python Documentation

Release 0.1

Cliburn Chan

July 19, 2010

CONTENTS

1 Scientific Scripting with Python for Computational Immunology **3**

1.1 Getting started . . . 3
1.2 Programming in Python . . . 6
1.3 Variables and assignments . . . 8
1.4 Operators . . . 9
1.5 Basic building blocks . . . 10
1.6 Functions . . . 13
1.7 Flow control . . . 15
1.8 Comprehensions . . . 16
1.9 File I/O . . . 17
1.10 Reading and writing data files . . . 18
1.11 Practice session 1 . . . 19
1.12 Solution to Practice session 1 Exercise 1 . . . 19
1.13 Solution to Practice session 1 Exercise 2 . . . 19
1.14 Solution to Practice session 1 Exercise 3 . . . 19
1.15 Solution to Practice session 1 Exercise 4 . . . 20
1.16 Introduction to scientific scripting . . . 20
1.17 Linear algebra with numpy . . . 22
1.18 Statistics and random numbers . . . 23
1.19 Introduction to scipy . . . 24
1.20 Introduction to matplotlib . . . 24
1.21 Statistics and linear regression . . . 24
1.22 Difference equations and linear algebra . . . 28
1.23 Series expansions and function approximations . . . 30
1.24 Solution to discrete logistic equation exercise . . . 33
1.25 Data analysis and visualization . . . 34
1.26 Fitting nonlinear models to data . . . 35
1.27 Modeling with ordinary differential equations (ODEs) . . . 38
1.28 Basic model of virus infection . . . 40
1.29 Individual/agent based modeling . . . 41

2 Indices and tables **45**

This is a tutorial introduction to quickly get you up and running with the Python programming language. It will be pretty fast-paced, moving from Python basics to scientific scripting. If you follow along the tutorial, you'll quickly see how Python can be incredibly useful for the sort of simple data manipulation, analysis and plotting that every immunology laboratory needs. Python can also be used to build sophisticated applications, but that's not our goal here.

Grab the **PDF version** of this tutorial.

Next, *Getting started*

CHAPTER

ONE

SCIENTIFIC SCRIPTING WITH PYTHON FOR COMPUTATIONAL IMMUNOLOGY

1.1 Getting started

..index:: single: Installation

1.1.1 Installation

I will assume that you have downloaded and installed the Enthought Python distribution. If you are on LInux and prefer to use the native package manager, you need to install the `ipython` `mercurial`, `numpy`, `scipy` and `matplotlib` packages. For example, in Debian or Ubuntu, type:

```
sudo apt-get install ipython mercurial python-numpy
python-scipy python-matplotlib python-sympy
```

That's it for installation of packages needed for the summer school. For reference, most Python packages can be installed easily using `easy_install` or `pip`. For example, if we want to install the symbolic algebra library `sympy`, either:

```
easy_install sympy
```

or:

```
pip install sympy
```

will do the job of finding the `networkx` package, download it from the web and install it for you.

1.1.2 Data files

Obtaining data files - **data.zip**. Unzipping this will create a directory called `data` with 3 files:

flow.txt # dna.txt # cfus.txt

These files will be used as examples and in the practice exercises.

1.1.3 Scripts

The Python scripts used in this session can be downloaded from **scripts.zip**. Unzipping this will create a directory called `scripts``with some ``.py` files that you can run by typing `python <filename>.py`.

1.1.4 Programming environment

We will be using Python *interatively* with the **ipython** program, and also writing programs with a programmer's text editor. To start using ipython, open a terminal (or command shell in Windows) and type: `ipython`. To check that you have everything installed correctly,type the following

```
In [1]: import numpy

In [2]: import scipy

In [3]: import matplotlib

In [4]: import fcm

In [5]: exit()
Do you really want to exit ([y]/n)? y
```

For longer programs that we may want to keep and reuse, we will use a text editor to write a program - this prgoram will be saved as `somename.py` and later run from the command line like this:

```
python somename.py <arguments>
```

If you already code with a favorite text editor that has Python syntax highlighting (e.g. vi, emacs, eclipse), go ahead and continue using it. If not, we will use the free Komodo editor. Download and install from http://www.activestate.com/komodo-edit/downloads. Mac and Windows versions come with installers; those of you on Linux can follow the instructions given at http://usefulubuntu.blogspot.com/2009/03/installing-komodo-edit-5.html.

After installation, please do the following configuration within Komodo Edit to add a command to run Python scripts easily:

```
go to Toolbox -> Add -> New Command...
in the top field enter the name 'Run Python file'
in the 'Command' field enter this text: %(python) %F
click Ok.
```

Now you can click the 'Run Python file' button and see your program execute.

1.1.5 Quick tour of Python

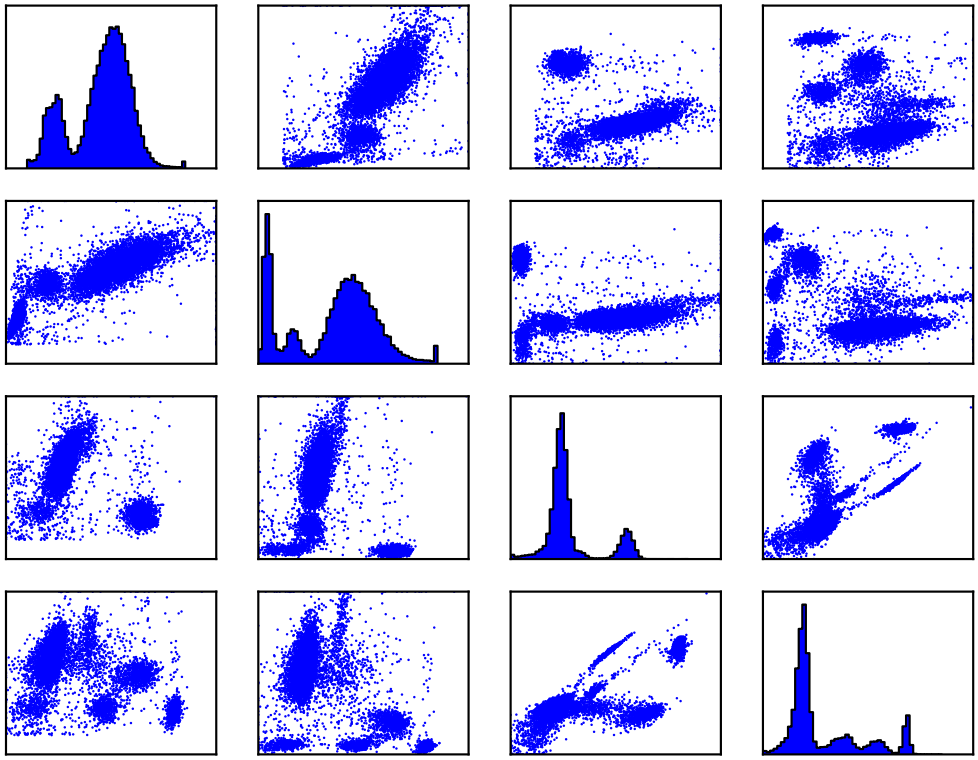
This session is loosely organzied to have aprpoximately 5 parts - the first 2 parts are an introductio to scientific programming, and the remaining 3 will use Python to computationally explore some of the statistical and mathematical topics that Tom will share.

1. introduction to the Python language
2. introduction to the main Python scientific modules - `numpy`, `scipy` and `matplotlib`
3. using statistics in Python to analyze data
4. using linear algebra and calculus in Python to help with modeling

5. playing with biological models in Python with differece equations, ordinary differential equations and individual based simulaitions

We will illustrate with an example of how concise scientific scripting with Python can be. For this example, we will read in some mysterious data from a text file `data.txt` and simply plot it for visualization.

```
import numpy
import pylab

if __name__ == '__main__':
    x = numpy.loadtxt('data.txt')
    n, p = x.shape
    nrows = ncols = p

    xmin, xmax = numpy.min(x), numpy.max(x)

    for row in range(nrows):
        for col in range(ncols):
            pylab.subplot(nrows, ncols, row*ncols+col+1)
            if row==col:
                pylab.hist(x[:,row], bins=50, histtype='stepfilled')
            else:
                pylab.scatter(x[::10,row], x[::10,col], s=1,
                              c='blue', edgecolors='none')
                pylab.axis([xmin, xmax, xmin, xmax])
            pylab.xticks([]); pylab.yticks([])

    # pylab.show()
    pylab.savefig('001.pdf')
```



The goal in this course is for you to be able to write similar scripts to pre-process, analyze and visualize your own data sets, and we will take you through many such examples. However, we first need to review some basics of the Python programming language, and then we will return to deconstruct the example

The course is extremely packed and you may feel like you are drinking from a fire hose, so if you have trouble understanding some of the examples or concepts, please discuss with one of the other students with programming experience, or talk to me. I will be here the whole week, and you can grab me at breaks, lunch or in the evenings.

Next, *Programming in Python*

1.2 Programming in Python

For those of you who have some previous programming experience, one of the unusual things about Python is that `white space` is significant (unless your previous language was **COBOL**, **Fortran** or **Haskell**). This means that indentation is important, and programming statements at the same "level" must line up vertically.

We will use Python in two ways - we will first use the Python **interpretor** *interactively* as this is a great way to get feedback fast, and then write **programs** using a text editor. For now, start the **ipython** interpretor by opening a **Teminal** and typiing:

```
ipython
```

You should see something like this (and more if this is the first time ipython is used)

```
$ ipython
Python 2.6.1 (r261:67515, Feb 11 2010, 00:51:29)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

You can enter Python expressions at the prompt `In [1]`, hit `Return` and the expression will be immediately evaluated. Try it!

