10. Input and output of data from python programmes.

In the Jupyter notebooks you have used so far we have have written our 'input' data directly into the python code as variables. We have also restricted the output of the data from the programme directly to the screen by using the 'print' statement or graphically using the MathPlotLib library. In practise, we often need to import data to our programme from an externally stored computer file. For example, we may have collected data from an experiment in a datafile that we now wish to process. Similarly, and especially for programmes that take a long time to execute, we want to save data to computer files (so we don't need to keep running the programme) or so that it can be processed later using further python programmes or other applications (e.g. Excel, Origin etc.).

In this notebook you will learn how to:

- provide a prompt for a user to enter data into your programme,
- read and write data to and from structured data text files using the Numpy library,
- read and write data to and from structured binary (archive) format files using the Numpy library,
- read and write arbitrary data to a text file as strings.

10.1 Entering data in response to a prompt - the input statement.

When writing programmes we often need to run the same programe using different input values. So far, in the jupyter notebooks we have achieved this by reassigning values to variables (normally at the beginning of the programme). This is not ideal, especially if you are writing the programme for someone else to run. To avoid this editing of code it is better to prompt the user for input and wait for them to type a response. To do this we use the python *input* statement. For example,

In [ ]:

```
a=input("Enter a number:")
print(a)
```

A common problem is that the data entered is not of the type expected. For example, in the code above there is nothing to say that the user cannot enter a text string rather than a number. This might cause the programme to fail with an error, or worse, it may do an unexpected type conversion and not produce the correct results.

To avoid these problems it is common to try and 'trap' errors with the input and ask the user to try again. To do this you need to check for any exceptions on the data input. In the example below, we are trying to read an integer number _b_. Provided the assignment of _b_ does not cause an error, there is no exception and the programme moves on to the code following the *while* statement. However by using the *try* ... *except* statement we can check to see if the user has indeed entered an integer number. If the input line produces an error, for example the user entered a text string or a floating point number, then the programme will ask again until the correct type of input value is used.

In [ ]:

```
test=True
while test :
    try:
        b=int(input("Enter an integer number"))
        test=False
    except :
        print("you need to enter a integer number!")
        test =True

print("You entered  ",b)
```

In the example above, the programme will ask to re-input the data, whatever caused the error. If you wish you can check to see if a specific error (or errors) has (or have) been produced. If done carefully, this allows you to produce more useful advice for the user depending on the error produced. For example, in the code below we specifically check to see if the error produced was a *ValueError*

In [ ]:

```
test=True
while test :
    try:
        c=int(input("Enter an integer number"))
        test=False
    except ValueError:
        print("you need to enter a integer number!")
        test =True

print("You entered  ",c)
```

You are allowed as many *except* statements after the *try* statement as you wish (much like the *elif* line for the _if_ statement). This allows you to trap and act on different types of error. A final *else* statement allows you to 'clean' up any errors that you didn't anticipate!

You might note that the *try* statement is not limited to checking input. You can include it anywhere in your code where you think an exception might be produced (for example an unwanted divide by zero error).

We won't cover any more about using exceptions. If you are the only one using the code you might be happy to accept that your programme will crash if it comes across an error or, for example, you enter text instead of a number. However, if you are expecting your code to be used by many people you will need to address these issues - users get very frustrated if a programme keeps crashing!

10.2 Output simple data to a file from your programme.

The Numpy library has some simple statements to conveniently save data in Numpy arrays to a data file. There are two kinds of file that can be produced: text files that are humanly readable and 'numpy' files in which the data is stored efficiently but in a way that is not humanly readable (binary files).

10.2.1 Saving data as text files.

The most convenient and simple way to save data from your program is to use the Numpy *savetxt* or *save* statements.

With *savetxt* you can save a Numpy 1D or 2D array to a named text file. The examples below show how this can be done. Note, it can be convenient to sometimes combine numpy arrays. In the example below the 1D arrays holding the _x_ and _y_ data have been combined into a single multicolumn 2D array with the Numpy *column.stack* function. These files may conveniently be read into, for example, excel spreadsheets later.

In [ ]:

```
import numpy as np
x=np.linspace(0,1,10)
y=np.sin(x)
xy=np.column_stack((x,y))
a=np.array([[1,2,3],[3,4,5],[5,6,7]])
np.savetxt("xdata.dat",x)
np.savetxt("ydata.dat",y)
np.savetxt("xydata.dat",xy)
np.savetxt("adata.dat",a)
```

Note that it is a good idea to give your files an extension, in this case, *dat* that shows that they are text files containing data. You should be able to read these data files by clicking on the filename on the Jupyter server or using any text editor on your computer. You should keep these files as they will be used later to look at reading data from files. You may wish to look up the Numpy *savetext* statement as there are additional parameters that can be set if you wish to take more control over the formatting of the data in the file.

10.2.2 Saving data as archive (binary) files.

A disadvantage of saving data as text files is that the precision in the data saved (especially if you format it to be more readable) may be reduced. An alternative way to save the data is as a binary (archive) file which is encoded in a similar way to the memory in the programme. This is efficient in space and preserves precision, however, the data cannot be read in the same way as a text file. In Numpy, binary data may be saved using the *save* statement or the *savez* statement. The latter saves the data in a compressed format and also allows mulitple arrays to be stored in the same file.

The following code shows examples of these statements.

