Week 7: Introduction to Python

January 15, 2017

1 Introduction

In the first three semesters, you learned about C programming, some VERILOG and some assembly language programming. In this lab, you are going to learn to code in Python.

Please download the book that has been uploaded on Moodle. It is an open source book that gives a simple introduction to Python. In addtion you can visit http://www.python.org and see other open source resources.

One book I have liked a lot has been *Beginning Python* (the first author being Peter Norton). It gives a very lucid introduction to basic python while tackling some surprisingly deep stuff at the same time.

In this assignment, we will explore Python the basic language and its features

2 Assignment for the Week

1. Python can handle basic numbers of different types. Use the following algorithm to compute the fibunacci series

```
n=1
nold=1
print 1,nold
print 2,n
repeat for k=3,4,...,10
  new=n+nold
  nold=n
  n=new
  print k,new
end loop
```

   and code it in both C and in Python. Compare the programs (they should be almost identical). **Note:** for Python, you can lookup the syntax of the *for* command and the *print* command.

2. Generate 1000 numbers using the following algorithm and use them to create a list:

```
n[0]=0.2
alpha=pi
for k in 1,2,...,999
    n[k]=fractional part of (n[k-1]+pi)*100
end for
```

   Print out the list generated. Each number in the list should have 4 digits.

   `Hint: look up` *`int`* `which takes a number and returns the integer part. To create a list, say "lst=[]". To add to a list, use "lst.append(number)"`

   Hint: Look up the *print* command for the formatting to use. It is very similar to printf.

   Do this both in C and in Python

3. A file has been uploaded in Moodle for this question. It is the text of *The Hound of Baskervilles* which is a famous Sherlock Holmes story. In Python, read in the file, and build up the frequency of the of words appearing in the text. This is nearly impossible to do in C, as you need to build a table with the word as the index and the frequency as the value. What is needed is a *hash table*, which is not a built in feature in C. In Python, dictionaries implement such tables.

   (a) To read in a file, use

   ```
   with open(filename,''r'') as f:
      contents=f.read()
   ```

   This will open the file for reading, and read the entire file as a single line into the variable *contents*.

   (b) To convert a string to words, use

   ```
   words=string.split()
   ```

   (c) To create and update a dictionary,

   ```
   d={} # creates an empty dictionary
   d[w]+=1 # adds 1 to d[w], but crashes if d[w] does not exist
   d[w]=d.setdefault[w,0]+1 # avoids crash by setting
      # d[w] to zero if it does not exist.
   if w in d: # this also works, and is quite fast
     d[w]+=1
   else:
     d[w]=1
   ```

   (d) To print out a list of words in dictionary

   ```
   for k in freq: # keys in arbitrary order
      print k,freq[k]
   for k in sorted(freq): # keys sorted
      print k,freq[k]
   for k in sorted(freq,key=freq.get): # values sorted
      print k,freq[k]
   for k in sorted(freq,key=freq.get,reverse=True):
      print k,freq[k]     # values sorted descending
   k=sorted(freq,key=freq.get) # k is now a list of
      # keys sorted in increasing value
   ```

3 The Python Language

When it comes to programming languages, there has been a long history of both compiler based languages and interpreter based languages. On the compiler side there was Fortran followed by Cobol and after quite a while, there was Pascal and then finally C. With each new language new features were added. Of course the older languages did not stay put and so there is a Fortran2003 which has many of the features of C++ that are relavent to scientific programming.

Similarly, on the interpreter side, there were early attempts which included Flex and Basic, as well as the language of the command shell. When UNIX came on the scene, the shell came with a sophisticated programming language. But UNIX was special in that it added special power to this shell via other utilities such as awk, sed, yacc etc. Awk was the first text processor. Sed was a stream editor, that filtered a text file to give another. And Yacc was a build on the run compiler generator. These were utilities of power never before seen by casual users, and many inventive uses were found for them. Around the same time,

on the IBM side, an interesting language called REXX also appeared which first invented one of the most important modern data structures, namely associative arrays.

