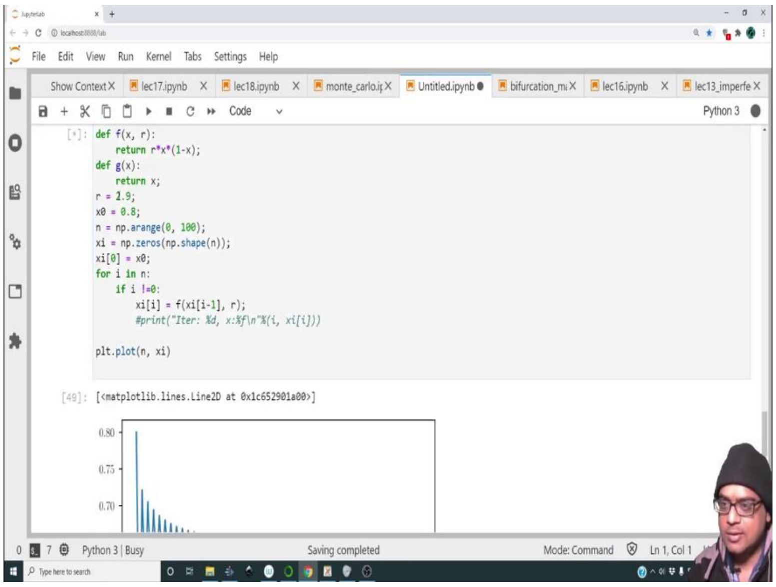
(Refer Slide Time: 36:06)


File Edit View Run Kernel Tabs Settings Help
Show Context lec17.ipynb lec18.ipynb monte_carlo.i Untitled.ipynb bifurcation_m lec16.ipynb lec13_imperfe
Code
Python 3
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
20
40
60
80
100
Python 3 | Idle
Saving completed
Mode: Command
Ln 1, Col 1

```
      plt.plot(n, xi)

[48]: [<matplotlib.lines.Line2D at 0x1c652659d90>]
```

So, after just a few one iteration maybe it settles down.

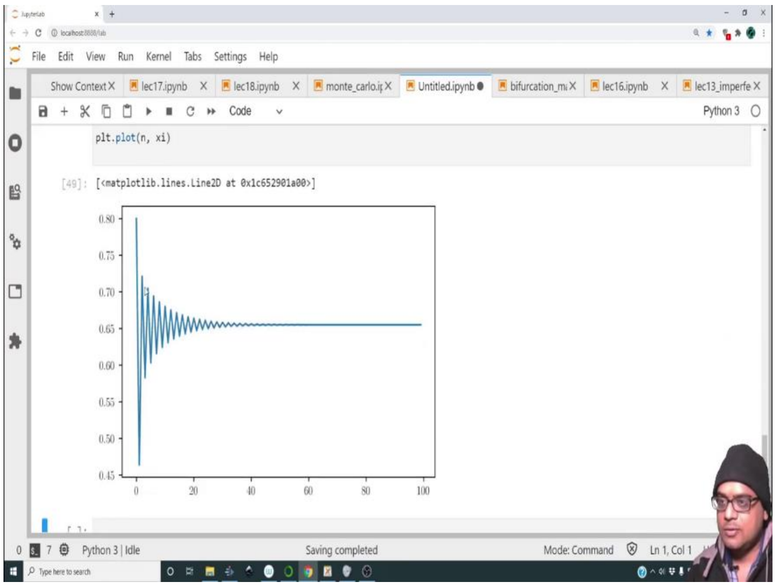
(Refer Slide Time: 36:16)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 2.9;
     x0 = 0.8;
     n = np.arange(0, 100);
     xi = np.zeros(np.shape(n));
     xi[0] = x0;
     for i in n:
         if i !=0:
             xi[i] = f(xi[i-1], r);
             #print("Iter: %d, x:%f\n"%(i, xi[i]))

     plt.plot(n, xi)

[49]: [<matplotlib.lines.Line2D at 0x1c652901a00>]
```

Let me change the value of r, let me make it 2.9 ok.

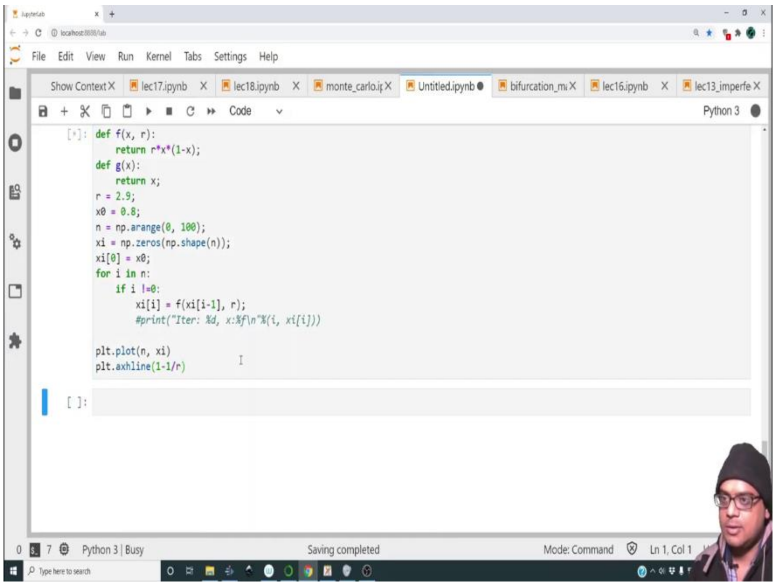
(Refer Slide Time: 36:19)



```
     plt.plot(n, xi)

[49]: [<matplotlib.lines.Line2D at 0x1c652901a00>]
```

So, it iterates, and it converges to 0.65, 0.65 should be equal to the value of 1 -1 /r. So, let us also plot that particular point.

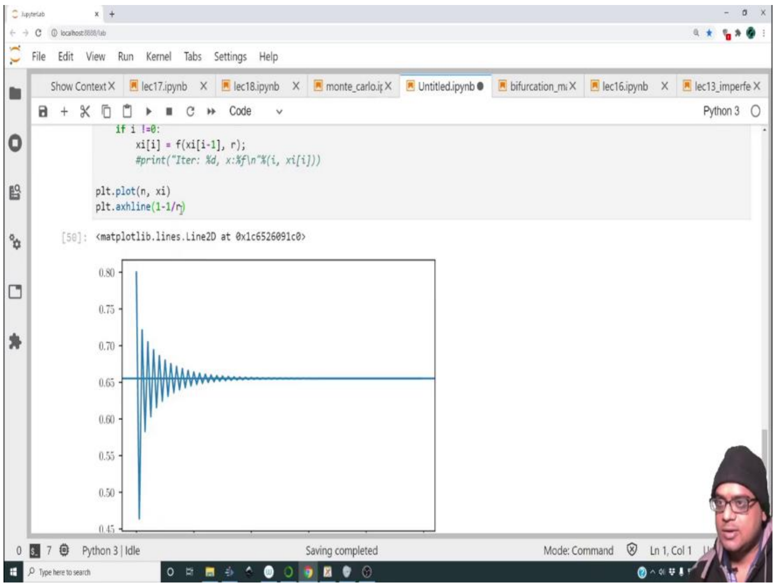
(Refer Slide Time: 36:35)



```
def f(x, r):
    return r*x*(1-x);
def g(x):
    return x;
r = 2.9;
x0 = 0.8;
n = np.arange(0, 100);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

plt.plot(n, xi)
plt.axhline(1-1/r)
```

So, plt.axhline and we will make it pass through 1 -1/r.

(Refer Slide Time: 36:44)

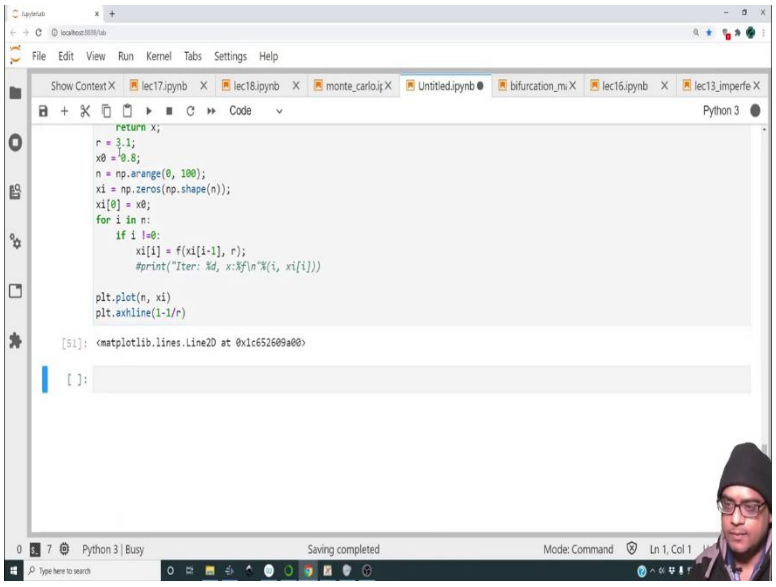


```
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[50]: <matplotlib.lines.Line2D at 0x1c6526091c0>
```

So, this is that fixed line, this is the root, this is the fixed point that we have calculated analytically.

(Refer Slide Time: 36:56)

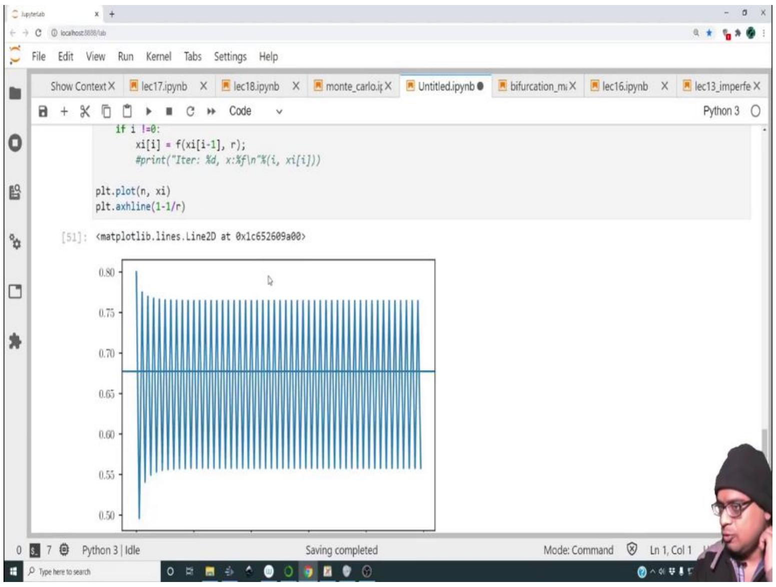


```
        return x;
    r = 3.1;
    x0 = 0.8;
    n = np.arange(0, 100);
    xi = np.zeros(np.shape(n));
    xi[0] = x0;
    for i in n:
        if i !=0:
            xi[i] = f(xi[i-1], r);
            #print("Iter: %d, x:%f\n"%(i, xi[i]))

    plt.plot(n, xi)
    plt.axhline(1-1/r)
```

```
[51]: <matplotlib.lines.Line2D at 0x1c652609a00>
```

Now, what happens when r exceeds the value of 3?

(Refer Slide Time: 36:58)

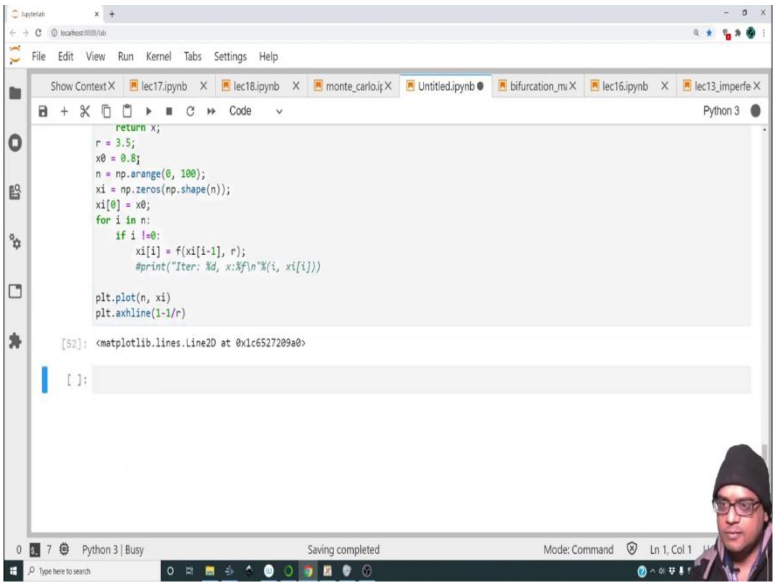


```
        if i !=0:
            xi[i] = f(xi[i-1], r);
            #print("Iter: %d, x:%f\n"%(i, xi[i]))

    plt.plot(n, xi)
    plt.axhline(1-1/r)
```

```
[51]: <matplotlib.lines.Line2D at 0x1c652609a00>
```

So, it appears to oscillate the values appear to oscillate in a periodic fashion and the period appears to be 2 meaning if I look at this point after 1, 2 it again repeats ok.

(Refer Slide Time: 37:24)

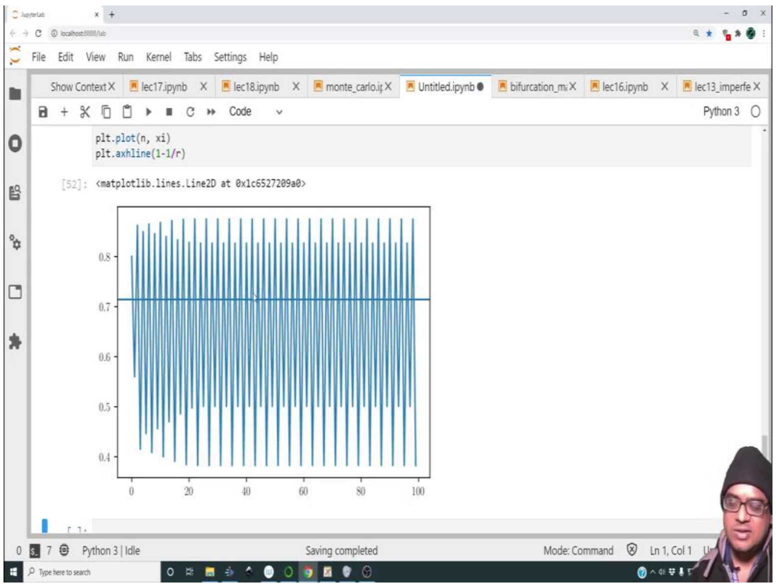


```
    return x;
r = 3.5;
x0 = 0.8;
n = np.arange(0, 100);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[52]: <matplotlib.lines.Line2D at 0x1c6527209a0>
```

So, let me now increase the value of this to 3.5. Now what happens?

(Refer Slide Time: 37:24)

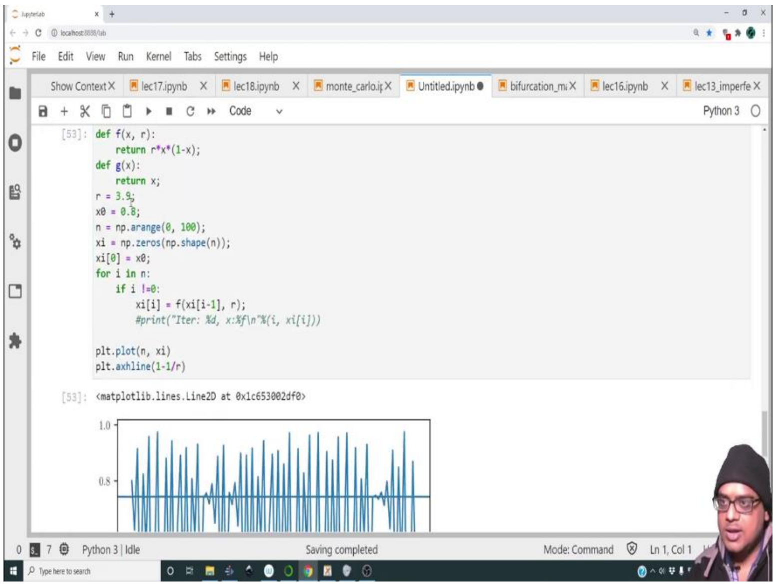


