Python/SciPy Tutorial

1 Basics

We can interact with Python is various ways, but for simulations the most useful way is to write a Python *script*. Literally, this is just a set of instructions, written in human readable text, that will be carried out by a program called the Python *interpreter*.

1.1 Editors and the Command Line

To create this script we need to use some editor, such as `gedit`, to open a plain text file. To do this, type at the command line

```
gedit first_program.py &
```

The file extension `.py` tells the `gedit` to go into Python mode, which will give you some helpful color coding. The ampersand allows you to get the command line back for more commands while `gedit` is still running.

Now, in the editor your first line has to be an instruction to fire up the Python interpreter, which we do with

```
#!/usr/bin/env python
```

Try these lines of code

```
a=6
print a
print a**2
```

Save this, and then back at the command-line prompt type

```
chmod u+x first_program.py
```

This is a one-time step for this file that allows you to great it as an executable file (read it as "change mode: user adds executable"). Now simply type the file name at the prompt, and see what results.

1.2 Variables and Assignment

The two built-in data types that we will be using are integers and floating point numbers. Integers are stored as a base-2 number and are exact, and calculations with integers are relatively fast. However, sometimes, numbers like `0.5` or `6.92101641398` are required, so integers aren't enough. Floating point variables can hold such numbers, though they come with two costs: computations with floating point numbers are slower, and they aren't exact. Python floating point numbers have a precision of about 1 part in 10^6, which for our purposes is plenty precise.

We can assign values to variables with the assignment operator, the `=` sign:

```
a = 6
b = 4 + a
c = 2.92 + a
print a, b, c
```

Here Python decides to make `a` and `b` into integers, but `c` into floating point.

Note that the assignment operator is not at all like the equals sign in math. You will frequently see things like this:

```
a = a + 1
```

which as a math equation makes no sense. Python takes the right hand side of an assignment and evaluates it. If it can't, perhaps because you gave it a variable that didn't have a value, it gives you an error. Once it's done evaluating the right hand side, it puts that value into the variable on the left. So this line makes perfect sense, and increments the variable `a` up by 1.

1.3 Loading Modules

Add another line to your program that says

```
print exp(a)
```

save this, and try running the code. You should get an error message the `exp` isn't defined. This happens because Python starts up with a fairly minimal set of "built-in" functions, and then you are expected to pull in whatever functions you need from modules. (This is done for efficiency purposes.) To load the exponential function from the beginning, we could have had as the second line in our program:

```
from scipy import exp
```

Add this line and try running the program again. Here we loaded a specific function from the `scipy` module. We could have loaded the whole module with the command

```
from scipy import *
```

There is yet another way to load the scipy module:

```
import scipy as sp
```

When you load a module in this fashion, you must precede the functions you are using with the module abbreviation you have *chosen*, here `sp`. In your program you would need

```
print sp.exp(a).
```

This "costs" a few extra keystrokes, but it makes it very clear where various functions come from. (This is important because there are similar functions with similar names in different modules.) In the rest of this handout, SciPy functions will all be preceded by `sp`.

We will discuss more features of the SciPy module in Section 4.

1.4 Comments

Computer programs should be documented with "comments" so that when you (or someone else) looks at the code long after it is written, it is possible to decipher the methods and goals of the programmer.

- For short comments it's easiest to use the `#` symbol. Everything to the right of the `#` sign will be ignored by the Python interpreter. Sample use:

  ```
  m = 3.4    # mass of the ball in kg
  ```

- For longer documentation you can use three quotation marks to surround the comments. For example,

  ```
  """
  The following two lines give the masses of the molecules in  AMUs.
  When m1 > m2 the recoil velocity is always less than the critical speed,
  but when m2 > m1 the result depends on the elasticity parameter
  """
  m1 = 3.2
  m2 = 1.5
  ```

2 Program Control

Programming is a sequence of commands, from beginning to end, but usually the tasks we want to do are more complicated than a simple list of commands. Very often we want to loop through a set of possible values, or make a decision about what to do next that depends on the values of some variable. In this section we describe these tasks.

2.1 `for` loops

Try the following snippet:

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

Might not be what you expected: the numbers looped from zero to 9, rather than from 1 to 10. But we did get 10 values.

Notice the indentation. This matters in Python. Often we will want to do multiple commands for each value of `i`, that is, there is a *block* of code controlled by the `for` command. In Python you use indentation to define the blocks:

```
for i in range(10):
    j = i**2
    print j
print "Now I'm done!"
```

You can have `for` loops within `for` loops, for example to sweep through a two-dimensional lattice:

```
count = 0
for i in range(10):
    for j in range(10):
        count = count + 1
    print count
```

Run this code. Does the result make sense to you? What would be different if you indented the `print count` command further? Or un-indented it?

2.2 `if` Statements

Decisions are often necessary in program control, and these are accomplished with the `if` statement. Consider the following:

```
for i in range(10):
    j = i**3
    if j < 300:
        print j
```

The result is that not all values are printed, only those which for which the `if` condition evaluates to be true. We might also want to do certain things in the case the condition is false, which we can do with the `else` and `elif` (else if) commands:

```
for i in range(10):
    if i < 4:
        print i
    elif i < 7:
        j = i**2
        print j
    else:
        j = i**3
        print j
```

An artificial example, but hopefully it makes the point.