With so many utilities and options things were a jumble and soon users began to demand for a more integrated language that had all these features built in. This ushered in the modern set of languages of which the premier one is PERL. PERL introduced many things into interpreted languages, so many that it is hard to enumerate them. But it tried to do too much and it is a sprawling language, one in which an expert can do magic, but one which requires quite a lot of studying to master.

Following PERL were some competitors of whom I consider Ruby and Python the best. Both are lovely languages from the user's point of view and both are very similar in their features. Ruby has developed into a terrific language for database programming. Python has become the primier language for science and engineering. This is why we will focus on Python in this course.

A third direction in which languages developed was the scratch pad type of language. It started in the sixties with early versions such as the "Online Math System" of UCLA. In the seventies, a powerful scientific development package called Basis was introduced. In the early eighties IBM had introduced scientific spreadsheets for laboratory use. This branch has led to modern tools such as Matlab, Mathematica and Labview.

Recently, a surprising development has taken place. People have developed modules (extra functionality) for Python that enables it to do most of the things that Matlab, Labview and Mathematica can do. This has created a special status for Python as far as technical users are concerned.

In this course we study Python. Along the way, most of the capabilities of Matlab will be demonstrated in Python, so that you can become a competent user of Matlab as well very quickly. In this assignment the focus is on Python basics.

4 Basic Features of Python

To start the interpreter, just type "ipython" at the command line. "ipython" stands for "interactive python" and it is the preferred python shell for this lab. You can then enter commands at the prompt. For example

```
$ ipython
Python 2.6.4 (r264:75706, Apr  1 2010, 02:55:51)  Type "copyright", "credits" or "l
IPython 0.10.2 -- 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]: a=range(10);a
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: b="This is a string";b
Out[2]: 'This is a string'
In [3]: a[2:5]
Out[3]: [2, 3, 4]
In [4]: quit()
$
```

The Python prompt is ">>>". The Ipython prompt is "In [nn]:". This short excerpt shows how to define a couple of variables and to quit the interpreter.

4.1 Basic Types

Python recognizes a variety of types but doesn't require the user to declare any of them. They include:

- Integers
- floats

- complex numbers
- boolean values
- strings
- "tuples" and lists (these are arrays)
- dictionaries (these are associative arrays)

(See the tutorials on the live CD for an introduction to these types). Pointers don't exist, as all memory management is handled invisibly. Python has the ability to define other types as well, such as matrices through the use of classes.

4.1.1 Examples

```
In [5]: a=4.5
```

This is an example of a real number.

```
In [6]: i=3
```

This is how we declare integers. The only thing to remember is that integer division can yield unexpected results:

```
In [7]: i=3;j=2;print i/j
1
```

The answer is not 1.5 but its truncated value. Also note that $3/(-2) = -2$! In Python version 3 onwards, integer division follows C syntax and returns the floating point answer, so beware!

```
In [8]: z=1+2j
```

Python handles complex numbers as native types. This is enormously useful to scientists and engineers which is one reason why it is popular with them. Note that "j" is identified as $\sqrt{-1}$ by context. So the following code will give peculiar answers:

```
In [9]: j=-2
In [10]: 1+2j
(1+2j)
In [11]: 1+j
-1
```

In the first complex assignment, Python knew that you meant a complex number. But in the second case it thought you meant the addition of 1 and the variable `j`. Also note that if you assign the answer to a variable Python does not print the answer. But if you do not assign it, the value is printed out.

```
In [12]: v=True
In [13]: j>2 or v
True
```

`True` and `False` (capitalized as shown) are the fundamental boolean values. They can be assigned to variables and they are also the result of logical expressions.

```
In [14]: ''This is the %dth sentence.'' % 4
'This is the 4th sentence.'
```

This is a most peculiar expression. We have seen strings already. But *any* string can be a `printf` string that follows C syntax. This must be followed by a % sign and a list of values.

```
In [15]: ‘‘This is the %(n)th %(type).’’ % {’type’:’sentence’,’n’:4}
```

A string can be more peculiar. It can take variable names for its entries so that they are represented by labels. Note that this is the general string, and can be used anywhere, not just where printing is going to happen.

```
In [16]: ‘‘’’’This is a multi
line string’’’’’
’This is a multi\nline string’
In [17]: print _
This is a multi
line string
```

Sometimes we might want embedded new lines in strings. Then the above syntax helps us. Everything between the triple quotes is part of the string. Also note the print command. It prints “_” which is Python’s variable that holds the last computed value that was not assigned to a variable.