```
plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[52]: <matplotlib.lines.Line2D at 0x1c6527209a0>
```

It appears to go so, 1, then 2, then 3, then 4 and then again it repeats. So, we see that the frequency has increased alright.

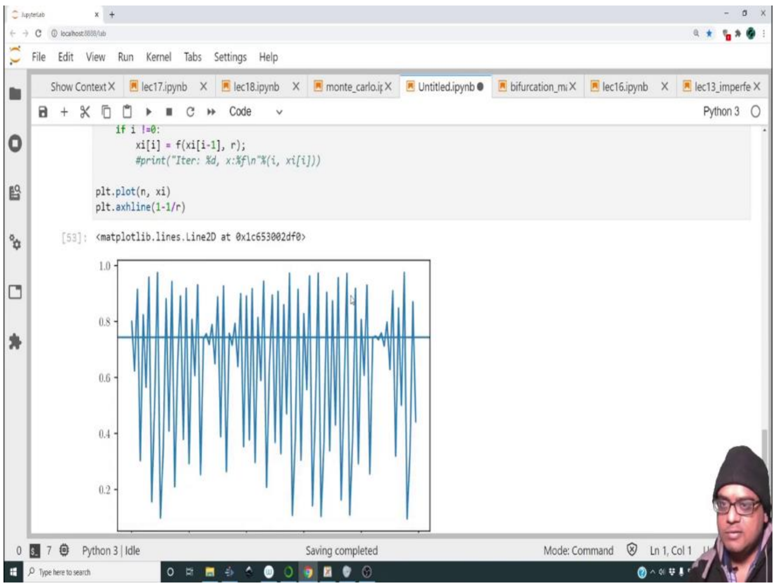
(Refer Slide Time: 37:41)



```
[53]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.9;
      x0 = 0.8;
      n = np.arange(0, 100);
      xi = np.zeros(np.shape(n));
      xi[0] = x0;
      for i in n:
          if i !=0:
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)
      plt.axhline(1-1/r)

[53]: <matplotlib.lines.Line2D at 0x1c653002df0>
```

And then let me make it 3.9.

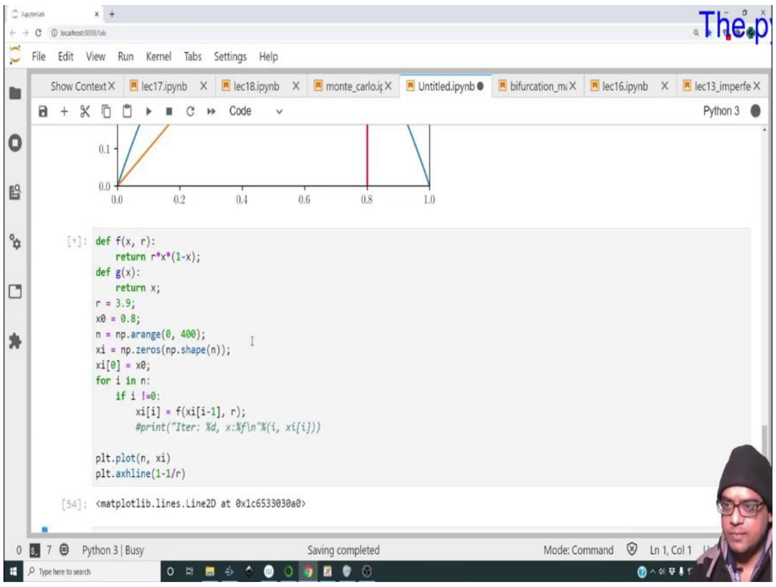
(Refer Slide Time: 37:43)



```
          if i !=0:
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)
      plt.axhline(1-1/r)

[53]: <matplotlib.lines.Line2D at 0x1c653002df0>
```

So, now, we have this. So, there is no discernible.

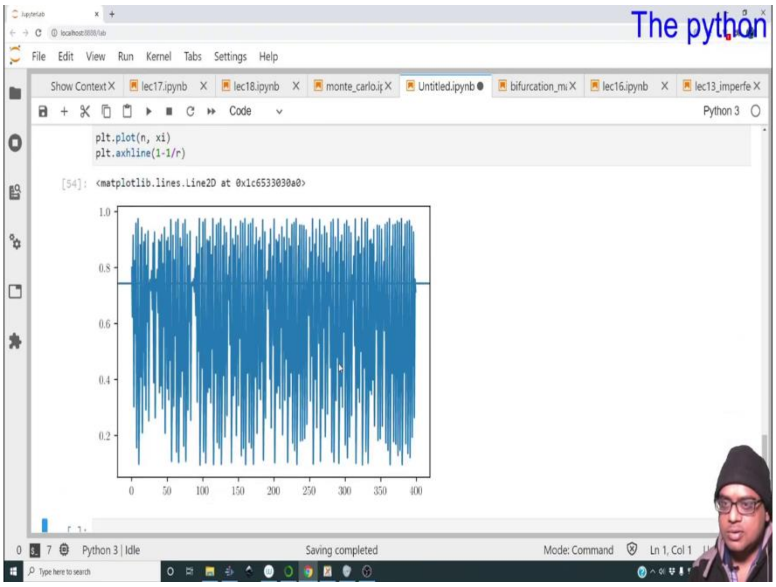
(Refer Slide Time: 37:49).



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 3.9;
     x0 = 0.8;
     n = np.arange(0, 400);
     xi = np.zeros(np.shape(n));
     xi[0] = x0;
     for i in n:
         if i !=0:
             xi[i] = f(xi[i-1], r);
             #print("Iter: %d, x:%f\n"%(i, xi[i]))

     plt.plot(n, xi)
     plt.axhline(1-1/r)

[54]: <matplotlib.lines.Line2D at 0x1c6533030a0>
```

So, let me increase the number of iterations to say 400.

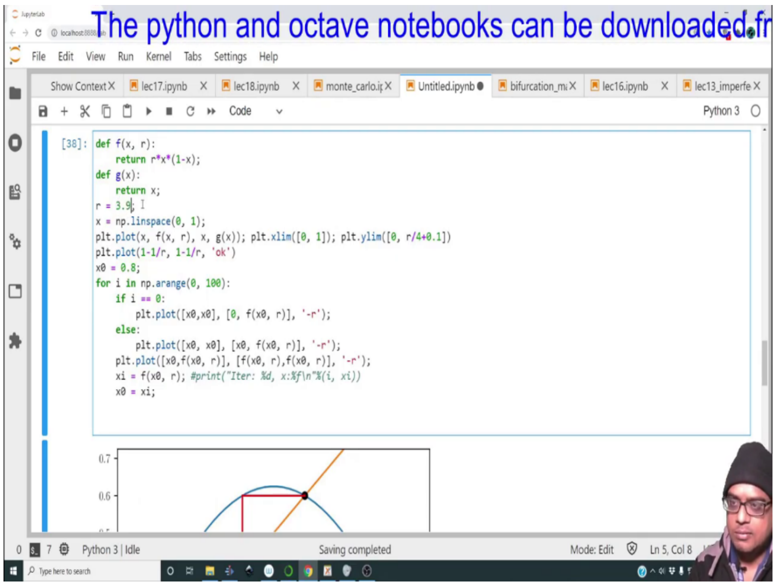
(Refer Slide Time: 37:51)



```
     plt.plot(n, xi)
     plt.axhline(1-1/r)

[54]: <matplotlib.lines.Line2D at 0x1c6533030a0>
```

So, there does not appear to be a well-defined iteration cycle.

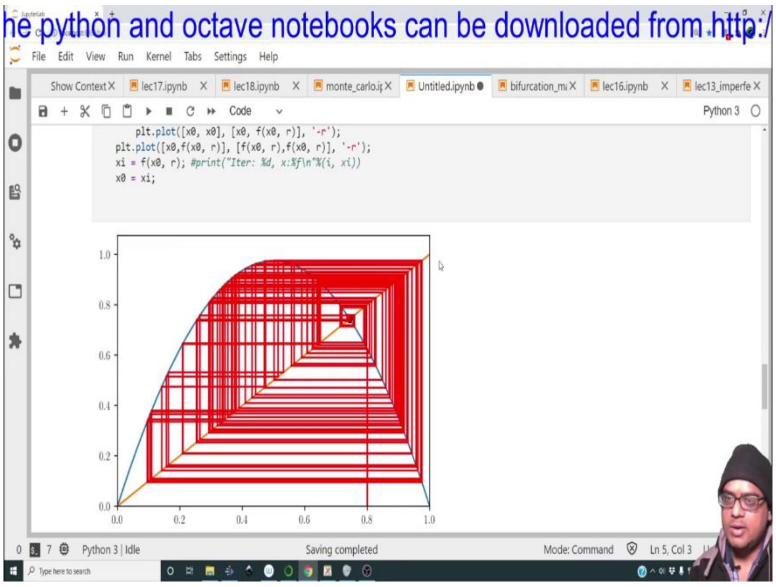
(Refer Slide Time: 38:05)



```
[38]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.9;
      x = np.linspace(0, 1);
      plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
      plt.plot(1-1/r, 1-1/r, 'ok')
      x0 = 0.8;
      for i in np.arange(0, 100):
          if i == 0:
              plt.plot([x0,x0], [0, f(x0, r)], '-r');
          else:
              plt.plot([x0, x0], [x0, f(x0, r)], '-r');
          plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
          xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
          x0 = xi;
```

And for 3.9, let us see how the cobweb looks like. So, we know that something funny happens, but how does it look like alright.

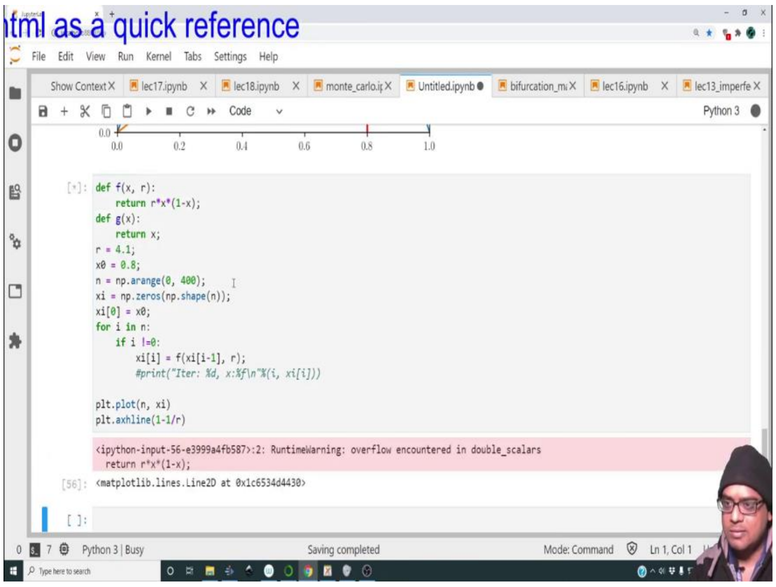
(Refer Slide Time: 38:07)



```
            plt.plot([x0, x0], [x0, f(x0, r)], '-r');
        plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
        xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
        x0 = xi;
```

So, that is the reason I mean it goes towards the fixed point, then its repelled away from the fixed point, it keeps on looping all over the place, but it is really never sort of settling down on the actual value and it appears to be like a limit cycle that we have studied in the non-linear dynamics part, but there is no periodicity.

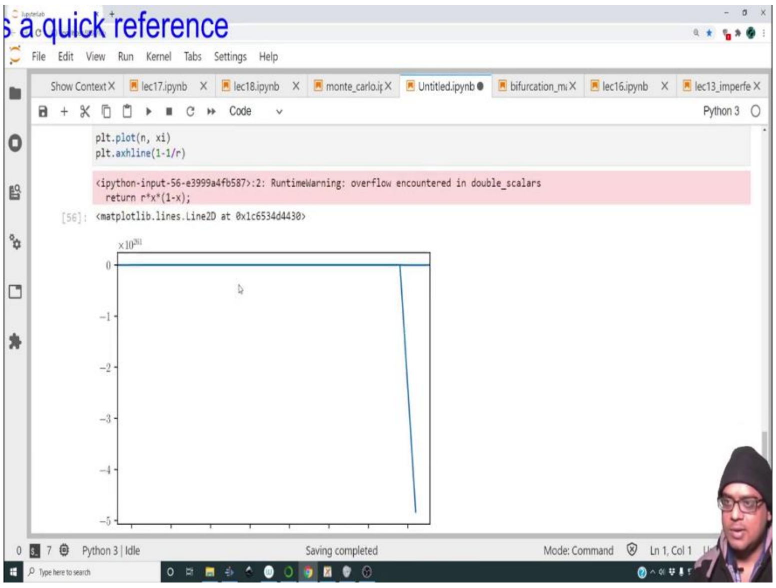
(Refer Slide Time: 38:39)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 4.1;
     x0 = 0.8;
     n = np.arange(0, 400);
     xi = np.zeros(np.shape(n));
     xi[0] = x0;
     for i in n:
         if i !=0:
             xi[i] = f(xi[i-1], r);
             #print("Iter: %d, x:%f\n"%(i, xi[i]))

     plt.plot(n, xi)
     plt.axhline(1-1/r)

<ipython-input-56-e3999a4fb587>:2: RuntimeWarning: overflow encountered in double_scalars
  return r*x*(1-x);
[56]: <matplotlib.lines.Line2D at 0x1c6534d4430>
```

Let me make it 4.1, let us see.

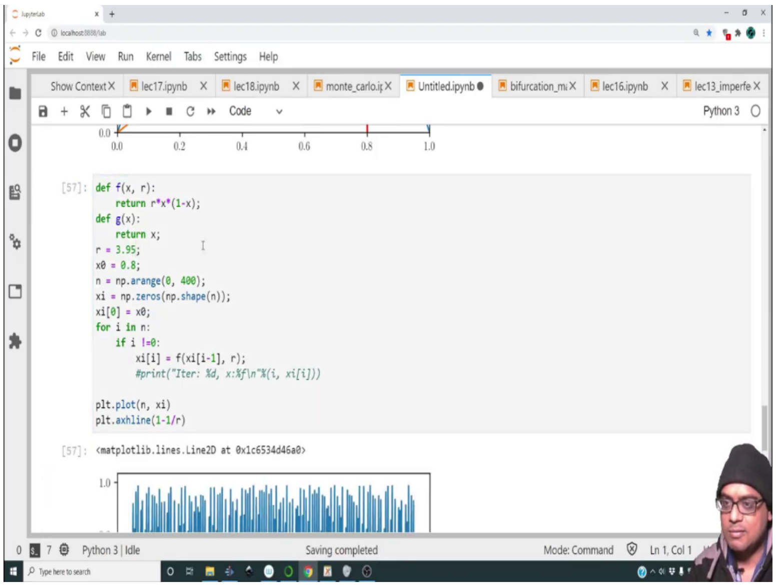
(Refer Slide Time: 38:42)



```
     plt.plot(n, xi)
     plt.axhline(1-1/r)

<ipython-input-56-e3999a4fb587>:2: RuntimeWarning: overflow encountered in double_scalars
  return r*x*(1-x);
[56]: <matplotlib.lines.Line2D at 0x1c6534d4430>
```

And it blows away to infinity well.

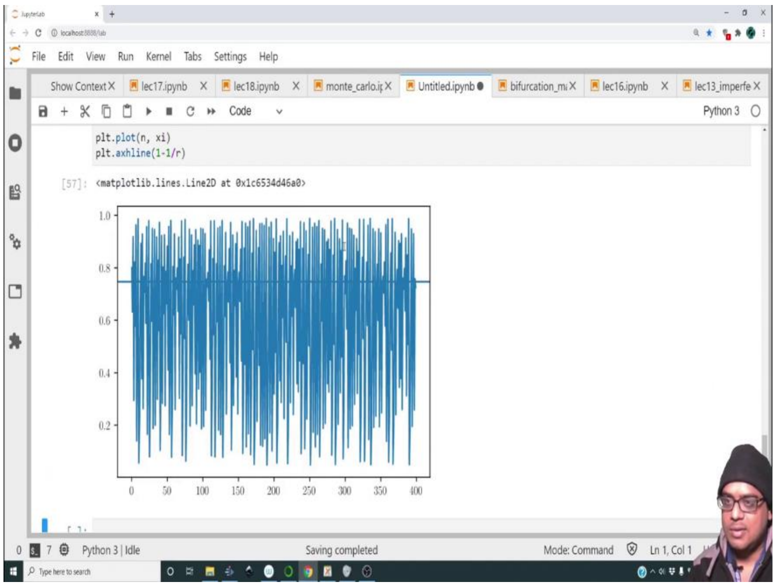
(Refer Slide Time: 38:47)