```
In [1]: 7*3
Out[1]: 21

In [2]: "hello world"[::-1]
Out[2]: 'dlrow olleh'

In [3]: sum([1,2,3])
Out[3]: 6

In [4]: help()
```

To exit ipython, type

```
In[5] quit()
```

and hit `Return`

Because python is an object-oriented language, there are **methods** associated with most constructs, and ipython makes it easy to find out what these methods are by **tab completion**. Here's an example

```
In [1]: greeting = "hello world"

In [2]: greeting.
```

Notice the `.` after `greeting` ... now press the TAB key twice

```
In [2]: greeting.
greeting.__add__                    greeting.__class__
greeting.__contains__               greeting.__delattr__
greeting.__doc__                    greeting.__eq__
greeting.__format__                 greeting.__getattribute__
greeting.__getitem__                greeting.__getnewargs__
greeting.__getslice__               greeting.__gt__
greeting.__hash__                   greeting.__init__
greeting.__le__                     greeting.__len__
greeting.__lt__                     greeting.__mod__
greeting.__mul__                    greeting.__ne__
greeting.__new__                    greeting.__reduce__
greeting.__reduce_ex__              greeting.__repr__
greeting.__rmod__                   greeting.__rmul__
greeting.__setattr__                greeting.__sizeof__
```

```
greeting.__str__                      greeting.__subclasshook__
greeting._formatter_field_name_split  greeting._formatter_parser
greeting.capitalize                   greeting.center
greeting.count                        greeting.decode
greeting.encode                       greeting.endswith
greeting.expandtabs                   greeting.find
greeting.format                       greeting.index
greeting.isalnum                      greeting.isalpha
greeting.isdigit                      greeting.islower
greeting.isspace                      greeting.istitle
greeting.isupper                      greeting.join
greeting.ljust                        greeting.lower
greeting.lstrip                       greeting.partition
greeting.replace                      greeting.rfind
greeting.rindex                       greeting.rjust
greeting.rpartition                   greeting.rsplit
greeting.rstrip                       greeting.split
greeting.splitlines                   greeting.startswith
greeting.strip                        greeting.swapcase
greeting.title                        greeting.translate
greeting.upper                        greeting.zfill
greeting.__ge__
```

and you see a huge list of methods that the string `greeting` knows about. For example, `split` will break the string at any whitespace, returning a list of strings. Try typing `sp` after `greeting.`, and hit TAB again at `greeting.sp`, and ipython will auto-complete to `greeting.split` and finish up by typing `()` to evaluate the split method.

```
In [2]: greeting.split()
Out[2]: ['hello', 'world']
```

We will next cover the basic elements of the Python programming language - please experiment within ipython as you move on to *Variables and assignments*

1.3 Variables and assignments

One of the most basic ideas in most languages is the idea of storing *values* in *variables*, and this done in Python with the **assignment** operator, which is just the = symbol:

```
x = 3
```

means take the value `3` and store it as `x`. Now the value of `x` is 3, and we can use x elsewhere:

```
x + 3
```

will give the answer `6`. We can *reassign* `x`:

```
x = 7
```

and now

```
x + 3
```

will be `10`.

Note that assignment is very different from mathematical equality, which is represented in Python by the == operator. So:

```
x == 7
```

is asking "Is x equal to 7?", and this would be `True` after the last assignment.

1.4 Operators

Python has a long list of operators that pretty much do as you'd expect them to.

Mathematical:

```
+, -, *, /, //, **, %,
```

The `//` operator is for integer division - i.e. the result of x//y is always rounded down to the nearest integer value:

```
In [1]: 2.0//3
Out[1]: 0.0
```

The `**` operator performs exponentiation:

```
In [2]: 10**3
Out[2]: 1000
```

and the `%` operator gives the modulus (remainder after division)

```
In [3]: 10 % 3
Out[3]: 1
```

Python also has a `divmod` function that returns both the dividend and remainder

```
In [4]: divmod(10, 3)
Out[4]: (3, 1)
```

Most of the arithmetic operators also have a convenient shorthand for repeated application:

```
x += 2
```

means:

```
x = x + 2
```

Logical:

```
and, or, not,   ==, <, <=, >, >=, !=
```

Logical operators return True or False and are often used to test for some condition in `if` and `while` loops often with the comparsion operators including == (is equal to) and `!=` (not equal).

```
In [5]: True and False
Out[5]: False

In [6]: True or False
Out[6]: True

In [7]: not True
Out[7]: False

In [8]: 3 > 4
Out[8]: False

In [9]: k = 0

In [9]: while k<3:
   ....:     print k
   ....:     k += 1
   ....:
0
1
2
```

For example, in the last example `In [9]`, `k` is initialized to be 0. The `while` loop checks the current value of `k` - if it is less than 3, the loop prints `k` and increments `k` by one. Eventually, `k=3` and the loop exits, hence only 0, 1 and 2 are printed.

Containment:

```
in
```

This checks that an element is in a list, or tuple, or dictionary, or ... anything that can be iterated through.

Bit manipulation:

```
<<. >>, &, |, ^, ~
```

Not covered here.

The only thing to note is that some operators are **overloaded** and may result in different actions depending on context. For example:

```
1 + 2
```

is addition and gives 3 while:

```
[1,2,3] + [4,,5,6]
```

is list concatenation and gives [1,2,3,4,5,6].

1.5 Basic building blocks

Some of the basic types in python are

`char`:

```
'a', 'z', '1'
```

boolean:

```
True, False
```

int:

```
1, 2, 43
```

float:

```
3.141519, 2.03e12
```

1.5.1 Iterables

Basic types by themselves are pretty limited, and the fun begins when we can work with *group* or *collections* or *iterables* of these basic types. Some of the more commonly collections in Python are `strings`, `tuples`, `lists`, `dictionaries` and `sets`.

Strings are just collections of `char`, so many of the useful tricks for dealing with collections that we will soon learn also apply to strings. The following are examples of strings:

```
'hello, world'
"goodbye, cruel world"
"""wait!
    I'm back!"""
```

Strings have many useful *methods* associated with them as we saw earlier.

1.5.2 Tuples and lists

Tuples and lists are the workhorses of Python. Anything separated by commas is a tuple in Python. For example:

```
1, 2, 3
```

is a tuple of integers. Conventionally, tuples are denoted with parentheses:

```
(1,2,3)
```

and can contain the same or different types:

```
(1, 'a', ('hello world', 'goodbye cruel world'))
```

is also a tuple (in this case, containing an integer, a character, and another tuple of strings).

Tuples are *immutable*, that is, the contents of a tuple cannot be changed without making a copy of the tuple. So tuples are good for storing information that is stable - if you intend to make changes to the contents, use a `list` instead. Lists are represented with square brackets and can also contain different types of elements:

```
[1, 'a', 'whatever']
```

1.5.3 Indexing

Lists and tuples can have their individual elements retrieved by positional **indexing**. Examples are helpful here:

```
In [24]: xs = [1,2,3,4,5]

In [25]: xs[0]
Out[25]: 1

In [26]: xs[1]
Out[26]: 2

In [27]: xs[4]
Out[27]: 5

In [28]: xs[-1]
Out[28]: 5
```

Both tuples and lists use square brackets for indexing. Note that Python *counts from 0*, and negative indices means *count from the end*. Trying to access indices for non-existent elements gives an error:

```
In [29]: xs[5]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

/Volumes/HD3/hg/summer-school/python/<ipython console> in <module>()

IndexError: list index out of range
```

Because lists are mutable and tuples are not, we can use indices to change the value of list elements but not tuple elements:

```
In [19]: alist = [1,2,3]

In [20]: alist[1] = 9

In [21]: alist
Out[21]: [1, 9, 3]

In [22]: atuple = (1,2,3)

In [23]: atuple[1] = 9
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/Volumes/HD3/hg/summer-school/python/<ipython console> in <module>()

TypeError: 'tuple' object does not support item assignment
```

We can use indexing to retrieve more than one element at a time with the **slice** notation `[start:stop:stride]` where start, stop and stride are integers. Start and stop are the start and stop positions, and stride is the interval between positions. So, a stride of 2 would mean selecting every other element between start and stop. Some examples:

```
In [30]: xs[1:-1]
Out[30]: [2, 3, 4]

In [31]: xs[::2]
```

```
Out[31]: [1, 3, 5]

In [32]: xs[0:-1:2]
Out[32]: [1, 3]

In [33]: xs[::-1]
Out[33]: [5, 4, 3, 2, 1]
```

In `In [30]`, we ask for the slice of `xs` that starts at position 1 and stops at position -1 (i.e. the last postion). Python slices do not include the stop position, so we get `[2, 3, 4]` back. When the `stride` is unspecified, it default to 1. In `In [31]` we see the effect of using a stride of 2. In this case, when start and stop are not specified, they default to 0 and the length of the iterable (which is one more than the last position), and `In [32]` shows an example where all 3 slice arguments are specified. The final example `In [33]` shows that Python interprets a stride of -1 to mean return the **reversed** iterable.

1.5.4 Dictionaries and sets

Dictionaries and sets differ from tuples and lists in that they are indexed not by integers, but by a **key** that can be any immutable item. Examples of keys are integers, strings, tuples.

A `dictionary` is a special container for storing `pairs` of items, where the first member is the `key` and is used to retrieve the second member or `value`. Dictionaries are represented with curly braces, with key, value pairs separated by a colon. For example, a simple phone book can be represented as a dictionary, and we can retrieve phone numbers by indexing on the keys:

```
In [34]:  phone_dict = {'tom' : '919-123-4567', 'cliburn': '919-345-6789'}
   ....:

In [35]: phone_dict['cliburn']
Out[35]: '919-345-6789'
```

The `set` is a collection that behave like mathematical sets - that is, sets do not contain duplicates, and you can perform set difference, intersection, union etc. Sets are created with the `set` keyword from another collection:

```
In [36]: set([1,1,2,2,3,3])
Out[36]: set([1, 2, 3])
```

Note the loss of duplicates when the set is created form a listL.

1.6 Functions

A **function** is a black box - it takes some input, and returns an output value. Functions are indispensable for programming in Python since they reduce code repetition. For example, `range` that we just used above is a built-in function that returns a range of integers. However, it is very easy to define our own functions in Python using the keyword `def`

```
In [1]: def square(x):
   ...:     """Returns the square of x."""
   ...:     return x*x
   ...:

In [2]: square(3.5)
Out[2]: 12.25
```

Here we define a `square` function that takes an input argument `x` and returns `x*x` as the output value. Note the triple quoted string just after the function definition - this is called the function `docstring` and Python will show this when someone asks for help on the function.

