Practical Kinetics

Exercise 1:

Introduction to Data Analysis in Python

Objectives:

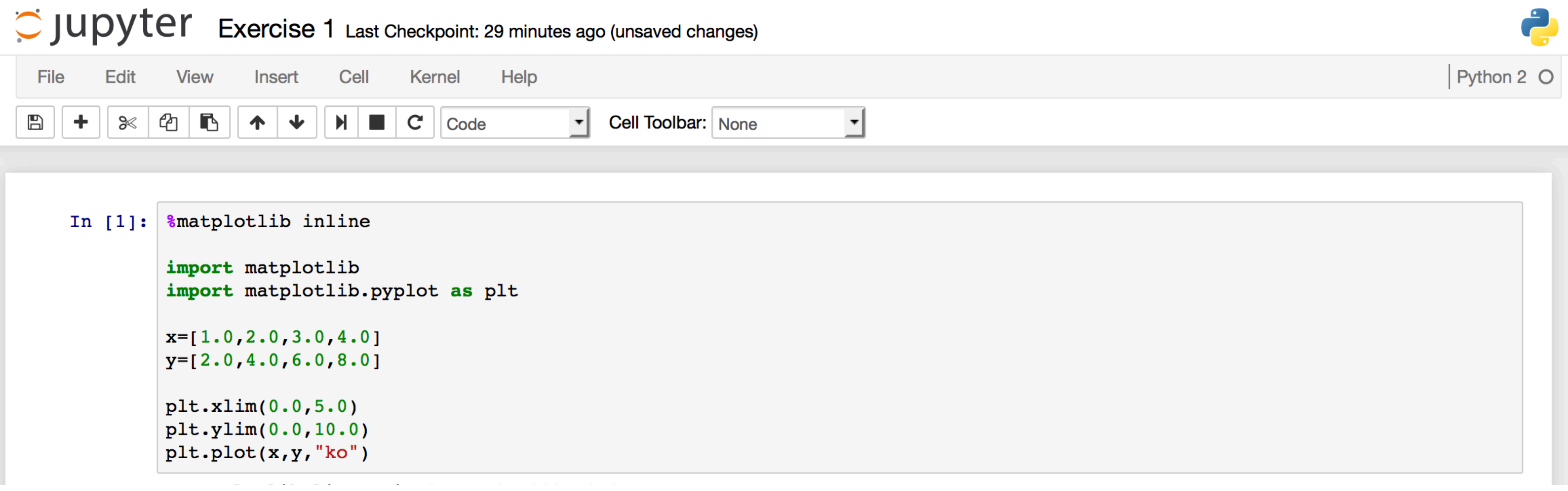
1. Make some simple graphs and perform some simple calculations with lists
2. Simulate a first-order kinetic system
3. Determine when the pre-equilibrium and steady state approximations are appropriate

Step 1: Plot a Straight Line

We’ll start by making a very simple graph. Type the following into the first cell:

```
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt

x=[1.0,2.0,3.0,4.0]
y=[2.0,4.0,6.0,8.0]

plt.xlim(0.0,5.0)
plt.ylim(0.0,10.0)
plt.plot(x,y,"ko")
```

Your window should look like this:

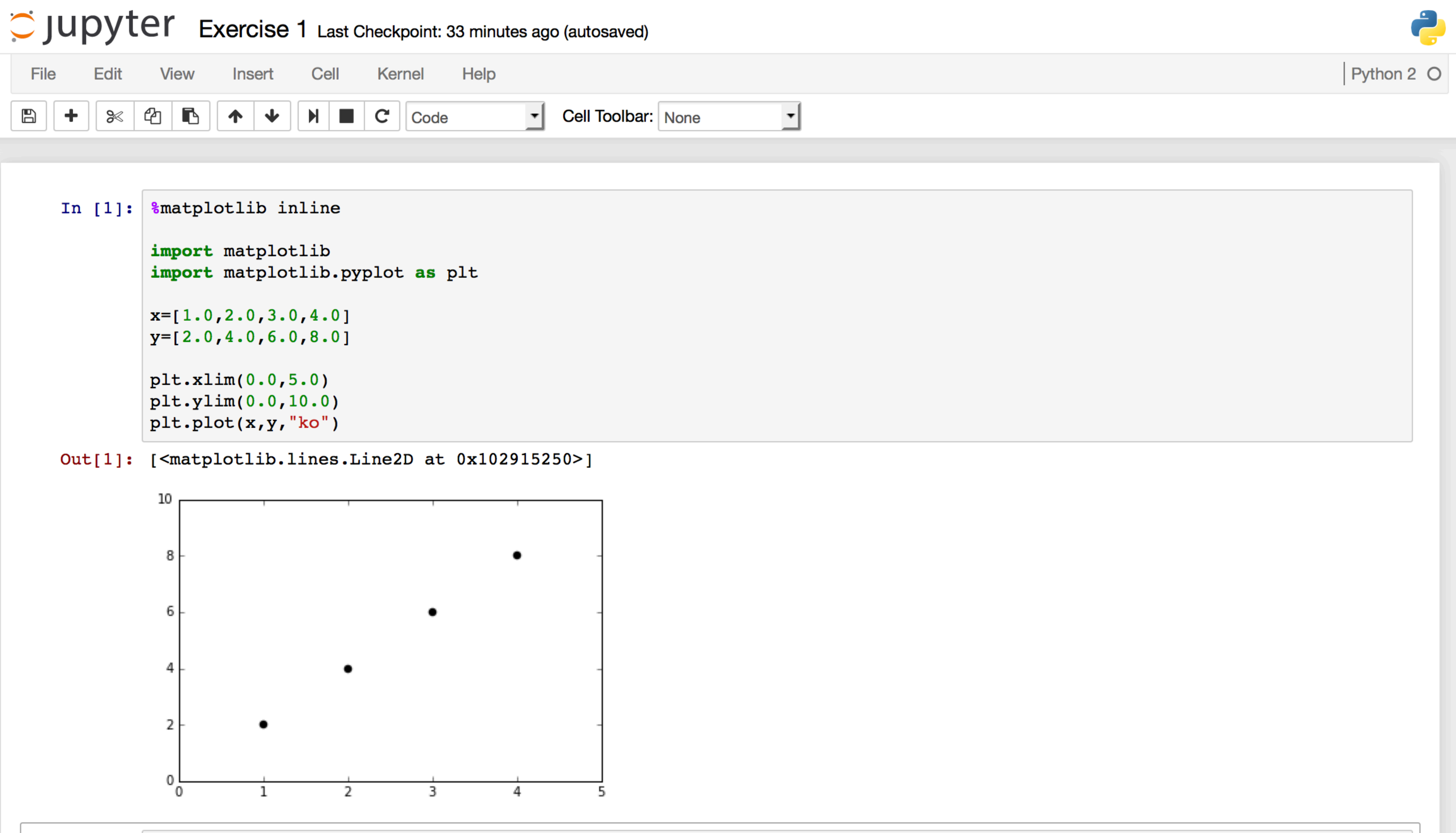


```
In [1]: %matplotlib inline

        import matplotlib
        import matplotlib.pyplot as plt

        x=[1.0,2.0,3.0,4.0]
        y=[2.0,4.0,6.0,8.0]

        plt.xlim(0.0,5.0)
        plt.ylim(0.0,10.0)
        plt.plot(x,y,"ko")
```

Step 1: Plot a Straight Line

Notice that pressing **enter** goes to the next line and does not execute anything.

To execute the code, press **shift-enter**. A plot will appear underneath:



```
In [1]: %matplotlib inline

        import matplotlib
        import matplotlib.pyplot as plt

        x=[1.0,2.0,3.0,4.0]
        y=[2.0,4.0,6.0,8.0]

        plt.xlim(0.0,5.0)
        plt.ylim(0.0,10.0)
        plt.plot(x,y,"ko")

Out[1]: [<matplotlib.lines.Line2D at 0x102915250>]
```

Step 1: Plot a Straight Line

```
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt

x=[1.0,2.0,3.0,4.0]
y=[2.0,4.0,6.0,8.0]

plt.xlim(0.0,5.0)
plt.ylim(0.0,10.0)
plt.plot(x,y,"ko")
```

These lines tell Python you want to make some plots in your browser window.

These are lists of numbers.

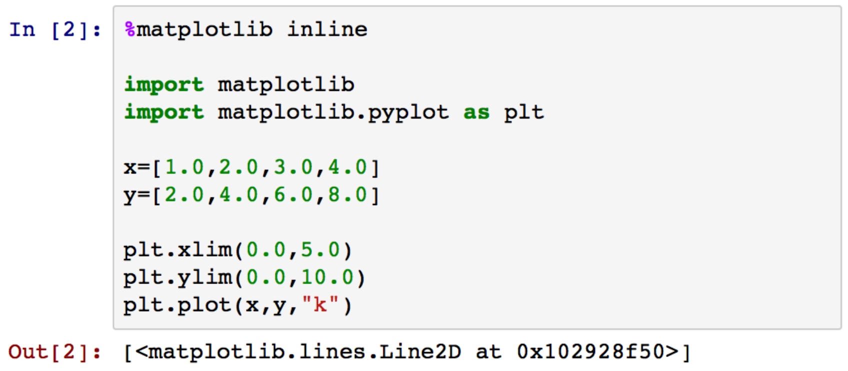
These lines set the x and y axis limits.

The last line plots y vs. x using black**k** circles (**o**).