```
[57]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.95;
      x0 = 0.8;
      n = np.arange(0, 400);
      xi = np.zeros(np.shape(n));
      xi[0] = x0;
      for i in n:
          if i !=0:
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)
      plt.axhline(1-1/r)

[57]: <matplotlib.lines.Line2D at 0x1c6534d46a0>
```

Let me make it 3.95.

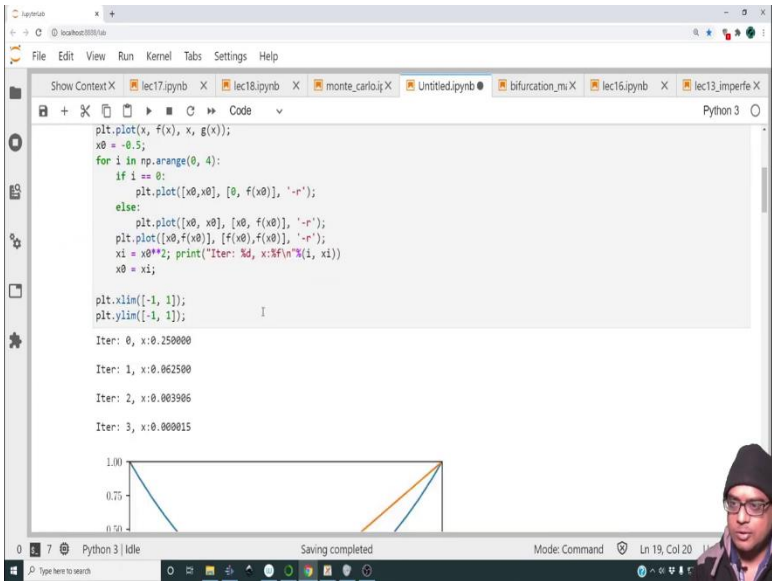
(Refer Slide Time: 38:51)



```
plt.plot(n, xi)
plt.axhline(1-1/r)

[57]: <matplotlib.lines.Line2D at 0x1c6534d46a0>
```

And we still have something like this. So, what is what happens beyond r = 3 ok, there is obviously, some period doubling that is happening and beyond that period doubling, it seems to go off the rails I mean there is no discernible pattern that we are able to observe and let us look at, let us go back a step and look at how the plot actually looks like.

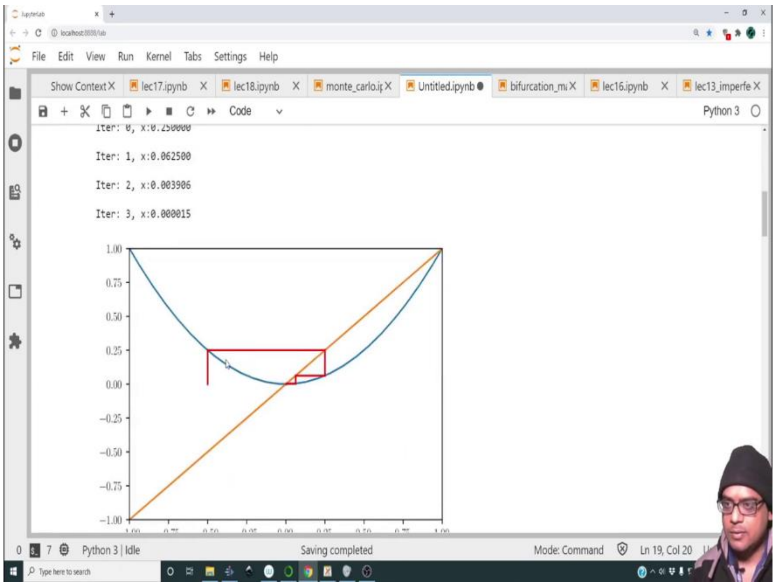
(Refer Slide Time: 39:23)



```
plt.plot(x, f(x), x, g(x));
x0 = -0.5;
for i in np.arange(0, 4):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0)], '-r');
    plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
    xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;

plt.xlim([-1, 1]);
plt.ylim([-1, 1]);
```

```
Iter: 0, x:0.250000

Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

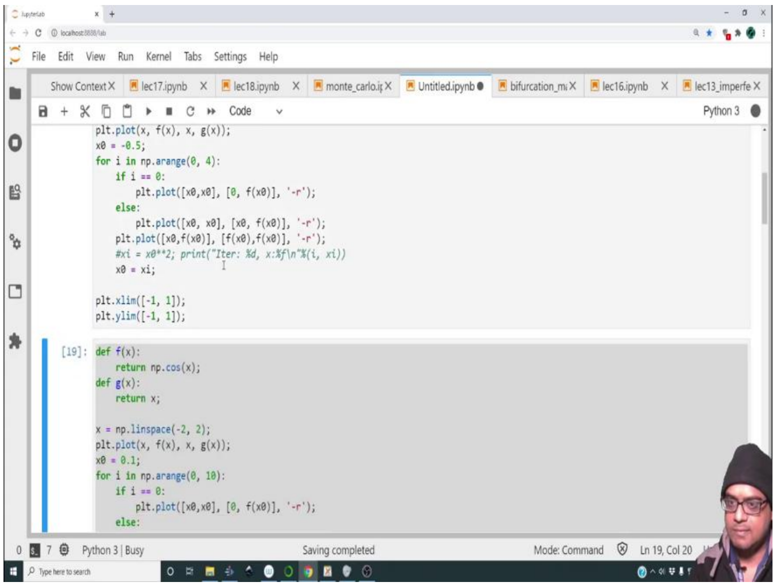
(Refer Slide Time: 39:24)



```
Iter: 1, x:0.062500

Iter: 2, x:0.003906

Iter: 3, x:0.000015
```

So, this was the original plot ok.

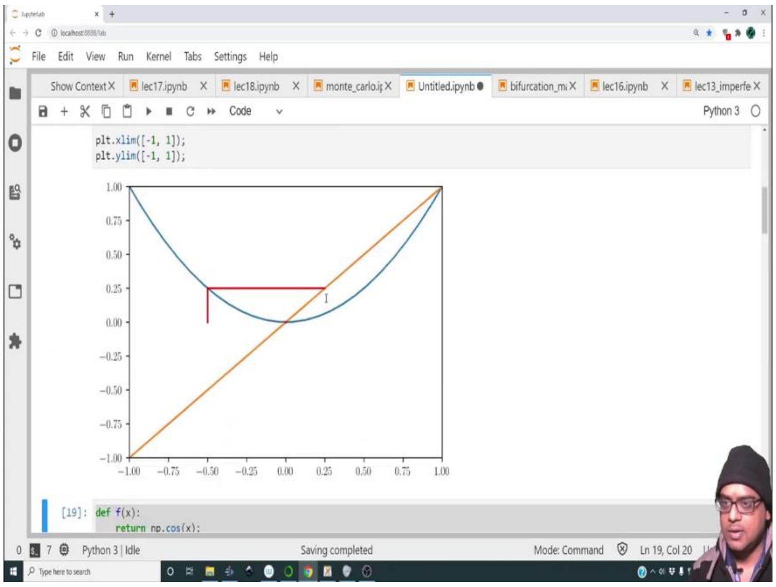
(Refer Slide Time: 39:28)



```
    plt.plot(x, f(x), x, g(x));
    x0 = -0.5;
    for i in np.arange(0, 4):
        if i == 0:
            plt.plot([x0,x0], [0, f(x0)], '-r');
        else:
            plt.plot([x0, x0], [x0, f(x0)], '-r');
        plt.plot([x0,f(x0)], [f(x0),f(x0)], '-r');
        #xi = x0**2; print("Iter: %d, x:%f\n"%(i, xi))
        x0 = xi;

    plt.xlim([-1, 1]);
    plt.ylim([-1, 1]);

[19]: def f(x):
        return np.cos(x);
    def g(x):
        return x;

    x = np.linspace(-2, 2);
    plt.plot(x, f(x), x, g(x));
    x0 = 0.1;
    for i in np.arange(0, 10):
        if i == 0:
            plt.plot([x0,x0], [0, f(x0)], '-r');
        else:
```

Let me suppress the printing of the iterates and.

(Refer Slide Time: 39:30)

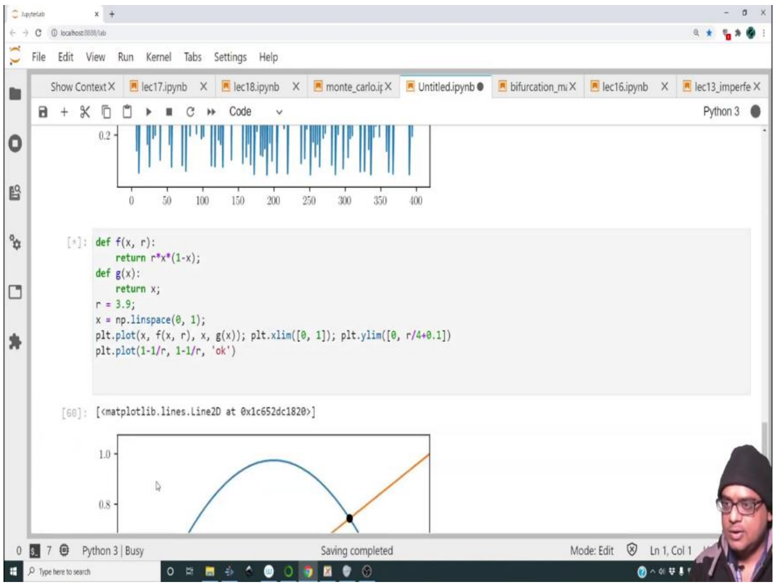


```
    plt.xlim([-1, 1]);
    plt.ylim([-1, 1]);
```

```
[19]: def f(x):
        return np.cos(x);
```

So, if we go back, let me this has to be there. So, this is a different map.

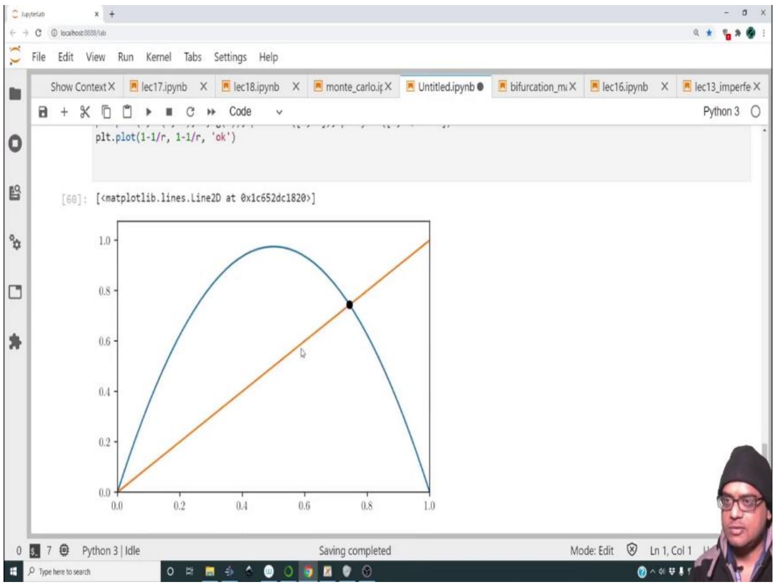
(Refer Slide Time: 39:50)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 3.9;
     x = np.linspace(0, 1);
     plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
     plt.plot(1-1/r, 1-1/r, 'ok')

[60]: [<matplotlib.lines.Line2D at 0x1c652dc1820>]
```

In fact, let me take this snippet, let me paste it over here so that we can see the map clearly. So, let me remove all these guess values and let me simply plot this ok.

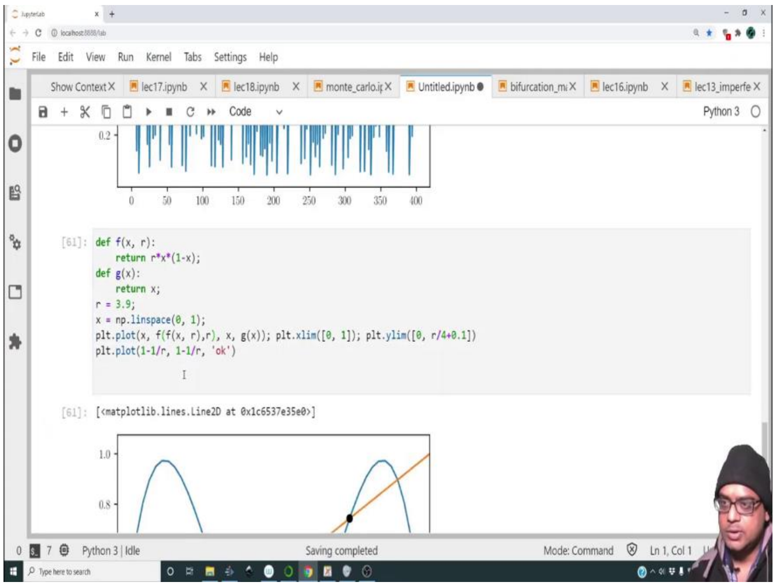
(Refer Slide Time: 40:01)



```
plt.plot(1-1/r, 1-1/r, 'ok')

[60]: [<matplotlib.lines.Line2D at 0x1c652dc1820>]
```

So, this is how the plot looks like and now, beyond r = 3 I mean this is not a stable fixed point, it is an unstable fixed point, it is not converging to that root.

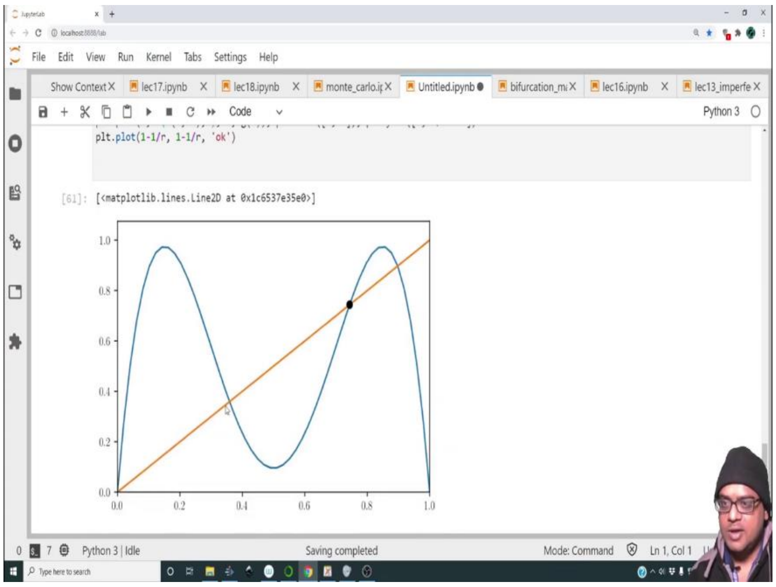
(Refer Slide Time: 40:28)



```
[61]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.9;
      x = np.linspace(0, 1);
      plt.plot(x, f(f(x, r),r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
      plt.plot(1-1/r, 1-1/r, 'ok')