Naturally strings are where a lot of built in functions exist and we will look at them soon.

4.1.2 Arrays

Arrays store items in order so that they can be accessed via their index. The indexing follows the C indexing (i.e., starts from 0). Unlike C, the elements of a list do not have to be identical in type.

```
In [18]: arr=[1,2,3,4,5];arr
[1, 2, 3, 4, 5]
```

Arrays can be created by just specifying the elements as in the first example above. The “constructor” is a set of comma separated values enclosed in square brackets.

```
In [19]: arr=range(10);arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [20]: arr=range(2,5);arr
[2, 3, 4]
In [21]: range(2.2,10.7,1.5)
[2, 3, 4, 5, 6, 7, 8, 9]
```

A common use for arrays is to create a sequence of uniformly spaced points. This can be done with the `range` command. Without a second argument, it returns a sequence of integers starting with zero upto the given argument (but not including it). With two arguments, the sequence starts at the first argument and goes to the second. Given three arguments, the third argument is the step. **Note that range expects integer arguments.** This is seen in the last example above. When floats are given, they are “coerced” (i.e. converted) into integer and then sent to the function.

```
In [22]: arr[2:7:2]
[2, 4, 6]
```

We can refer to a portion of an array, using what is known as a “slice”. `2:7:2` means “start with 2, increment by 2 upto but not including 7”.

```
In [23]: arr=[1,2,[4,5,6],’’This is a string’’]
In [24]: arr[0]
1
In [25]: arr[2]
[4, 5, 6]
In [26]: arr[3]
’This is a string’
```

Arrays do not need to have elements of a single type. In the example above, we have mixed integers with another list and with a string.

```
In [27]: arr[2][1]
5
```

This shows the way to access the element of a list that is an element of a second list. The syntax is very similar to 2-dimensional arrays in C but clearly pointers are involved everywhere, but invisibly.

4.2 Dictionaries

Python also has associative arrays whose values are indexed not by number but by strings.

```
In [28]: hash={‘‘a’’:’’argentina’’, ’’b’’:’’bolivia’’,’’c’’:[1,2,3]}
In [29]: hash[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 0
In [30]: hash["a"]
’argentina’
In [31]: hash["c"]
[1, 2, 3]
```

The first attempt `hash[0]` shows that this is a new beast. Its elements are not accessed by an index. The error message says that “0” is not a key found in the dictionary. A “dictionary” consists of a set of “key-value” pairs. Each pair has a string called the key and some value. The value could be another string or it could be anything else. In the above example it is a list of three elements. Dictionaries were (to my knowledge) first introduced in Rexx and perfected in Perl are invaluable in modern programming.

In the current assignment, you will use Dictionaries to hold the node names. Each time you read in a node name, check if it is already known, and if not, associate it with a new number:

```
nodes=dict()
nodeidx=1   # index 0 reserved for ground
....
if w not in nodes:
  nodes[w]=nodeidx
  nodeidx+=1
....
for n in nodes:
  print n,nodes[n]
```

There is another type of list in Python called the “tuple”. It merely means a constant list, i.e. a list whose values cannot change. It follows the same syntax except its values are enclosed in round brackets.