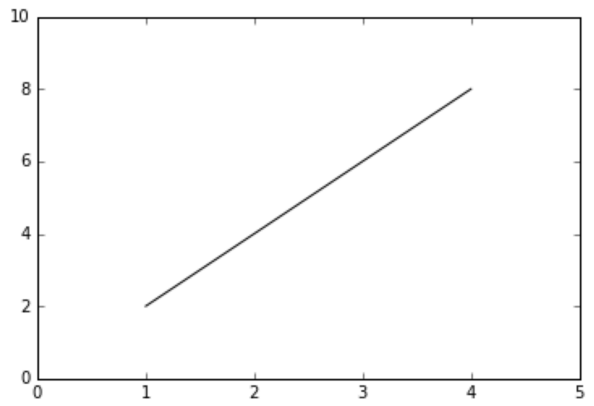
In a new cell, delete the "o" and press **shift-enter** again to re-evaluate the cell. (That's a lowercase o, as in "oak.")

Step 2: Formatting Plots

Without the “o,” just the line is plotted:



```
In [2]: %matplotlib inline

        import matplotlib
        import matplotlib.pyplot as plt

        x=[1.0,2.0,3.0,4.0]
        y=[2.0,4.0,6.0,8.0]

        plt.xlim(0.0,5.0)
        plt.ylim(0.0,10.0)
        plt.plot(x,y,"k")

Out[2]: [<matplotlib.lines.Line2D at 0x102928f50>]
```



You can plot both the line and the points if you wish. Add the original line underneath:

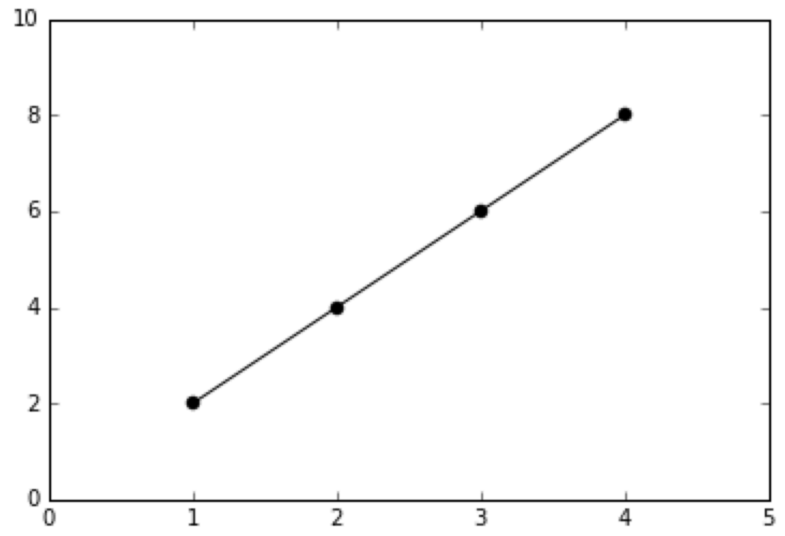
```
plt.plot(x,y,"ko")
plt.plot(x,y,"k")
```

Step 3: Formatting Plots

When multiple plot statements are present, the plots are automatically overlaid:

the format strings

```
plt.plot(x,y,"ko")
plt.plot(x,y,"k")
```



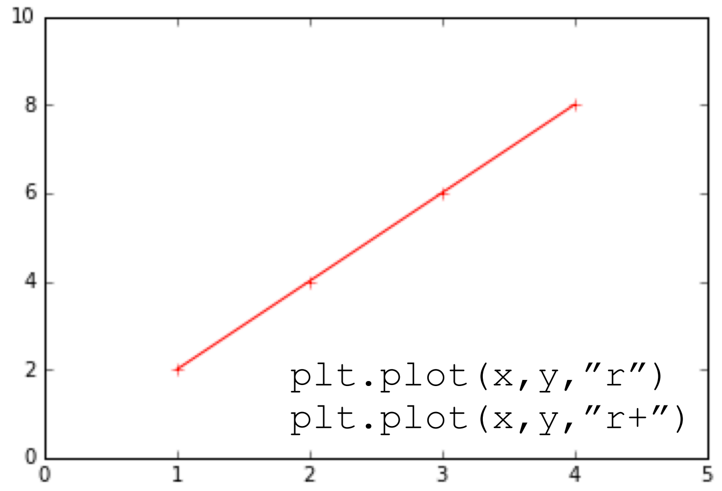
To customize further, create a format string by combining a color abbreviation with a line style abbreviation:

colors: r = red, b = blue, g = green, k = black

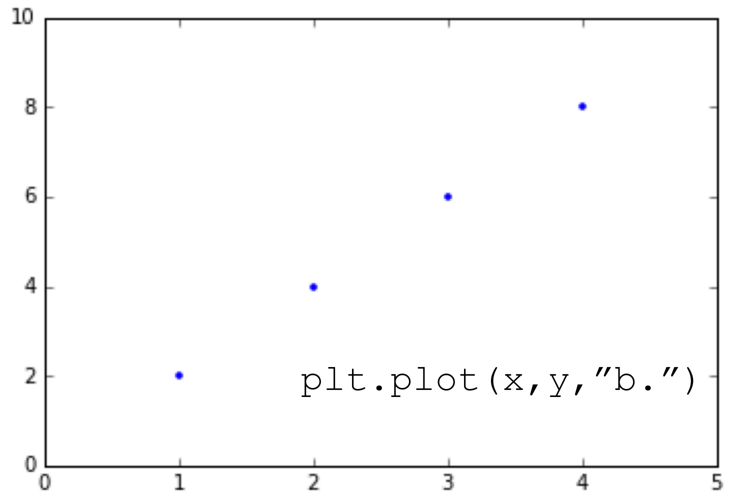
line styles: nothing or - = connect points with straight, solid lines
-- = connect lines with straight, dashed lines
. = mark points with dots
o = mark points with circles
+ = mark points with crosses

Step 3: Formatting Plots

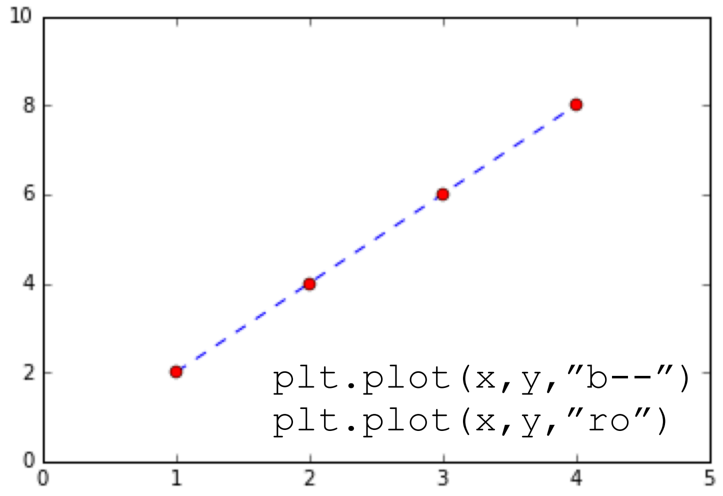
Here are a few examples:



```
plt.plot(x,y,"r")
plt.plot(x,y,"r+")
```



```
plt.plot(x,y,"b.")
```



```
plt.plot(x,y,"b--")
plt.plot(x,y,"ro")
```

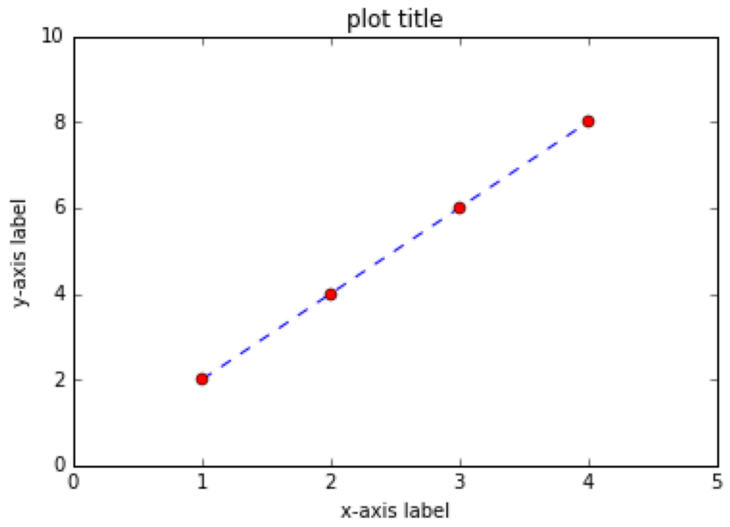
Notice that the color goes first and the line style goes second.

Step 3: Labeling Plots

Try adding the following:

```
plt.xlabel("x-axis label")
plt.ylabel("y-axis label")
plt.title("plot title")
```

Note: Further customization of fonts, legends, line styles, etc. is possible. See: http://matplotlib.org/users/pyplot_tutorial.html



Saving plots: If you want to save your plot to disk, add a line:

```
plt.savefig("my_plot.png")
```

This will save a graphic to the directory you started IPython Notebook in.

You can replace `.pdg` with `.pdf` to make PDF files.

