A Crash Course in Python for Scientists

Rick Muller (http://www.cs.sandia.gov/~rmuller/), Sandia National Laboratories

version 0.5, Released 7/3/2013, Alpha

Modified by Douglas Applegate, Argelander Institute for Astronomy, University of Bonn, January 2014

Further Modified by Carolin Villforth, University of St Andrews, October 2014

Why Python?

Python is the programming language of choice for many scientists to a large degree because it offers a great deal of power to analyze and model scientific data with relatively little overhead in terms of learning, installation or development time. It is a language you can pick up in a weekend, and use for the rest of one's life.

Benefits:

- It's free
- It's everywhere
- Flexible like a scripting language (bash, perl), but can be nearly as fast as C
- No compiling
- Automatic memory management (Nearly forget about Segfaults!)
- Plenty of scientific libraries
- Can call C code (with not too much work).

0. Useful References

Python Language

- Python Tutorial (http://docs.python.org/2/tutorial/) First stop reference for basic tasks (looping, string formating, classes, functions, etc)
- Python Standard Library (http://docs.python.org/2/library/) The definitive reference for everything Python can do out of the box. Many common and not so common tasks are already taken care of.
- Python for Matlab Users (http://mathesaurus.sourceforge.net/matlab-numpy.html)
- Python for IDL Users (http://mathesaurus.sourceforge.net/idl-numpy.html)

Numpy

The Numpy library is the math workhorse of Python. Everything is vectorized, so `a*b+c**d` makes sense if a is a number or an n-dimensional array.

- Numpy Example List (http://wiki.scipy.org/Numpy_Example_List_With_Doc) Examples using every Numpy function. Keep close at hand!
- Numpy for Matlab Users (http://wiki.scipy.org/NumPy_for_Matlab_Users) Know Matlab? Then use this translation guide.
- Numpy Reference (http://docs.scipy.org/doc/numpy/reference/) Main documentation

Scipy

Scipy gathers many useful odds and ends into one place, such as minimizers, statistical functions, fourier transforms, interpolation, etc.

- Scipy Documentation (http://docs.scipy.org/doc/scipy/reference/)

Plotting with Matplotlib

Matplotlib provides Matlab-style plotting for Python.

- Pylab Commands (http://matplotlib.org/api/pyplot_summary.html) Documentation and examples for plotting. Keep close at hand!

Other Useful Documentation

- Astropy (http://docs.astropy.org/en/stable/) Useful tools from the Astronomy Community (FITS files, catalogs, cosmology, VO, etc)
- IPython Notebook (http://ipython.org/ipython-doc/stable/interactive/notebook.html) Documentation for the Notebook interface you are viewing now.
- PyMC (http://pymc-devs.github.io/pymc/) Bayesian Modelling Software and MCMC Sampler

Additional Tutorials and References

- Learning Python the Hard Way (http://learnpythonthehardway.org/book/) Excellent set of basic Python tutorials
- Scientific Python Tutorials (http://jrjohansson.github.io/scientific-python-lectures)
- A collection of Notebooks for using IPython effectively (https://github.com/ipython/ipython/tree/master/examples/notebooks#a-collection-of-notebooks-for-using-ipython-effectively)
- A gallery of interesting IPython Notebooks (https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks)

I. Python Overview

This section covers some basics. You should feel free to evaluate and alter any of the calculations, otherwise

This section covers some basics. You should feel free to evaluate and alter any of the calculations, otherwise you can just skim through this section. None of this should be particularly confusing if you've programmed before.

The lessons that follow make use of the IPython notebooks. There's a good introduction to notebooks in the IPython notebook documentation (http://ipython.org/notebook.html) that even has a nice video (http://www.youtube.com/watch?v=H6dLGQw9yFQ#!) on how to use the notebooks. You should probably also flip through the IPython tutorial (http://ipython.org/ipython-doc/dev/interactive/tutorial.html) in your copious free time.

How to use a Notebook (the thing you are looking at now)

Briefly, notebooks have code cells (that are generally followed by result cells) and text cells. The text cells are the stuff that you're reading now. The code cells start with "In []:" with some number generally in the brackets. If you put your cursor in the code cell and hit Shift-Enter, the code will run in the Python interpreter and the result will print out in the output cell. You can then change things around and see whether you understand what's going on. If you need to know more, see the IPython notebook documentation (http://ipython.org/notebook.html) or the IPython tutorial (http://ipython.org/ipython-doc/dev/interactive/tutorial.html).

Useful keyboard shortcuts:

- `Shift-Enter` : Execute code cell and move to the next cell.
- `Ctrl-Enter` : Execute code cell, but stay in current cell.
- `Ctrl-m h` (Ctrl and m at the same time, then h) : Show list of other keyboard shortcuts.

Using Python as a Calculator

Many of the things I used to use a calculator for, I now use Python for:

```
In [31]: 2+2
```

```
Out[31]: 4
```

```
In [32]: (50-5*6)/4
```

```
Out[32]: 5
```

(If you're typing this into an IPython notebook, or otherwise using a notebook file, you hit shift-Enter to evaluate a cell.)

There are some gotchas compared to using a normal calculator.

```
In [33]: 7/3
```

```
Out[33]: 2
```

But if one of the numers is a float, then divison works fine.

```
In [34]: 7/float(3)

Out[34]: 2.3333333333333335
```

```
In [35]: 7./3   #Notince the dot after the 7. That makes 7 a float.

Out[35]: 2.3333333333333335
```

In the last few lines, we have sped by a lot of things that we should stop for a moment and explore a little more fully. We've seen, however briefly, two different data types: **integers**, also known as *whole numbers* to the non-programming world, and **floating point numbers**, also known (incorrectly) as *decimal numbers* to the rest of the world.

We've also seen the first instance of an **import** statement. Python has a huge number of libraries included with the distribution. To keep things simple, most of these variables and functions are not accessible from a normal Python interactive session. Instead, you have to import the name. For example, there is a **math** module containing many useful functions. To access, say, the square root function, you can either first

```
from math import sqrt
```

and then

```
In [36]: sqrt(81)

Out[36]: 9.0
```

or you can simply import the math library itself

```
In [37]: import math
         math.sqrt(81)

Out[37]: 9.0
```

You can define variables using the equals (=) sign:

```
In [38]: width = 20
         length = 30
         area = length*width
         area

Out[38]: 600
```

If you try to access a variable that you haven't yet defined, you get an error:

```
In [39]: volume

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

```
<ipython-input-39-0c7fc58f9268> in <module>()
----> 1 volume

NameError: name 'volume' is not defined
```

and you need to define it:

```
In [40]: depth = 10
         volume = area*depth
         volume

Out[40]: 6000
```

You can name a variable *almost* anything you want. It needs to start with an alphabetical character or "_", can contain alphanumeric charcters plus underscores ("_"). Certain words, however, are reserved for the language:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Trying to define a variable using one of these will result in a syntax error:

```
In [41]: return = 0

           File "<ipython-input-41-2b99136d4ec6>", line 1
             return = 0
                    ^
         SyntaxError: invalid syntax
```

Strings

Strings are lists of printable characters, and can be defined using either single quotes

```
In [42]: 'Hello, World!'