4.3 Basic Programming

The basic structures in Python are the if block, the for block and functions. There are others, but these are the important ones.

```
In [32]: if j>2 or v:
...    print ‘‘Too many inputs: %d’’ % j
...
Too many inputs: -2
In [33]: sum=0
```

Note that blocks start with a statement ending in a colon. The code in the block must be indented evenly - the extent of indentation is how python knows which block the statement belongs to. The above code is the standard `if` block. The general structure is

```
if condition:
    commands
elif condition:
    commands
elif condition:
    commands
...
else:
    commands
```

There is no switch command in Python, and this construction has to do the job of the select - case statement as well.

```
In [34]: for i in range(20):
...      sum += i
...
In [35]: print ‘‘The sum of 0, 1, ... 19 = %d’’ % sum
The sum of 0, 1, ... 19 = 190
```

The for block in Python iterates over an array of values. Here it is iterating over a set of integers.

```
In [36]: words=[‘‘Water’’, ‘‘water’’, ‘‘everywhere’’, ‘‘but’’, ‘‘not’’, ‘‘a’’, ‘‘drop’’, ‘‘to’’, ‘‘d
In [37]: for w in words:
...      print ‘‘%s’’ % w,
...
Water water everywhere but not a drop to drink
```

This for block prints out all the values of array words. Note that the print command ends in a comma, which replaces the carriage return with a single space.

```
In [38]: sum=0;i=0
In [39]: while i<20:
...      i += 1
...      if i%3 == 0: continue
...      sum += i
...
In [40]: print ‘‘The sum = %d’’ % sum
The sum = 147
```

The while construct is self-evident. It iterates till the condition fails. The if command skips to the next iteration every time a multiple of 3 is encountered. Also note that the “continue” statement can be placed on the if line itself if desired. This is not actually recommended though.

```
In [41]: def factorial(n=0):
...      ‘‘’’’ Computes the factorial of an integer ‘‘’’’
...      fact=1
...      for i in range(1,n+1):
...          fact *= i
...      return(fact)
...
In [42]: factorial(8)
```

```
40320
In [43]: factorial()
1
In [44]: factorial(4.5)
__main__:4: integer argument expected, got float
In [45]: factorial.__doc__
' Computes the factorial of an integer '
```

Functions are defined using blocks as well. Arguments are like C and can have "default" values. Here the default value of `n` is zero, which is the value assumed for `n` if no argument is passed. If a float is passed or a complex, an error is generated. This is not an argument mismatch error - python is very forgiving about those. Rather it is saying that `range(1,n+1)` does not make sense unless `n` is an integer.

The final line shows the self-documenting nature of Python functions. Anything entered as a string at the beginning of a function is retained as documentation for the function. The help string for a function can always be retrieved through the "`__doc__`" method.

4.4 Modules

Python comes with a huge variety of helper functions bundled in modules. For now all we need to know is that modules are collections of functions that have their own "namespace". You import the functions of a module by using the import command

```
In [46]: import os
In [47]: print os.system.__doc__
system(command) -> exit_status

Execute the command (a string) in a subshell.
```

The "namespace" comment is that to access the system command in module "os", we had to say "os.system", not "system". So there is a hierarchy in the names of functions and variables rather like the filesystem. Modules are like sub-directories.

You can import a module in the current space, by typing

```
from os import *
```

and you can import just some functions by typing

```
from scipy.special import j0
```

which imports the Bessel function $J_0(x)$

5 Example: Counting word length frequencies in a File

We need to open a file and read in the words. Then build up a frequency table of the word lengths.

5.1 Opening a file and reading in the contents

There are many commands for file management, but we will only look at two. In the first we will read in a file as a list of strings:

```
import sys # for commandline arguments
if len(sys.argv)<2:
  print ‘‘usage: python1.py filename’’
  sys.exit(1)
with open(sys.argv[1],'r') as f:
  lines=f.readlines()
```

What this does is to open the file for reading and assign it to the file pointer "f". The last line reads in the entire file into "lines". Lines is now a list of strings, one string per line in the file. **Note:** The file is automatically closed when we complete the block.

The other approach is to read in the file as a giant string:

```
import sys
if len(sys.argv)<2:
  print ''usage: python1.py filename''
  sys.exit(1)
with open(sys.argv[1],'r') as f:
  s=f.read()
```

Now "s" contains the entire file in the string "s".