[61]: [<matplotlib.lines.Line2D at 0x1c6537e35e0>]
```

But now, let me plot instead of f(x), we saw that the period doubled instead of converging it to two oscillations so, let me plot f(f(x)), let me plot this.

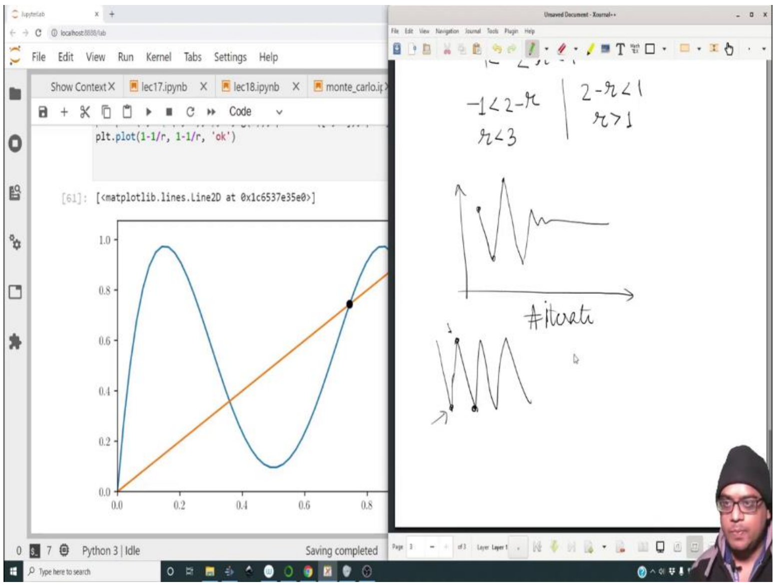
(Refer Slide Time: 40:34)



```
      plt.plot(1-1/r, 1-1/r, 'ok')

[61]: [<matplotlib.lines.Line2D at 0x1c6537e35e0>]
```

So, now, we have we saw that it was oscillating between two roots, but not two roots, but two iterates and those iterates were actually this and this, but now, we do not know whether these points are stable, we need to guarantee that the slopes of f`(f(f(x))) alright.

(Refer Slide Time: 40:59)

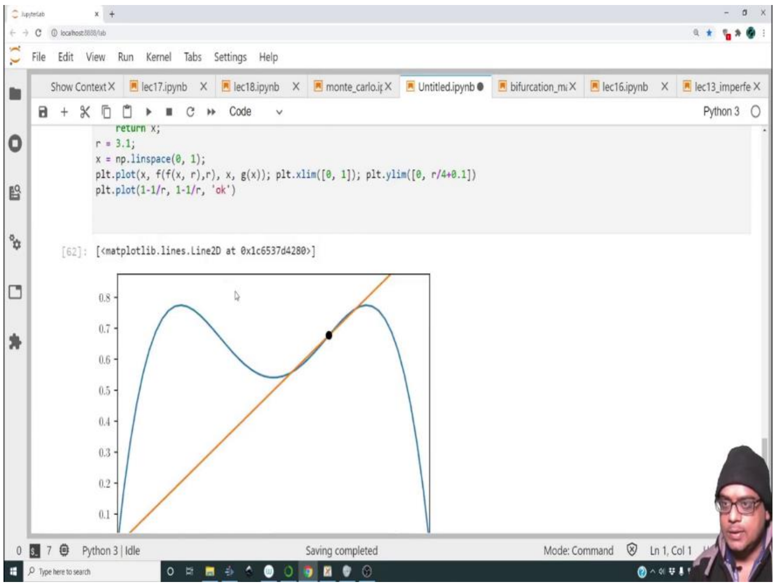


```
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[61]: [<matplotlib.lines.Line2D at 0x1c6537e35e0>]
```

So, we are trying to explain why there were two oscillations like this, this and this were being repeated meaning it is going to this point, then it is thrown away to this point and then, it is again thrown away to this point over all successive iterations, it is not converging.

(Refer Slide Time: 41:21)

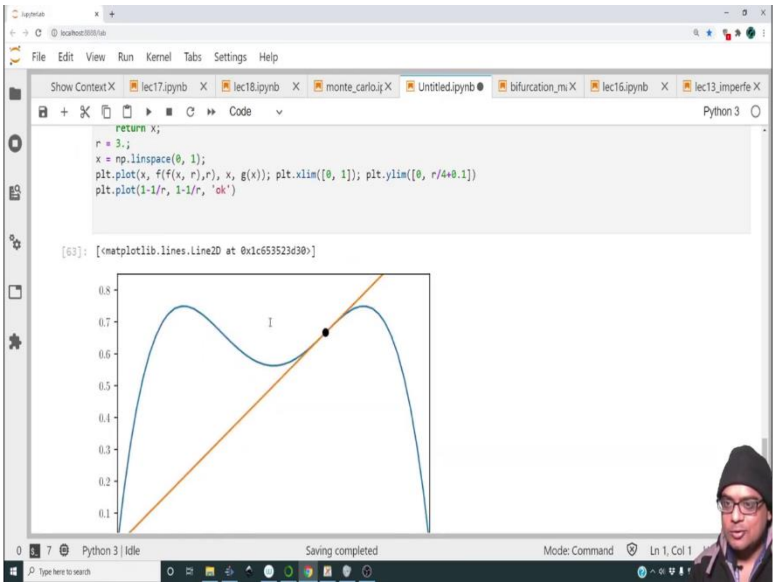


```
    return x;
r = 3.1;
x = np.linspace(0, 1);
plt.plot(x, f(f(x, r),r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[62]: [<matplotlib.lines.Line2D at 0x1c6537d4280>]
```

So, let me make it 3.1 ok.

(Refer Slide Time: 41:25)

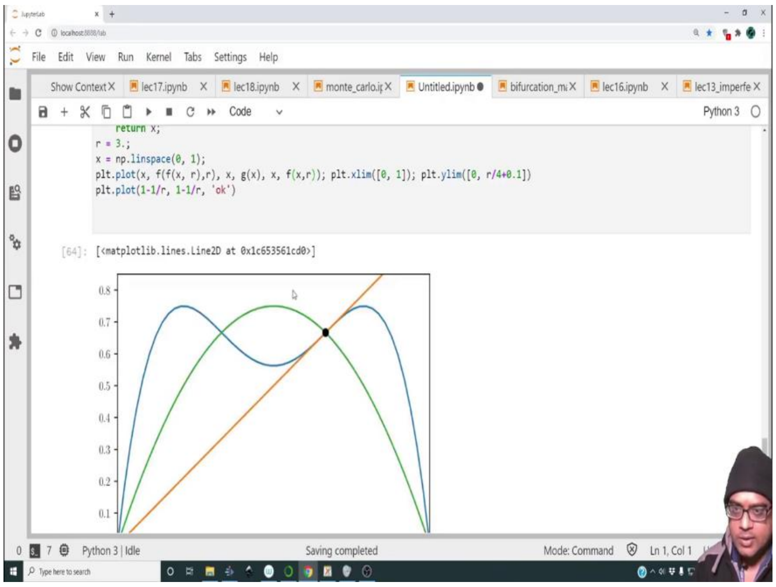


```
    return x;
r = 3.;
x = np.linspace(0, 1);
plt.plot(x, f(f(x, r),r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[63]: [<matplotlib.lines.Line2D at 0x1c653523d30>]
```

So, in fact, let me make it this 3, let me show you. So, at 3, it is just tangent and that is the boundary of stability. Below 3, we had the single map that is f(x) which was converging.

(Refer Slide Time: 41:45)

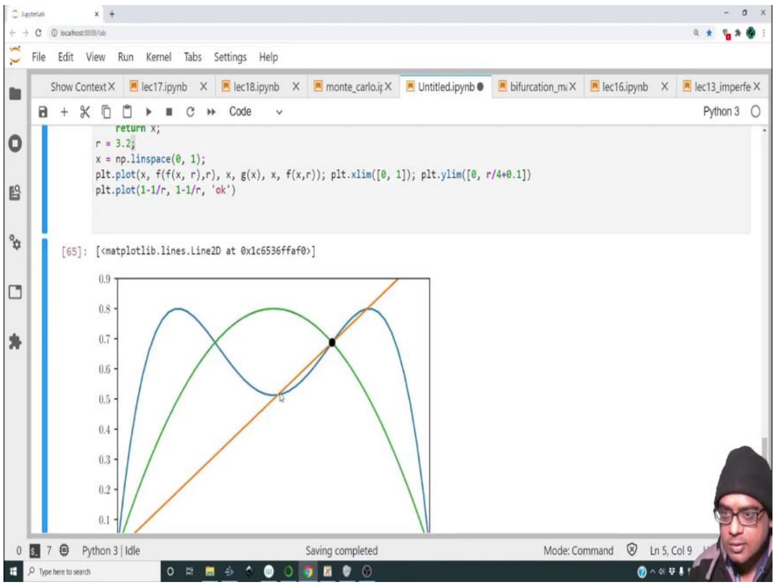


```
    return x;
r = 3.;
x = np.linspace(0, 1);
plt.plot(x, f(f(x, r),r), x, g(x), x, f(x,r)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[64]: [<matplotlib.lines.Line2D at 0x1c653561cd0>]
```

So, let me also plot x,f(x,r). So, the green curve intersects the orange curve and at this parameter where r = 3, it was a stable point, but beyond 3, it becomes an unstable point and beyond 3, after it becomes an unstable point, we see that it oscillates between two iterates.

(Refer Slide Time: 42:15)



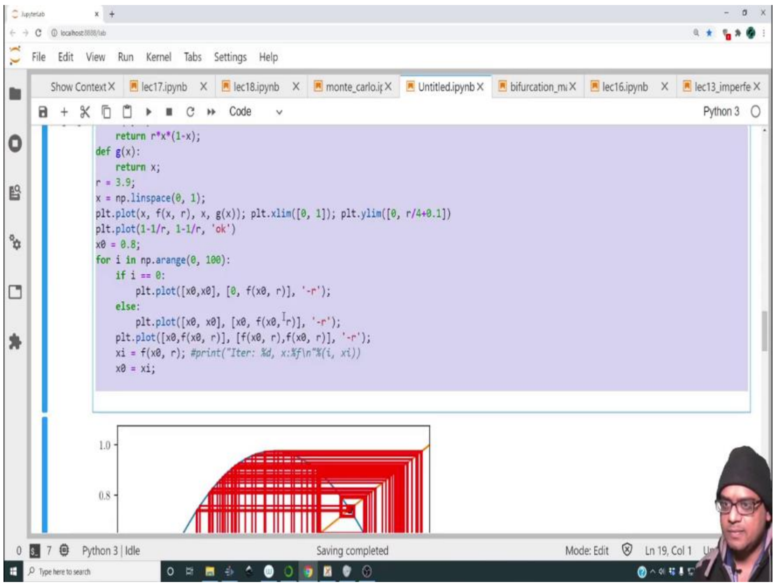
```
    return x;
r = 3.2;
x = np.linspace(0, 1);
plt.plot(x, f(f(x, r),r), x, g(x), x, f(x,r)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[65]: [<matplotlib.lines.Line2D at 0x1c6536ffaf0>]
```

And those two iterates are these roots, this root and this root while this root becomes unstable. So, it is like a bifurcation that occurs where the original root becomes unstable and it goes to two different roots and you can revisit the earlier lectures on bifurcations to figure out what kind of bifurcation it is. I am not going to discuss all that over here, but you can imagine that it becomes tangential and now you have two roots appearing, this root and this root.

And this root, the slope appears to be low, but at this point also the slope appears to be 0. So, these two points, these two iterates are stable points, but whether or not the iterates are oscillating around these two points remains to be seen.

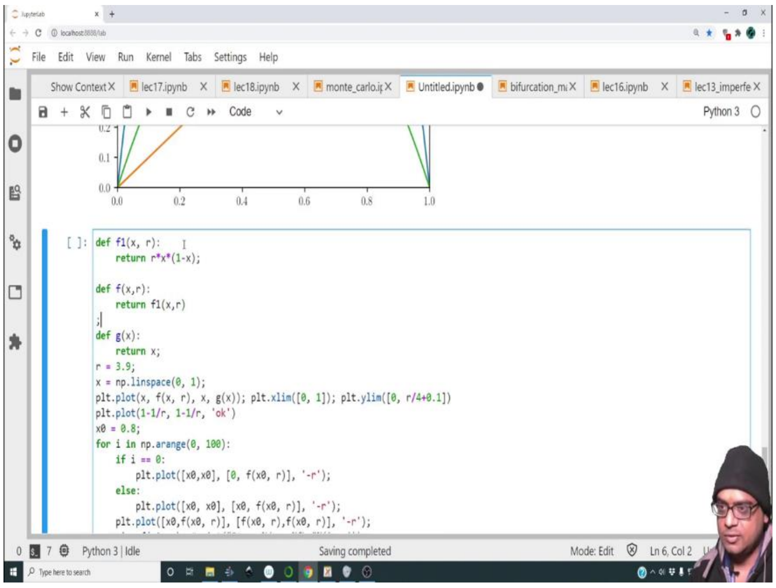
(Refer Slide Time: 43:05)



```
    return r*x*(1-x);
def g(x):
    return x;
r = 3.9;
x = np.linspace(0, 1);
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
x0 = 0.8;
for i in np.arange(0, 100):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, let us go back to this diagram, let me copy this.

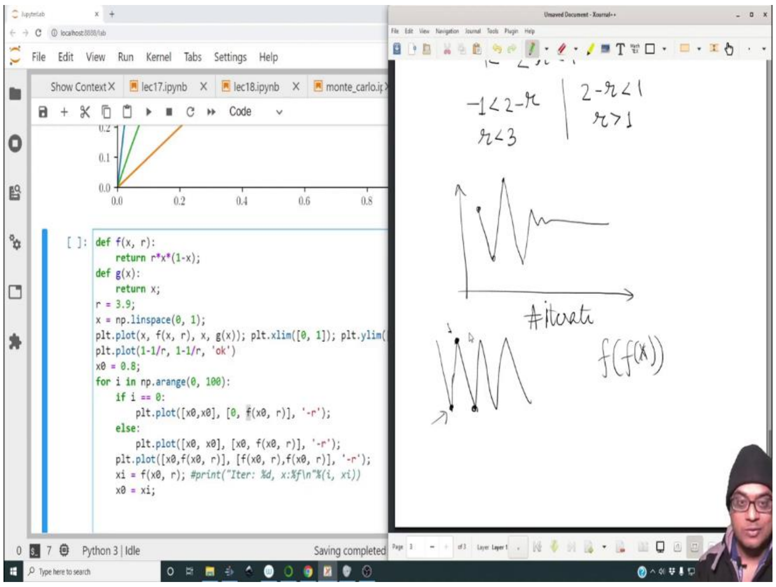
(Refer Slide Time: 43:09)



```
def f1(x, r):
    return r*x*(1-x);

def f(x,r):
    return f1(x,r)
;
def g(x):
    return x;
r = 3.9;
x = np.linspace(0, 1);
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
x0 = 0.8;
for i in np.arange(0, 100):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
```

Let me go over here, paste this.