Out[42]: 'Hello, World!'
```

or double quotes

```
In [43]: "Hello, World!"

Out[43]: 'Hello, World!'
```

But not both at the same time, unless you want one of the symbols to be part of the string.

```
In [44]: "He's a Rebel"
```

```
Out[44]: "He's a Rebel"
```

```
In [45]: 'She asked, "How are you today?"'
```

```
Out[45]: 'She asked, "How are you today?"'
```

Just like the other two data objects we're familiar with (ints and floats), you can assign a string to a variable

```
In [46]: greeting = "Hello, World!"
```

The **print** statement is often used for printing character strings:

```
In [47]: print greeting
```

```
Hello, World!
```

But it can also print data types other than strings:

```
In [48]: print "The area is ",area
```

```
The area is  600
```

In the above snipped, the number 600 (stored in the variable "area") is converted into a string before being printed out.

You can use the + operator to concatenate strings together:

```
In [49]: print "This " + "is " + "a " + "concatenated " + "statement. That is " + str(7)
```

```
This is a concatenated statement. That is 7x too long.
```

If you have a lot of words to concatenate together, there are other, more efficient ways to do this. But this is fine for linking a few strings together. Notice how we had to tell python to treat 7 as a string.

Getting Help

Don't know what an object or function does? There are two quick ways to get help about something.

```
In [50]: alist = [1,4,'a',[]]
         help(alist)
```

```
Help on list object:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
```

```
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __delitem__(...)
 |      x.__delitem__(y) <==> del x[y]
 |
 |  __delslice__(...)
 |      x.__delslice__(i, j) <==> del x[i:j]
 |
```

In [51]:
```
help(len)
```

```
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Perhaps this is easier. Add a `?` after the function or object, and then use 'ctrl-enter'. A subwindow in the notebook will pop open to give you help.

In [52]:
```
len?
```

Lists

Very often in a programming language, one wants to keep a group of similar items together. Python does this using a data type called **lists**.

In [53]:
```
days_of_the_week = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday",
```

You can access members of the list using the **index** of that item:

In [54]:
```
days_of_the_week[2]
```

Out[54]:
```
'Tuesday'
```

Python lists, like C, but unlike Fortran, use 0 as the index of the first element of a list. Thus, in this example, the 0 element is "Sunday", 1 is "Monday", and so on. If you need to access the *n*th element from the end of the list, you can use a negative index. For example, the -1 element of a list is the last element:

In [55]:
```
days_of_the_week[-1]
```

Out[55]:
```
'Saturday'
```

You can also get a range of items from the list.

```
In [56]: days_of_the_week[2:4]   #start at the 2nd element, and stop before the 4th.
```

```
Out[56]: ['Tuesday', 'Wednesday']
```

Or skip along the list.

```
In [57]: days_of_the_week[1:-1:2]   #start at the 1st element, stop before the last eleme
```

```
Out[57]: ['Monday', 'Wednesday', 'Friday']
```

You can add additional items to the list using the .append() command:

```
In [58]: languages = ["Fortran","C","C++"]
         languages.append("Python")
         print languages
```

```
         ['Fortran', 'C', 'C++', 'Python']
```

The **range()** command is a convenient way to make sequential lists of numbers:

```
In [59]: range(10)
```

```
Out[59]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that range(n) starts at 0 and gives the sequential list of integers less than n. If you want to start at a different number, use range(start,stop)

```
In [60]: range(2,8)
```

```
Out[60]: [2, 3, 4, 5, 6, 7]
```

The lists created above with range have a *step* of 1 between elements. You can also give a fixed step size via a third command:

```
In [61]: evens = range(0,20,2)
         evens
```

```
Out[61]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Lists do not have to hold the same data type. For example,

```
In [62]: ["Today",7,99.3,""]
```

```
Out[62]: ['Today', 7, 99.3, '']
```

Iteration, Indentation, and Blocks

One of the most useful things you can do with lists is to *iterate* through them, i.e. to go through each element one at a time. To do this in Python, we use the **for** statement:

In [63]:
```
for day in days_of_the_week:
    statement = "Today is " + day
    print statement
```

```
Today is Sunday
Today is Monday
Today is Tuesday
Today is Wednesday
Today is Thursday
Today is Friday
Today is Saturday
```

This code snippet goes through each element of the list called **days_of_the_week** and assigns it to the variable **day**. It then executes everything in the indented block (in this case only one line of code, the print statement) using those variable assignments. When the program has gone through every element of the list, it exists the block.

(Almost) every programming language defines blocks of code in some way. In Fortran, one uses END statements (ENDDO, ENDIF, etc.) to define code blocks. In C, C++, and Perl, one uses curly braces {} to define these blocks.

Python uses a colon (":"), followed by indentation level to define code blocks. Everything at a higher level of indentation is taken to be in the same block.

White space and indents matter in Python!

The **range()** command is particularly useful with the **for** statement to execute loops of a specified length:

In [64]:
```
for i in range(20):
    print "The square of ",i," is ",i*i
```

```
The square of  0  is  0
The square of  1  is  1
The square of  2  is  4
The square of  3  is  9
The square of  4  is  16
The square of  5  is  25
The square of  6  is  36
The square of  7  is  49
The square of  8  is  64
The square of  9  is  81
The square of  10  is  100
The square of  11  is  121
The square of  12  is  144
The square of  13  is  169
The square of  14  is  196
The square of  15  is  225
The square of  16  is  256
The square of  17  is  289
The square of  18  is  324
The square of  19  is  361
```

We can combine range with looping over a list with the **enumerate** command.

```
In [65]: for i, day in enumerate(days_of_the_week):
             statement = day + " is day " + str(i) + " of " + str(len(days_of_the_week))
             print statement
```

```
Sunday is day 0 of 7.
Monday is day 1 of 7.
Tuesday is day 2 of 7.
Wednesday is day 3 of 7.
Thursday is day 4 of 7.
Friday is day 5 of 7.
Saturday is day 6 of 7.
```

Booleans and Truth Testing

We have now learned a few data types. We have integers and floating point numbers, strings, and lists to contain them. We have also learned about lists, a container that can hold any data type. We have learned to print things out, and to iterate over items in lists. We will now learn about **boolean** variables that can be either True or False.