```
 In [3]: help(square)

Help on function square in module __main__:

 square(x)
     Returns the square of x.
```

To exit help, press the **Q** key. We can try this with other functions, for example, the `range` function

```
In [4]: help(range)

Help on built-in function range in module __builtin__:

range(...)
  range([start,] stop[, step]) -> list of integers

  Return a list containing an arithmetic progression of integers.
  range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
  When step is given, it specifies the increment (or decrement).
  For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
  These are exactly the valid indices for a list of 4 elements.
```

and we see that `range` actually takes 3 input arguments, but `start` and `step` have defaults so range(10) really means range(0, 10, 1). Notice that range starts from 0 by default, and does *not* include the stop number.

To add a default argument to a function, just assign a value to the input argument in the function definition

```
In [5]: def power(x, n=2):
   ...:     """Returns x raised to the nth power."""
   ...:     return x**n
   ...:

In [7]: power(3)
Out[7]: 9

In [8]: power(3,3)
Out[8]: 27

In [9]: power(n=3, x=2)
Out[9]: 8
```

where we also see the use of **named arguments**.

Function arguments are **dummy** variables - they take on the value of whatever was passed in when the function was called, and have no relationhip with any variables outside the function that may have the same name. For example

```
In [43]: x = 10

In [44]: def f(x):
   ....:   return x+2
   ....:
In [45]: f(3)
Out[45]: 5
```

The `x` argument in `f(x)` has nothing to do with the variable `x` outside the function that has been assigned the value 10. Hence `f(3)` is 5 and not 12.

Functions can be applied to lists of numbers in two ways - using list comprehensions

```
In [10]: [square(x) for x in range(3, 10, 2)]
Out[10]: [9, 25, 49, 81]
```

or the built-in function `map` that takes 2 arguments - the first is the function, and the second is a collection of stuff to apply the function to

```
In [11]: map(square, range(3, 10, 2))
Out[12]: [9, 25, 49, 81]
```

Related functions in Python are grouped together in **modules**, and we need to `import` the module to use it. For example, the **numpy** module is indispensable for numerical computation, and we get access to it by

```
In [12]: import numpy

In [13]: x = numpy.arange(1, 5, 0.5)

In [14]: x
Out[14]: array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])

In [15]: x**2
Out[15]: array([  1.  ,   2.25,   4.  ,   6.25,   9.  ,  12.25,  16.  ,  20.25])
```

Note that `x` is a numpy **array**, created by the `arange` function that is like range, but allows non integer arguments. Since arange is a function in the numpy module, it is called by prefacing with `numpy.`. Notice also that numpy arrays are **vectorized** - the same operation (or numpy function) is applied to all members of the array without the need for `map` or `comprehensions`. Two other modules we will be making use of are **scipy** for scientific functions, and **matplotlib** for graphics.

1.7 Flow control

Sometimes, we need to make decisions. The simplest way is with the `if` statement:

```
if x%2==0:
  print "x is even"
else:
  print "x is odd'
```

The if statement can also be extended with `elif` before the filnal `else`:

```
if 0 < x <= 2:
  print "tiny"
elif 2 < x <= 5:
  print "small"
elif 5 < x <= 10:
  print "big"
else:
   print "huge"
```

We also often need to do similar stuff many times, and this can be done with the `for` statement:

```
for i in range(10):
  print i*i
```

and this is not restricted to numbers:

```
for name in ('Ann', 'Bob', 'Charlie'):
  print "hello", name
```

If we need to repeat until some condition is met, we can use the `while` statement:

```
x = 0
while x < 5:
  x += 1
```

1.8 Comprehensions

You may be familiar with *set definitions* in Mathematics, for example:

```
s = {n^2-4 : n is an integer; and 0 <= n <= 9}
```

is the set `s` of the twenty smallest integers that are four less than perfect squares. Such sets are easily generated in Python using **list comprehensions** that mimic the mathematical notation:

```
s = [n**2-4 for n in range(10)]
```

where:

```
range(n) = [0,1,2,...,n-1]
```

We can even add in **filter** clauses to winnow down the generator list:

```
s1 = [n^2-4 for n in range(10) if n%3==0]
```

where the filter is the `if` part. Try it in ipython

```
In [15]: [n**2-4 for n in range(10)]
Out[15]: [-4, -3, 0, 5, 12, 21, 32, 45, 60, 77]

In [16]: [n**2-4 for n in range(10) if n%3==0]
Out[16]: [-4, 5, 32, 77]
```

List comprehensions are a much more compact way of writing `for` loops such as:

```
s = []
for n in range(10):
  if n%3==0:
    s.append(n**2-4)
```

For fans of **functional programming**, this can also be expressed as:

```
map(lambda n: n**2-4, filter(lambda n: n%3==0, range(10)))
```

However, this style of programming with liberal use of `map`, `filter` and `lambda` (nameless functions) is not very common in Python and will not be elaborated on here.

1.9 File I/O

We will only cover how to read in simple text files and delimited format files here. Working with files usually involves a 3 stage process:

1. opening the file
2. doing something with the contents (read, write or append)
3. closing the file.

There are two ways of reading a text file - line by line, or all at once. Here is a line by line example::

```
fin = open('examples/flow.txt', 'r')
for line in fin:
   print line,
 fin.close()
```

The built-in function `open` opens the file given by the first argument `examples/data.txt` for *reading* as indicated by the `'r'` in the second argument. If we are only using the file once, the `fin` variable is gratuitous and we simply write:

```
for line in open('examples/flow.txt'):
   print line,
```

and file closing is handled automatically.

Actually, the above script will not work on Windows machines for two reasons - the path separator in Windows is `\` and not `/`, and the line termination indicator is different in Windows. If you do cross-platform data analysis, you will need a bit more code to make it bulletproof:

```
import os
fin = open(os.path.join('examples', 'flow.txt'), 'rU')
for line in fin:
  print line,
fin.close()
```

where the path is assembled in a platform-independent way using the `os.path` module's `join` function, and the second argument is `rU` instead of just `r` - the `U` for Universal tells Python to do the appropriate thing for each platform.

We can also read the entire contents of the file into a string:

```
string = fin.read()
```

or as a list of strings (one string for each line in the original file):

```
strings = fin.readlines()
```

To create a new file and write something to it, we can write:

```
fout = open('examples/foo.txt', 'w')
fout.write('hello world\n')
fout.close()
```

When the second argument to `open` is `w`, a new file is created for writing. If there is an existing file with the same name, it will be clobbered. Sometimes it is prudent not to overwrite existing files:

```
import os
filename = 'examples/foo.txt'
if not os.path.exists(filename):
  fout = open(filename, 'w')
...
```

We can also open a file for appending:

```
fout = open('examples/foo.txt', 'a')
fout.write('goodbye cruel world\n')
fout.close()
```

and the contents of `examples/foo.txt` will now be:

```
hello world
goodbye cruel world
```

1.10 Reading and writing data files

Many data files have a tabular format, with multiple rows each containing the same number of columns separated by some delimiter. For example, CSV files exported from Excel have this format. Sometimes, they may also have one or more lines of text, either comments or corresponding to column headers.

For example, the file may look like this:

```
234.000000    58.000000      648.000000      487.000000
503.000000    239.000000     287.000000      613.000000
511.000000    542.000000     305.000000      180.000000
```

where the file consists of many rows of 4 columns separated by tabs, and typically we want to read this directly into (or write out from an array), and numpy provides convenient functions to do so:

```
import numpy
x = numpy.loadtxt('examples/data.txt')
```

will create a numpy array `x` with the data from the file.

If necessary, you can skip rows (e.g. column headers), ignore comments, or only import certain columns:

```
x = numpy.loadtxt("examples/data.txt", skiprows=1, comments='#',
                                  delimiter=',', usecols=(0,2))
```

Alternatively, you may want to save a numpy array to a text file:

```
import numpy
xs = numpy.array([1,1,2,3,5,8,13,21])
numpy.savetxt('examples/fibonacci.txt', xs)
```

Time to for some hands on practice *Practice session 1*

1.11 Practice session 1

1. Calculate the sum of the first 1000 consecutive integers in Python (starting from 1). *Solution to Practice session 1 Exercise 1*
2. Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz". (This is an infamous programming interview question, because it is claimed that *the majority of comp sci graduates can't.* http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/) *Solution to Practice session 1 Exercise 2*
3. Read in the file "dna.txt" in the downloaded data directory and find the number of A, C, G and Ts in the file. *Solution to Practice session 1 Exercise 3*
4. Write a function that takes two input arguments - the first is a string, and the second is a character. The function returns the number of characters in the string that are the same as the second argument. *Solution to Practice session 1 Exercise 4*

In the next session, we will move on to *Introduction to scientific scripting*

1.12 Solution to Practice session 1 Exercise 1

```
print sum(range(1, 1001))
```

1.13 Solution to Practice session 1 Exercise 2

```
for i in range(1, 101):
    if i%15.0 == 0:
        print "FizzBuzz"
    elif i%3 == 0:
        print "Fizz"
    elif i%5 == 0:
        print "Buzz"
    else:
        print i
```

1.14 Solution to Practice session 1 Exercise 3

```
# read in file and strip trailing empty space
s = open('dna.txt').read().strip()
len(s)
s

# method 1 - use built-in count method of strings
s.count('A')
s.count('C')
s.count('T')
s.count('G')
```

```
# method 2 - use sum and a generator comprehension
sum(1 for x in s if x=='A')
sum(1 for x in s if x=='C')
sum(1 for x in s if x=='T')
sum(1 for x in s if x=='G')

# method 3 - use a defaultdict with default value = 0
import collections
counter = collections.defaultdict(int)
for x in s:
    counter[x] += 1
print counter

# method 4 - use vectorized equality operation after conversion of
# string to numpy character array
import numpy
sum(numpy.array(s, 'c')=='A')
sum(numpy.array(s, 'c')=='C')
sum(numpy.array(s, 'c')=='T')
sum(numpy.array(s, 'c')=='G')
```

1.15 Solution to Practice session 1 Exercise 4

```
def coumt_letter(text, ch):
    return text.count(ch)

if __name__ == '__main__':
    # read in file and strip trailing empty space
    s = open('dna.txt').read().strip()

    for ch in 'ACTG':
        print ch, count_letter(s, ch)
```

1.16 Introduction to scientific scripting

There are many specialized Python libraries written for scientific research, but the most generally useful are **numpy**, **scipy** and **matplotlib**. In this session, we will learn how to use these libraries for data analysis and visualization.

The current numpy documentation is a little dry for beginners to wade through, so I will give a longish basic overview here. Once you have gone through this section and want to see what else numpy can do, there is a vast example list available at http://www.scipy.org/Numpy_Functions_by_Category.

1.16.1 Introduction to numpy

The numpy library provides data structures and functions for numerical computation and linear algebra. Critically, the numpy library is written in C and highly optimized, so that equivalent mathematical calculations done in numpy are much faster than those done in pure Python. Here is a trivial example:

```
$ python -m timeit -s "s=0" "for i in range(1000000): s+=i"
10 loops, best of 3: 85.3 msec per loop
```

```
$ python -m timeit "sum(range(1000000))"
10 loops, best of 3: 46.3 msec per loop

$ python -m timeit -s "import numpy" "numpy.sum(numpy.arange(1000000))"
100 loops, best of 3: 4.87 msec per loop
```

Note the 20-fold increase in speed from using a for loop to using the `numpy.sum` and `numpy.arange` functions.

Numpy arrays

The basic data type in numpy is the **array**, and this can be in an arbitrary number of dimesions. The simplest way to create an array is to convert them from a list

```
In [1]: import numpy

In [2]: numpy.array([1,2,3])
Out[2]: array([1, 2, 3])

In [3]: numpy.array([1.,2.,3.])
Out[3]: array([ 1.,  2.,  3.])