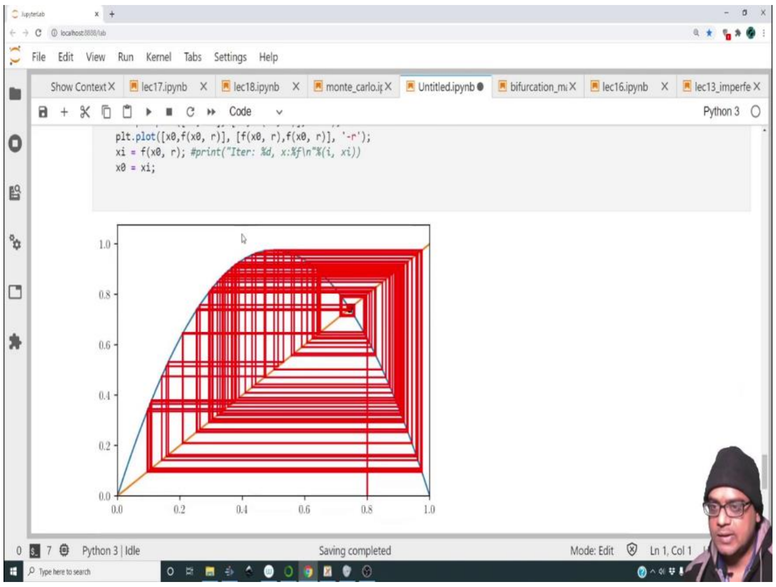
(Refer Slide Time: 43:24)



```
def f(x, r):
    return r*x*(1-x);
def g(x):
    return x;
r = 3.9;
x = np.linspace(0, 1);
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim(
plt.plot(1-1/r, 1-1/r, 'ok')
x0 = 0.8;
for i in np.arange(0, 100):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

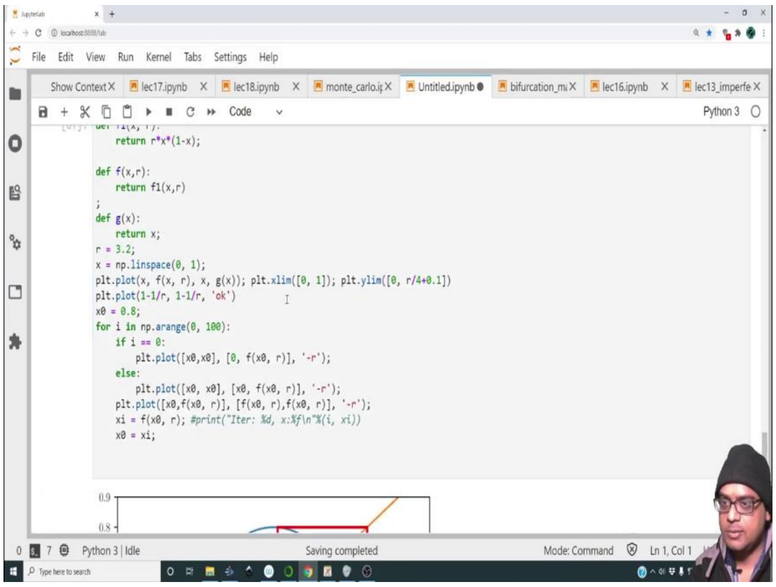
Now, let me define f(x, r) as something else. So, here, we want the double iterate meaning f(f(x)) because it is going in a two cycle so, we want to see the iterate of the iterate because that appears to give us two roots. So, let me modify this code so that it returns us the f(f(x)). So, let me define this as f1 and let me define f(x,r) and we will return f1(x,r). So, it is like calling that function within this function. So, let me plot this.

(Refer Slide Time: 44:03)



```
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

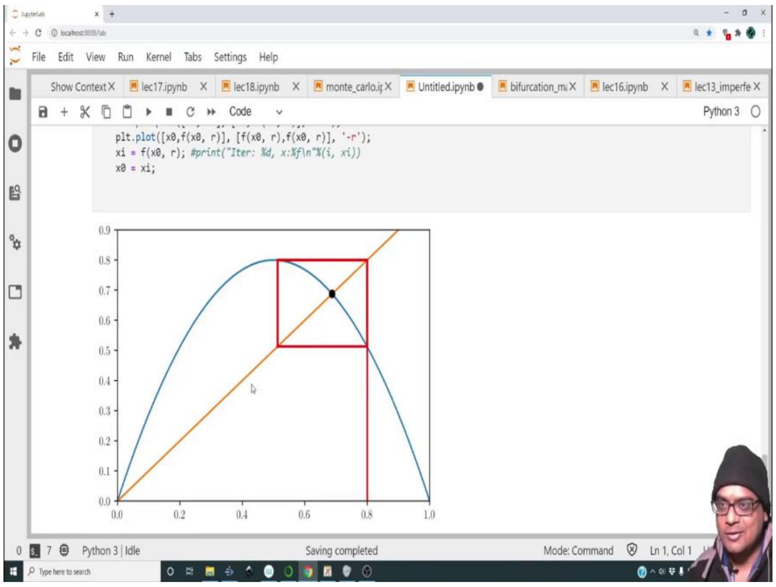
(Refer Slide Time: 44:21)



```
    return r*x*(1-x);

def f(x,r):
    return f1(x,r)
;
def g(x):
    return x;
r = 3.2;
x = np.linspace(0, 1);
plt.plot(x, f(x, r), x, g(x)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
x0 = 0.8;
for i in np.arange(0, 100):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

And in fact, let me plot, let me reduce the parameter.

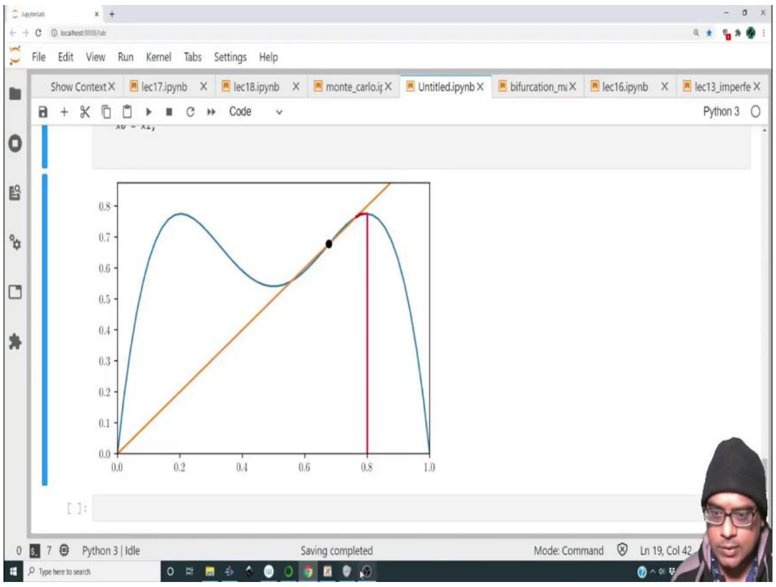
(Refer Slide Time: 44:23)



```
plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
x0 = xi;
```

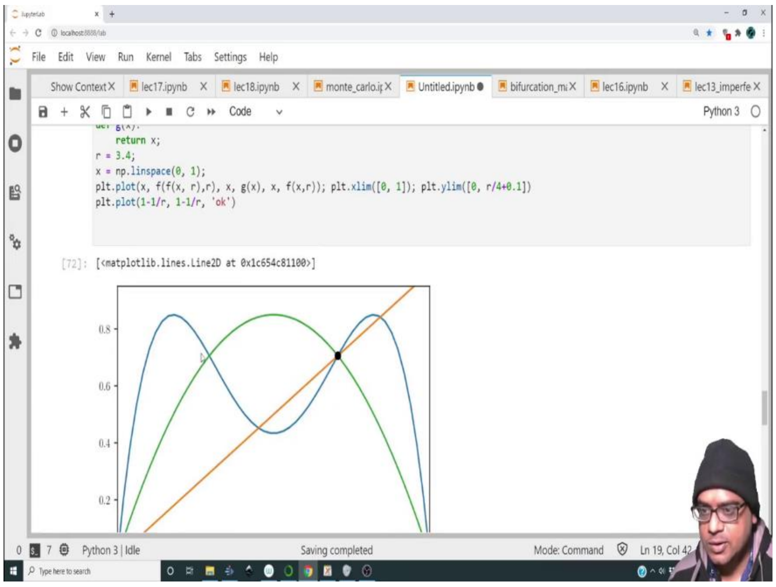
After reducing the parameter, let me. Well, you can see that it is settling on these two roots. So, if I start from here, it is like going in I mean in a metaphorical circle, it is going in a rectangle across those two iterates but let us try to show it also.

(Refer Slide Time: 44:43)



So, if you try to plot the how the iterates occur in a map of f of; in the map of f of f of; in the map of f(f(x)), it appears to converge to this root, but the reality is in the in actual map, it is actually going across these two points and that is clear by looking at these oscillations.

(Refer Slide Time: 45:24)

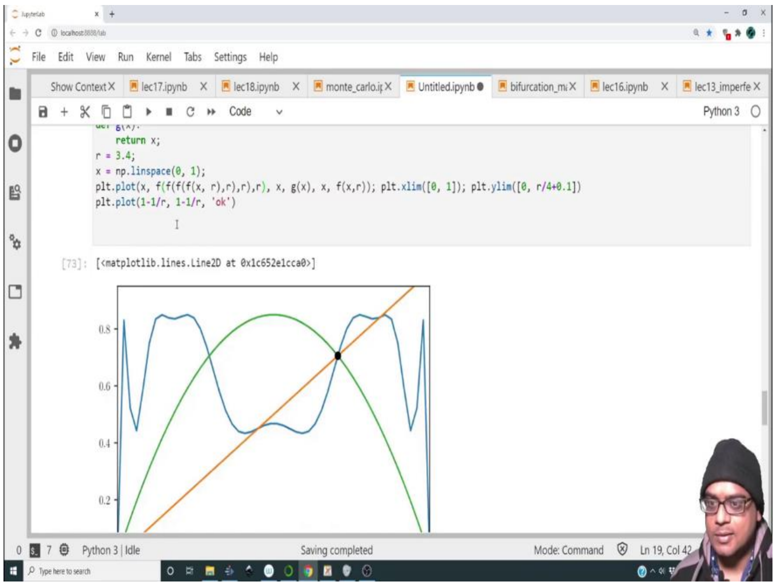


```
    return x;
r = 3.4;
x = np.linspace(0, 1);
plt.plot(x, f(f(x, r),r), x, g(x), x, f(x,r)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[72]: [<matplotlib.lines.Line2D at 0x1c654c81100>]
```

So, now, let us increase the value of the parameter ok. So, now, at a certain point, it will also become unstable.

(Refer Slide Time: 45:33)

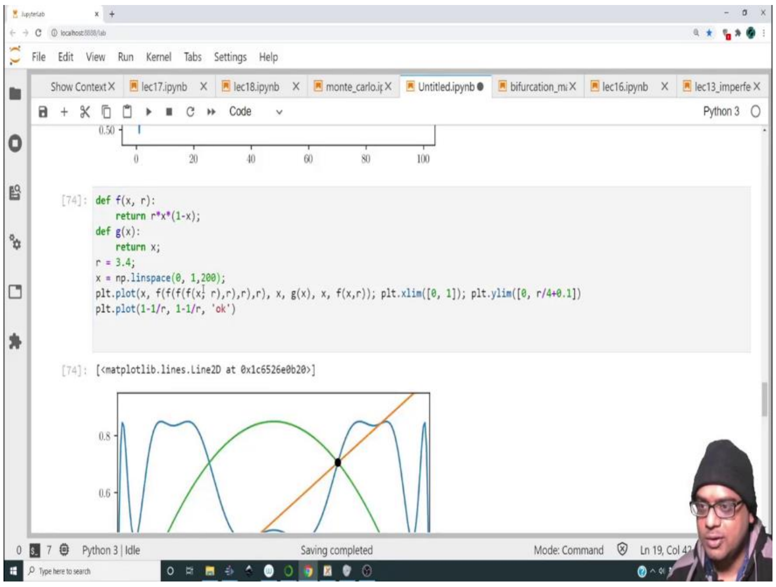


```
    return x;
r = 3.4;
x = np.linspace(0, 1);
plt.plot(x, f(f(f(f(x, r),r),r),r), x, g(x), x, f(x,r)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
plt.plot(1-1/r, 1-1/r, 'ok')
```

```
[73]: [<matplotlib.lines.Line2D at 0x1c652e1cca0>]
```

So, probably we have to plot f of this and because we see that the period becomes 4 so, we need to plot f(f(f(f(x)))) ok.

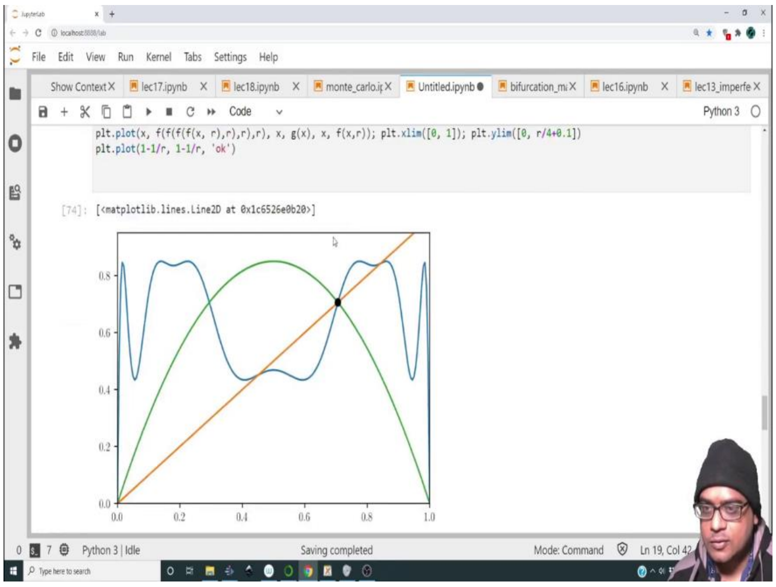
(Refer Slide Time: 45:54)



```
[74]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.4;
      x = np.linspace(0, 1,200);
      plt.plot(x, f(f(f(f(x, r),r),r),r), x, g(x), x, f(x,r)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
      plt.plot(1-1/r, 1-1/r, 'ok')

[74]: [<matplotlib.lines.Line2D at 0x1c6526e0b20>]
```

So, now, it will intersect over here where the slope is mild. So, we need obviously, more number of points over here let me choose this to have 200 points.

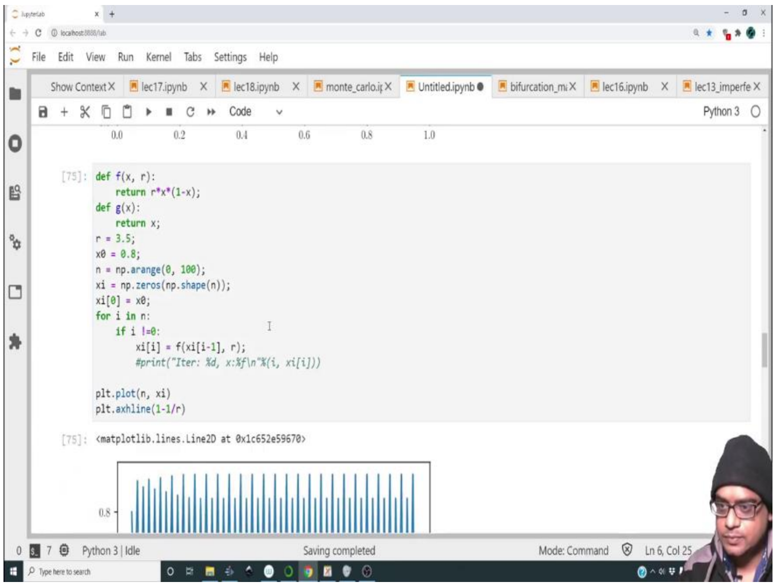
(Refer Slide Time: 45:56)



```
      plt.plot(x, f(f(f(f(x, r),r),r),r), x, g(x), x, f(x,r)); plt.xlim([0, 1]); plt.ylim([0, r/4+0.1])
      plt.plot(1-1/r, 1-1/r, 'ok')

[74]: [<matplotlib.lines.Line2D at 0x1c6526e0b20>]
```

So, it will intersect over here, then over here so, these two become the stable points, but then eventually, if you keep changing the parameter, you will see more intersection points and so, as you keep changing this, as you keep taking a larger value of r, this stable point will become unstable, it will give rise to more and more points afterwards it will descend into chaos as we have already seen.

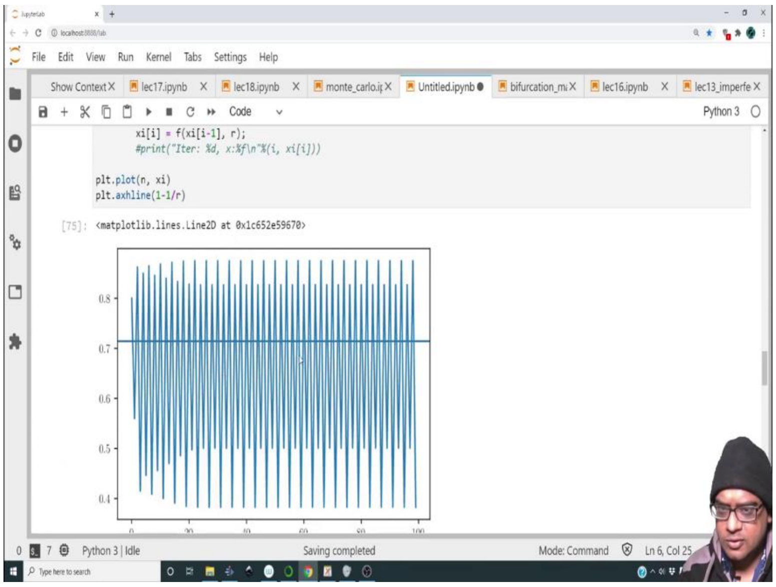
(Refer Slide Time: 46:26)