For the example here, the second variant is useful - we don't need to know line information. For the assignment, we need to work line by line, and the first variant is better.

5.2 Count the words

```
count=[0 for i in range(10)]
```

This creates a 10 element list filled with zeros. The '0' could have been any expression.

```
words=s.split()
```

We now have to count up the number of words of different lengths.

```
for w in words:
  count[len(w)]+=1
```

This is slow, and we will see later how to speed it up. But we now have the word counts.

6 Example: Converting the value field to a number

6.1 The Regular Expression Module

One of the most important modules in Python is the regular expression module. This allows you to search a text string for patterns. It is very useful for parsing our value field to convert it into a proper number. I will just give the code here since it is difficult to come up with it without a lot of reading up on internet:

Let "w" contain the value string, for example 1.5e4k. We want to extract the value and the modifier.

First the value. It has an optional sign, followed by a number, followed by an optional 'e' followed by optional integer. Here is a search:

```
import re
k=re.search(''^[+-]?[0-9]*[\.]?[0-9]*'',w)
print k.group()
> 1.5
```

So the string was searched for an optional sign followed by 0 or more digits, followed by an optional '.' followed by 0 or more digits. The '^' at the start says that this should begin the word.

But we also want the exponential part. We change the search string

```
import re
k=re.search(''^([+-]?[0-9]*[\.]?[0-9]*)(e[+-]?[0-9]+)'',w)
print k.groups()
> ('1.5','e4')
```

The brackets ask python to return the two parts separately. Suppose we only had w="1.5k". What then? We get an error, since the second part wasn't found. So instead we try

```
import re
w="1.5k"
k=re.search("^([+-]?[0-9]*[\.]?[0-9]*)(e[+-]?[0-9]+)?",w)
print k.groups()
> ('1.5',None)
```

Python says the exponential wasn't found. The '?' permitted the string to be present or absent.

Now for the modifier.

```
import re
w="1.5e4k"
k=re.search("^([+-]?[0-9]*[\.]?[0-9]*)(e[+-]?[0-9]+)?(.*)",w)
print k.groups()
> ('1.5','e4','k')
w="1.5e4"
> ('1.5','e4','')
```

With this, we can now reconstruct the number and handle the modifier. The modifier is particularly easy:

```
mods={'p':1.e-12,'n':1e-9,'u':1e-6,'m':1e-3,'':1,'k':1e3,'meg':1e6}
k=re.search(...,w)
print mods[k.groups()[2]]
> 1000.0
```

associative arrays are extremely convenient in interpreted language programming. However, **NOTE:** they are very inefficient. Use with care.

6.2 Scientific Python (Not for current assignment)

Python has some packages (called modules) that make Python very suitable for scientific use. These are `numpy`, `scipy` and `matplotlib`. `numpy` makes available numerical (and other useful) routines for Python. `scipy` adds special functions and complex number handling for all functions. `matplotlib` adds sophisticated plotting capabilities.

Here is a simple program to plot $J_0(x)$ for $0 < x < 10$. (type it in and see)

```
In [48]: from pylab import *
In [49]: x=arange(0,10,.1)
In [50]: y=jv(0,x)
In [51]: plot(x,y)
In [52]: show()
```

The `import` keyword imports a module as discussed above. The "pylab" module is a super module that imports everything needed to make python seem to be like Matlab.

The actual code is four lines. One defines the x values. The second computes the Bessel function. The third plots the curve while the last line displays the graphic.

6.3 Plotting

Plotting is very simple. Include the plot command and at the end of the script add a `show()`. The plot command has the following syntax:

```
plot(x1,y1,style1,x2,y2,style2,...)
```

just as in Matlab.