In [4]: numpy.array([[1,0],[0,2]])
Out[4]:
array([[1, 0],
       [0, 2]])
```

Note that the first array was a 1D array of 3 integers, the second a 1D array of 3 floats, and the final a 2D array. Arrays can also be created from special **factory** functions

```
In [5]: numpy.zeros((4,2))
Out[5]:
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])

In [6]: numpy.ones(10)
Out[6]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

In [7]: numpy.arange(2,11,2)
Out(7]: array([ 2,  4,  6,  8, 10])

In [8]: numpy.identity(3)
Out[8]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

We can do mathematical operations on arrays

```
In [10]: a = numpy.arange(1,11)

In [11]: b = numpy.arange(10,0,-1)

In [12]: a
Out[12]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [13]: b
Out[13]: array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])

In [14]: a + b
Out[14]: array([11, 11, 11, 11, 11, 11, 11, 11, 11, 11])

In [15]: a + 10
Out[15]: array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20])

In [16]: a**2
Out[16]: array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])

In [17]: numpy.log(a)
Out[17]:
array([ 0.        ,  0.69314718,  1.09861229,  1.38629436,  1.60943791,
        1.79175947,  1.94591015,  2.07944154,  2.19722458,  2.30258509])

In [18]: numpy.sum(a), numpy.prod(a)
Out[18]: (55, 3628800)

In [19]: numpy.min(a), numpy.max(a)
Out[19]: (1, 10)
```

Next up, linear algebra and statistics with numpy: *Linear algebra with numpy*

1.17 Linear algebra with numpy

```
In [20]: m1 = numpy.array([[1,0],[0,2]])

In [21]: numpy.dot(m1, m1)
Out[21]:
rray([[1, 0],
      [0, 4]])

In [22]: numpy.diag(m1)
Out[22]: array([1, 2])

In [23]: numpy.trace(m1)
Out[23]: 3

In [24]: import numpy.linalg as la

In [25]: la.det(m1)
Out[25]: 2.0

In [26]: la.inv(m1)
Out[26]:
rray([[ 1. ,  0. ],
      [ 0. ,  0.5]])

In [27]: e, v = la.eig(m1)

In [27]: e
Out[27]: array([ 1.,  2.])
```

```
In [28]: v
Out[28]:
rray([[ 1.,  0.],
      [ 0.,  1.]])

In [29]: la.cholesky(m1)
Out[29]:
array([[ 1.        ,  0.        ],
       [ 0.        ,  1.41421356]])
```

Suppose we wanted to solve a set of linear equations

$$3x + 4y = 10$$
$$x - 2y = 3$$

In matrix notation, this would be

$$\begin{pmatrix} 3 & 4 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 10 \\ 3 \end{pmatrix}$$

We can easily solve and check this with numpy

```
In [3]: A = [[3,4],[1,-2]]

In [4]: B = [10,3]

In [5]: X = la.solve(A, B)

In [6]: X
Out[6]: array([ 3.2,  0.1])

In [7]: numpy.dot(A, X)
Out[7]: array([ 10.,   3.])
```

1.18 Statistics and random numbers

Numpy also has some elementary statistical functions and random number distributions useful for **simulations**.

```
In [17]: numpy.random.normal(3, 2, 4)
Out[17]: array([ 1.44387144,  2.13959912,  1.36089619,  1.78945005])

In [18]: numpy.random.binomial(10, 0.3, 4)
Out[18]: array([2, 2, 4, 4])
```

We can do statistics on arrays and see the *law of large numbers* in action

```
In [23]: numpy.mean(numpy.random.normal(3, 2, 5))
Out[23]: 3.2453640076788668

In [24]: numpy.std(numpy.random.normal(3, 2, 5))
Out[24]: 1.2214649679712186

In [27]: numpy.mean(numpy.random.normal(3, 2, 50000))
Out[27]: 2.9945709290502984
```

```
In [28]: numpy.std(numpy.random.normal(3, 2, 50000))
Out[28]: 2.0060104334714044
```

Next up, scipy.rst *Introduction to scipy*

1.19 Introduction to scipy

There are a huge number of modules in scipy, including modules for numerical integration, optimization, interpolation, clustering, fourier transforms, signal processing, linear algebra, statistics and image processing. It is far too much to cover, and besides, the reference documentation at http://docs.scipy.org/doc/scipy/reference/ is excellent and easily followed once you have a basic understanding of how numpy works.

1.20 Introduction to matplotlib

Matplotlib is also extremely easy to use, and the available functions are very clearly described at http://matplotlib.sourceforge.net A fast way to generate plots for your research is to scan the examples at http://matplotlib.sourceforge.net/gallery.html. Clicking on any example will bring up the source code that you can modify to suit your needs.

So, instead of going through another long, boring discussion of what scipy and matplotlib can do, we will demonstrate their use with real data sets provided by Tom. Now for some *Statistics and linear regression*.

1.21 Statistics and linear regression

We will first gain some experience with *The Weak Law of Large Numbers*, which simply says that the sample average converges to the expected value as the number of trials increases.

Suppose we flip a coin - it can be land heads (H) or tails (T). If it is a fair coin, the result of a coin toss is equally likely to be H or T, i.e. $P(H) = P(T) = 0.5$. There program `cointoss.py` is a simulation of repeated coin tosses with the following interface - press `T` to *toss* and after some number of tosses, press `G` to guess if the coin was fair (F) or biased (B). Download the program from **examples/cointoss.py** and type:

```
python examples/cointoss.py
```

Next, we will analyze some simple PCR data found in `data/dilution.csv`. Start up `ipython` as usual with the `-pylab` argument:

```
ipython -pylab
```

and quickly scan the contents of the `dilution.csv` file.

```
In [1]: less dilution.csv
```

We can see that dilution has a row of headers and is comma-separated. For now, we are only interested in fitting the first 2 columns of the file, so we load the file using `numpy.loadtxt`

```
In [2]: import numpy

In [3]: xs = numpy.loadtxt('dilution.csv', skiprows=1, delimiter=',', usecols=(0,1))
```

We want to express the dilution factor as the number of dilutions, so we will assign x to be log2(column1 values) and y to be column2 values

```
In [4]: x = log2(xs[:,0])

In [5]: y = xs[:,1]

In [6]: plot(x, y, 'x')
```

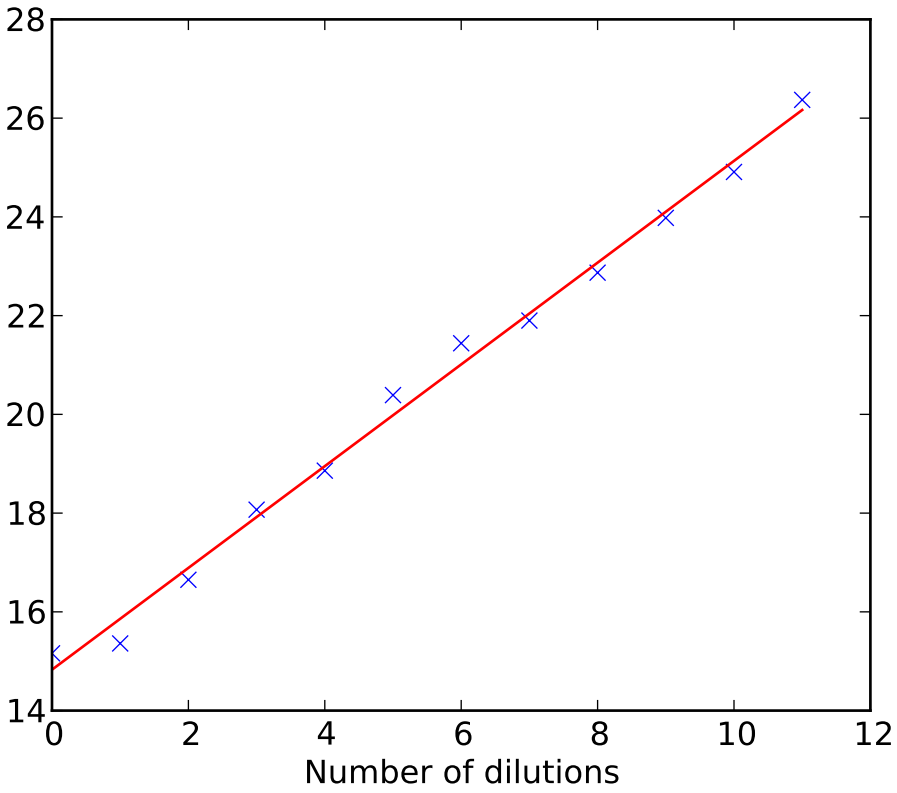
Looking at the plot, a straight line fit seems reasonable, so we will try to do a linear regression using the `scipy.stats.linregress` function that returns the slope, intercept, and measures of how good the fit is (r-value, p-value and standard error)

```
In [6]: from scipy import stats

In [7]: slope, intercept, r_value, p_value, std_err = stats.linregress(x,y)

In [8]: plot(x, intercept + slope*x, 'r-')
```

where the last equation plotted is the fitted line using the familiar equation of the line $y = ax + b$.