```
[75]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.5;
      x0 = 0.8;
      n = np.arange(0, 100);
      xi = np.zeros(np.shape(n));
      xi[0] = x0;
      for i in n:
          if i !=0:
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)
      plt.axhline(1-1/r)

[75]: <matplotlib.lines.Line2D at 0x1c652e59670>
```

So, when it becomes quite large.

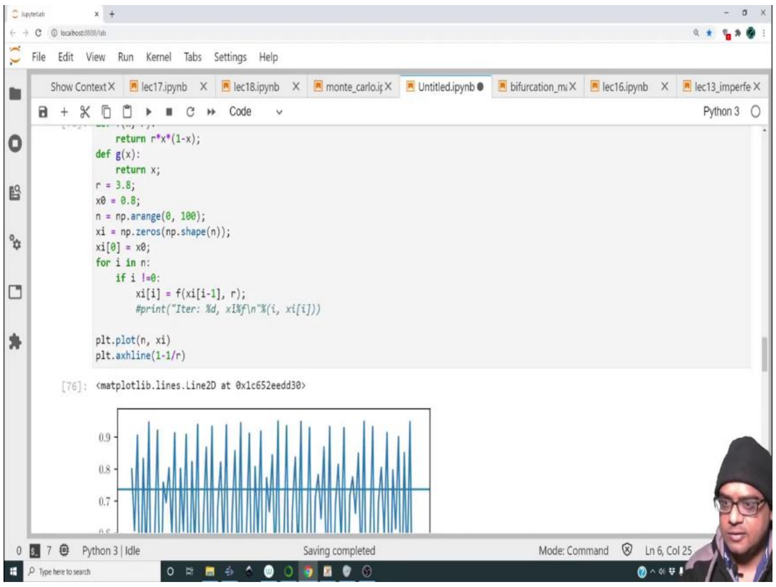
(Refer Slide Time: 46:28)



```
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)
      plt.axhline(1-1/r)

[75]: <matplotlib.lines.Line2D at 0x1c652e59670>
```

So, for this also there is a four cycle going on.

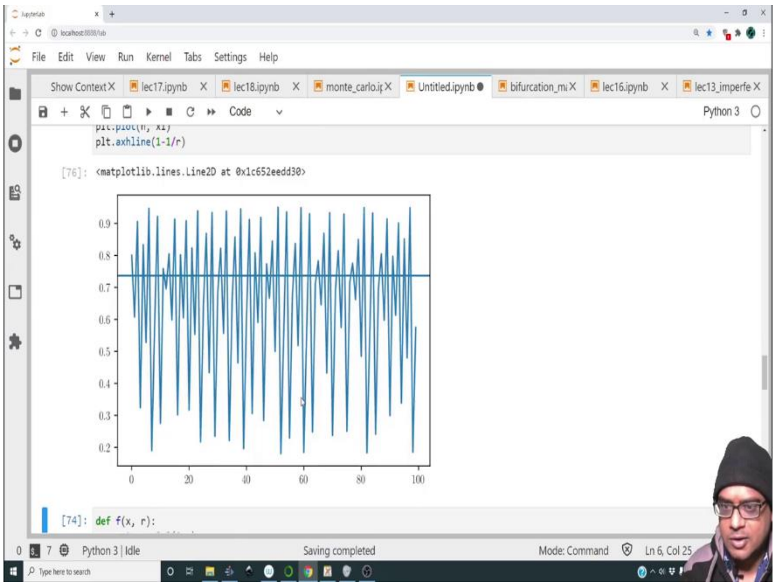
(Refer Slide Time: 46:32)



```
    return r*x*(1-x);
def g(x):
    return x;
r = 3.8;
x0 = 0.8;
n = np.arange(0, 100);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, xI%f\n"%(i, xi[i]))

plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[76]: <matplotlib.lines.Line2D at 0x1c652eedd30>
```

(Refer Slide Time: 46:33)



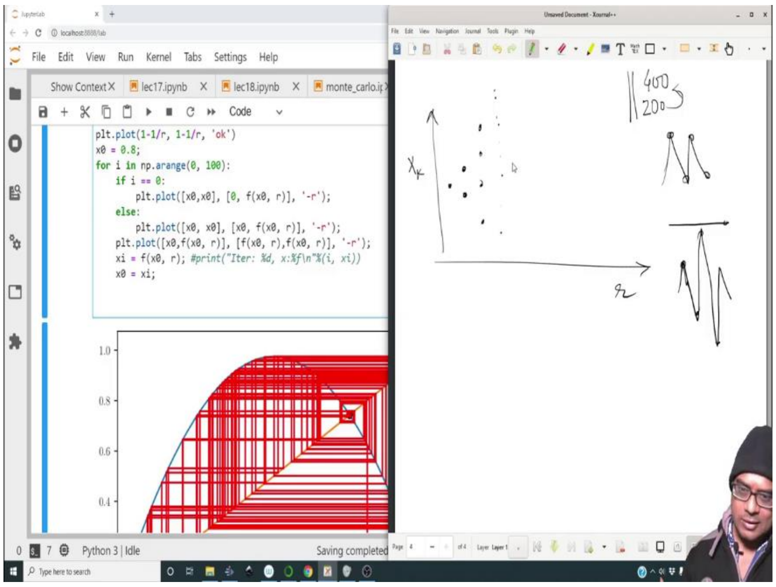
```
plt.axhline(1-1/r)
```

```
[76]: <matplotlib.lines.Line2D at 0x1c652eedd30>
```

```
[74]: def f(x, r):
```

For this and there is something weird going on, there appear to be some intermittent moments, but so, how do we quantify all this I mean there is something going on I mean as we change the parameter r, either it is iterating between I mean it goes to a fixed point when the parameter is less than 3, when it is more than 3, it goes into a 2 cycle and into a 4-cycle, into an 8 cycle, then completely erratic so, how do we make sense of this and the way is to make a bifurcation map ok. So, let me first tell you how that map will be generated.

(Refer Slide Time: 47:27)



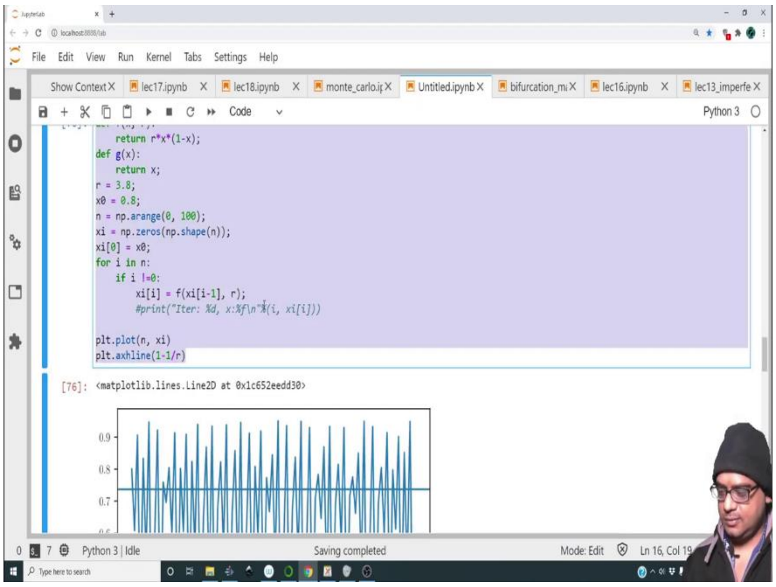
```
plt.plot(1-1/r, 1-1/r, 'ok')
x0 = 0.8;
for i in np.arange(0, 100):
    if i == 0:
        plt.plot([x0,x0], [0, f(x0, r)], '-r');
    else:
        plt.plot([x0, x0], [x0, f(x0, r)], '-r');
    plt.plot([x0,f(x0, r)], [f(x0, r),f(x0, r)], '-r');
    xi = f(x0, r); #print("Iter: %d, x:%f\n"%(i, xi))
    x0 = xi;
```

So, for each value of r so, suppose on the x axis, we have the values of r and we will plot so, we will do say 400 iterations and we will plot the 200th to 400th iteration whatever iterates we have, we will plot those values ok. So, if it goes into a two cycle, we will do see two points over here.

So, here is x k going from 200 to 400 so, if it settles down to this two, if it goes flat, then we will see one point, if it has something like this like 1, 2, 3, 4 and then it repeats so, we will have 1, 2, 3, 4 something like this. So, this will call; this will call as a bifurcation map.

So, let us see how we can assess the way this kind of cascading occurs and once it becomes very erratic, we will expect to see a lot of points, but will it actually have a cascade or will it not have a cascade so, let us see, let us see how to do that.

(Refer Slide Time: 48:51).

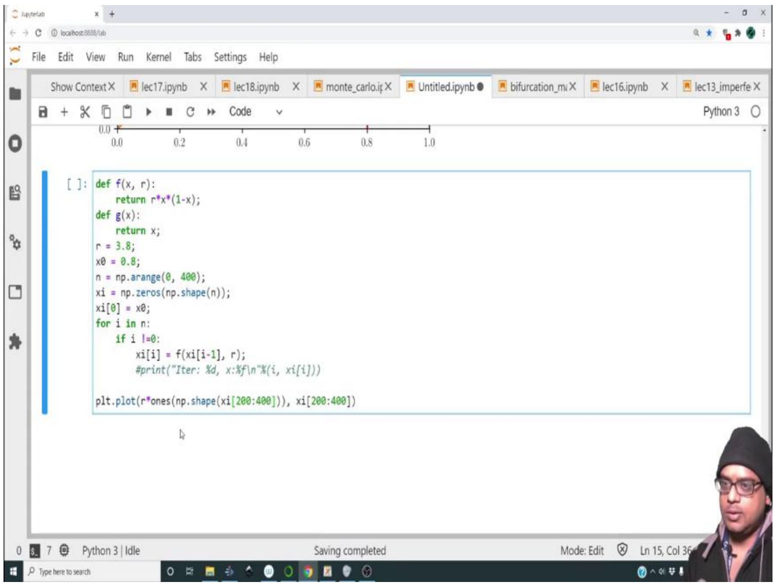


```
    return r*x*(1-x);
def g(x):
    return x;
r = 3.8;
x0 = 0.8;
n = np.arange(0, 100);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[76]: <matplotlib.lines.Line2D at 0x1c652eedd30>
```

So, we will reuse some of our code as usual. We will take this; this appears to be a nice snippet to work with.

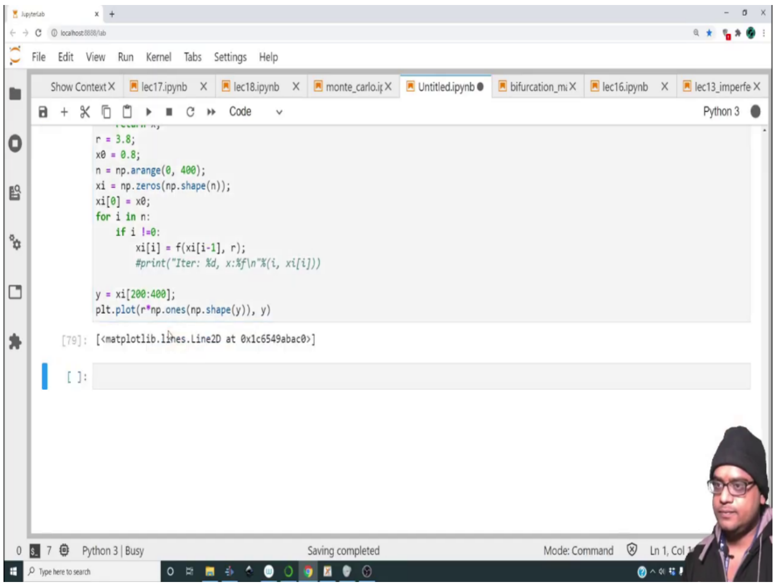
(Refer Slide Time: 48:55)



```
[ ]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;
     r = 3.8;
     x0 = 0.8;
     n = np.arange(0, 400);
     xi = np.zeros(np.shape(n));
     xi[0] = x0;
     for i in n:
         if i !=0:
             xi[i] = f(xi[i-1], r);
             #print("Iter: %d, x:%f\n"%(i, xi[i]))

     plt.plot(r*ones(np.shape(xi[200:400])), xi[200:400])
```

So, over here, we will increase the iterations to 400 and we will plot not the time series so, plotting n not the time series, but the iteration series so, n versus xi that will be the iteration series, but we do not want this in fact, we do not want any of this. So, we will plot, we have to plot r versus xi going from 200 to 400 I mean by that I mean going to 399. But now, r is a scalar value so, we need to have ones multiplying this. So, when r times one's np.shape(xi[200:400]).

(Refer Slide Time: 49:55)

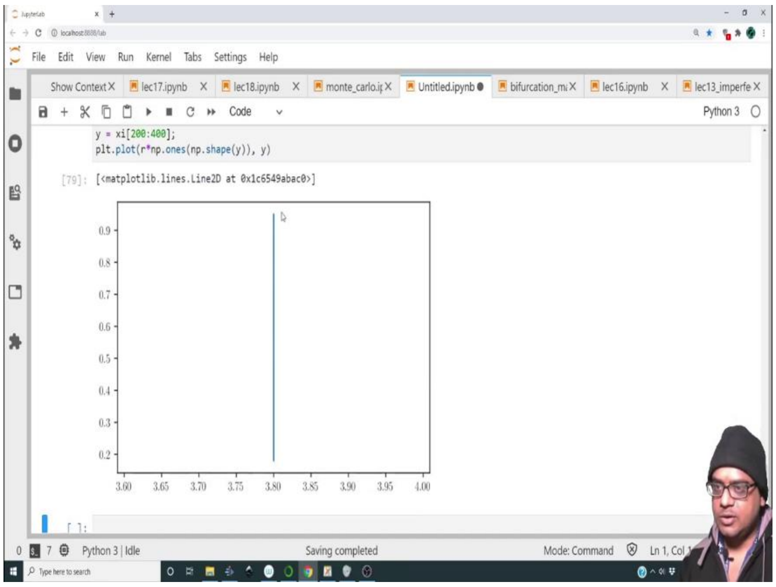


```
r = 3.8;
x0 = 0.8;
n = np.arange(0, 400);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y)
```

```
[79]: [<matplotlib.lines.Line2D at 0x1c6549abac0>]
```

Or rather let us define this as the y and ok. So, let me, sorry alright, this has to be np.ones.

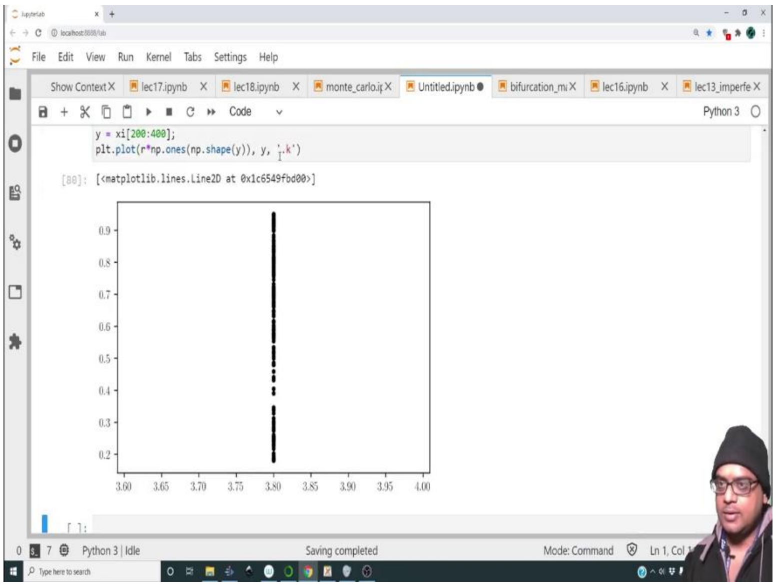
(Refer Slide Time: 50:18)