6.4 Array operations

With these modules, we can now create arrays of greater complexity and manipulate them. For example,

```
In [56]: from scipy import *
In [57]: A=[[1,2,3],[4,5,6],[7,8,9]];A
In [58]: print A
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
In [59]: A=array([[1,2,3],[4,5,6],[7,8,9]]);A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
In [60]: B=ones((3,3));B
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
In [61]: C=A*B;C
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
In [62]: D=dot(A,B);D
array([[  6.,  6.,  6.],
       [ 15., 15., 15.],
       [ 24., 24., 24.]])
```

As can be seen in the examples above, operations on arrays are carried out element by element. If we meant matrix multiplication instead, we have to use `dot` instead. This is a problem that is unavoidable since these modules are built on top of python, which does not permit "dot" operators.

`Numpy` and `scipy` also define "matrix" objects for which "*" means matrix multiplication. However, I think it is not worth it. Just use the array objects.

Some important things to know about arrays:

- Array elements are all of one type, unlike lists. This is precisely to improve the speed of computation.
- An array of integers is different from an array of reals or an array of doubles. So you can also use the second argument to create an array of the correct type. Eg:

  ```
  x=array([[1,2],[3,4]]),dtype=complex)
  ```

- Arrays are stored row wize by default. This can be changed by setting some arguments in numpy functions. This storage is consistent with C.
- The `size` and `shape` methods give information about arrays. In above examples,

  ```
  D.size   # returns 9
  D.shape  # returns (3, 3)
  len(D)   # returns 3
  ```

  So `size` gives the number of elements in the array. `Shape` gives the dimensions while `len` gives only the number of rows.
- Arrays can be more than two dimensional. This is a big advantage over Matlab and its tribe. Scilab has hypermatrices, but those are slow. Here arrays are intrinsically multi-dimensional.
- The `dot` operator does tensor contraction. The sum is over the last dimension of the first argument and the first dimension of the second argument. In the case of matrices and vectors, this is exactly matrix multiplication.

6.5 Finding elements

Sometimes we want to know the indices in a matrix that satisfy some condition. The method to do that in Python is to use the `where` command. To find the even elements in the above matrix we can do

```
from scipy import *
A=array([[1,2,3],[4,5,6],[7,8,9]]);A
i,j=where(A%2==0)   # returns the coords of even elements
print A[i,j]
```

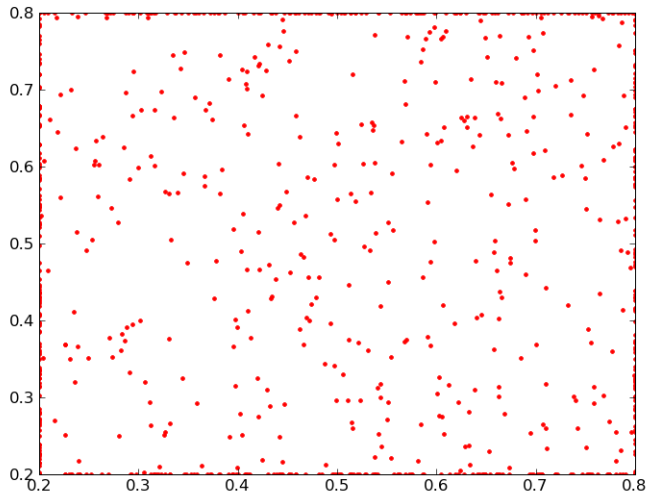
The ouptut is `[[2 4 6 8]]` which has no memory of the shape of A, but does contain the values. Note that the row and column indices start with zero, not one.

```
B=array([[6,6,6],[4,4,4],[2,2,2]])
i,j=where((A>3)*(B<5)>0)
```

Here we have a new trick. We want to know those elements of A and B where the element of A is greater than 3 while the corresponding element of B is less than 5. In Matlab (or Scilab) we would say `find(A>4 and B<5)`. However, Python does not like this construction since `A>4` is not well defined. Are we to take all the truth values, and them all together and return a single true or false? Or something else? So Python does not permit such an operation to be part of a logical statement. However, we `can` do element by element operations. which is what we are doing here.