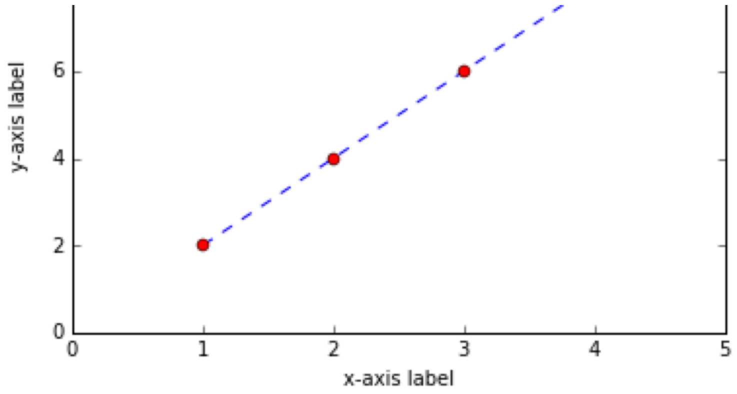
Step 4: Introduction to Lists

In the previous code, we had:

```
x=[1.0,2.0,3.0,4.0]
y=[2.0,4.0,6.0,8.0]
```

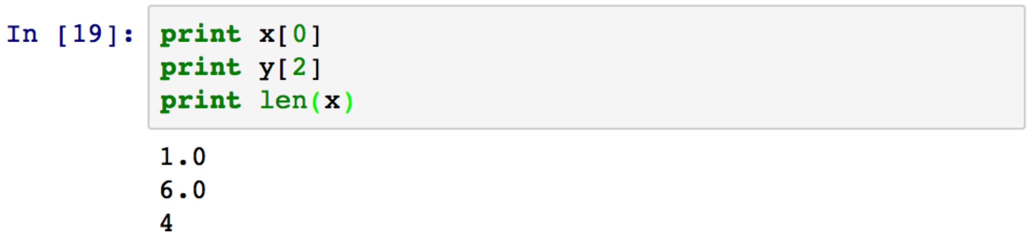
This makes two **lists**, one called "x" and another called "y." You can access lists "by element." *In a new cell*, try typing:

```
print x[0]
print y[2]
print len(x)
```



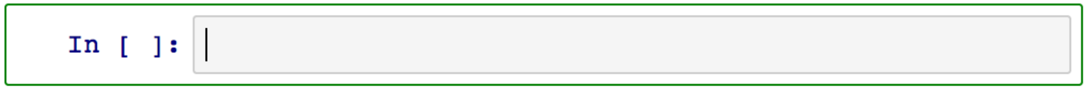
Press **shift-enter** to evaluate again.

Notice that Python "remembers" x and y, even though they were defined above.



```
In [19]: print x[0]
         print y[2]
         print len(x)

1.0
6.0
4
```

A new cell appears as well:



```
In [ ]: |
```

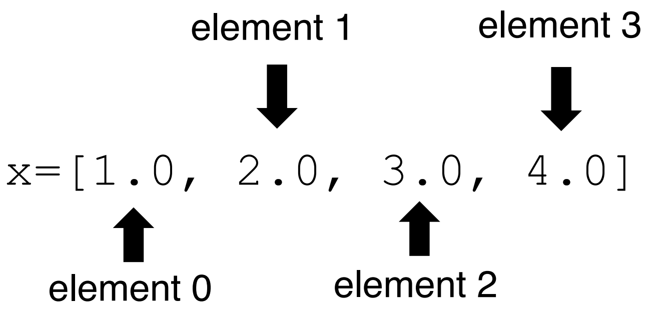
Step 4: Introduction to Lists

In the previous code, we wrote:

```
x=[1.0,2.0,3.0,4.0]
y=[2.0,4.0,6.0,8.0]
```

These are called **lists**. We'll deal with lists that contain numbers, but they can contain other things too.

Each *element* has an *index:*



```
x=[1.0, 2.0, 3.0, 4.0]
```

Notice that the numbering starts at 0, not 1.

You can access each element directly:

```
print x[0]
   1.0
```

Step 4: Introduction to Lists

The symbol `x` is the name of the list:

```
x=[1.0, 2.0, 3.0, 4.0]
```

⬆
name of list

To <u>change an element</u> of `x`:

```
x[0]=8.0                    result:
print x                     [1.0, 2.0, 3.0, 4.0]
```

To <u>append</u> to `x`:

```
x.append(5.0)               result:
print x                     [8.0, 2.0, 3.0, 4.0, 5.0]
```

To find out <u>how many elements</u> `x` contains:

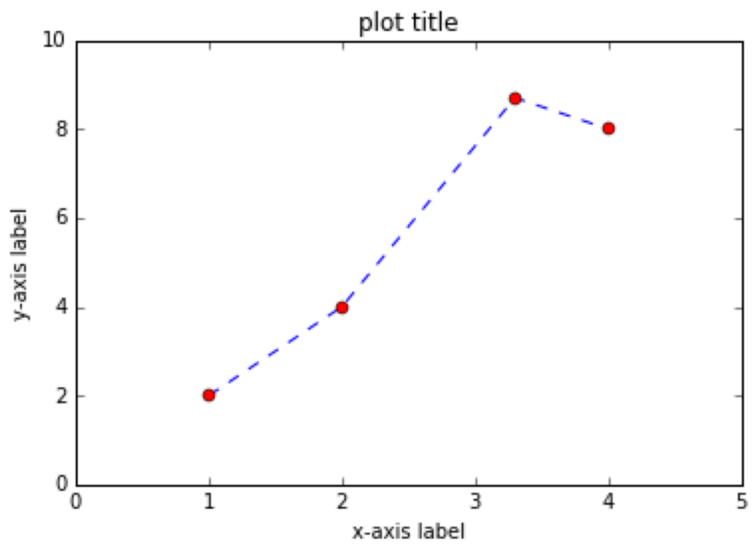
```
                            result:
print len(x)                5
```

Step 4: Introduction to Lists

In a new cell:

Change the **third** (index = 2) point to (3.3, 8.7). Re-plot the result.

```
In [8]: x[2]=3.3
        y[2]=8.7
        plt.xlim(0.0,5.0)
        plt.ylim(0.0,10.0)
        plt.plot(x,y,"b--")
        plt.plot(x,y,"ro")

        plt.xlabel("x-axis label")
        plt.ylabel("y-axis label")
        plt.title("plot title")

Out[8]: <matplotlib.text.Text at 0x10a0d1c10>
```



Notice that the previous plots (not shown here) remain unchanged. However, the lists `x` and `y` have now changed in memory. Plotting them again will reflect the changes.

Step 5: List Comprehensions

Instead of entering numbers into lists manually, we can use a mathematical expression.

The simplest is to use the `range` command:

```
list1 = range(5)            Result:
print list1                 [0, 1, 2, 3, 4]
print len(list1)            5
```

Typing in `range(n)` returns the numbers from 0 to n, exclusive, as a list.

```
range(n) = [0, 1, ..., n-1 ]
```

Step 5: List Comprehensions

We can use range to generate more complicated lists:

```
list1 = range(5)
list1 = [0, 1, 2, 3, 4]

list2 = [ i*2 for i in list1 ]          Result:
print list2                             [0, 2, 4, 6, 8]
```

This is called a **list comprehension**. In English, this means:

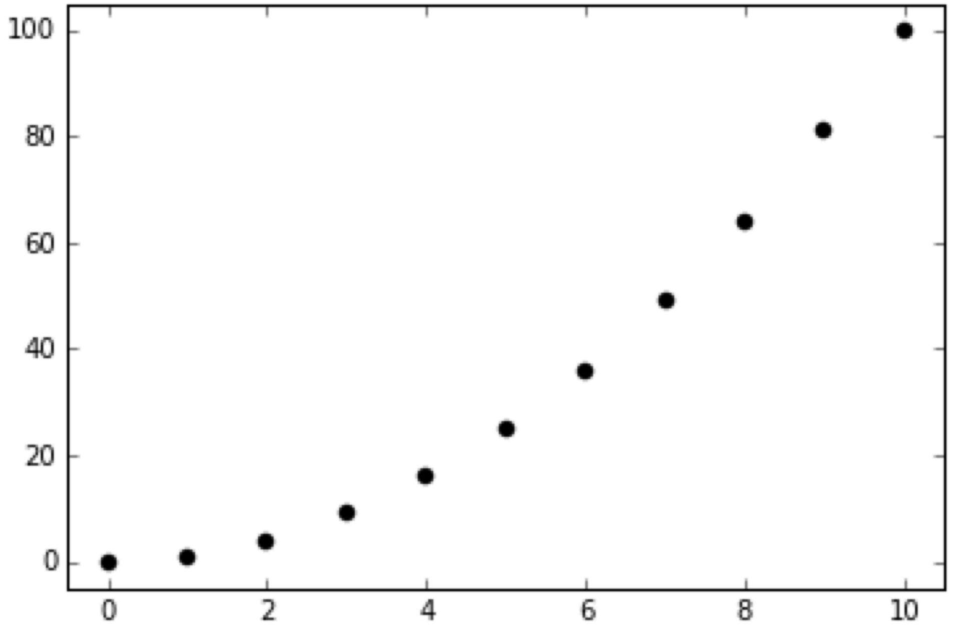
"Take each item in `list1`, one at a time, and call it `i`. Multiply `i` by 2 and put the result in `list2`."

Taking items one at a time from a list is called **iteration**.