We invariably need some concept of *conditions* in programming to control branching behavior, to allow a program to react differently to different situations. If it's Monday, I'll go to work, but if it's Sunday, I'll sleep in. To do this in Python, we use a combination of **boolean** variables, which evaluate to either True or False, and **if** statements, that control branching based on boolean values.

For example:

```
In [66]: if day == "Sunday":
             print "Sleep in"
         elif day == "Saturday":
             print "Do chores"
         else:
             print "Go to work"
```

```
Do chores
```

(Quick quiz: why did the snippet print "Go to work" here? What is the variable "day" set to?

Hint: When was the last time you used `day`?

You can compare any data types in Python:

```
In [67]: 1 == 2
```

```
Out[67]: False
```

```
In [68]: 50 == 2*25
```

```
Out[68]: True
```

```
In [69]: 3 < 3.14159
```

```
Out[69]: True
```

```
In [70]: 1 == 1.0
```

```
Out[70]: True
```

```
In [71]: 1 != 0
```

```
Out[71]: True
```

```
In [72]: 1 <= 2
```

```
Out[72]: True
```

```
In [73]: 1 >= 1
```

```
Out[73]: True
```

We see a few other boolean operators here, all of which which should be self-explanatory. Less than, equality, non-equality, and so on.

Particularly interesting is the 1 == 1.0 test, which is true, since even though the two objects are different data types (integer and floating point number), they have the same *value*. There is another boolean operator **is**, that tests whether two objects are the same object:

```
In [74]: 1 is 1.0
```

```
Out[74]: False
```

We can do boolean tests on lists as well:

```
In [75]: [1,2,3] == [1,2,4]
```

```
Out[75]: False
```

```
In [76]: [1,2,3] < [1,2,4]
```

```
Out[76]: True
```

Finally, note that you can also string multiple comparisons together, which can result in very intuitive tests:

```
In [77]: hours = 5
         0 < hours < 24
```

```
Out[77]: True
```

This is something of an advanced topic, but ordinary data types have boolean values associated with them, and, indeed, in early versions of Python there was not a separate boolean object. Essentially, anything that was a 0 value (the integer or floating point 0, an empty string "", or an empty list []) was False, and everything else was true. You can see the boolean value of any data object using the **bool()** function.

```
In [78]: bool(1)

Out[78]: True
```

```
In [79]: bool(0)

Out[79]: False
```

```
In [80]: bool(["This "," is "," a "," list"])

Out[80]: True
```

Functions

We have already used a number of functions, such as `range`, `len`, and `bool`. A function is a collection of code that is cann be called by typing the name of the function followed by (). To define a function, we use `def`. Lets have a look at a simple function.

```
In [81]: def sayhello():
             print 'Hello, Python here!'
```

Once our function is defined, we can call it like any other function.

```
In [82]: sayhello()

         Hello, Python here!
```

But so far we've always passed arguments on to the functions. For example, we typed range(10) and the function used the input, how do we do that?

```
In [83]: def sayhello_v2(name):
             print 'Hello %s, Python here!' % name
```

Let's see what it does.

```
In [84]: sayhello_v2('you')

         Hello you, Python here!
```

We can also pass several parameters on to the function.

```
In [85]: def addnumbers(a, b):
             output = a + b
```

```
    print '%s + %s = %s' % (a, b, output)
    return output
```

See the return on the last line? What this does is return the results, rather than just print it out so it can be used further in the code.

```
In [86]: addnumbers(12, 13)

         12 + 13 = 25

Out[86]: 25
```

See the 'Out' showing the result, this is what return does, but we can use this in a smart way.

```
In [87]: result = addnumbers(12, 13)

         12 + 13 = 25

In [88]: result

Out[88]: 25
```

Lets see what else we can do with functions:

```
In [89]: def linearfunct(x, slope, intercept=0):
             y = intercept + slope * x
             return y
```

As you see, this function now has three parameters, but one of them is set already. This can be useful for increasing the flexibility of functions without making them too difficult to use.

```
In [90]: linearfunct(2, 0.5)

Out[90]: 1.0

In [91]: linearfunct(2, 0.5, 4)

Out[91]: 5.0
```

We can also call the function like this:

```
In [92]: linearfunct(x=2, slope=1.3, intercept=14)

Out[92]: 16.6

In [93]: linearfunct(intercept=12, x=3.16, slope=13)

Out[93]: 53.08
```

This is what happens when a function is called incorrectly:

```
In [94]: linearfunct(intercept=14)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-94-5895107c3571> in <module>()
----> 1 linearfunct(intercept=14)

TypeError: linearfunct() takes at least 2 arguments (1 given)
```

The function was expecting at least x and slope, but only got one argument.

```
In [95]: linearfunct(intercept=14, 7, 3.8)
```

```
  File "<ipython-input-95-704d4cf686af>", line 1
    linearfunct(intercept=14, 7, 3.8)
SyntaxError: non-keyword arg after keyword arg
```

This does not work either because the function expects the arguments given without specifying the keyword to be given first.

Example: The Fibonacci Sequence

The Fibonacci sequence (http://en.wikipedia.org/wiki/Fibonacci_number) is a sequence in math that starts with 0 and 1, and then each successive entry is the sum of the previous two. Thus, the sequence goes 0,1,1,2,3,5,8,13,21,34,55,89,...

Use what you have learned to calculate the Fibonacci sequence. Write a function that takes an integer n and returns the first n elements of the Fibonacci sequence.

```
In [96]: def fibonacci(n):
             '''Compute the fibonacci sequence : n_i = n_(i-1) + n_(i-2)'''

             if n == 0:
                 return [0]
             if n == 1:
                 return [0,1]
             seq = [0,1]
             for i in range(2, n):
                 seq.append(seq[-2] + seq[-1])

             return seq
```

```
In [97]: ### Testing code ###
         fib10 = fibonacci(10)
         assert(type(fib10) == list and fib10 == [0,1,1,2,3,5,8,13,21,34])
```

Two More Data Structures: Tuples and Dictionaries

Before we end the Python overview, I wanted to touch on two more data structures that are very useful (and thus very common) in Python programs.