How do we use the `where` command? Suppose we want to clip values to between 0.2 and 0.8. We execute the following code:

```
A=random.rand(1000,2)
i,j=where(A<0.2)
A[i,j]=0.2
i,j=where(A>0.8)
A[i,j]=0.8
plot(A[:,0],A[:,1],'r.')
show()
```

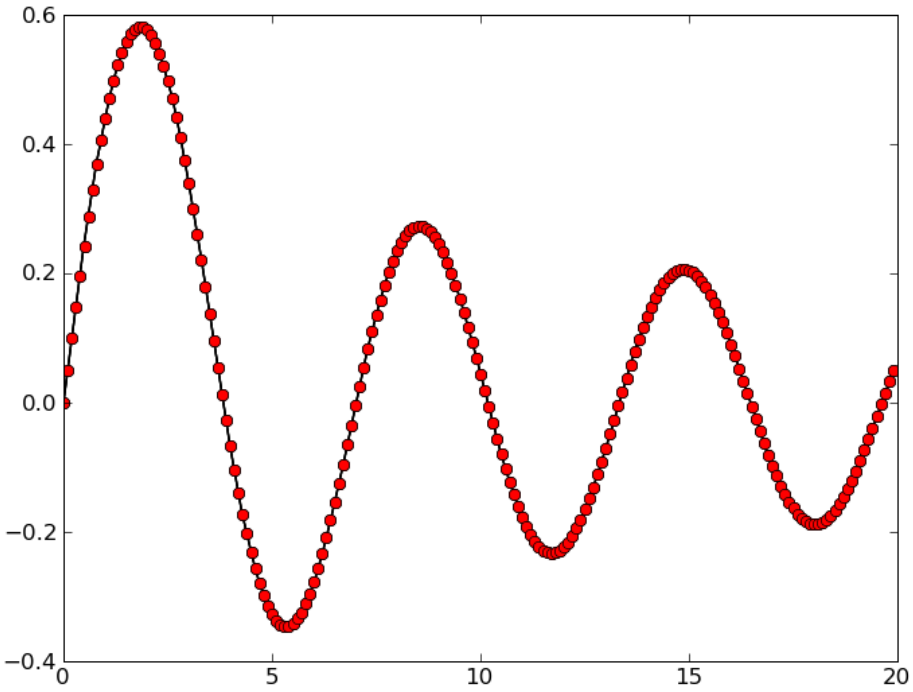


As can be seen, the points are all clipped to between 0.2 and 0.8.

6.6 Simple File I/O with Numpy

There are two simple functions that do most of what we want with numbers, namely `loadtxt()` and `savetxt()`. Both are in the `numpy` module. I used one of them to create the `plot.py` script. Here is a simple way to use them. (I will drop the python prompt from here one. That way you can cut and paste more easily)

```
from scipy import *
import scipy.special as sp
from matplotlib.pyplot import *
x=arange(0,20,.1)    # x values
y=sp.jv(1,x)            # y values
plot(x,y,'k')
savetxt('test.dat',(x,y),delimiter=' ')
w,z=loadtxt('test.dat')
plot(w,z,'ro')
show()
```



As can be seen, the red dots (the plot of w vs z) lie on the black line (the plot x vs y). So the reading and writing worked correctly.

`savetxt` simply writes out vectors and arrays. In the above example, if you look at `test.dat`, you will see two rows of data each with 200 columns. `loadtxt` reads in the objects in the file. Again, in this case, it reads in a 2 by 200 matrix, which is assigned to two row vectors.

In the assignment of this week, we directly use file i/o to access the contents of a file. That shows the power of Python that is available at any time.

Certain things which are very intuitive in matlab like languages are less easy in Python. To stack two vectors as columns we use

```
A=[x,y]
```

in matlab, provided the vectors are column vectors. In python we must write

```
A=c_[x,y]
```

since `[x,y]` defines a list with two elements and not a matrix. The function `c_[...]` takes the arguments and stacks them as columns to form a numpy array.

13 ⟨* 13⟩≡