Step 5: List Comprehensions

In a new cell, plot the function $y=x^2$ from 0 to 10 using a list comprehension.

```
In [11]: x2 = range(11)
         y2 = [ x**2 for x in x2 ]
         plt.xlim(-0.5,10.5)
         plt.ylim(-5,105)
         plt.plot(x2,y2,"ko")

Out[11]: [<matplotlib.lines.Line2D at 0x10a29e550>]
```

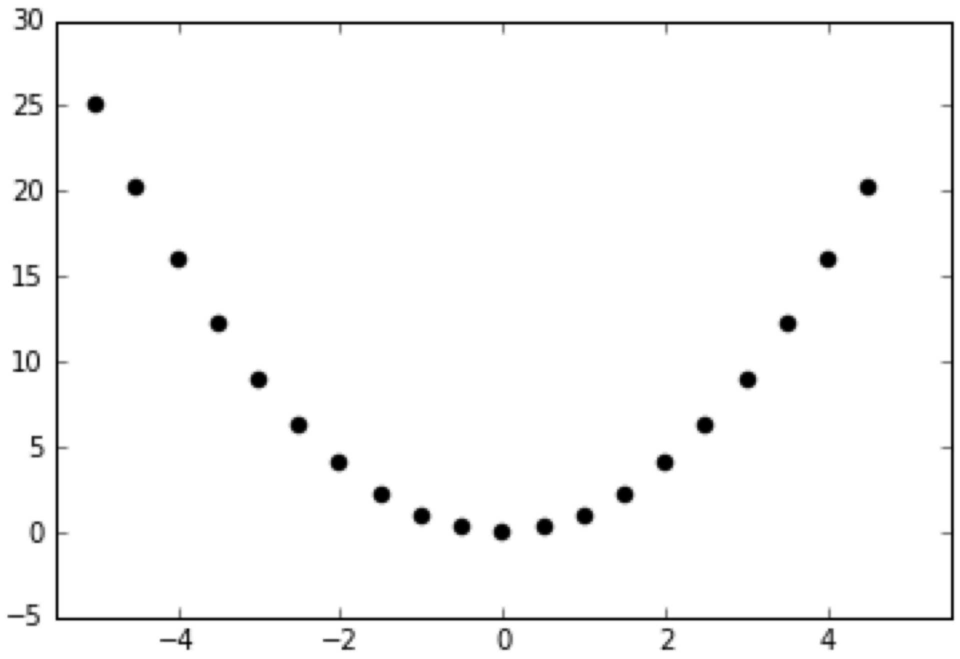


Notice that we need `range(11)`, not `range(10)`. `x**2` means x raised to the power of 2 in Python.

Step 5: List Comprehensions

What if we wanted to plot from –5 to +5, every 0.5? Instead of `range`, we need a more generalized function called `np.arange`:

```
In [17]: import numpy as np
         x3 = np.arange(-5.0,5.0,0.5)
         y3 = [ x**2 for x in x3 ]
         plt.xlim(-5.5,5.5)
         plt.ylim(-5,30)
         plt.plot(x3,y3,"ko")

Out[17]: [<matplotlib.lines.Line2D at 0x10a704750>]
```



Step 5: List Comprehensions

```
import numpy as np
```

This loads the NumPy library. A “library” is a collection of code that has been written by someone else to perform common tasks.

To refer to a NumPy function instead of a regular Python function, we need to preface the function’s name with `np`.

`sqrt` (regular Python)
`np.sqrt` (numpy)

```
np.arange(-5.0, 5.0, 0.5)
```

“Fill a list with the numbers from –5 to +5 every 0.5.” Just like with the `range` function, the last point is not included.

The general syntax is:

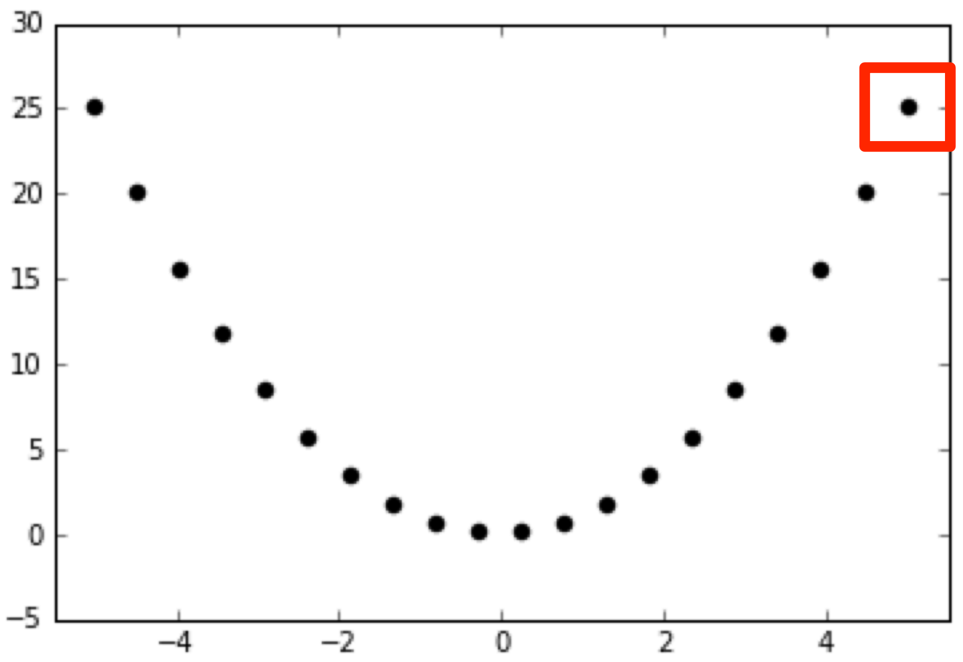
```
np.arange(start, stop, stepsize)
```

Step 5: List Comprehensions

If you wanted to fill the range from –5 to +5 evenly with 20 points, you can use `np.linspace`:

```
In [18]: x4 = np.linspace(-5.0,5.0,20)
         y4 = [ x**2 for x in x4 ]
         plt.xlim(-5.5,5.5)
         plt.ylim(-5,30)
         plt.plot(x4,y4,"ko")
```

```
Out[18]: [<matplotlib.lines.Line2D at 0x10a800390>]
```



Note that with `np.linspace`, the last point *is* included.

Step 6: First-Order Kinetics

Recall that for a first-order reaction:

$$A \xrightarrow{k_1} B$$

$$\frac{d[A]}{dt} = -k_1[A]$$

$$[A] = [A]_0 \exp\left(-k_1 t\right)$$

$$\log[A] = \log[A]_0 - k_1 t$$

To simulate the timecourse of a reaction, enter the following *in a new cell:*

```
from math import exp
import numpy as np

time = np.arange(0.0,10.0,0.1)
initial_concentration = 1.0
rate_constant = 1.0
concentration = [ initial_concentration *
exp(-rate_constant*t) for t in time ]
plt.plot(time, concentration, "k.")
```

Place this on one line.

Step 6: First-Order Kinetics

```
from math import exp
```

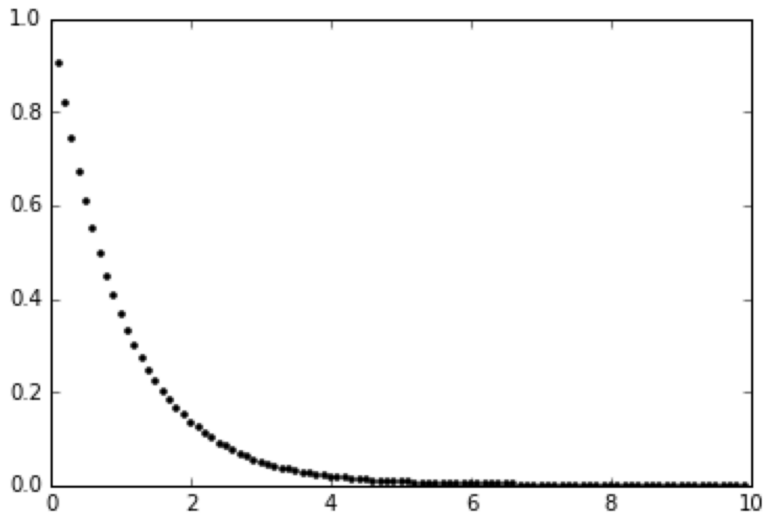
“From now on, whenever I type `exp`, interpret that to mean the `exp` function from the `math` library.”

```
In [36]: from math import exp
         import numpy as np

         time = np.arange(0.0,10.0,0.1)
         initial_concentration = 1.0
         rate_constant = 1.0
         concentration = [ initial_concentration * exp(-rate_constant*t) for t in time ]
         plt.plot(time, concentration, "k.")
Out[36]: [<matplotlib.lines.Line2D at 0x105f48b10>]
```

We need some mathematical functions.

0 to 10 seconds in 0.1 s steps

the integrated rate law as a list comprehension



Step 7: Make a Log Plot

Recall that a log plot will show a straight line:

$$A \xrightarrow{k_1} B$$

$$\frac{d[A]}{dt} = -k_1[A]$$

$$[A] = [A]_0 \exp\left(-k_1 t\right)$$

$$\log[A] = \log[A]_0 - k_1 t$$

You can perform log calculations with:

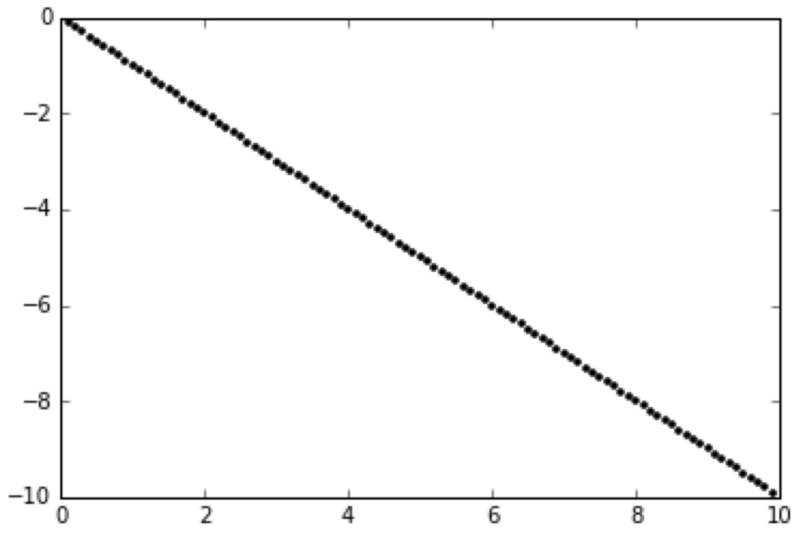
```
from math import log

log_concentration = [ log(conc) for conc in concentration ]
```

Try this out *in a new cell*.