A **tuple** is a sequence object like a list or a string. It's constructed by grouping a sequence of objects together with commas, either without brackets, or with parentheses:

```
In [98]: t = (1,2,'hi',9.0)
         t
```

```
Out[98]: (1, 2, 'hi', 9.0)
```

Tuples are like lists, in that you can access the elements using indices:

```
In [99]: t[1]
```

```
Out[99]: 2
```

However, tuples are *immutable*, you can't append to them or change the elements of them:

```
In [100]: t.append(7)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-100-50c7062b1d5f> in <module>()
----> 1 t.append(7)

AttributeError: 'tuple' object has no attribute 'append'
```

```
In [101]: t[1]=77
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-101-03cc8ba9c07d> in <module>()
----> 1 t[1]=77

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

Dictionaries are an object called "mappings" or "associative arrays" in other languages.

The index in a dictionary is called the *key*, and the corresponding dictionary entry is the *value*. A dictionary can use (almost) anything as the key. Whereas lists are formed with square brackets [], dictionaries use curly brackets {}:

```
In [102]: animalfood = {"Giraffe": "Leaves", "Lion": "Zebras", "Cat": "Mice"}
```

Lets now access one of the entries in the dictionary.

```
In [103]: animalfood["Lion"]
```

```
Out[103]: 'Zebras'
```

How can we figure out what is in a dictionary?

```
In [104]: animalfood.keys()
```

```
Out[104]: ['Lion', 'Giraffe', 'Cat']
```

Note that the order of the animals here is different from the order in the definition above, this is because a dictionary has no specific order. Therefore, we also cannot access the "first" entry in a dictionary like we would for a list or tuple.

```
In [105]: animalfood[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-105-d8fc453996f2> in <module>()
----> 1 animalfood[0]

KeyError: 0
```

```
In [106]: len(t)
```

```
Out[106]: 4
```

```
In [107]: len(animalfood)
```

```
Out[107]: 3
```

Plotting with Matplotlib

We can generally understand trends in data by using a plotting program to chart it. Python has a wonderful plotting library called Matplotlib (http://matplotlib.sf.net). The IPython notebook interface we are using for these notes has that functionality built in.

As an example, we have looked at two different functions, the Fibonacci function, and the exponential function, both of which grow faster than polynomially. Which one grows the fastest? Let's plot them. First, let's generate the Fibonacci sequence of length 20:

```
In [108]: fibs = fibonacci(10)
```

Next lets generate the exponentials.

```
In [109]: from math import exp
```

```
In [110]: exps = [exp(i) for i in range(10)]
```

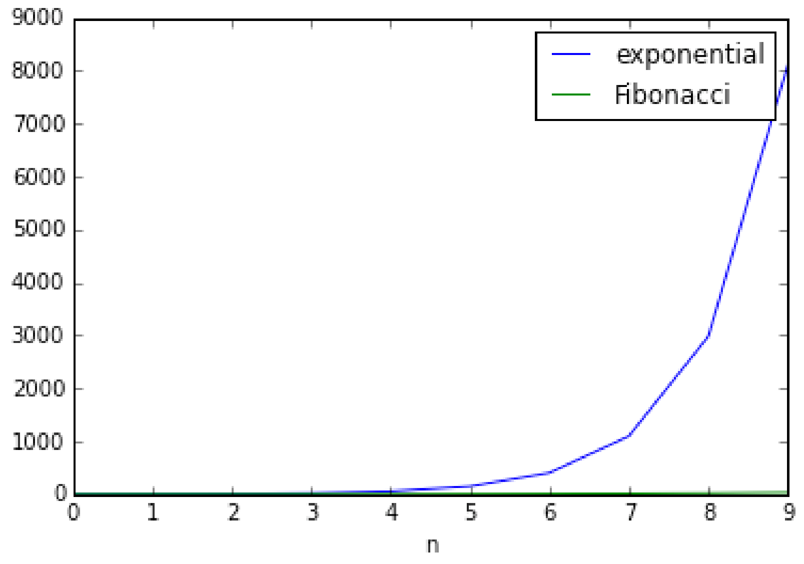
Oh wow, that looks weird! It's called a 'list comprehension.' Think of it like mathematical set notation.

Now we use the Matplotlib function **plot** to compare the two.

```
In [111]: import pylab
```

```
In [112]: pylab.plot(exps,label="exponential")
          pylab.plot(fibs,label="Fibonacci")
          pylab.xlabel("n")
          pylab.legend()
```

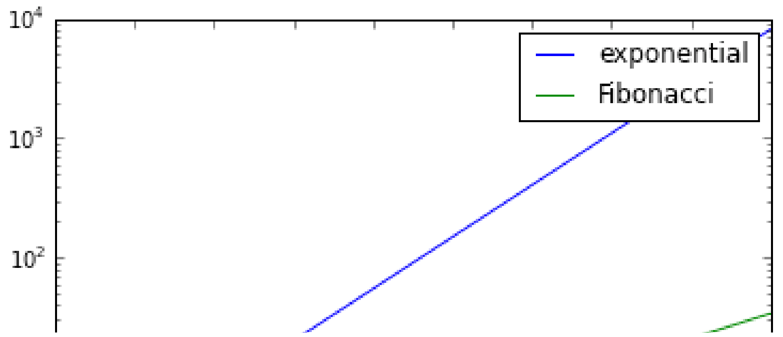
```
Out[112]: <matplotlib.legend.Legend at 0x10e5a9d10>
```

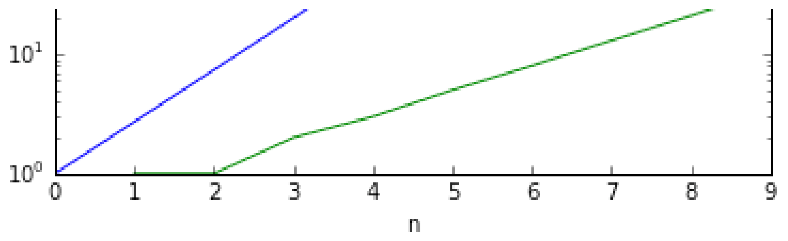


The exponential function grows much faster. In fact, you can barely see the Fibonacci sequence. Let's plot these on a semilog plot so we can see them both a little more clearly:

```
In [113]: pylab.semilogy(exps,label="exponential")
          pylab.semilogy(fibs,label="Fibonacci")
          pylab.xlabel("n")
          pylab.legend()
```

```
Out[113]: <matplotlib.legend.Legend at 0x10e550490>
```





There are many more things you can do with Matplotlib. We'll be looking at some of them in the sections to come. In the meantime, if you want an idea of the different things you can do, look at the Matplotlib Gallery (http://matplotlib.org/gallery.html). Rob Johansson's IPython notebook Introduction to Matplotlib (http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-4-Matplotlib.ipynb) is also particularly good.

II. Numpy and Scipy

Numpy (http://numpy.org) contains core routines for doing fast vector, matrix, and linear algebra-type operations in Python. Scipy (http://scipy) contains additional routines for optimization, special functions, and so on. Both contain modules written in C and Fortran so that they're as fast as possible. Together, they give Python roughly the same capability that the Matlab (http://www.mathworks.com/products/matlab/) program offers. (In fact, if you're an experienced Matlab user, there a guide to Numpy for Matlab users (http://www.scipy.org/NumPy_for_Matlab_Users) just for you.)

Making vectors and matrices

Fundamental to both Numpy and Scipy is the ability to work with vectors and matrices. You can create vectors from lists using the **array** command. But before that, we need to import numpy.

