Python lab session 4

Dr Ben Dudson, Department of Physics, University of York

18^{th} February 2011

Important: Remember to create a new document to write your answers in and put your name at the top. For each task, put your code and the result in a different section. At the end of the lab session, email this document to `phys-python@york.ac.uk`. Anything submitted after 14:00 **today** will not be marked. To avoid disappointment, please double-check that you have attached the document.

This lab session introduces reading and writing data to files, and starts to look at handling errors.

Writing text files

Variables can be converted to text using the str() function. Text can then be stored in variables, and connected together using '+'. For example:

```
a = "Hello "
b = "World!"
c = a + b
print c          -> 'Hello World!'
```

To write text to a file, you need to first open the file, write the text, then close the file. The following creates a 1D array and then write it to file

```
x = linspace(0, 4, 5)
f = open('output.txt', 'w')
f.write(str(x))
f.close()
```

This results in a file which contains

```
[ 0.  1.  2.  3.  4.]
```

Writing arrays as rows is fairly straightforward, but usually data is stored in columns instead. This is more convenient in some situations, for example when numbers are generated one by one, but makes writing and reading the files a little more difficult. This is something you might like to try on your own.

Task 1 Writing files (15%)

Change this program to write out each number in the array separated by spaces, so that the output file contains

```
0.  1.  2.  3.  4.
```

i.e. without the '[' and ']' brackets. To do this you'll have to loop through each number in the array, and write them to the file separately.

Task 2 Writing files 2 (20%)

a) Turn your program from task 1 into a function called writeArray to write an array on a single line, finishing with a new line ("\n"). Use the len() function so write works with any size of 1D array.

```
def writeArray(file, data):
    ####

x = linspace(0, 4, 5)
f = open('output.txt', 'w')
writeArray(f, x)
f.close(f)
```

b) Use this function to write x and $\sin(x)$ as two separate lines in a file where x is an array of 50 values between 0 and 2π.

c) Use Excel to open this file and use it to make a plot of the data

Copy your program, the text file 'output.txt', and Excel plot into your lab document.

Reading text files

In order to read text files back into Python, we need to be able to split text into pieces. Text strings can be split up into words (characters separated by spaces) using the split() method: if we have a variable s which contains some text, then s.split() will return a list of words. For example,

```
t = "this is some text"
print t.split()
```

will result in

```
['this', 'is', 'some', 'text']
```

The following code opens a file 'output.txt', reads it in one line at a time, splits the line into pieces separated by spaces, then prints the number of words in the line.

```
f = open('output.txt', 'r')    # Open the file for reading ('r')
while True:                    # Keep repeating until 'break'
    line = f.readline()        # Read in one line
    if len(line) == 0:         # If nothing in the line
        break                  # Then exit from loop
    columns = line.split()     # Split into words
    print len(columns)         # Print number of words in line
f.close()                      # Close the file when finished
```

Task 3 Reading files (20%)

Adapt the above program so that it counts the total number of lines and words in a file. Using the file you made in task 2, your program should print out something like

```
Lines: 2
Words: 100
```

In our case, each word represents a number e.g. "0.362". Note though that the way this is stored is completely different to how a floating point number is stored: text is stored as a list of bytes, one per character. As far as Python is concerned, "0.362" is no more a number than "spam" is.

To convert some text "0.362" into the number 0.362, we need to use float () to do the conversion. Example:

```
s = "0.362"
t = "4.0"
print s + t
```

```
0.3624.0
```

What Python has done is treat s and t as text, so '+' just joins the text together. Instead,

```
s = "0.362"
t = "4.0"
print float(s) + float(t)
```

```
4.362
```

This now converts s and t into floating point numbers, then adds the two numbers together.

Task 4 Read file and sum rows (20%)

Extend your program from task 3 so that it calculates the total for each line in the file. The output should look like:

```
Total of line 1: 100.0
Total of line 2: 19.86748
Lines: 2
Words: 100
```

Handling errors

When your program isn't just doing calculations, but has to start dealing with the outside world, you need to start handling errors. This is because if you're getting input, either directly from a user or from a file, you can't completely control what comes in.

You've probably experienced using a program which has suddenly crashed or given you an error. This is usually (though not always) caused because something happened which the programmers didn't expect. In our case, what happens if the file 'output.txt' doesn't exist (e.g. been moved or deleted)?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'output.txt'
```

This is an error which stops your program. Instead, we would like to be able to recover and either try a different file or tell the user about the error in a nicer way.

The easiest way to handle problems like this is to use **exceptions**. As the name implies, these are for exceptional circumstances which means something has gone wrong. In this case, the error message says IOError so this is the type of exception. We could modify the code to

```
try:
    f = open('output.txt', 'r')
except IOError:
    print "Couldn't open the file"
else:
    print "Opened file"
```

If 'output.txt' doesn't exist, this now just prints out

```
Couldn't open the file
```

Here **try** starts a protected block of code which could contain more than one command. If an error occurs, it jumps to the first **except** block. If the type of error (here IOError) matches the type after **except**, then it runs that block of code. If a different type of error happens which doesn't match, then the error will stop the program. For example:

```
try:
    f = open('output.txt', 'r') # Open the file
    line = f.readline()         # Read one line
    columns = line.split()      # Split into words
    val = float(columns[0])     # Convert first word to float
except IOError:
    print "Couldn't open the file"
else:
    print "Opened file"
```

If a file contains a word which can't be converted to a number (e.g. "eggs"), then this will produce

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for float(): eggs
```

Task 5 Error handling (10%)

Adapt the above code so it can recover from this error, and print out a message saying

"File contains invalid numbers"

Test by putting just the word "eggs" in 'output.txt' and running your program.

Task 6 Error handling 2 (15%)

Adapt your program from task 4 so that it can cope with the following errors and print a user-friendly message:

a) The file doesn't exist. Your program should print a message like

```
File 'output.txt' doesn't exist
```

b) A word can't be converted to a number. In this case, print out which line and word number caused the error, and what the word was. The message should look like

```
On line 2, word 4 ('spam') couldn't be converted to a number
```

Extra tasks

That's the end of the assessed lab, but here are some things you could try if you have time:

Task 7 Pickling and animations

From last time, calculating the Mandelbrot set images took quite a long time at high resolution. To produce an animation, it would be better to calculate all the results and save them to file. A second program could then read in these results and animate them.

a) Adapt your Mandelbrot calculation into a function so you give it a range of x and y and it returns a 2D array for the result.

```
import pickle
def mandelbrot(xmin, xmax, ymin, ymax):
    ###

file = open(###)
### loop, changing the range
    data = mandelbrot(####)
    pickle.dump(data, file)
file.close()
```

b) Write a program which loads the result arrays one at a time using pickle.load() and makes a contour plot animation

```
### loop until end of file
    data = pickle.load(file)
    ### contour plot, update z values
```

Task 8 Reading data in columns

Think about how to read data which is stored in columns. Each time you read in a line of text, it contains one number from each array.

a) If you know how big the arrays are going to be, you can create arrays first

b) One way to do this is to use the first line in the file to store the size of the arrays. Read in the size of the arrays, create the arrays, then read in the values

c) Finally, you can read the numbers in and append them to the arrays. There are several ways to do this, some of which are faster than others.