Step 7: Make a Log Plot

This is a straight line with unit negative slope.

```
In [38]: from math import log

         log_concentration = [ log(conc) for conc in concentration ]
         plt.plot(time, log_concentration, "k.")

Out[38]: [<matplotlib.lines.Line2D at 0x1060735d0>]
```



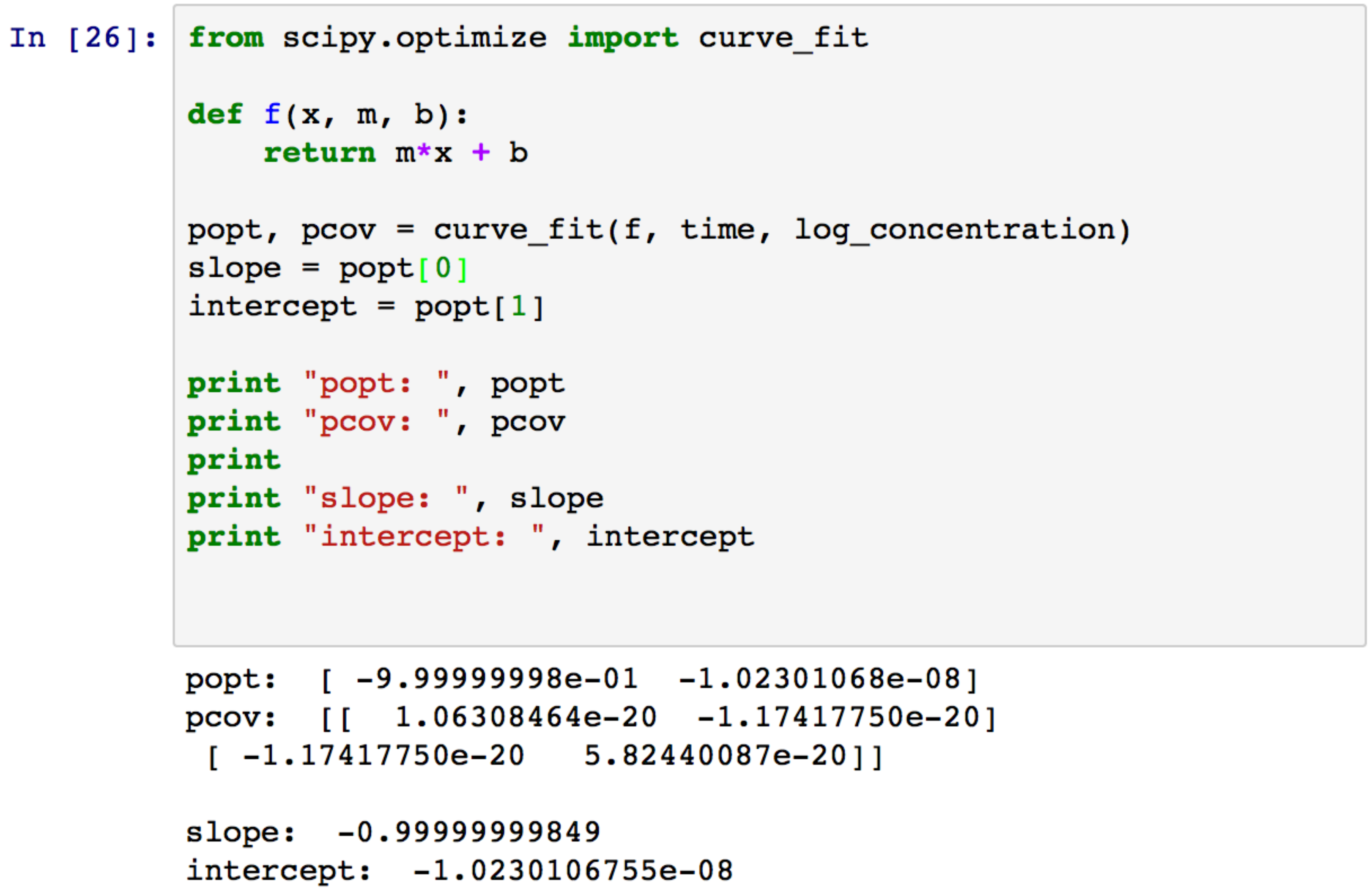
The list comprehension “iterates” over `concentration` to produce a new list:

“Take each value in the `concentration` list, one at a time, take the logarithm and add it to a new list called `log_concentration`.”

Note that `log` means the natural logarithm. You can use `log10` for base 10.

Step 8: Fit a Straight Line

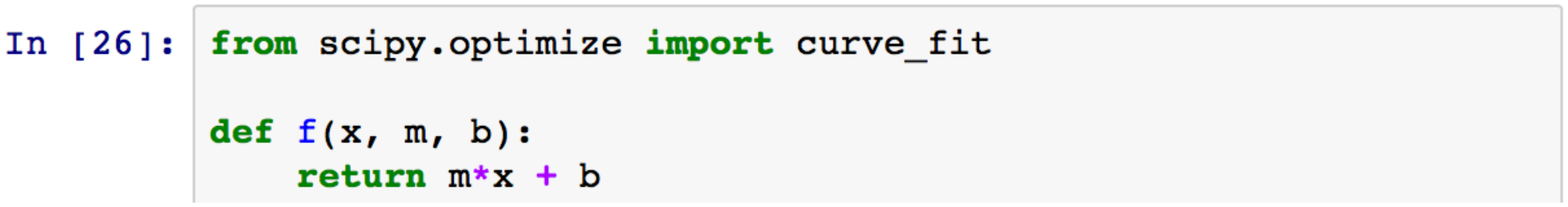
What if you wanted to fit a straight line to get the slope and intercept?



In [26]:
```
from scipy.optimize import curve_fit

def f(x, m, b):
    return m*x + b

popt, pcov = curve_fit(f, time, log_concentration)
slope = popt[0]
intercept = popt[1]

print "popt: ", popt
print "pcov: ", pcov
print
print "slope: ", slope
print "intercept: ", intercept
```

```
popt:  [ -9.99999998e-01  -1.02301068e-08]
pcov:  [[  1.06308464e-20  -1.17417750e-20]
 [ -1.17417750e-20   5.82440087e-20]]

slope:  -0.99999999849
intercept:  -1.0230106755e-08
```

Step 8: Fit a Straight Line



```
In [26]: from scipy.optimize import curve_fit

         def f(x, m, b):
             return m*x + b
```

```
from scipy.optimize import curve_fit
```

SciPy is a standard Python library that contains many routines for doing scientific computing. In this case, we are importing the `curve_fit` function from the SciPy optimization library.

```
def f(x, m, b):
    return m*x + b
```

“Define a function `f` that takes three parameters: `x`, `m`, and `b`. Return `m` x `x` + `b` whenever `f` is called.”

Step 8: Fit a Straight Line

```
def f(x, m, b):
    return m*x + b

popt, pcov = curve_fit(f, time, log_concentration)
```

`curve_fit` is a function in `scipy.optimize`.

We “call” it with three parameters:

`curve_fit(f, time, log_concentration)`

`f`, the functional form of the curve we want to fit
`time`, the x values
`log_concentration`, the y values

`curve_fit` adjusts `m` and `b` in `f` to minimize the sum of squares between `f(time, m, b)` and `log_concentration`.

Step 8: Fit a Straight Line

```
def f(x, m, b):
    return m*x + b

popt, pcov = curve_fit(f, time, log_concentration)
```

Which parameter is which?

`curve_fit` determines this by looking at the order in which the parameters are passed to it. That is, the function always comes first, the x values second, and the y values third. The actual names of the parameters are not important.

In the parameter list for the function `f(x, m, b)`, the independent variable must come first. Any subsequent parameters are adjusted to minimize the sum of squares.

Step 8: Fit a Straight Line

```
popt, pcov = curve_fit(f, time, log_concentration)
slope = popt[0]
intercept = popt[1]
```

```
popt:  [ -9.99999998e-01  -1.02301068e-08]
pcov:  [[  1.06308464e-20  -1.17417750e-20]
 [ -1.17417750e-20   5.82440087e-20]]

slope:  -0.99999999849
intercept:  -1.0230106755e-08
```

```
popt, pcov = curve_fit(f, time, log_concentration)
```

When `curve_fit` is finished, it places the results in two lists, `popt` and `pcov`.

`popt` contains the optimized values of the parameters, in the same order as defined in `f`. Thus, the slope *m* is `popt[0]` and the intercept *b* is `popt[1]`.

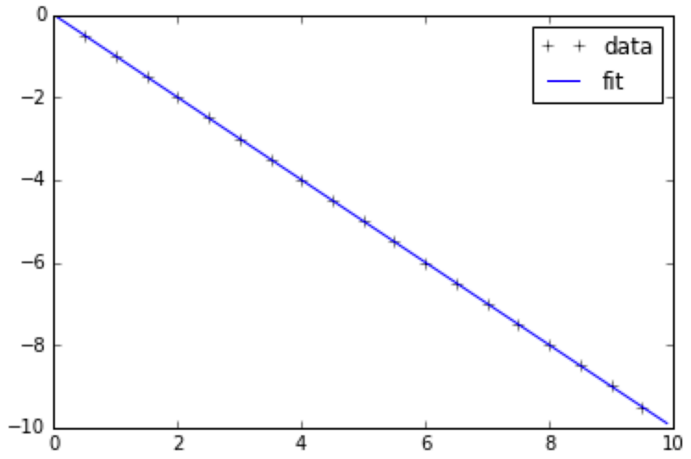
Technical Note:

`pcov` contains the (symmetric) covariance matrix for the parameter estimates. The nested square brackets mean that this is really a 2x2 matrix. The diagonal entries `pcov[0][0]` and `pcov[1][1]` represent the uncertainties of the slope and intercept, respectively. The off-diagonal entries indicate how the uncertainty in slope and intercept are related. In this case, the uncertainties are very small and meaningless because no error bars were passed to `curve_fit`.