```
y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y)
```

```
[79]: [<matplotlib.lines.Line2D at 0x1c6549abac0>]
```

(Refer Slide Time: 50:21)

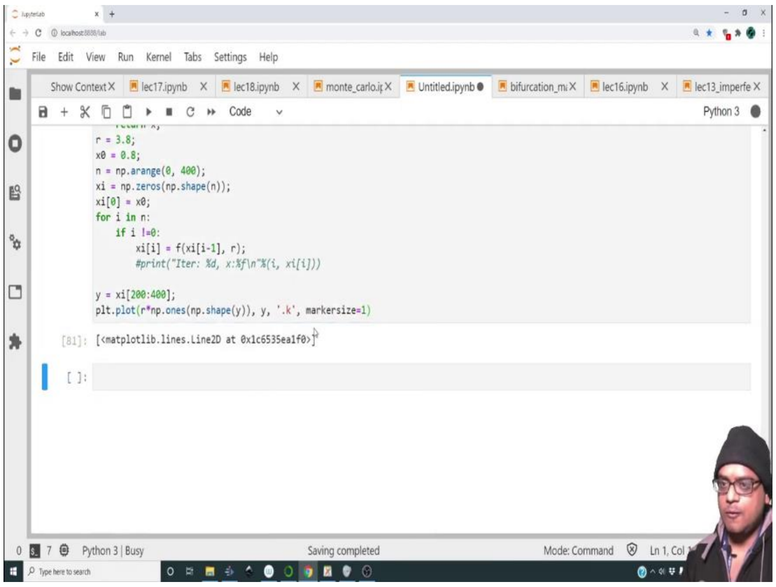


```
y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y, '.k')
```

```
[80]: [<matplotlib.lines.Line2D at 0x1c6549fbd00>]
```

So, we have a bunch of, we need to actually only plot the points, we do not need to connect them by a line. Again, it looks something like this because it is sort of iterating and get getting across all these points one after the other, there is no sequence inherent to this.

(Refer Slide Time: 50:44)

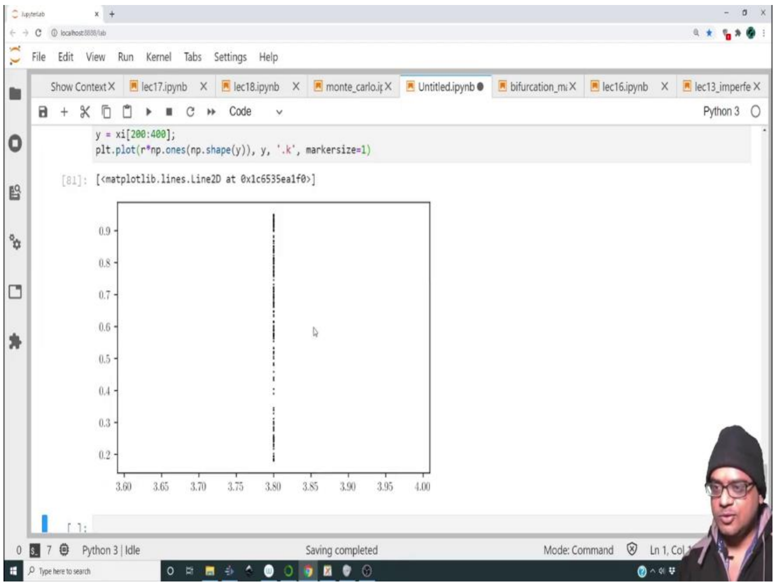


```
r = 3.8;
x0 = 0.8;
n = np.arange(0, 400);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

```
[81]: [<matplotlib.lines.Line2D at 0x1c6535ea1f0>]
```

So, now, what we need to do is to sweep over the values of r, we need to sweep over all the values of r and let me change the markersize, let me make it 1.

(Refer Slide Time: 50:49)

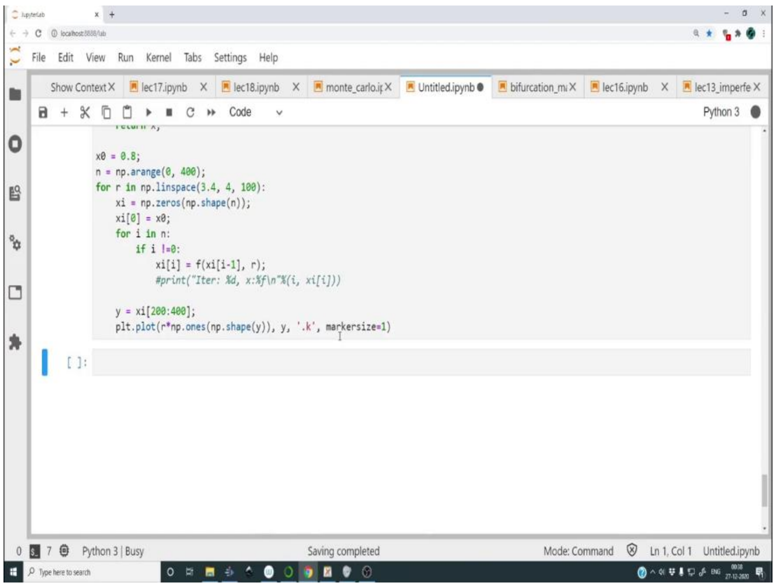


```
y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

```
[81]: [<matplotlib.lines.Line2D at 0x1c6535ea1f0>]
```

It looks much better; it looks much more polished.

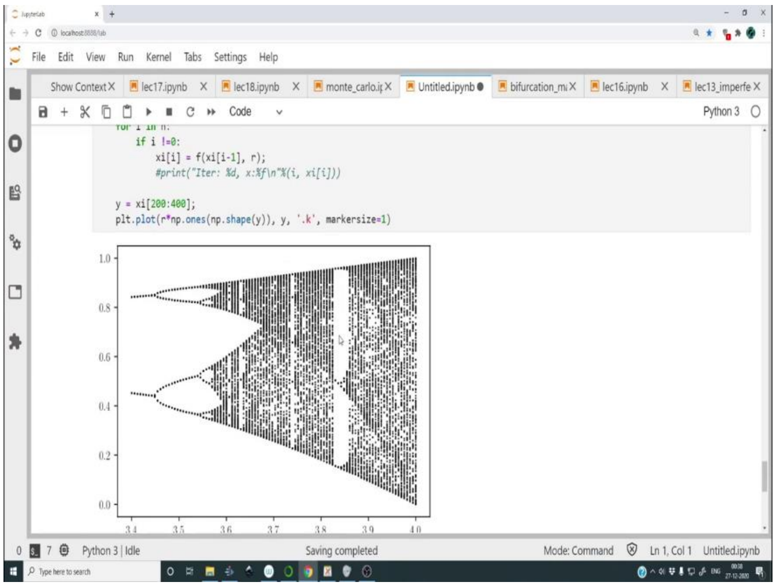
(Refer Slide Time: 51:20)



```
x0 = 0.8;
n = np.arange(0, 400);
for r in np.linspace(3.4, 4, 100):
    xi = np.zeros(np.shape(n));
    xi[0] = x0;
    for i in n:
        if i !=0:
            xi[i] = f(xi[i-1], r);
            #print("Iter: %d, x:%f\n"%(i, xi[i]))

    y = xi[200:400];
    plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

So, now let me remove this and let me perform a loop over all the values of r ok. So, this does not change, we initialize it for each set n does not change alright. So, for r in np.linspace let me just check the parameter value 3.4 to 4, let me take a 100 points. So, it will do all this plot and it may take a bit of time. So, let me run this ok.

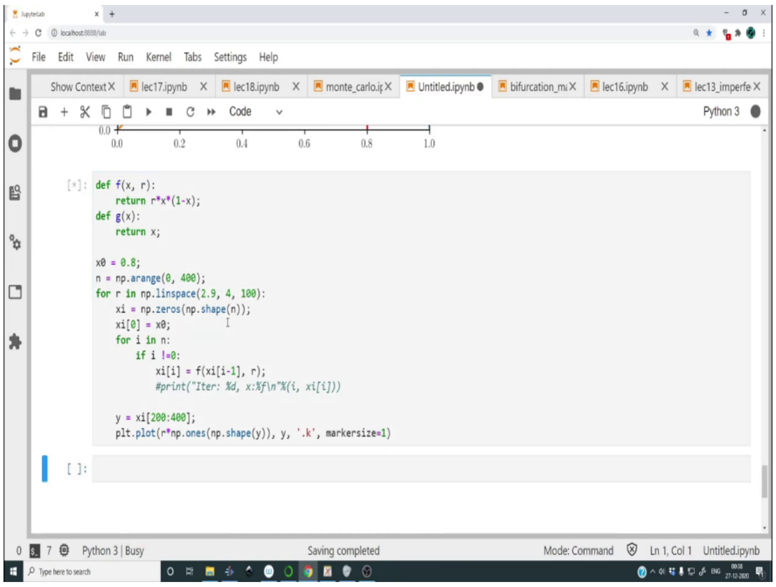
(Refer Slide Time: 51:55)



```
        for i in n:
            if i !=0:
                xi[i] = f(xi[i-1], r);
                #print("Iter: %d, x:%f\n"%(i, xi[i]))

        y = xi[200:400];
        plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

We get this wonderful looking diagram and what we do see is ok.

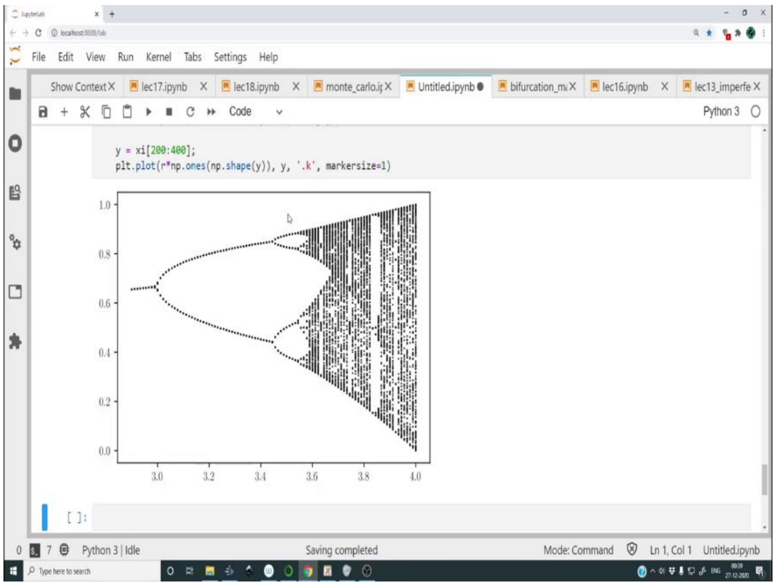
(Refer Slide Time: 52:09)



```
[*]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;

     x0 = 0.8;
     n = np.arange(0, 400);
     for r in np.linspace(2.9, 4, 100):
         xi = np.zeros(np.shape(n));
         xi[0] = x0;
         for i in n:
             if i !=0:
                 xi[i] = f(xi[i-1], r);
                 #print("Iter: %d, x:%f\n"%(i, xi[i]))

         y = xi[200:400];
         plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)

[ ]:
```

Let me do it all the way from 2.9 because we want to take r less than 3 as well because when r is less than 3, we will have only one iterate. So, let me run this again.

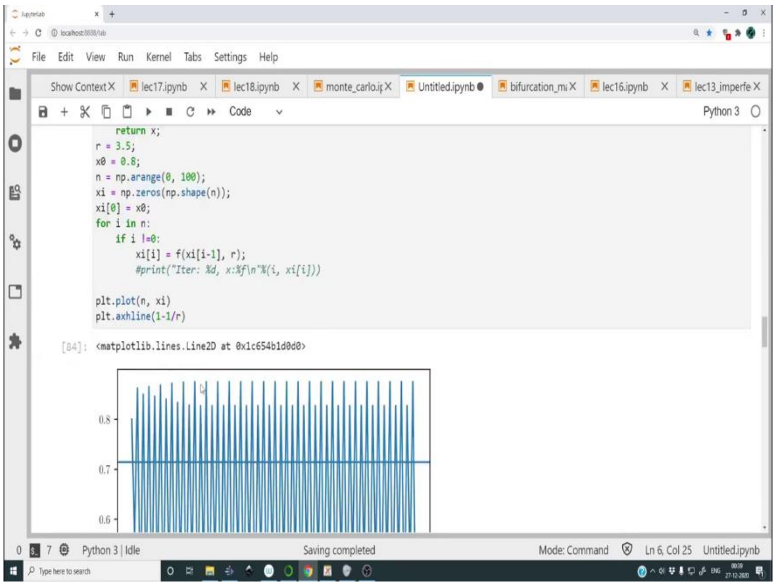
(Refer Slide Time: 52:17)



```
y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

So, we have one iterate like this and it splits into two iterates, it remains like this until 3.4 something and it splits again into 4 cycles because see at this transect, in this particular transect, we have 1, 2, 3, 4. So, you have the iterations over a large span sort of going over these four points again and again ok.

(Refer Slide Time: 52:52)

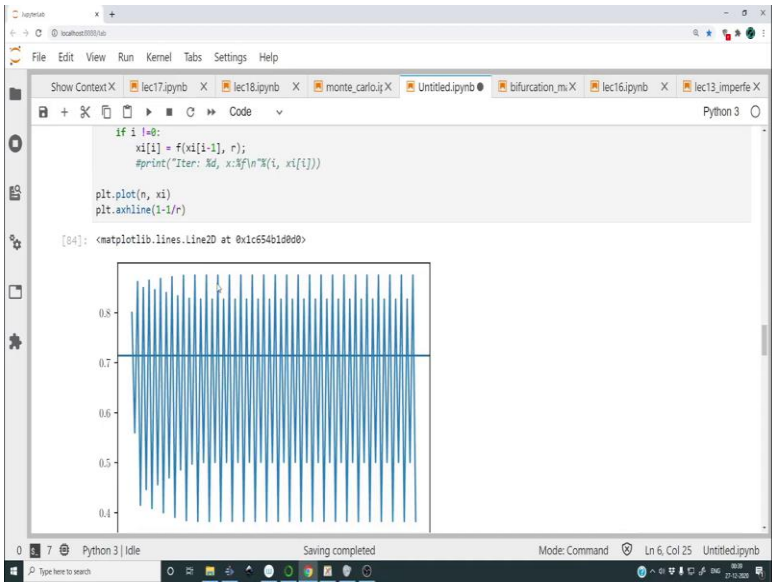


```
    return x;
r = 3.5;
x0 = 0.8;
n = np.arange(0, 100);
xi = np.zeros(np.shape(n));
xi[0] = x0;
for i in n:
    if i !=0:
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[84]: <matplotlib.lines.Line2D at 0x1c654b1d0d0>
```

So, let us see, let us choose 3.5 just to show what those four points are.

(Refer Slide Time: 52:53)

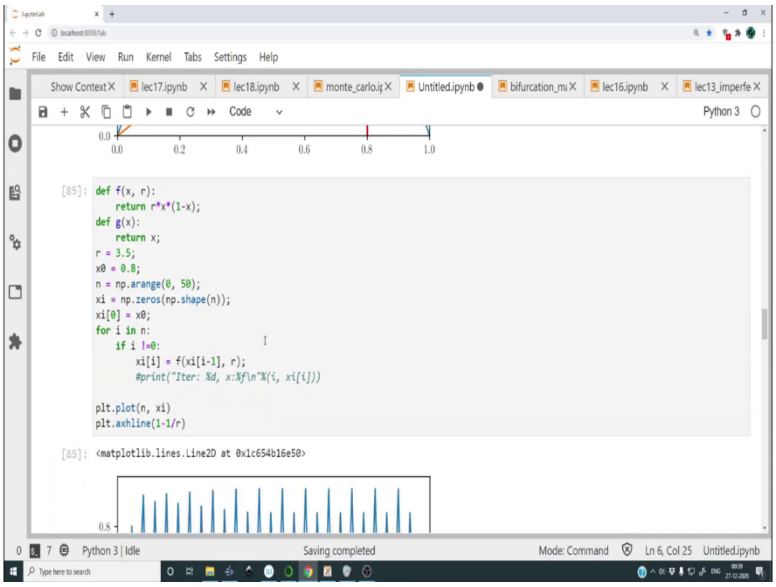


```
        if i !=0:
            xi[i] = f(xi[i-1], r);
            #print("Iter: %d, x:%f\n"%(i, xi[i]))

    plt.plot(n, xi)
    plt.axhline(1-1/r)
```

```
[84]: <matplotlib.lines.Line2D at 0x1c654b1d0d0>
```

So, it is this point.