```
In [114]: import numpy
```

```
In [115]: numpy.array([1,2,3,4,5,6])

Out[115]: array([1, 2, 3, 4, 5, 6])
```

You can pass in a second argument to **array** that gives the numeric type. There are a number of types listed here (http://docs.scipy.org/doc/numpy/user/basics.types.html) that your matrix can be. Some of these are aliased to single character codes. The most common ones are 'd' (double precision floating point number), 'D' (double precision complex number), and 'i' (int32). Thus,

```
In [116]: numpy.array([1,2,3,4,5,6],'d')

Out[116]: array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

```
In [117]: numpy.array([1,2,3,4,5,6],'D')

Out[117]: array([ 1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j,  5.+0.j,  6.+0.j])
```

```
In [118]: numpy.array([1,2,3,4,5,6],'i')
```

```
Out[118]: array([1, 2, 3, 4, 5, 6], dtype=int32)
```

To build matrices, you can either use the array command with lists of lists:

```
In [119]: numpy.array([[0,1],[1,0]],'d')
```

```
Out[119]: array([[ 0.,  1.],
                 [ 1.,  0.]])
```

You can also form empty (zero) matrices of arbitrary shape (including vectors, which Numpy treats as vectors with one row), using the **zeros** command:

```
In [120]: numpy.zeros((3,3),'d')
```

```
Out[120]: array([[ 0.,  0.,  0.],
                 [ 0.,  0.,  0.],
                 [ 0.,  0.,  0.]])
```

The first argument is a tuple containing the shape of the matrix, and the second optional argument is the data type argument, which follows the same conventions as in the array command. Thus, you can make vectors and columns of any size.

```
In [121]: numpy.zeros(3)
```

```
Out[121]: array([ 0.,  0.,  0.])
```

```
In [122]: numpy.zeros((2,3))
```

```
Out[122]: array([[ 0.,  0.,  0.],
                 [ 0.,  0.,  0.]])
```

```
In [123]: numpy.zeros((3,4,2))
```

```
Out[123]: array([[[ 0.,  0.],
                  [ 0.,  0.],
                  [ 0.,  0.],
                  [ 0.,  0.]],

                 [[ 0.,  0.],
                  [ 0.,  0.],
                  [ 0.,  0.],
                  [ 0.,  0.]],

                 [[ 0.,  0.],
                  [ 0.,  0.],
                  [ 0.,  0.],
                  [ 0.,  0.]]])
```

There's also an **identity** command that behaves as you'd expect:

```
In [124]: numpy.identity(4,'d')
```

```
Out[124]: array([[ 1.,  0.,  0.,  0.],
                 [ 0.,  1.,  0.,  0.],
                 [ 0.,  0.,  1.,  0.],
                 [ 0.,  0.,  0.,  1.]])
```

as well as a **ones** command.

Linspace, matrix functions, and plotting

The **linspace** command makes a linear array of points from a starting to an ending value.

```
In [125]: numpy.linspace(0,1)
```

```
Out[125]: array([ 0.        ,  0.02040816,  0.04081633,  0.06122449,  0.08163265,
                  0.10204082,  0.12244898,  0.14285714,  0.16326531,  0.18367347,
                  0.20408163,  0.2244898 ,  0.24489796,  0.26530612,  0.28571429,
                  0.30612245,  0.32653061,  0.34693878,  0.36734694,  0.3877551 ,
                  0.40816327,  0.42857143,  0.44897959,  0.46938776,  0.48979592,
                  0.51020408,  0.53061224,  0.55102041,  0.57142857,  0.59183673,
                  0.6122449 ,  0.63265306,  0.65306122,  0.67346939,  0.69387755,
                  0.71428571,  0.73469388,  0.75510204,  0.7755102 ,  0.79591837,
                  0.81632653,  0.83673469,  0.85714286,  0.87755102,  0.89795918,
                  0.91836735,  0.93877551,  0.95918367,  0.97959184,  1.        ])
```

If you provide a third argument, it takes that as the number of points in the space. If you don't provide the argument, it gives a length 50 linear space.

```
In [126]: numpy.linspace(0,1,11)
```

```
Out[126]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

linspace is an easy way to make coordinates for plotting. Functions in the numpy library (all of which are imported into IPython notebook) can act on an entire vector (or even a matrix) of points at once. Thus,

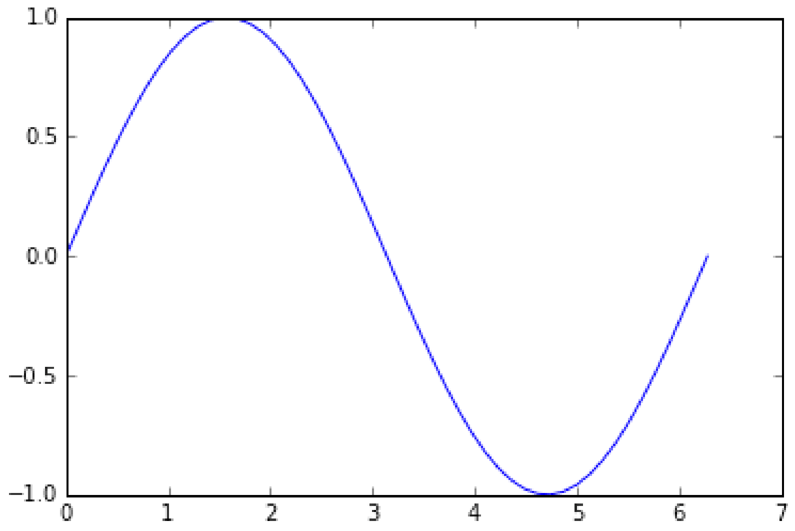
```
In [127]: x = numpy.linspace(0,2*pi)
          numpy.sin(x)
```