Step 8: Fit a Straight Line

Let’s plot the result:

```
In [28]: best_fit = [ f(x, slope, intercept) for x in time ]
         plt.plot(time[::5], log_concentration[::5], "k+", label="data")
         plt.plot(time, best_fit, "b", label="fit")
         plt.legend()

Out[28]: <matplotlib.legend.Legend at 0x10c36add0>
```



The fit is perfect because this the data are synthetically generated.

The `time[::5]` notation tells Python to take every fifth point in the `time` list and create a new list. I did that so the points aren’t jammed together.

Step 9: Non-Linear Curve Fitting

There is no reason that `f` must be linear! Instead of fitting log(concentration) vs. time, let's fit concentration vs. time directly.

In a new cell, type this:

```
def f2(t, initial_concentration, k):
    return initial_concentration*np.exp(-k*t)

popt, pcov = curve_fit(f2, time, concentration)
fitted_initial_concentration = popt[0]
fitted_k = popt[1]
print "fitted initial concentration:", fitted_initial_concentration
print "fitted rate constant:", fitted_k
best_fit2 = [ f2(t, fitted_initial_concentration, fitted_k) for t in time ]
plt.plot(time[::5], concentration[::5], "k+", label="data")
plt.plot(time, best_fit2, "b", label="fit")
plt.legend()
```

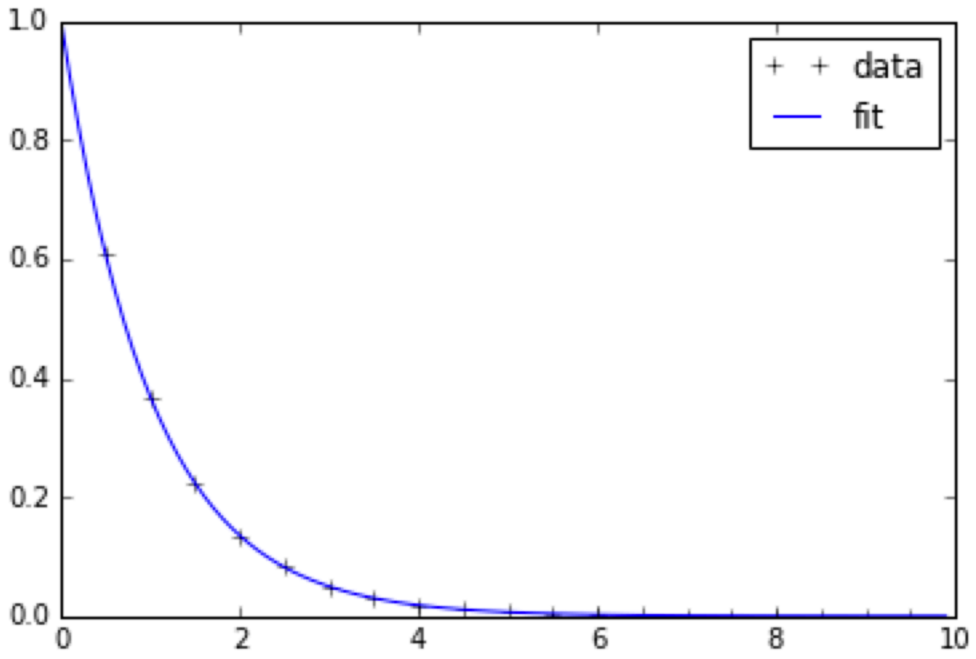
Technical Note:

We need `np.exp` instead of `exp` because the latter does not work on arrays.

Step 9: Non-Linear Curve Fitting

Of course, the fit is perfect and the parameters are exactly recovered:

```
         fitted initial concentration: 1.0
         fitted rate constant: 1.0

Out[29]: <matplotlib.legend.Legend at 0x10c4b4050>
```



Technical Notes: The `initial_concentration` variable inside `f2` is *shadowed*, meaning that its value is local the scope of the function. It is not the same as the `initial_concentration` variable defined in previous cells. You can think of it as a dummy variable that disappears as soon as the `f2` evaluates.

Step 10: The Two Step System

$$A \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} B \xrightarrow{k_2} C$$

There are three ways to treat this system:

1. Pre-Equilibrium Approximation

Assume that the ratio $K = [B]/[A]$ is maintained at its thermodynamic value, k_1/k_{-1}. This is valid if the subsequent rate constant, k_2, is relatively slow.

2. Steady State Approximation

More generally, we can assume that $d[B]/dt \approx 0$. More precisely, we assume that [B] changes much more slowly than [C]. Put another way, we assume that the amount of [B] is determined kinetically, rather than thermodynamically. This is valid if k_2 is relatively fast.

3. Differential Equations

If we solve the differential equations exactly, we can see when the pre-equilibrium and steady state approximations are valid.

Step 10: The Two Step System

$$A \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} B \xrightarrow{k_2} C$$

The integrated rate laws for this system are:

$$p = k_1 + k_{-1} + k_2, \quad q = \sqrt{p^2 - 4k_1k_2}$$

$$\lambda_2 = \frac{p+q}{2.0}, \quad \lambda_3 = \frac{p-q}{2.0}$$

$$[A] = \frac{k_1[A]_0}{\lambda_2 - \lambda_3}\left(\frac{\lambda_2 - k_2}{\lambda_2}e^{-\lambda_2 t} - \frac{\lambda_3 - k_2}{\lambda_3}e^{-\lambda_3 t}\right)$$

$$[B] = \frac{k_1[A]_0}{\lambda_2 - \lambda_3}\left(e^{-\lambda_3 t} - e^{-\lambda_2 t}\right)$$

The derivation uses the Laplace Transform (*J. Chem. Educ.* **1999**, *76*, 1578). I have presented the solution into a more convenient form (Chemical Kinetics and Catalysis Notes 2011, Professor Clark Landis, University of Wisconsin-Madison).

Step 10: The Two Step System

In a new cell, compute the timecourse of the following reaction:

$$A \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} B \xrightarrow{k_2} C$$

$$p = k_1 + k_{-1} + k_2, \quad q = \sqrt{p^2 - 4k_1 k_2}$$

$$\lambda_2 = \frac{p+q}{2.0}, \quad \lambda_3 = \frac{p-q}{2.0}$$

$$[A] = \frac{k_1 [A]_0}{\lambda_2 - \lambda_3}\left(\frac{\lambda_2 - k_2}{\lambda_2} e^{-\lambda_2 t} - \frac{\lambda_3 - k_2}{\lambda_3} e^{-\lambda_3 t} \right)$$

$$[B] = \frac{k_1 [A]_0}{\lambda_2 - \lambda_3}\left(e^{-\lambda_3 t} - e^{-\lambda_2 t} \right)$$

Simulation Parameters:

```
k_1       = 10.0
k_minus1  = 100.0
k_2       = 0.1
A_initial = 1.0
```

Plot from 0.0 to 5.0 seconds.

If this looks hard, don’t worry, because we’ll do it step by step.

Step 10: The Two Step System

First, setup some variables:

```
In [36]: k_1 = 10.0
         k_minus1 = 100.0
         k_2 = 0.1
         A_initial = 1.0
```

We can reevaluate this cell later when we want to change the rate constants.

Next, calculate:

$$p = k_1 + k_{-1} + k_2, \quad q = \sqrt{p^2 - 4k_1k_2}$$

$$\lambda_2 = \frac{p+q}{2.0}, \quad \lambda_3 = \frac{p-q}{2.0}$$

None of these variables depends on time, so it makes sense to calculate them ahead of time.

Step 10: The Two Step System

$$p = k_1 + k_{-1} + k_2, \quad q = \sqrt{p^2 - 4k_1k_2}$$

$$\lambda_2 = \frac{p+q}{2.0}, \quad \lambda_3 = \frac{p-q}{2.0}$$

In [31]:

```python
from math import sqrt

p = k_1 + k_minus1 + k_2
q = sqrt(p**2 - 4*k_1*k_2)
lambda_2=(p+q)/2.0
lambda_3=(p-q)/2.0
```

Note that we needed to import the `math.sqrt` function.

Step 10: The Two Step System

$$[A]=\boxed{\frac{k_1[A]_0}{\lambda_2-\lambda_3}}\left(\frac{\lambda_2-k_2}{\lambda_2}e^{-\lambda_2 t}-\frac{\lambda_3-k_2}{\lambda_3}e^{-\lambda_3 t}\right)=c_1\left(c_2e^{-\lambda_2 t}-c_3e^{-\lambda_3 t}\right)$$

$$[B]=\frac{k_1[A]_0}{\lambda_2-\lambda_3}\left(e^{-\lambda_3 t}-e^{-\lambda_2 t}\right)=c_1\left(e^{-\lambda_2 t}-e^{-\lambda_3 t}\right)$$

Examining the expressions for [A] and [B], we find that there are constants that also do not depend on time. I highlighted one of them.

Let's define those, too:

```
In [32]: c_1 = (k_1*A_initial)/(lambda_2-lambda_3)
         c_2 = (lambda_2-k_2)/lambda_2
         c_3 = (lambda_3-k_2)/lambda_3
```

Remember, if we change the rate constants later, we'll have to re-evaluate all of these cells to update everything.