(Refer Slide Time: 53:01)

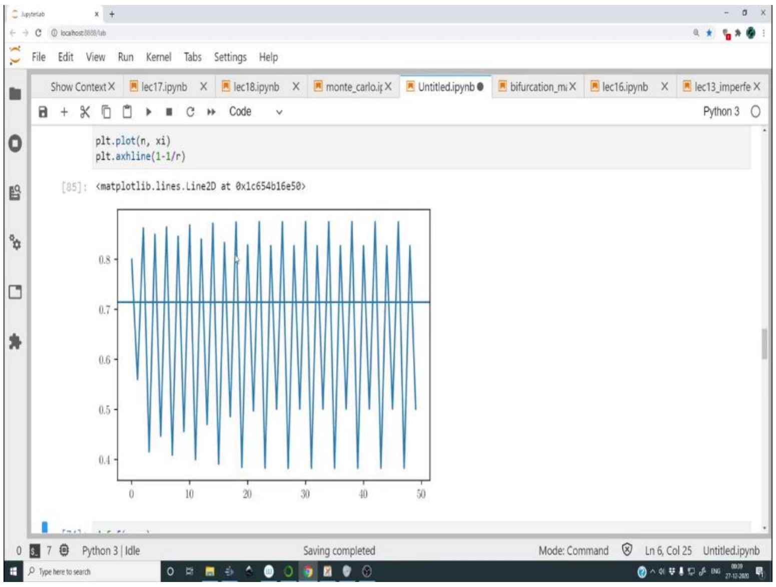


```
[85]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;
      r = 3.5;
      x0 = 0.8;
      n = np.arange(0, 50);
      xi = np.zeros(np.shape(n));
      xi[0] = x0;
      for i in n:
          if i !=0:
              xi[i] = f(xi[i-1], r);
              #print("Iter: %d, x:%f\n"%(i, xi[i]))

      plt.plot(n, xi)
      plt.axhline(1-1/r)
```

```
[85]: <matplotlib.lines.Line2D at 0x1c654b16e50>
```

In fact, let me choose only 50 ok.

(Refer Slide Time: 53:03)

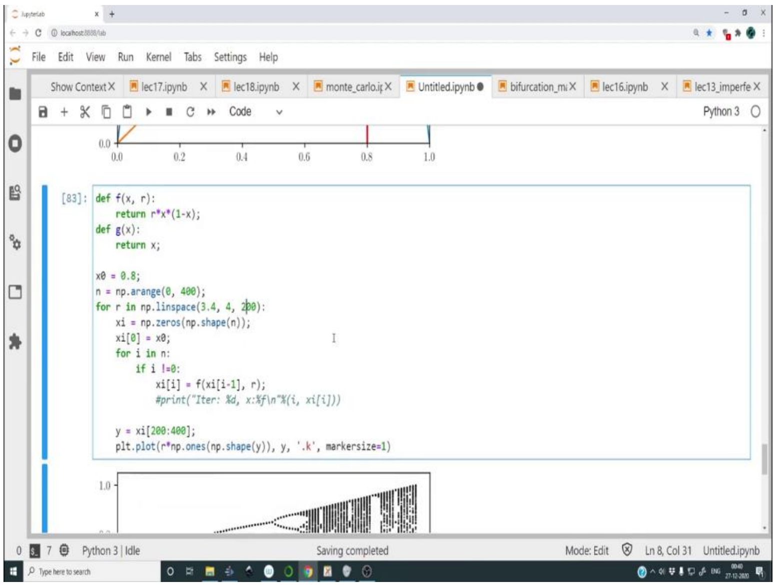


```
plt.plot(n, xi)
plt.axhline(1-1/r)
```

```
[85]: <matplotlib.lines.Line2D at 0x1c654b16e50>
```

So, it is this point, then this point, then this point, then this point so, it is four distinct points over each cycle over which this set of iterates keep on happening and that is reflected over in this particular plot.

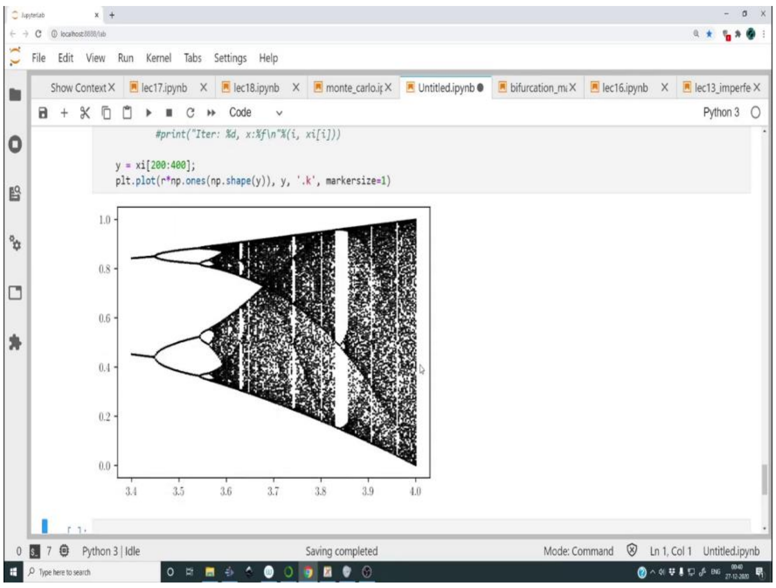
(Refer Slide Time: 53:29)



```
[83]: def f(x, r):
          return r*x*(1-x);
      def g(x):
          return x;

      x0 = 0.8;
      n = np.arange(0, 400);
      for r in np.linspace(3.4, 4, 200):
          xi = np.zeros(np.shape(n));
          xi[0] = x0;
          for i in n:
              if i !=0:
                  xi[i] = f(xi[i-1], r);
                  #print("Iter: %d, x:%f\n"%(i, xi[i]))

          y = xi[200:400];
          plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

Let us now do this plot but let me choose a denser sort of let me do it from 3.4; let me do it from 3.4 to 4 and let me take 200 points. So, this may take a while.

(Refer Slide Time: 53:39)



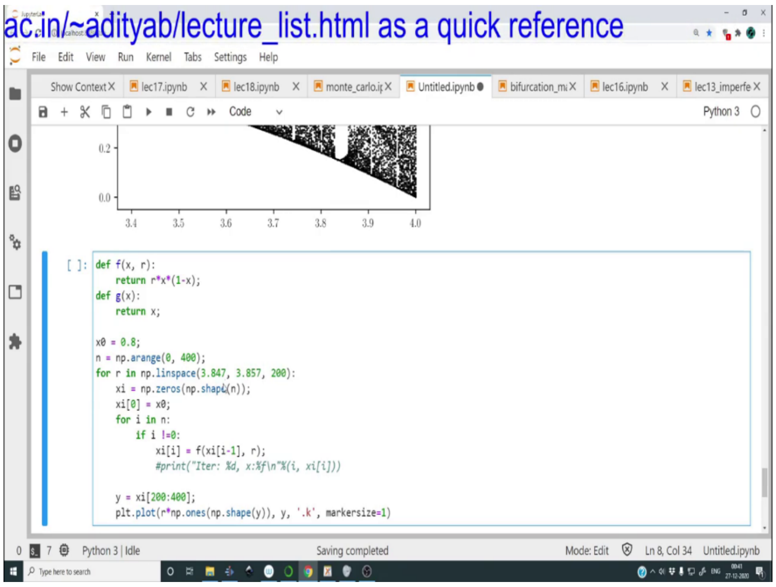
```
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

y = xi[200:400];
plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

So, this particular diagram is called as the bifurcation map. There are various interesting things to be noted over here. First is there is a period doubling bifurcation that goes on and then, it descends into chaos, but in between those doubling cascades, there are windows where it does not really sort of double anymore, it goes back to a period three in fact.

There is only 1, 2, 3 and there is a large window where there are no iterates, it is iterating over this branch, this branch and this branch and there seems to be a sort of they seem to be these slits where suddenly the bifurcation does not happen anymore and goes through a; goes to a period three cycle where.

There are lot of odd things that happen and I will give you some links where you can explore all this with the help of some of these tools that we have developed, some of the programs that we have developed. Let me in fact, make another diagram, but over different range, I want to show you something interesting.

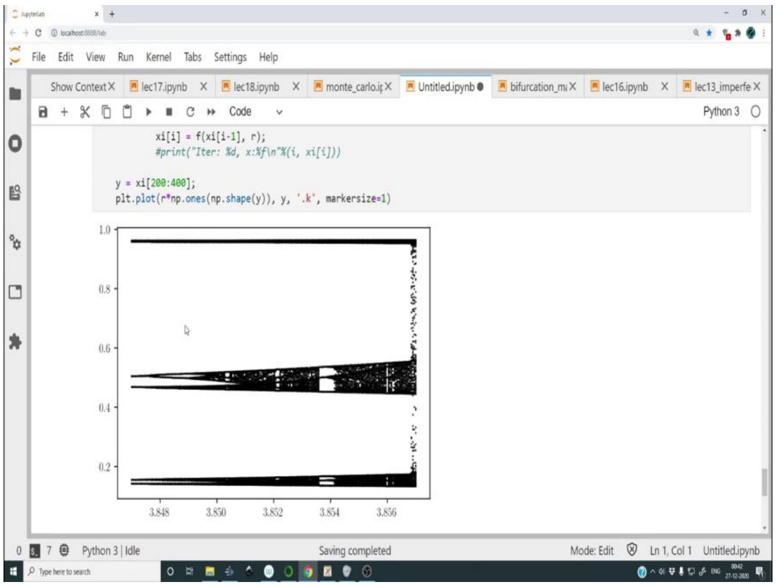
(Refer Slide Time: 54:49)


ac.in/~adityab/lecture_list.html as a quick reference

```
[ ]: def f(x, r):
         return r*x*(1-x);
     def g(x):
         return x;

     x0 = 0.8;
     n = np.arange(0, 400);
     for r in np.linspace(3.847, 3.857, 200):
         xi = np.zeros(np.shape(n));
         xi[0] = x0;
         for i in n:
             if i !=0:
                 xi[i] = f(xi[i-1], r);
                 #print("Iter: %d, x:%f\n"%(i, xi[i]))

         y = xi[200:400];
         plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

So, let me choose it between 3.847 and 3.857. Let me run this.

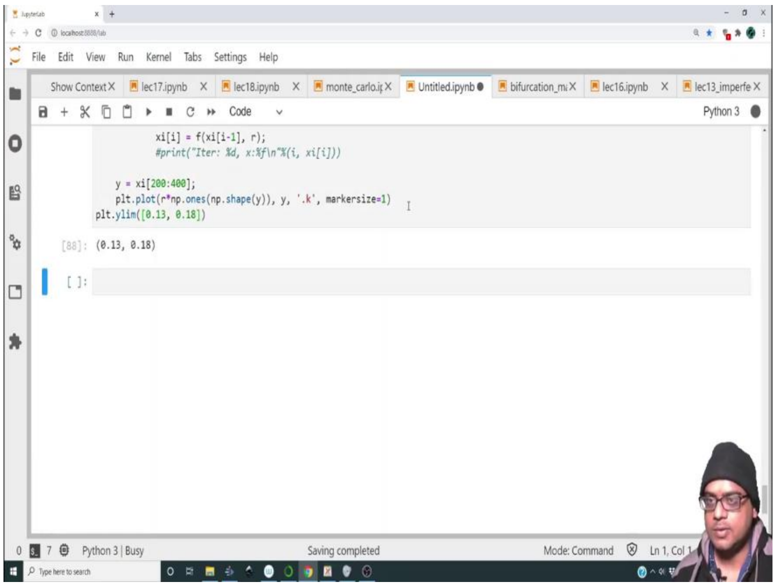
(Refer Slide Time: 55:12)



```
            xi[i] = f(xi[i-1], r);
            #print("Iter: %d, x:%f\n"%(i, xi[i]))

    y = xi[200:400];
    plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
```

So, essentially by dropping the initial ones, we are sort of plotting the long-time dynamics. It was we did something like this for the Poincare map as well.

(Refer Slide Time: 55:32)

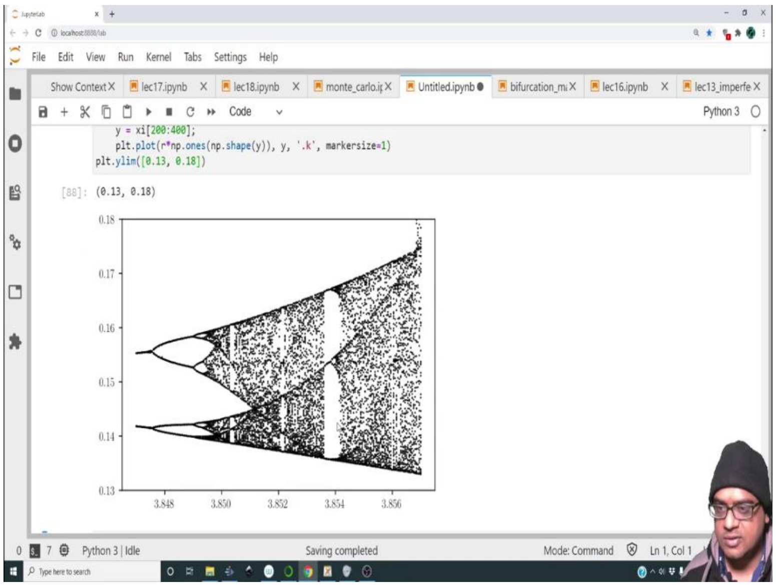


```
        xi[i] = f(xi[i-1], r);
        #print("Iter: %d, x:%f\n"%(i, xi[i]))

    y = xi[200:400];
    plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
plt.ylim([0.13, 0.18])
```

```
[88]: (0.13, 0.18)
```

And let me show only this lower branch. So, let me just change the plt.ylim going from 0.13 to 0.18. So, we run it again.

(Refer Slide Time: 55:46)



```
    y = xi[200:400];
    plt.plot(r*np.ones(np.shape(y)), y, '.k', markersize=1)
plt.ylim([0.13, 0.18])
```

```
[88]: (0.13, 0.18)
```

It runs rather fast on my PC, but ok. So, this is the diagram that we have and so, this diagram that we have plotted for a smaller window on the previous curve, it does appear to be the same kind of diagram that happens over here. So, there is a certain cell similarity to this entire bifurcation diagram.

Again once again we see 2, then 4, then 8 descending into chaos and windows where nothing really goes on, 3 cycles appearing large gaps where there are no iterates and so on.

So, it is quite I mean unusual and you would never have predicted that such a simple map such a logistic map could give such a rich dynamics ok.

So, you can play around with the parameters, you can see what really happens, you can zoom into the different zones by creating your own the small segment where you want to make the plot to zoom into the appropriate location and you will keep on seeing this kind of stuff happening ok. So, this is called as the bifurcation for the bifurcation diagram for the logistic map.

And the such kinds of things do appear for a lot of; for a lot of physical processes as well such as transition to turbulence and so on even synchronization of big swarms of things. So, it is not just confined to the analysis of a logistic map, but it can extend to a lot of areas of physics and mathematics and I will put some links in the description as well, you can have a look and feel free to explore this and this gives you the platform to perform numerical experiments.

So, all these things you cannot really go on field and take measurements, but in many cases, such kinds of measurements have shown this kind of a bifurcation pattern as well. The most famous experiment is the leaking faucet experiment, and we will see in the last week whether we can achieve, we can record those data's and we can construct this diagram also.

So, with this, I request you to play around, have fun, explore the depths of various maps and some assignment problems you can really attack them with all your might and with this, I conclude this particular lecture and I will see you next time with random numbers, bye.