2.3 `while` Loops

Frequently you want to loop through a number of cases as long as some condition is true, rather than specifying a range. This can be done with the `while` command.

```
x = 0
while sp.exp(x) < 1.0e5:
    print x,sp.exp(x)
    x = x + 1.0
```

3 Defining Functions

It can be useful to define a function to perform a task, in the sense of making the program more readable or eliminating repetition. For example, a function might take a pair of numbers and return the larger of the two:

```
def max(x,y):
    if x > y:
        return x
    else:
        return y

print max(10,0)
print max(5, 6.3)
```

The command we use to define a function is `def`, followed by the function name and then an argument list. The guts of the function definition is an indented block. The `return` command determines what is returned as the value of the function. Test out this code and see if it makes sense to you.

When is it worth defining a function, rather than just using the code of the function definition? Either when it will be used many times, so the function definition makes your code more compact and easier to read, or when it just makes logical sense.

For example, we could define a function that takes as an argument a spin lattice, and returns as a value the magnetization per spin. Or the energy per spin.

4 The SciPy Module

4.1 Introduction to SciPy

SciPy is a collection of mathematical algorithms and convenient functions for scientific computation within Python. It is built on the Numpy module, and includes Numpy itself when SciPy is imported. SciPy/Numpy are particularly helpful in this project in setting up the data structures we need.

More information can be found on the following web pages.

SciPy tutorial:

`http//docs.scipy.org/doc/scipy/reference/tutorial/index.html`

SciPy "Getting Started" page:

`http://www.scipy.org/Getting_Started`

SciPy array tip sheet:

`http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/arrays.html`

4.2 SciPy/Numpy arrays

The central data objects in SciPy/Numpy are homogeneous multidimensional *arrays*. An array is a table of elements (usually numbers), all of the same type, indexed by a set of positive integers (technically a Python tuple). For those familiar with Python, arrays are similar in some respects to Python lists, but they are better-suited to numerical computation.

We can make a simple one-dimensional array as follows:

```
a = sp.array([1,2,5])
print a
```

or a two-dimensional array as

```
b = sp.array([[1,2],[3,4],[5,6]])
print b
```

For the Ising simulation we want a lattice of spin variables (should we use integers or floating point for the spins?). This can be set up easily as an array using the `zeros` function from the SciPy module. Here's an example:

```
n = 10
s = sp.zeros( (n,n), int )
print s
print s[6,2]
```

What happened here? We told `zeros` to make an `n`×`n` array of integers (spins) and also to assign them the value zero. Then we printed the entire array, and then a particular one of those spins. And we got zero. Good.

What if we wanted to initialize the spins to all be one instead of zero? We could use `zeros` to create the spin lattice and then loop through all the spins and set them to be one. Or, we could create them with the value one instead of the value zero by using the `scipy` function `ones`. This is imported and used just like `zeros`. Try changing the code above and see that you get one as the result for `s[6,2]`.

4.3 Random numbers in SciPy/Numpy

Once the SciPy module has been imported, you can pick random real numbers between 0 and 1 using

```
sp.rand()
```

Other random number functions are part of the `random` *sub-module* of SciPy. To get a random integer between 0 and 6, use

```
sp.random.randint(0,7)
```

To get a 10×10 array of 1's and 0's we can use

```
sp.random.randint(0,2,(size,size))
```

Suppose that we wanted to decide randomly to do x 30% of the time, but to do y the other %70. We can use `random` to do this:

```
p = 0.3
if sp.rand() < p:
    #  Do whatever we wanted to do 30% of the time
else:
    #  Do instead what we wanted to do 70% of the time
```

The logic here is a bit tricky, if you're new to it. Since `rand()` will return all values between 0 and 1 with equal probability, that means %30 of the time it will return a value less than 0.3.

4.4 Input/Output of SciPy arrays

SciPy provides a very convenient means of reading external files into SciPy arrays, and writing SciPy arrays to files. Start by using `gedit` to create a simple file, `in.dat` containing the following "data":

```
1 1
2 4
3 8
```

The following Python code reads in the data, prints the array, modifies one element of the array, and writes the modified array to a new file.

```
a = sp.loadtxt('in.dat')
print a
a[2,1] = 9
sp.savetxt('out.dat',a)
```

After running this script you should examine the contents of the newly-created file `out.dat`.

4.5 Appending rows to SciPy arrays

```
a = sp.array([[1,2],[3,4]])
print a
a = sp.append(a,[[5,6]],axis=0)
print a
```

4.6 Combining 1-d SciPyarrays

Let's say you have a set of times, $t = 1, 2, 3$ along with a set of speeds at those times, $v = 1, 4, 9$, and you want to combine them into a single array containing a set of (t, v) pairs. Try the following:

```
t = sp.array([1,2,3])
v = sp.array([1,4,9])
print sp.vstack((t,v))
print transpose(sp.vstack((t,v)))
```