Step 10: The Two Step System

$$[A]=\frac{k_1[A]_0}{\lambda_2-\lambda_3}\left(\frac{\lambda_2-k_2}{\lambda_2}e^{-\lambda_2 t}-\frac{\lambda_3-k_2}{\lambda_3}e^{-\lambda_3 t}\right)=c_1\left(c_2e^{-\lambda_2 t}-c_3e^{-\lambda_3 t}\right)$$

$$[B]=\frac{k_1[A]_0}{\lambda_2-\lambda_3}\left(e^{-\lambda_3 t}-e^{-\lambda_2 t}\right)=c_1\left(e^{-\lambda_2 t}-e^{-\lambda_3 t}\right)$$

Now, let's create functions for [A] and [B]. We will calculate [C] by mass balance later.

```
In [33]: def A(t):
             return c_1 * (c_2 * exp(-lambda_2*t) - c_3 * exp(-lambda_3*t))

         def B(t):
             return c_1 * (exp(-lambda_3*t)-exp(-lambda_2*t))
```

Note that these functions depend on variables that are outside their scope. For example, `c_1` appears inside `A(t)` even though it is not defined there. This is perfectly acceptable in Python.

Step 10: The Two Step System

Now we need to run the simulation.

```
time = np.arange(0.0,5.0,0.01)
conc_A = [ A(t) for t in time ]
conc_B = [ B(t) for t in time ]
conc_C = [ A_initial - conc_A[i] - conc_B[i] for i in range(len(time)) ]
```

First, we fill up the `time` list with values from 0.0 to 5.0 in steps of 0.01.

Then, we iterate over `time`. For each value `t` in `time`, we call the function `A` to get the value `A(t)`. These values are placed, one at a time, in `conc_A`.

We do the same thing for [B]. For [C], we use mass balance.

Step 10: The Two Step System

```
time = np.arange(0.0,5.0,0.01)
conc_A = [ A(t) for t in time ]
conc_B = [ B(t) for t in time ]
conc_C = [ A_initial - conc_A[i] - conc_B[i] for i in range(len(time)) ]
```

Step by step:

`len(time)` returns the number of time points (i.e., the number of elements in `time`)

`range(len(time))` returns a list `[0, 1, ..., len(time)-1]`. We can use this to iterate over each list in parallel, since each list has the same number of elements.

Finally,

```
conc_C = [ A_initial - conc_A[0]   - conc_B[0],
           A_initial - conc_A[1]   - conc_B[1],
           ...,
           A_initial - conc_A[n-1] - conc_B[n-1] ]
```

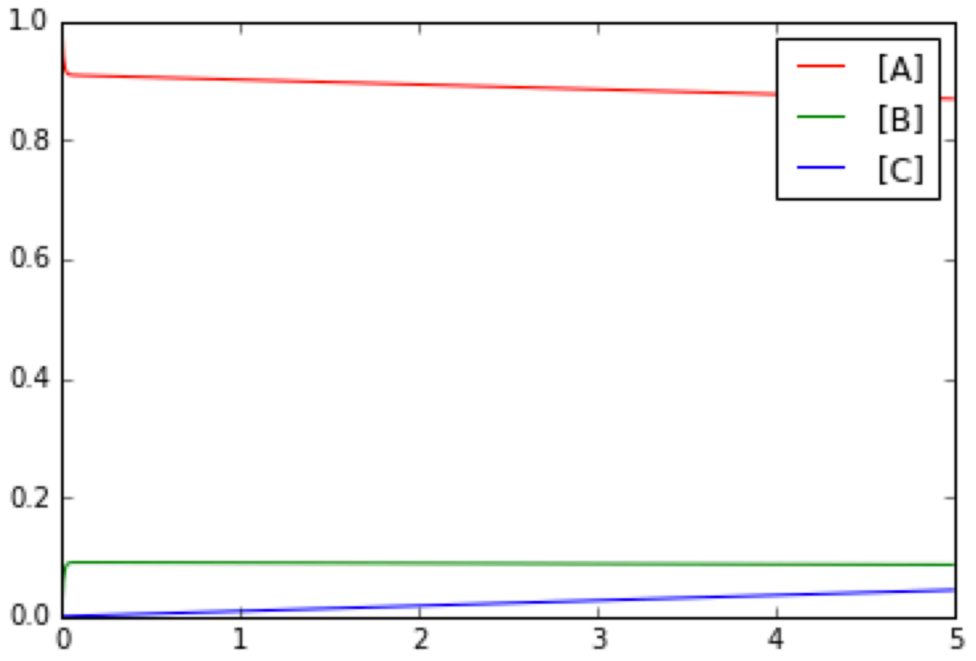
where n = len(time).

In English, this is the total concentration minus [A] minus [B] for every point in time.

Step 10: The Two Step System

Here is the plot of the result:

```
In [41]: plt.plot(time, conc_A, "r", label="[A]")
         plt.plot(time, conc_B, "g", label="[B]")
         plt.plot(time, conc_C, "b", label="[C]")
         plt.legend()

Out[41]: <matplotlib.legend.Legend at 0x10c700190>
```



Would you call this pre-equilibrium, steady state, or neither?

Step 11: Pre-Equilibrium or Steady State?

Plot the ratio of [B]/[A] over the course of the reaction to find out.

On one line, type:

```
conc_ratio = [ conc_B[i] / conc_A[i] if conc_B[i] > 0.0 else
0.0 for i in range(len(time)) ]
```

This divides [B]/[A] for each point in time. The expression:

```
conc_B[i] / conc_A[i] if conc_B[i] > 0.0 else 0.0
```

means that we should only divide if [A] ≠ 0.

The “foreach” expression

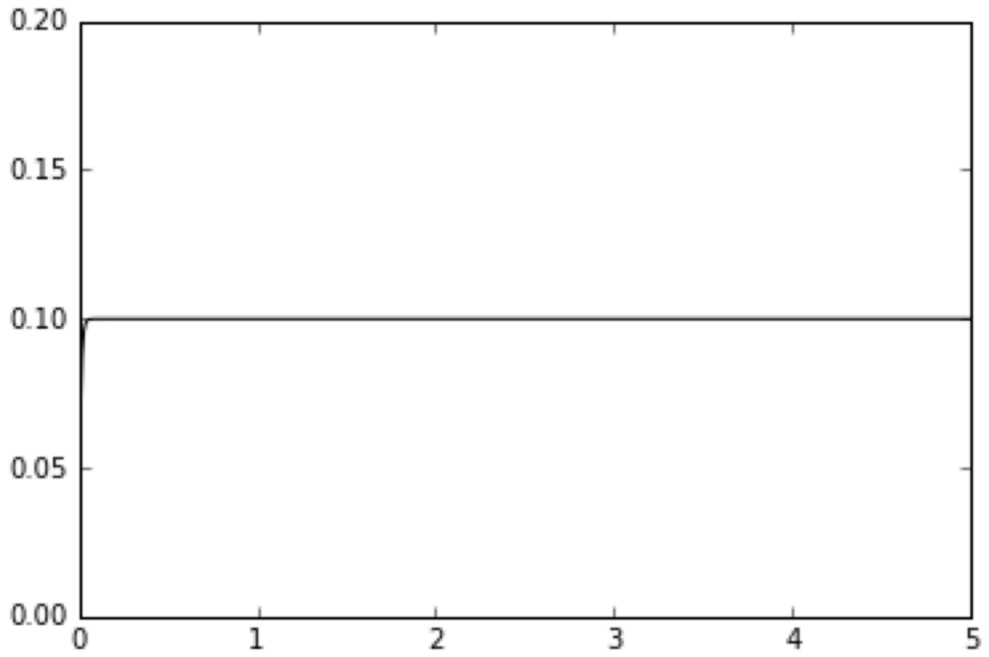
```
for i in range(len(time))
```

iterates over `[0, 1, ..., len(time)-1]`, the indices of `conc_A` and `conc_B`.

Step 11: Pre-Equilibrium or Steady State?

```
In [79]: conc_ratio = [ conc_B[i] / conc_A[i] if conc_B[i] > 0.0 else 0.0\
                       for i in range(len(time)) ]
         plt.ylim(0.0,0.2)
         plt.plot(time, conc_ratio, "k")

Out[79]: [<matplotlib.lines.Line2D at 0x1088df290>]
```

(This backslash lets me put the expression on one line. On your computer, you can just put the whole expression on one line.)