```
Out[127]: array([  0.00000000e+00,   1.27877162e-01,   2.53654584e-01,
                   3.75267005e-01,   4.90717552e-01,   5.98110530e-01,
                   6.95682551e-01,   7.81831482e-01,   8.55142763e-01,
                   9.14412623e-01,   9.58667853e-01,   9.87181783e-01,
                   9.99486216e-01,   9.95379113e-01,   9.74927912e-01,
                   9.38468422e-01,   8.86599306e-01,   8.20172255e-01,
                   7.40277997e-01,   6.48228395e-01,   5.45534901e-01,
                   4.33883739e-01,   3.15108218e-01,   1.91158629e-01,
                   6.40702200e-02,  -6.40702200e-02,  -1.91158629e-01,
                  -3.15108218e-01,  -4.33883739e-01,  -5.45534901e-01,
                  -6.48228395e-01,  -7.40277997e-01,  -8.20172255e-01,
                  -8.86599306e-01,  -9.38468422e-01,  -9.74927912e-01,
                 -9.95379113e-01,  -9.99486216e-01,  -9.87181783e-01,
                 -9.58667853e-01,  -9.14412623e-01,  -8.55142763e-01,
                 -7.81831482e-01,  -6.95682551e-01,  -5.98110530e-01,
                 -4.90717552e-01,  -3.75267005e-01,  -2.53654584e-01,
                 -1.27877162e-01,  -2.44929360e-16])
```

In conjunction with **matplotlib**, this is a nice way to plot things:

```
In [128]: pylab.plot(x,sin(x))
```

```
Out[128]: [<matplotlib.lines.Line2D at 0x10e9f41d0>]
```



Array operations

Array objects act sensibly when multiplied by scalars:

```
In [129]: 0.125*numpy.identity(3,'d')
```

```
Out[129]: array([[ 0.125,  0.   ,  0.   ],
                 [ 0.   ,  0.125,  0.   ],
                 [ 0.   ,  0.   ,  0.125]])
```

as well as when you add two arrays together. (However, the arrays have to be the same shape.)

```
In [130]: numpy.identity(2,'d') + numpy.array([[1,1],[1,2]])
```

```
Out[130]: array([[ 2.,  1.],
                 [ 1.,  3.]])
```

Something that confuses Matlab users is that the times (*) operator give element-wise multiplication rather than matrix multiplication:

```
In [131]: numpy.identity(2)*numpy.ones((2,2))
```

```
Out[131]: array([[ 1.,  0.],
                [ 0.,  1.]])
```

Note: Numpy does 'by element' operations, not linear algebra!

(unless you tell it do do linear algebra)

Let's see whether this works for our sin example from above:

III. Intermediate Python

Input Parsing

Let us read in a text file. Included with this notebook is a text file containing the quasar catalog.

```
In [132]: data = numpy.loadtxt('/Users/cvillforth/Teaching/PythonTutorial/FirstYearTutoria
```

numpy.loadtxt is a function that loads text files.

Have a look at the file you loaded. You'll see it contains several lines starting with '#'. numpy.loadtxt ignores all those lines and treats them as comments, rather than data.

Let's see how the data got stored in memory:

```
In [133]: data

Out[133]: array([[ -2.32500000e+01,   0.00000000e+00,   0.00000000e+00, ...,
                    3.17486000e+03,   5.45700000e-01,   1.99000000e+01],
                 [ -2.27126000e+01,   0.00000000e+00,   0.00000000e+00, ...,
                    4.35346000e+03,   5.51200000e-01,   2.01990000e+01],
                 [ -2.51981000e+01,   0.00000000e+00,   0.00000000e+00, ...,
                    4.80353000e+03,   5.28200000e-01,   1.76740000e+01],
                 ...,
                 [ -2.25556000e+01,   0.00000000e+00,   0.00000000e+00, ...,
                    3.80426000e+03,   5.41200000e-01,   2.12340000e+01],
                 [ -2.35384000e+01,   0.00000000e+00,   0.00000000e+00, ...,
                    2.76996000e+03,   5.70000000e-01,   1.94240000e+01],
                 [ -2.40940000e+01,   0.00000000e+00,   0.00000000e+00, ...,
                    2.79901000e+03,   5.70200000e-01,   1.89700000e+01]])
```

The function stored all the information in the file into a numpy array. But how can we access a line or column?

First, let's see how many lines and columns there are.

```
In [134]: numpy.shape(data)
```

```
Out[134]: (7211, 9)
```

What does this mean? numpy.shape shows us the shape of the array. The first number indicates the number of lines, the second number is the number of columns.

Now let's have a look at the first line.

```
In [135]: data[0]
```

```
Out[135]: array([ -2.32500000e+01,   0.00000000e+00,   0.00000000e+00,
                  -1.00000000e+00,   4.44274000e+01,   4.42860000e+01,
                   3.17486000e+03,   5.45700000e-01,   1.99000000e+01])
```

Remember that Python counts from 0, so this is the first line. Now lets access the first entry of the first line.

```
In [136]: data[0,0]
```

```
Out[136]: -23.25
```

What did we do here?

The [] indicates that we are accessing parts of the array, the number before the comma is the index of the line we have choosen, the number after the comma is the index of the column we have choosen.

So, how do we choose all data in the first column?

```
In [137]: data[:,0]
```

```
Out[137]: array([-23.25  , -22.7126, -25.1981, ..., -22.5556, -23.5384, -24.094 ])
```

Compare this to the syntax in the line above. We now replaced the 0 before the comma with a ':'. The ':' means 'all entries'. So, to choose the first line, we could also do this:

```
In [138]: data[0,:]
```

```
Out[138]: array([ -2.32500000e+01,   0.00000000e+00,   0.00000000e+00,
                  -1.00000000e+00,   4.44274000e+01,   4.42860000e+01,
                   3.17486000e+03,   5.45700000e-01,   1.99000000e+01])
```

We can also choose data in a more complex manner. For example, lets choose the first three columns and the last line.

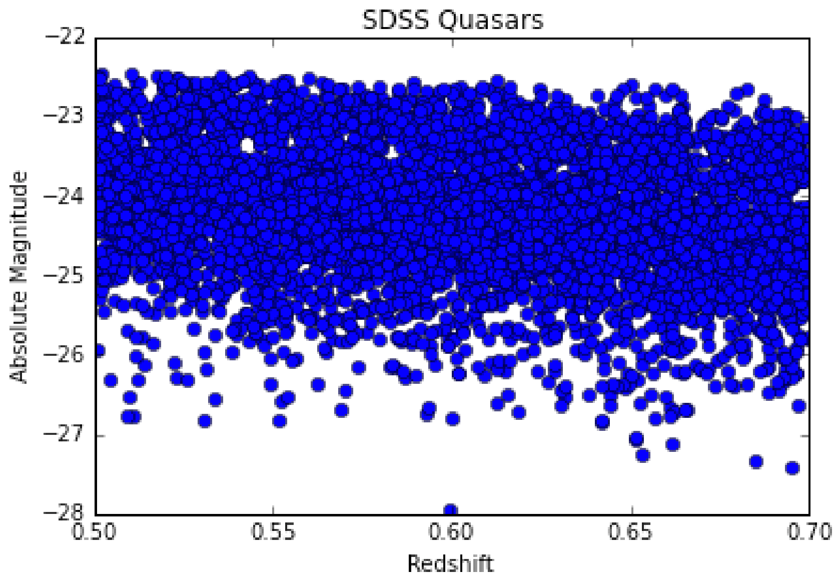
```
In [139]: data[0:3,-1]
```

```
Out[139]: array([ 19.9  ,  20.199,  17.674])
```

Now we have learned how to access the data. Lets now plot it!

```
In [140]: pylab.plot(data[:,7], data[:,0], 'bo')
          pylab.xlabel('Redshift')
          pylab.ylabel('Absolute Magnitude')
          pylab.title('SDSS Quasars')
```

```
Out[140]: <matplotlib.text.Text at 0x10e9a2b90>
```



Smart Array Access

Here is a great feature of working with Numpy arrays. What if we want to look at only part of the data? Let's print out all redshifts z > 0.6 in the table.

```
In [141]: data[:,7][data[:,7] > 0.6]
```

```
Out[141]: array([ 0.6521,  0.6513,  0.6598, ...,  0.6965,  0.6912,  0.683 ])
```

What happened here?

data[:,7] is the column containing the redshift data. But what does the expression [data[:,7] > 0.6] mean? Let's have a look.

```
In [142]: data[:,7] > 0.6
```

```
Out[142]: array([False, False, False, ..., False, False, False], dtype=bool)
```

This expression return an array which contains a 'False' value for every entry for which the expression is true and a 'True' value for every entry for which the expression is true. This array acts as a mask. When we write it after the redshift array, all the values that are not > 0.6 are masked out.

This is a powerful tool we can use to deal with data.

Let's have a look at a simple example.

```
In [143]: maskarray = numpy.array([0,14,0,13,25,79,0])
          maskarray
```

```
Out[143]: array([ 0, 14,  0, 13, 25, 79,  0])
```

We now have an array that contains some zeros. First, let's create a mask for the zeros.

```
In [144]: mask = maskarray == 0
          mask
```

```
Out[144]: array([ True, False,  True, False, False, False,  True], dtype=bool)
```

```
In [145]: maskarray[mask]
```

```
Out[145]: array([0, 0, 0])
```

Great! It worked. But we can use this also to replace all '0' entries in the array with another value, like this:

```
In [146]: maskarray[mask] = -999
          maskarray
```

```
Out[146]: array([-999,   14, -999,   13,   25,   79, -999])
```

Basic Statistics

We've learned to plot and manipulate data. Now we can do some simple calculations:

numpy and scipy have a large number of functions for statistics.

Let's start with something simple. What is the mean and standard deviation of the data in an array? What is the minimum and maximum?

```
In [147]: numpy.min(data[:, 7])
```

```
Out[147]: 0.5
```

```
In [148]: numpy.max(data[:,7])
```

```
Out[148]: 0.69999999999999996
```

```
In [149]: numpy.mean(data[:,7])
```

```
Out[149]: 0.60053003744279465
```

```
In [150]: numpy.std(data[:,7])
```

```
Out[150]: 0.056896074602847943
```

We can also do more complex things. Let's fit a line using numpy.polyfit (check the documentation here http://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html (http://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html))

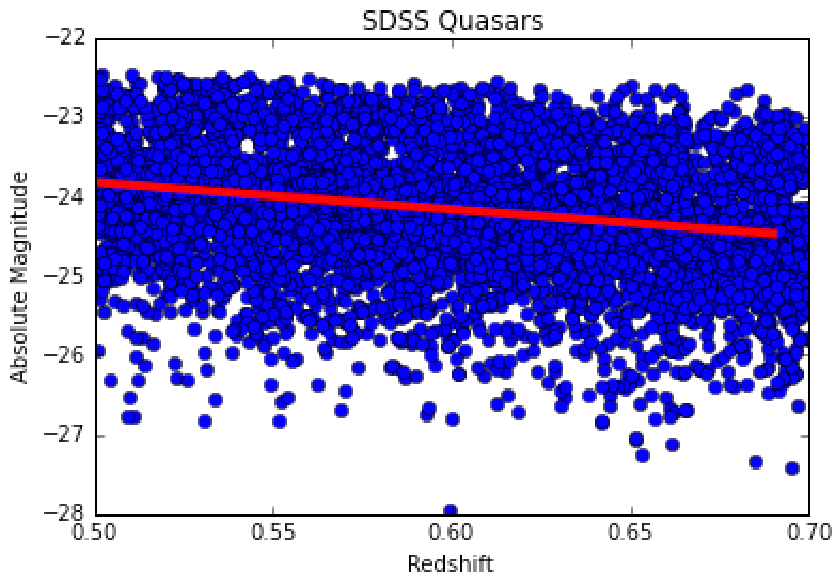
```
In [151]: fitline =numpy.polyfit(data[:,7], data[:,0], 1)
          fitline

Out[151]: array([ -3.39636304, -22.12638289])
```

Let's have a look at the fit we made to the data:

```
In [152]: x = numpy.arange(0.5, 0.7, 0.01)
          pylab.plot(data[:,7], data[:,0], 'bo')
          pylab.plot(x, fitline[1] + x*fitline[0], ls='-', c='r', lw=4)
          pylab.xlabel('Redshift')
          pylab.ylabel('Absolute Magnitude')
          pylab.title('SDSS Quasars')