The complete program is given below:

```
import numpy
import pylab
from scipy import stats

if __name__ == '__main__':
    xs = numpy.loadtxt('dilution.csv', skiprows=1,
                       delimiter=',', usecols=(0,1))
    x = numpy.log2(xs[:,0])
    y = xs[:,1]
    pylab.plot(x, y, 'x')
    slope, intercept, r_value, p_value, std_err = stats.linregress(x,y)
    pylab.plot(x, intercept + slope*x, 'r-')
    pylab.xlabel('Number of dilutions')
    pylab.show()
```



Let's examine an actual PCR experiment from the `pcr.csv` data set, where the first column contains the number of PCR cycles, and the second the measured intensity. We will load and plot the log2(intensity) - this should be quite familiar now.

```
In [9]: xs = numpy.loadtxt('pcr.csv', skiprows=1, delimiter=',', usecols=(0,1))

In [10]: x = xs[:,0]

In [11]: y = log2(xs[:,1])

In [12]: plot(x, y, 'x')
```

However, looking at the plot, the linear part of the curve where exponential doubling occurs lies approximately between the log intensities of 8 and 15, and we need to extract that part of the data for linear regression. We can use *logical indexing* to extract just that portion - a logical index gives the locations in a numpy array where a test condition is True. For example,

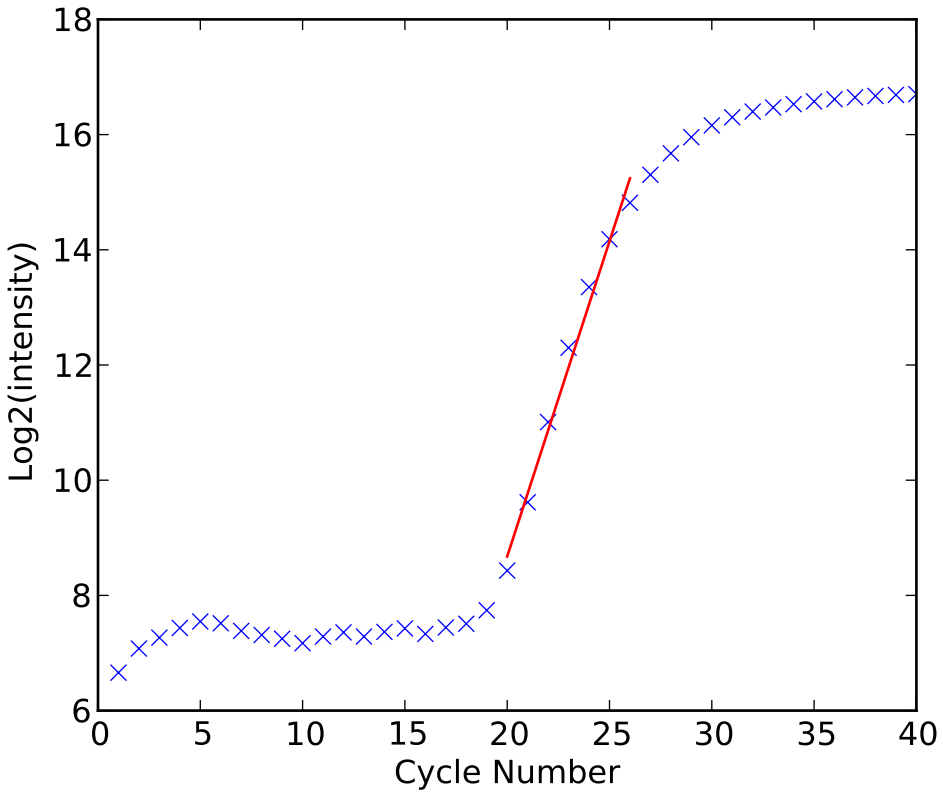
```
In [13]: a = numpy.arange(10)

In [14]: a
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: a > 5
Out[15]: array([False, False, False, False, False, False,  True,  True,  True,  True], dtype=bool)
```

The array returned in `Out[15]` is a logical index of the positions with value > 5. Knowing this, we can easily extract the linear part of the PCR data:

```
In [60]: idx = (y > 8) & (y < 15)

In [61]: x1 = x[idx]

In [62]: y1 = y[idx]

In [63]: slope, intercept, r_value, p_value, std_err = stats.linregress(x1,y1)

In [64]: plot(x1, intercept + slope*x1, 'r-')
```

And the complete program and fitted linear regression is

```
import numpy
import pylab
from scipy import stats

if __name__ == '__main__':
    xs = numpy.loadtxt('pcr.csv', skiprows=1,
                       delimiter=',', usecols=(0,1))
    x = xs[:,0]
    y = numpy.log2(xs[:,1])
    pylab.plot(x, y, 'x')

    # extract the linear part into new variables x1 and y1
    idx = (y > 8) & (y < 15)
    x1 = x[idx]
    y1 = y[idx]

    slope, intercept, r_value, p_value, std_err = stats.linregress(x1,y1)
    pylab.plot(x1, intercept + slope*x1, 'r-')
    pylab.xlabel('Cycle Number')
    pylab.ylabel('Log2(intensity)')
    pylab.show()
```



1.21.1 Exercise

1. Repeat the linear regression fit for columns 1 and 3 of `dilution.csv`
2. Repeat the linear regression fit for columns 1 and 3 of `pcr.csv` - can we reject the null hypothesis that the slope is 2 (i.e. that the measured slope is 1, since log2(2) = 1)?

Next, *Difference equations and linear algebra*.

1.22 Difference equations and linear algebra

When events occur in discrete steps, we can use *difference equations* to model them. A class of such models of interest in immune modeling are equations of the form $x(t+1) = f(x(t))$, where the variable of interest x at time $t+1$ only depends on the value of x at time t. We will briefly review the computational analysis of such *discrete dynamical systems*.

A famous historical example is the model of replicating rabbits proposed by Leonardo of Pisa, also known as Fibonacci. The following description is lifted straight from Wikipedia:

In the West, the sequence was studied by Leonardo of Pisa, known as Fibonacci, in his Liber Abaci (1202).[10] He considers the growth of an idealised (biologically unrealistic) rabbit population, assuming that: a newly-born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?

Let `a` be the number of adult pairs, and `j` the number of juvenile pairs. We start with 1 adult pair. Then the model for the next 10 years is

```
In [12]: a = 1

In [13]: j = 0

In [14]: for t in range(10):
   ....:     a, j = a+j, a
   ....:     print a,
   ....:
   ....:
1 2 3 5 8 13 21 34 55 89
```

If we want to know the number of adult and juvenile pairs simultaneously, we can write a difference equation

$$\begin{pmatrix} a_{t+1} \\ j_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_t \\ j_t \end{pmatrix}$$

Take a while to convince yourself that this equation correctly describes the model described by Fibonacci. It follows that for 2 time steps,

$$\begin{pmatrix} a_{t+2} \\ j_{t+2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{t+1} \\ j_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_t \\ j_t \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} a_t \\ j_t \end{pmatrix}$$

In general, after k generations, the solution will be

$$\begin{pmatrix} a_{t+k} \\ j_{t+k} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} a_t \\ j_t \end{pmatrix}$$

We can, of course, do this by brute force in Python

```
In [22]: r = numpy.mat([[1],[0]])

In [23]: m = numpy.mat([[1,1],[1,0]])

In [24]: m*m*m*m*m*m*m*m*m*m*r
Out[24]:
matrix([[89],
        [55]])
```

Or slightly less hideously as

```
In [25]: numpy.linalg.matrix_power(m, 10)*r
Out[25]:
matrix([[89],
        [55]])
```

However, there is a much more elegant solution using*eigenvectors* and *eigenvalues* (which Tom should hopefully have covered), and this can be done in Python as follows:

```
In [33]: v, e = eig(m)

In [34]: e*diag(v**10)*e.T*r
Out[34]:
matrix([[ 144.],
        [ 89.]])
```

1.22.1 Exercise

A famous difference equation is the *discrete logistic model* $x(t+1) = rx(x-1)$. Write a program to find and plot the steady state values of x after discarding *transient* solutions for a range of values of r between 0 and 4. Specifically, assume the initial value of :math"x is 0.1 for each value of r, calculate and plot $x(500)$ to $x(550)$ against r, ignoring $x(0)$ to $x(499)$. Start with $r = 0$, and increase by 0.01 until $r = 4$. The behavior of the steady state solution as r approaches 4 is very interesting. *Solution to discrete logistic equation exercise*

Next, *Series expansions and function approximations*.

1.23 Series expansions and function approximations

Python is great for numerical computations, but we can also do *symbolic algebra* with the `sympy` library. This is far too large and complex a topic to cover in this session, so I will simply show some examples that may be useful for finding some of the solutions to Tom's problem sets if you've forgotten your calculus.

```
In [1]: from sympy import *

In [2]: x = symbols('x')

In [3]: f = Function("f")

# calculate Taylor series expansions
In [4]: (1/(1-x)).series(x,0,5)
Out[4]: 1 + x + x**2 + x**3 + x**4 + O(x**5)

# find limits as x->0 and as x -> oo (infinity is written as 2 o's)
In [5]: limit(sin(x)/x, x, 0)
Out[5]: 1

In [6]: limit(1/x, x, oo)
Out[6]: 0

# first and second derivatives
In [7]: diff(x**3, x, 1)
Out[7]: 3*x**2

In [8]: diff(x**3, x, 2)
Out[8]: 6*x

# indefinite integrals
In [9]: integrate(6*x)
Out[9]: 3*x**2

In [10]: integrate(3*x**2)
Out[10]: x**3

# definite integrals
In [11]: integrate(sin(x), (x, 0, pi))
Out[11]: 2

# solving algebraic equations
In [12]: a, b, c = symbols('abc')

In [13]: solve(a*x**2 + b*x + c, x)
Out[13]: [-(b + (-4*a*c + b**2)**(1/2))/(2*a), (-b + (-4*a*c + b**2)**(1/2))/(2*a)]
```

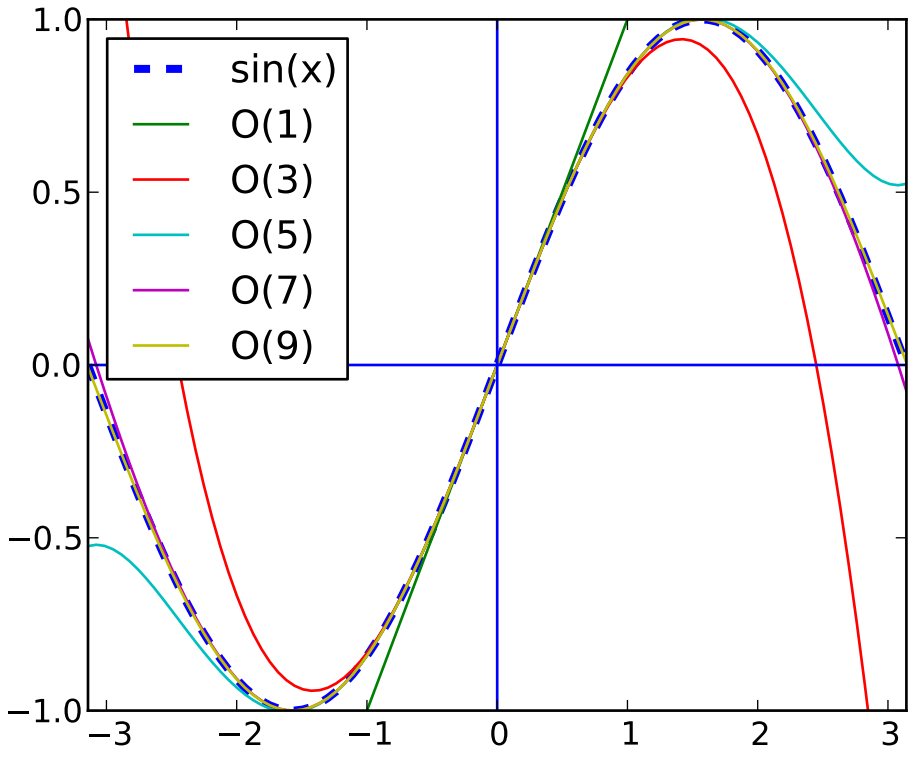
A simple program to illustrate successive approximations to $sin(x)$ using Taylor series.