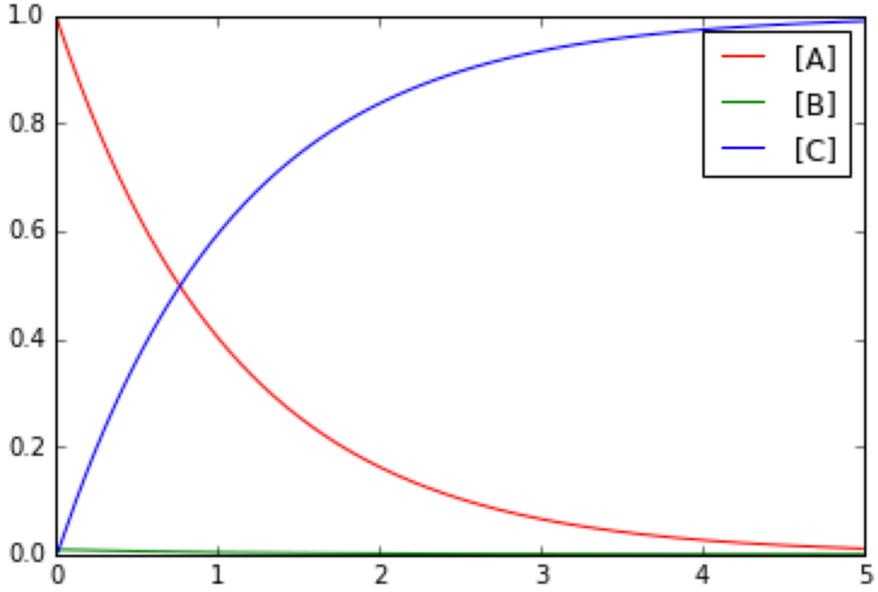


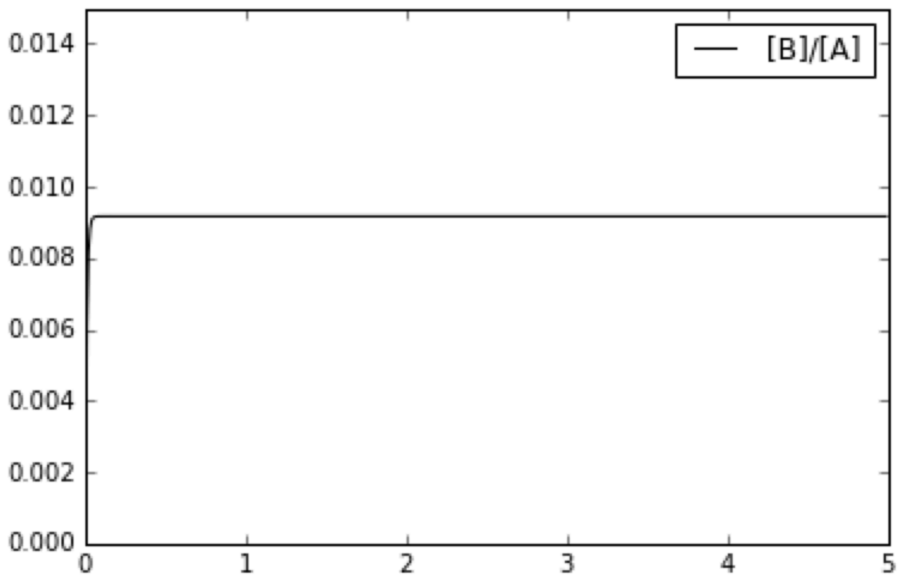
The ratio is 1:10 for most of the reaction. Recall, `k_1 = 10.0`, `k_minus1 = 100.0`, `k_2 = 0.1`. The thermodynamic ratio is $k_1/k_{-1} = 0.1$, so this is pre-equilibrium. This happens when k_2 is slow relative to k_1 and k_{-1}.

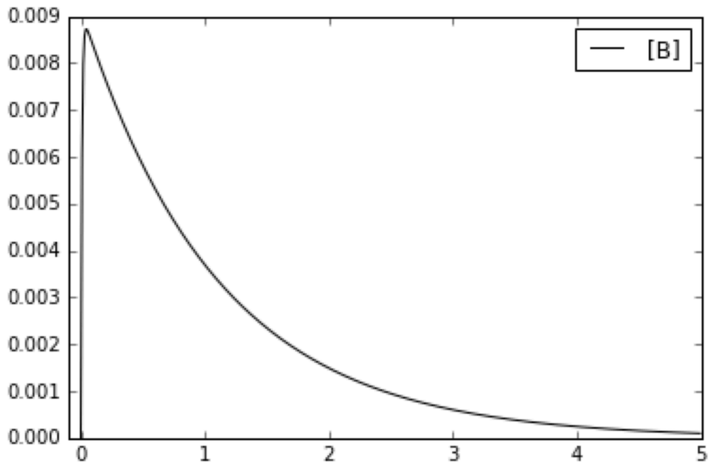
Step 11: Pre-Equilibrium or Steady State?

What if k_2 is fast? Re-run the simulation with:

```
k_1 = 1.0        k_minus1 = 10.0       k_2 = 100.0
```







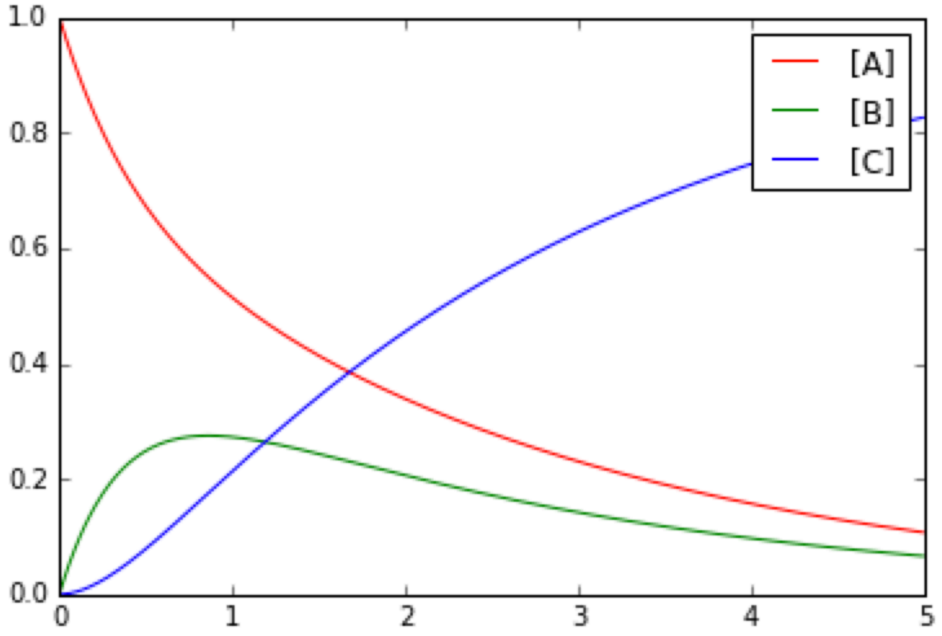
This is steady-state:
$$\frac{\partial[B]}{\partial t} = 0 = k_1[A] - k_{-1}[B] - k_2[B]$$

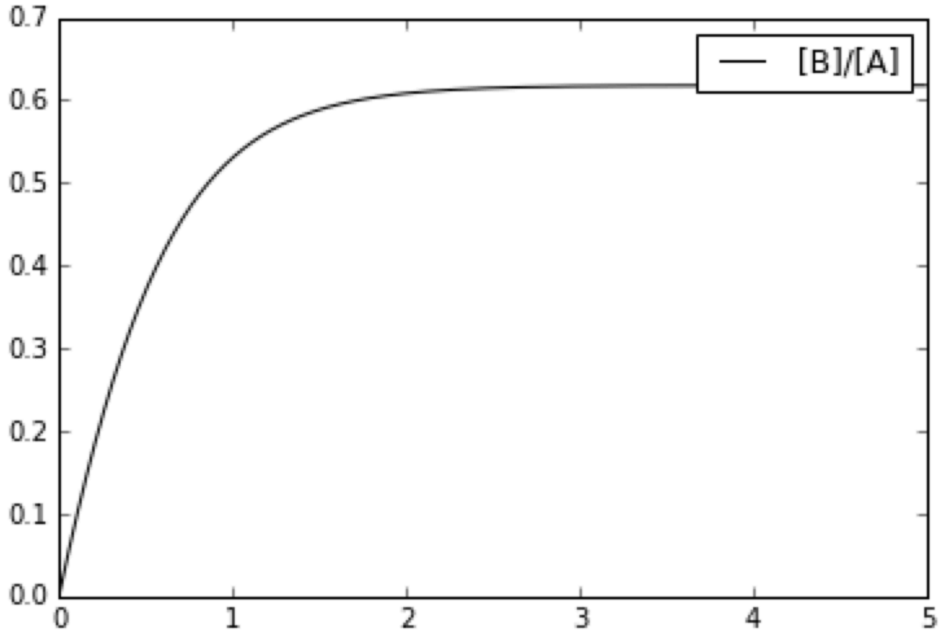
$$\frac{[B]_{SS}}{[A]} = \frac{k_1}{k_{-1} + k_2}$$

In this case, [B]/[A] = 1/(10+100) = 0.009, matching the figure above. This is the “kinetic equilibrium” value.

Step 12: Competitive Rates

To conclude this exercise, let's examine the case where all the rate constants are set to 1.0:





The plot on the right shows that *neither* the pre-equilibrium nor steady state conditions apply until late in the reaction. This scenario is rare.

Summary

Congratulations! You now know how to perform simple kinetic analyses in Python! Here are some short code fragments that summarize what you learned:

Plotting

```
# comments start with a hashtag

# import libraries
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

# multiple plot statements will
# automatically overlay
plt.plot(x,y,"ko”,label=“abc”)
plt.plot(x2,y2,”b”,label=“def”)

# set plot boundaries
plt.xlim(0.0,5.0)
plt.ylim(0.0,10.0)

# add a legend
plt.legend()

# add plot labels
plt.xlabel("x-axis label")
plt.ylabel("y-axis label")
plt.title("plot title”)

# save the plot
plt.savefig("my_plot.png”)
```

For a more in-depth introduction, I recommend:

https://www.codecademy.com/learn/python

More self-guided tutorials are available at:

http://learnpythonthehardway.org/book/index.html

A good reference for NumPy and SciPy is:

http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf

```
# fitting
```

Summary

Lists

```
# basic list operations
list1 = [1.0, 3.0, 7.0]
list1[0] = 1.0
list1[1] = 3.0
list1[2] = 7.0
len(list1) = 3

# to
# range(n) gives 0, 1, ..., n-1
list2 = [ i for i in range(5) ]
list2 = = [0, 1, 2, 3, 4]

# to take every n-th item
list3 = [ i for i in range(10) ]
list3[::5] = [0, 2, 4, 6, 8]
```

```
# avoid dividing by zero
# with a ternary expression

list4 = [0.0, 2.0, 4.0]

list5 = [ 1.0 / i if i > 0.0 else 0.0 for i in list3 ]
list5 = [0.0, 0.5, 0.25]
```

Math

```
# use ".0" after numbers

# addition, subtraction
1.0+2.0, 4.0-2.0

# multiplication, division
2.0*3.0, -4.0/5.0

# exponents
x ** y  # x raised to the y
```

```
from math import exp, log
import numpy as np

# exponentials
exp(x) or np.exp(x)

# logarithms
log(x) or np.log(x)

# linearly spaced values
np.arange(start,stop,stepsize)
np.linspace(start,stop,number_of_steps)
```