Out[152]: <matplotlib.text.Text at 0x10ef2d450>
```



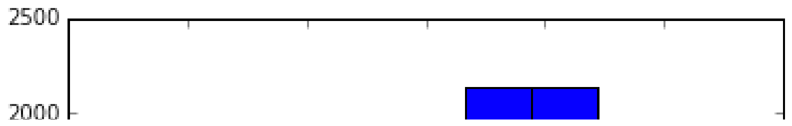
Advanced Plotting

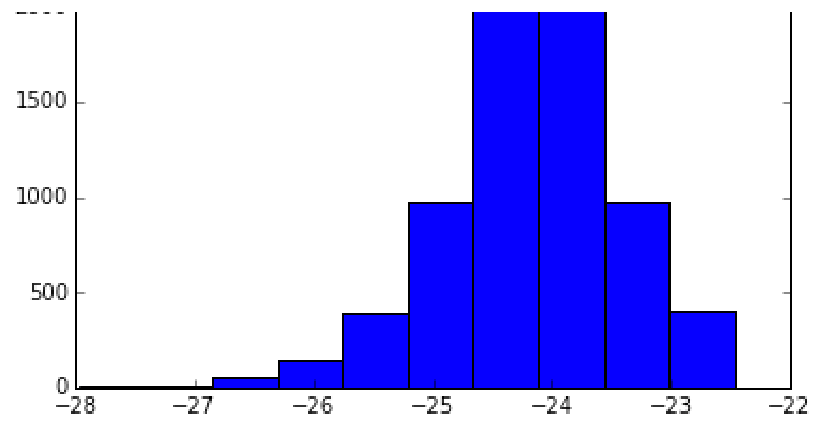
matplotlib is a powerful plotting tool.

Here are a few examples. Let's start with a histogram.

```
In [153]: pylab.hist(data[:,0])

Out[153]: (array([  2.00000000e+00,   5.00000000e+00,   5.40000000e+01,
                    1.42000000e+02,   3.86000000e+02,   9.67000000e+02,
                    2.13900000e+03,   2.13800000e+03,   9.76000000e+02,
                    4.02000000e+02]),
           array([-27.9568 , -27.40772, -26.85864, -26.30956, -25.76048, -25.2114 ,
                 -24.66232, -24.11324, -23.56416, -23.01508, -22.466   ]),
           <a list of 10 Patch objects>)
```

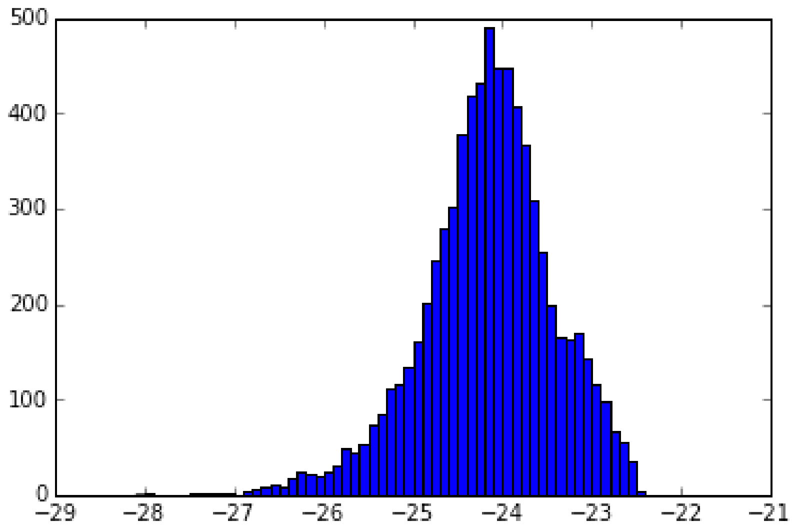




The histogram looks nice, but the bins are a bit large for my taste, we can adjust them manually like this:

```
In [154]: pylab.hist(data[:,0], numpy.arange(-28.1, -22, 0.1))
```

```
Out[154]: (array([   0.,    1.,    0.,    0.,    0.,    0.,    1.,    1.,    1.,
              1.,    2.,    0.,    5.,    7.,    9.,   10.,    9.,   17.,
             24.,   21.,   20.,   25.,   32.,   48.,   44.,   53.,   74.,
             84.,  112.,  116.,  135.,  160.,  201.,  245.,  279.,  303.,
            379.,  419.,  431.,  491.,  447.,  448.,  408.,  367.,  308.,
            254.,  200.,  166.,  164.,  169.,  143.,  117.,   97.,   67.,
             56.,   35.,    5.,    0.,    0.,    0.,    0.]),
           array([-28.1, -28. , -27.9, -27.8, -27.7, -27.6, -27.5, -27.4, -27.3,
                  -27.2, -27.1, -27. , -26.9, -26.8, -26.7, -26.6, -26.5, -26.4,
                  -26.3, -26.2, -26.1, -26. , -25.9, -25.8, -25.7, -25.6, -25.5,
                  -25.4, -25.3, -25.2, -25.1, -25. , -24.9, -24.8, -24.7, -24.6,
                  -24.5, -24.4, -24.3, -24.2, -24.1, -24. , -23.9, -23.8, -23.7,
                  -23.6, -23.5, -23.4, -23.3, -23.2, -23.1, -23. , -22.9, -22.8,
                  -22.7, -22.6, -22.5, -22.4, -22.3, -22.2, -22.1, -22. ]),
           <a list of 61 Patch objects>)
```



Much better!

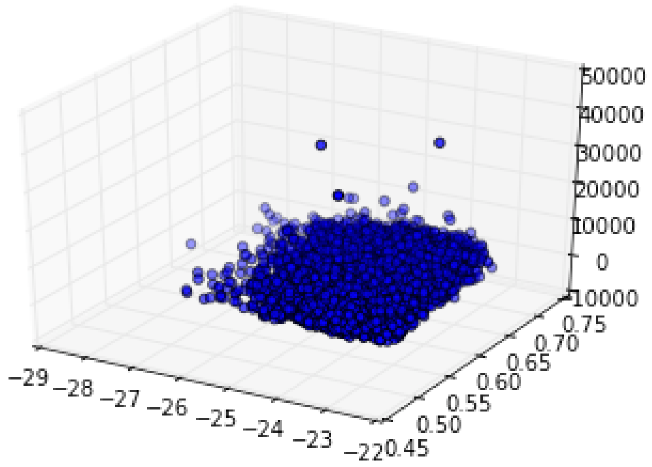
And finally, let's do something really exciting, a 3D plot! We'll first need to import a special set of functions.

```
In [155]: from mpl_toolkits.mplot3d import Axes3D
```

This will look a bit more complicated, we'll first have to create a figure (that is basically just an empty canvas). Then we can add an axes to that figure. Finally, we can plot the data.

```
In [156]: fig = pylab.figure()
          ax = fig.add_subplot(111, projection='3d')
          ax.scatter(data[:,0], data[:,7], data[:,6])

Out[156]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1111c4890>
```



V. License