In [14]: sin(x).series(x, 0, 10)

Out[14]:

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} + O(x^{10})$$

```
from sympy import *
import pylab

x = symbols('x')
sin(x).series(x, 0, 10)

xs = pylab.linspace(-pylab.pi, pylab.pi, 100)
pylab.plot(xs, sin(xs), '--',
           label='sin(x)', linewidth=3)
pylab.plot(xs, xs, '-',
           label='O(1)')
pylab.plot(xs, xs - xs**3./6, '-',
           label='O(3)')
pylab.plot(xs, xs - xs**3./6 + xs**5./120, '-',
           label='O(5)')
pylab.plot(xs, xs - xs**3./6 + xs**5./120 - xs**7./5040,  '-',
           label='O(7)')
pylab.plot(xs, xs - xs**3./6 + xs**5./120 - xs**7./5040 + xs**9./362880, '-',
           label='O(9)')
pylab.axvline(0)
pylab.axhline(0)
pylab.legend(loc='best')
pylab.axis([-pylab.pi, pylab.pi, -1, 1])
pylab.show()
```



1.23.1 Pretty printing with isympy

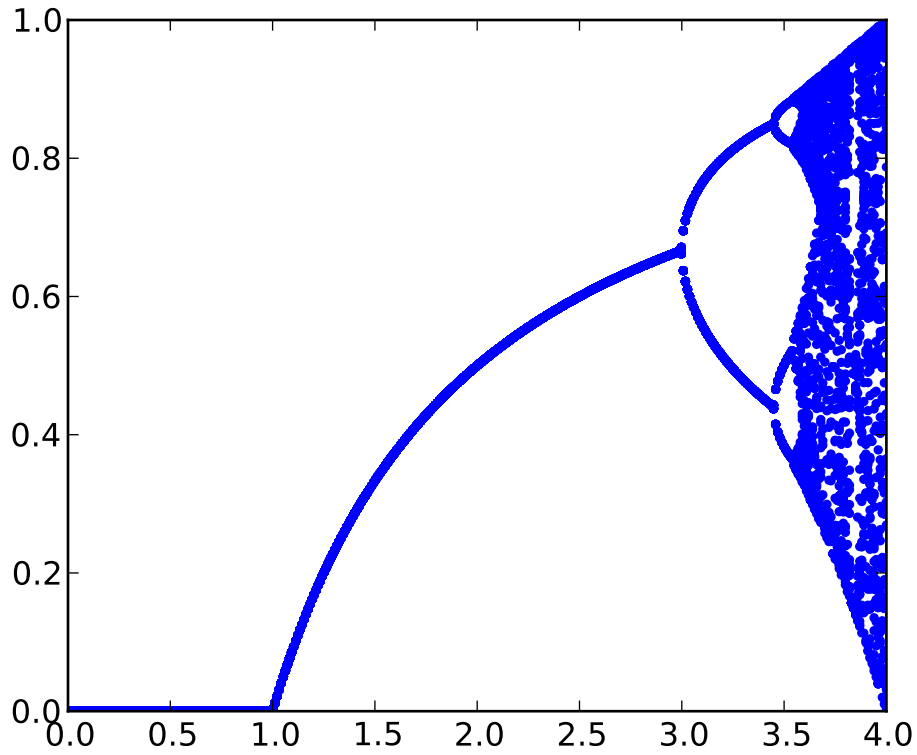
If you install `sympy`, it will also install a program called `isympy` that provides pretty printing of the computer algebra>. Since you can import numpy from within isympy as usual, it is possible to mix and match both numerical and symbolic computation in a single session.

```
paster:flow cliburn$ isympy
Python 2.6.5 console for SymPy 0.6.7

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z = symbols('xyz')
>>> k, m, n = symbols('kmn', integer=True)
>>> f, g, h = map(Function, 'fgh')

Documentation can be found at http://sympy.org/

In [1]: diff(f(x)*g(x), x)
Out[1]:
d               d
--(f(x))g(x) + --(g(x))f(x)
dx              dx
```

Next, *Modeling with ordinary differential equations (ODEs)*.

1.24 Solution to discrete logistic equation exercise

```
import numpy
import pylab

def f(x, r):
    """Discrete logistic equation with parameter r"""
    return r*x*(1-x)

if __name__ == '__main__':
    # initial condition for x
    ys = []
    rs = numpy.linspace(0, 4, 400)
    for r in rs:
        x = 0.1
        for i in range(500):
            x = f(x, r)
        for i in range(50):
            x = f(x, r)
            ys.append([r, x])

    ys = numpy.array(ys)
    pylab.plot(ys[:,0], ys[:,1], '.')
    pylab.show()
```



1.25 Data analysis and visualization

We are going to try to fit a model to experimental data on bacterial growth rates. Bacteria were plated under different conditions, and the number of CFUs on the plate counted at specific time intervals. To keep the CFU density low enough to be counted, bacterial samples were diluted 10-fold if necessary prior to plating.

We start with a simple data set found in `data/cfus.txt`. The file contains data for 2 replicate experiments in a tab-delimited format with 4 columns:

1. Time (mins)
2. CFUs for replicate 1
3. CFUs for replicate 2
4. Number of 10-fold dilutions

As is typical of most experimental data, some pre-processing is necessary to simplify modeling. First, we need to account for the different dilutions. One simple way to put things on the same scale is to assume that the *true* CFU is equal to the *observed* CFU $\times 10^D$, where D is the number of dilutions. It will also be convenient to use $\log_{10}$ CFUs rather than the raw values.

We will explore the data set using `ipython -pylab` from the command line with the `-pylab` flag that makes interactive data visualization more convenient

..sourcecode: ipython

In [1]: t, x1, x2, d = loadtxt('examples/cfu.txt', unpack=True)

Note that when ipython is started with the `-pylab` flag, both `numpy` and `matplotlib` functions are loaded into the global `namespace`, so that we can call `loadtxt` directly instead of `numpy.loadtxt`. We also use `loadtxt` with the extra argument `unpack = True` that *transposes* the data so that individual columns are returned. Because there are 4 such columns, we assign them to `t` (time), `x1` (CFUs in experiment 1), `x2` (CFUs in experiment 2) and `d` (dilurtion factor) respectively.

Now we need to *transform* the observed `x1` and `x2` CFUs to log10 "real" CFUs that we shall call `y1` and `y2` by multiplying by $\times 10^D$ and taking $\log_{10}$ of the resulting values.

..sourcecode: ipython

In [2]: y1 = log10(x1*10**d)

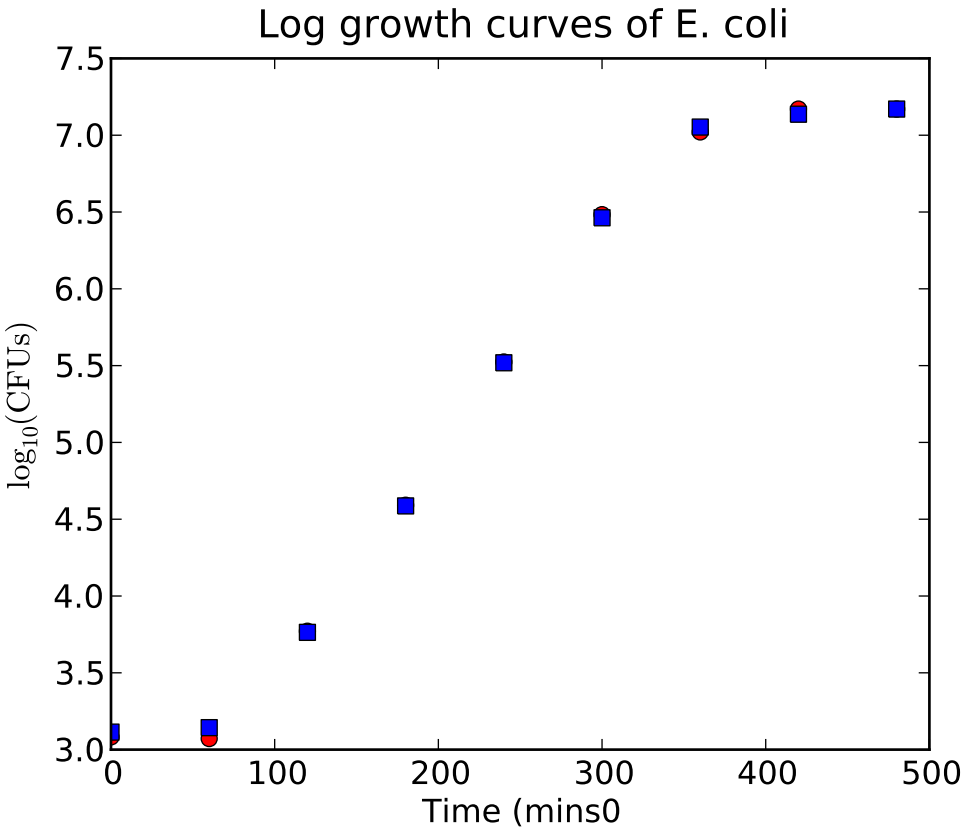
In [3]: y2 = log10(x2*10**d)

Let's see what the data looks like

We ask for two superimposed plots - the first plot is of `t` on the x-axis with `y1` on the y-axis, while the second is also of `t` on the x-axis but with `y2` on the y-axis. For `y1`, we asked that the data be shown as *red circles* using the shorthand `ro` and for `y2`, we get *blue squares* with `bs`. A full python program with comments (lines starting with `#`) is given below.

```
import pylab
import numpy

if __name__ == '__main__':
    # load the data
    t, x1, x2, d = numpy.loadtxt('cfu.txt', unpack=True)

    # transform the observed CFU values
    y1 = numpy.log10(x1*10**d)
    y2 = numpy.log10(x2*10**d)
```

```
# plot the data
pylab.plot(t, y1, 'ro', t, y2, 'bs')
pylab.title('Log growth curves of E. coli')
pylab.xlabel('Time (mins0')
pylab.ylabel('$\log_{10}(\mathrm{CFUs})$')

# show is necessary to display the plot when
# not in interactive mode
pylab.show()
```



We shall next see how to fit a model to the data in *Fitting nonlinear models to data*

1.26 Fitting nonlinear models to data

We will fit a phenomenological **Hill function** to the curve, where

$$f(t) = a + b\frac{t^n}{t^n + k^n}$$

The Hill function generates an S-shaped curve, where t is the variable of interest (time in our example), a is the *baseline*, $a + b$ is the maximum or saturated value, k is the value of t when the function has 50% of its maximal value, and n, the Hill coefficient, determines how steep the slope of the function is at k.

We first need to define the function in Python:

```
def f(t, a, b, k, n):
  return a + b*(t**n)/(t**n + k**n)
```

To fit the Hill function to our bacterial data, we also need another function that tells us how far the fitted curve is away from the actual data. This is known as the **residual** function, and is given by:

```
def resid(p, y, t):
    a, b, k, n = p
    return y - f(t, a, b, k, n)
```

The residual function needs the four parameters a, b, k, n as well as the independent variable t and the experimental data y'. The parameters are given as a list `p` because the optimization routine we will requires the residual function to have only 3 arguments. However, we unpack `p` back into `a, b, c, d` since that is how our Hill function `f(x)` expects its arguments.

Now we use one of the scipy optimization functions to find the Hill function that minimizes the sum of squared distances between the data and the fitted model. This function is in the `scipy.optimize` module and goes by the name `leastsq`. After looking at the help on `leastsq`, we figure out we can fit the Hill function to one set of data $y1$ as follows

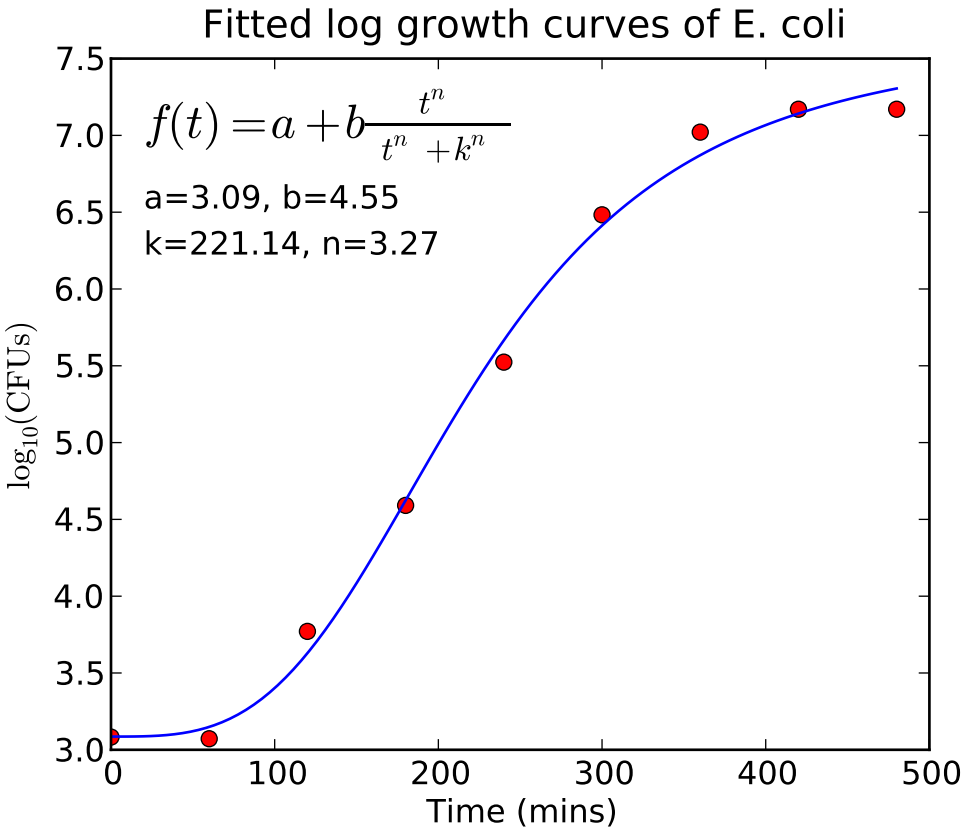
[a, b, k, n], flag = optimize.leastsq(resid, [a0, b0, k0, n0], args=(y1, t))

We need to provide initial values for the parameters as `a0, b0, k0, n0`, and it returns a tuple where the first value is the list or parameters, and the second is a flag that lets us know if the optimization succeeded or not. We can guess reasonable values for `a0, b0, k0, n0` by looking at the data points on the growth plot, but I simply used 1 for all of them out of sheer laziness. Sometimes, the fit may not succeed if the initial guesses are very poor, but in this case it doesn't matter. We unpack the returned list of parameters back into `a, b, c, d`.

A full Python program showing the fitted curve is shown below that incorporates the above code snippets.

```
import pylab
import numpy
from scipy import optimize

def f(t, a, b, k, n):
    return a + b*(t**n)/(t**n + k**n)

def resid(p, y, t):
    a, b, k, n = p
    return y - f(t, a, b, k, n)

if __name__ == '__main__':
    # load the data
    t, x1, x2, d = numpy.loadtxt('cfu.txt', unpack=True)

    # transform the observed CFU values
    y1 = numpy.log10(x1*10**d)
    y2 = numpy.log10(x2*10**d)

    y = numpy.concatenate([y1, y2])

    a0, b0, k0, n0 = 1, 1, 1, 1

    [a, b, k, n], flag  = optimize.leastsq(resid, [a0, b0, k0, n0],
                                           args=(y1, t))

    print flag, a, b, k, n
```

```
# plot the data
pylab.plot(t, y1, 'ro')

# plot the smooth model fit
ts = numpy.linspace(t[0], t[-1], 100)
pylab.plot(ts, f(ts, a, b, k, n))

pylab.title('Fitted log growth curves of E. coli')
pylab.text(20, 6.8,
           r'$f(t) = a +b \frac{t^n}{t^n + k^n}$'  ,
           fontsize=18)
pylab.text(20, 6.5, 'a=%.2f, b=%.2f' % (a, b))
pylab.text(20, 6.2, 'k=%.2f, n=%.2f' % (k, n))
pylab.xlabel('Time (mins)')
pylab.ylabel('$\log_{10}(\mathrm{CFUs})$')

# show is necessary to display the plot when
# not in interactive mode
pylab.show()
```



We shall next how see bacterial growth can be modeled using ordinary differential equations, and how the model solution relates to the phenomenological Hill function taht we used to fit the data in *Modeling with ordinary differential equations (ODEs)*

1.27 Modeling with ordinary differential equations (ODEs)

We will use an extremely simple model of bacterial growth from the classic textbook *Mathematical Models in Biology* by Leah Edelstein-Keshet. This model has the following variables, c is the concentration of nutrients in the medium, n is the concentration of bacteria (assumed to be proportional to the number of CFUs), and that bacteria will grow with a rate $k(c)$ that is directly proportional to the nutrient concentration c - i.e., $k(c) = kc$. We assume that α units of nutrients are consumed in producing one unit of bacteria. With these assumptions, we can set up a system of ODEs to model the system:

$$\begin{array}{ll} \frac{dn}{dt} & = k(c)n \quad = kcn \\ \frac{dc}{dt} & = -\alpha\frac{dn}{dt} \quad = -\alpha kcn \end{array}$$

Actually, this model is so simple that we can solve it analytically with some calculus, but most ODE models of biological phenomena are not solvable in closed form, and we must use **simulation** to study their short term or transient behavior, and **qualitative analysis** to understand their asymptotic solutions. So we will now explore how to simulate ODE systems in Python.

We need to define the ODEs as a function:

def f(y, t, k, alpha):

    return (k*y[0]*y[1], -alpha*k*y[0]*y[1])

where `y` is a the vector `n(t), c(t)` of bacterial and nutrient concentrations at time `t`.

We now need to specify some initial conditions for `y` and the time span over which to simulate, as well as some values for the parameters `k` and `alpha`:

```
y0 = [0.1, 0.1]
t0 = 0
t = range(0, 51)
k=1
alpha = 0.01
```

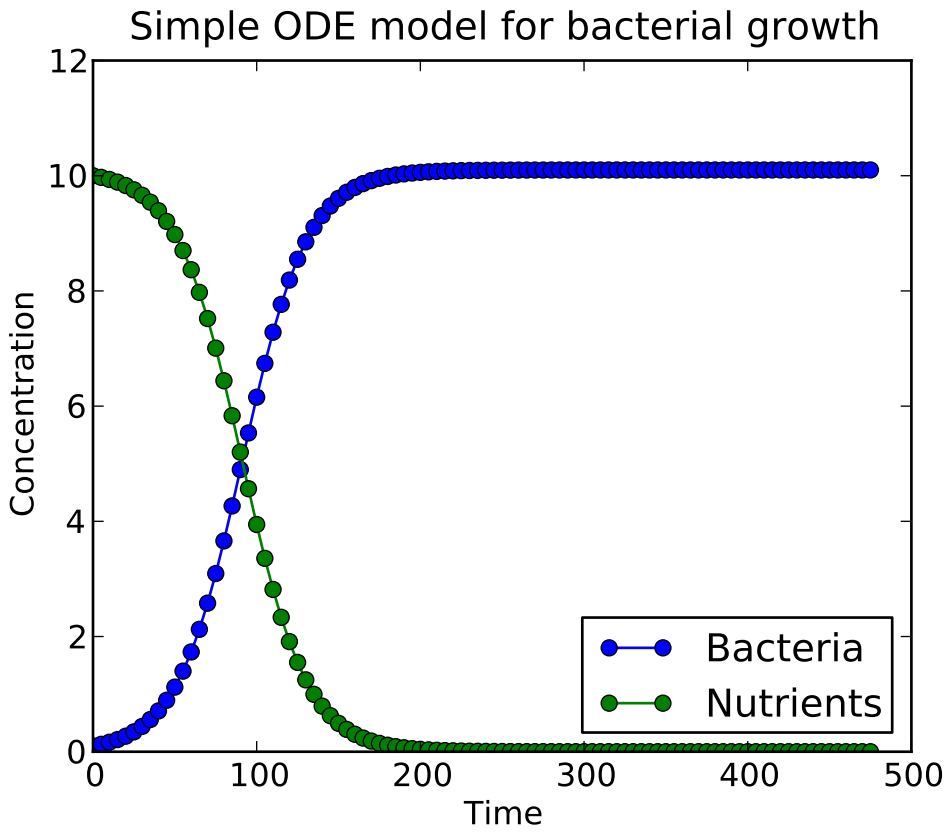
And now we can use the scipy function integrate.odeint to find simulate the model:

```
from scipy import integrate
r = integrate.odeint(f, y0, t, args=(k, alpha))
```

Here is the full Python program:

```
from scipy import integrate
import pylab

def f(y, t, k, alpha):
  return (k*y[0]*y[1],
          -alpha*k*y[0]*y[1])

if __name__ == '__main__':

  y0 = [0.1, 10]
  t = range(0, 480, 5)
  k = 0.005
  alpha = 1

  r = integrate.odeint(f, y0, t, args=(k, alpha))

  pylab.plot(t, r, '-o')
```

```
pylab.legend(['Bacteria', 'Nutrients'], loc='lower right')
pylab.xlabel('Time')
pylab.ylabel('Concentration')
pylab.title('Simple ODE model for bacterial growth')
pylab.show()
```



Next, *Individual/agent based modeling*.

1.27.1 Exercise

- The closed form solution to this system of equations is given by

$$n(t) = \frac{n_0 c_o/\alpha}{n_0 + (c_0/\alpha - n_0)e^{-kc_o t}}$$

where n_0 is the initial bacterial concentration. Try to fit this equation to the bacterial count data. Is there a good fit or not? What are the limitations of this model and how might they be addressed?

- In the book *Virus dynamics: mathematical principles of immunology and virology By Martin A. Nowak, Robert McCredie May*, the basic model of virus infection is given by the equations:

$$\begin{aligned} \frac{dx}{dt} &= \gamma - dx - \beta xv \\ \frac{dy}{dt} &= \beta xv - ay \\ \frac{dv}{dt} &= ky - uv \end{aligned}$$

where x is uninfected cells, y is infected cells, v is virus, γ is the rate of production of new uninfected cells, d is the death rate of uninfected cells, β is the rate of infection, a is the death rate of infected cells, k is the birth rate of new

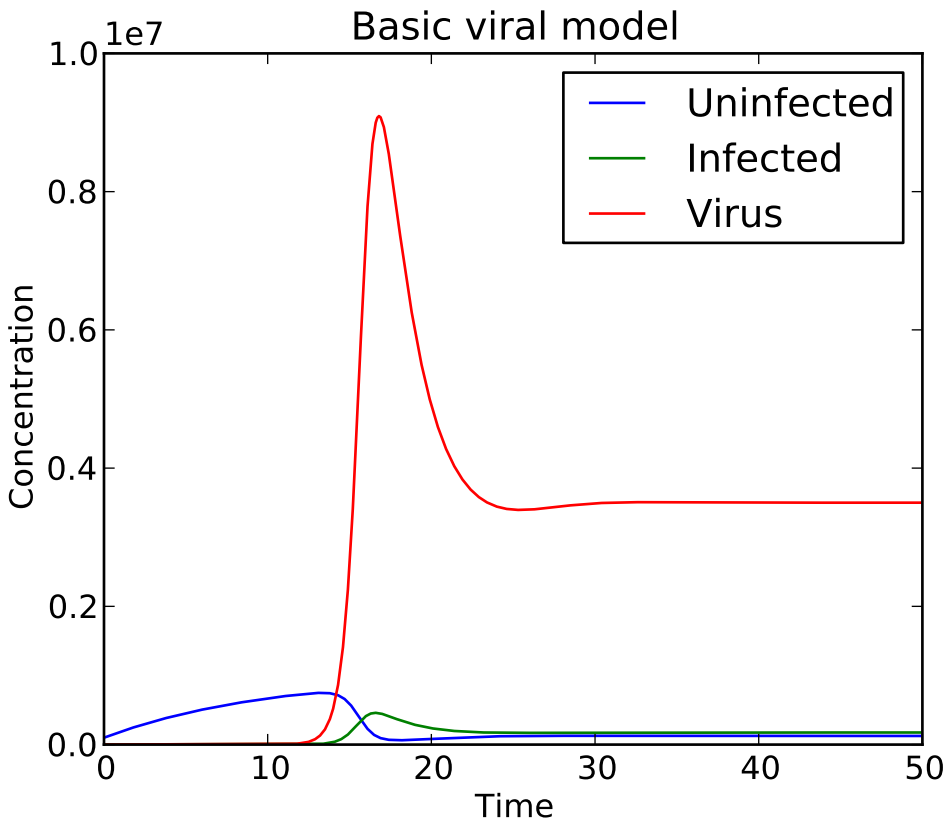
virus, and u is the death rate of virus. Write a program to simulate this system of equations from t=0 to t=50, using the following values:

$$\gamma = 10^5, d = 0.1, a = 0.5, \beta = 2 \times 10^{-7}, k = 100, u = 5$$

Plot the solutions for virus, uninfected and infected cells. *Basic model of virus infection*

1.28 Basic model of virus infection

```
import numpy
from scipy import integrate
import pylab

# x = y[0] = uninfected cells
# y = y[1] = infected cells
# v = y[2] = virus

def f(y, t, gamma, d, a, beta, k, u):
  return (gamma - d*y[0]- beta*y[0]*y[2],
          beta*y[0]*y[2] - a*y[1],
          k*y[1] - u*y[2])

if __name__ == '__main__':

  y0 = [1e5, 0, 1]
  t = numpy.arange(0, 50, 0.1)
  gamma = 1e5
  d = 0.1
  a = 0.5
  beta = 2e-7
  k = 100
  u = 5

  r = integrate.odeint(f, y0, t, args=(gamma, d, a, beta, k, u))

  pylab.plot(t, r, '-')
  pylab.legend(['Uninfected', 'Infected', 'Virus'], loc='upper right')
  pylab.xlabel('Time')
  pylab.ylabel('Concentration')
  pylab.title('Basic viral model')
  pylab.show()
```



1.29 Individual/agent based modeling

We close by noting that there is another class of models that works at the level of individual cells, rather than populations of cells as with difference and differential equations. These models are sometimes known as *individual-* or *agent-based models*, and Matt will be providing an overview on Wednesday.

We will not have time to describe how to construct and program individual based models, but one of the simplest variants of these models is the **cellular automata**, made famous by the Game of Life invented by John Conway.

I will just leave you with two example programs - they use quite a lot of Python optimization trickery to run fast, and may not be easy to understand, but I'd be happy to chat about them if anyone is interested.

1.29.1 The Game of Life

```
"""Game of life in Python with numpy tricks for compuational efficiency.

Note: the program does not show any output, but simply makes a series
of png images, one for each step. For example, if you have the
ImageMagick program installed, type 'animate -delay 100 life*png' to
see a movie with delay of 100ms between framew.
"""

import numpy
import pylab
```

```
import time

shape = (252,252)
old = numpy.random.randint(0,2,shape)
new = numpy.zeros(shape)

num_steps = 100
frames = numpy.zeros([num_steps, shape[0], shape[1]])

print "Life at step: ",
for i in range(num_steps):
    print i,
    frames[i] = old
    num = (old[0:-2,0:-2] + old[0:-2,1:-1] + old[0:-2,2:] +
           old[1:-1,0:-2]                  + old[1:-1,2:] +
           old[2:,0:-2]   + old[2:,1:-1]   + old[2:,2:])
    birth = (num==3) & (old[1:-1,1:-1]==0)
    survive = ((num==2) | (num==3)) & (old[1:-1,1:-1]==1)
    new[1:-1,1:-1][birth|survive]  = 1
    new[1:-1,1:-1][~(birth|survive)] = 0
    old = new
print

print "Making images: ",
for i, f in enumerate(frames):
    print i,
    pylab.matshow(f)
    pylab.savefig('life%06d.png' % i)

print
```

Movie from code **examples/life_movie.avi**

1.29.2 Bacterial colony growth with crowding

```
"""Simulation of bacterial colony patterns with CA

From http://www.math.ubc.ca/~keshet/pubs/BardLeahCA.pdf
JTB 1993, 160:97

"""

import numpy
import pylab

if __name__ == '__main__':
    a1 = 60 # food for growth
    a2 = 10 # food for sustenance
    T = 8 # time interval beteen cell divisions
    delta = 0.01 # fraction diffusing per time step
    theta = 3950 # threshold for new bacterial cell
    # crowding function influences likelihood of new bacteria formation

    nx = 301 # size in x direction
    ny = 301 # size in y direction
    shape = numpy.array([nx, ny])
```

```
nsteps = 1000 # number of time steps

crowd = numpy.array([0,40,40,40,30,20,10,0,0])
food = 100.0*numpy.ones(shape)
cells = numpy.zeros(shape, 'int')
# seed with initial bacterium in center
cells[nx/2, ny/2] = 1

mean = numpy.zeros(shape)
newcells = numpy.zeros(shape, 'int')
nbrs = numpy.zeros(shape, 'int')

k = 0
for t in range(nsteps):
    print t
    # diffuse
    mean[1:-1,1:-1] = (
        food[0:-2,0:-2]+4*food[0:-2,1:-1]  +food[0:-2,2:] +
        4*food[1:-1,0:-2]                  +4*food[1:-1,2:] +
        food[2:,0:-2]  +4*food[2:,1:-1]  +food[2:,2:])/20.0
    eaten = newcells*a1 + cells*a2
    food = (1-delta)*food + delta*mean - eaten
    food[food<0] = 0

    if t%T == 0:
        nbrs[1:-1,1:-1] =  (
            cells[0:-2,0:-2]+cells[0:-2,1:-1]+cells[0:-2,2:] +
            cells[1:-1,0:-2]                 +cells[1:-1,2:] +
            cells[2:,0:-2]  +cells[2:,1:-1]  +cells[2:,2:])
        f = numpy.choose(nbrs, crowd)
        f *= food

        r = numpy.random.random(shape)

        newcells[(f > theta) & (r < 0.5)] = 1
        cells[(cells==1) | (newcells==1)] = 1
        pylab.imsave('img%04d.png' % k, cells)
        k += 1
    # print food
```

Note that the Python programs only generate a series of still images in the PNG format named img0001.png, img0002.png etc. To convert them into a movie, I used the `ffmpeg` program (http://www.ffmpeg.org/download.html):

```
ffmpeg -qscale 5 -r 20 -b 9600 -i img%04d.png bacteria.avi
```

Movie from code **examples/bacteria.avi**

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*
- *glossary*