In [ ]:

```
np.save("xdata",x)
np.save("ydata",y)
np.save("adata",a)
np.savez("xdata",xvals=x)
np.savez("ydata",yvals=y)
np.savez("adata",xvals=x,yvals=y,xydata=xy,avals=a)
```

We can see that there is a subtle difference between these functions. *save* only allows a single 1D or 2D array to be saved whereas we can save multiple arrays using *savez* . Note that we haven't included an extension for the filenames in this case. Numpy will assign the defaults extension *.npy* and *.npz* for files saved with *save* and *savez* respectively. You will find that you cannot open these files on the Jupyter server. If you save them to your computer and open them with the a text editor (e.g. notepad) you will find they contain gobbledegook. Furthermore, if you edit them in a text editor you will corrupt the data. We will see how to read these files shortly.

10.3 Input data from a file into your programme.

10.3.1 loading data stored as text.

The code below shows how you can read the data we wrote to the files *xydata.dat* and *adata.dat* back into python arrays. (NB. They do not need to have the same variable name).

In [ ]:

```
data=np.loadtxt("xydata.dat")
print(data)
data2=np.loadtxt("adata.dat")
print(data2)
```

The datafiles do not have to be created using python. Provided the data in the file is ordered as an array it may be read using *loadtxt* . By default python uses 'space delimited' data files. Data from other programmes (for example excel) may be stored in 'comma delimited' *csv* data files (the numbers are separated by commas) or some other delimiter. In this case you will need to specifiy the delimiter used in the data file in your *loadtxt* function. For example, if you have a *csv* file named *test.csv* the appropriate syntax to load the data is to use the keyword 'delimiter' i.e.,

data=np.loadtxt("test.csv",delimiter=',')

10.3.2 loading data stored using the Numpy *save* function

Numpy provides a function that will read any array data stored in *npy* files using the *save* function. The syntax is straightforward. i.e.

In [ ]:

```
xyin=np.load("xdata.npy")
print (xyin)
ain=np.load("adata.npy")
print(ain)
```

Note you need to include the file extension in the load statement.

10.3.3 load data stored using the Numpy *savez* function.

Reading data from the data stored in *npz* using the Numpy *savez* function uses the same Numpy load function. However, Numpy will assume that the data file will contain one or more arrays that may be of different sizes (provided you have used the *npz* extension). To properly read the file we need to know the names of the arrays stored in the data file. If you created the file and know the names this is fine. However, if you don't know the names of the arrays you use the *keys* method to recover the names. i.e.

In [ ]:

```
xdatin=np.load("xdata.npz")
print(xdatin)
print(list(xdatin.keys()))
alldata=np.load("adata.npz")
print(alldata)
print (list(alldata.keys()))
```

Note that the variables *xdatin* and *alldata* are not the original arrays but 'Npzfile' objects that contain the data. In our programme, to recover the original arrays as numpy arrays we use e.g. the following syntax.

In [ ]:

```
x2 = alldata['xvals']   # recover the original xvals array to the array x2
y2 = alldata['yvals']   # recover the original yvals array to the array x2
xy2 =alldata['xydata']  # recover the original xydata array to the array xy2
a2 = alldata['avals']   # recover the original avals array to the array a2
#
#   Print out some of the arrays to check.
#
print(a2)
print(xy)
```

Note, we don't need to use the original array names.

10.4 Opening, reading, writing and closing more general text files.

The Numpy functions described above for reading and saving data are very useful and simple to use. However, there are many times when you would like to write and read data to less structured files. For example, keeping a log of progress through your programme (i.e. similar to the print statements you'd use to write to the screen) or reading less structured data from another application.

10.4.1 Writing arbitrary data to a file.

If you are able to use and format *print* statements to the screen then writing data to a file is similar. Firstly, you need to create and open a file that you may 'write' to. Once you have the file open you use a *myfile.write()* statement (where *myfile* is the object you've assigned to your file) to write data to the file. At the end make sure that you close the file. If your programme crashes while the file is open you may need to close it manually from the console.

Note. The write statement allows a lot of flexibility in how the data is output but the argument to it must be a single string. The string needs to contain a placeholder {} at the point in the string at which the variable is to be output. The string should be followed by a '.format()' method that gives the variable for each placeholder. Further formatting in the way the variable is output can be included in the format method but we will not consider it here.

Finally, you should add an '\n' at the end of the string to make sure that the next write statement is on the next line.

The example below should hopefully give you the idea.

Important If the file already exists then it will be overwritten. If you wish to append data to an existing file then change the "w" to and "a" in the *open* statement. It is possible to check to see if a file exists before attempting to open a new one using a *try* ... *except* structure as we used above but this is beyond the scope of this course.

You may wish to inspect the contents of the file 'test.dat' that should have been written to the same folder as this notebook. Note it is all in readable text.

In [ ]:

```
import numpy as np
outfile=open("test.dat","w")
outfile.write("This is where I've put my data\n")
outfile.write("sin(pi/3) is {}\n".format(np.sin(np.pi/3.)))
ang=np.pi/4.
outfile.write(" Angle              sin                 cos                 tan\n")
outfile.write("{} {} {} {}\n".format(ang,np.sin(ang),np.cos(ang),np.tan(ang)))
outfile.close()
```

10.4.2 Reading data from an arbitary text file.

Note this will only work for a file that contains 'normal' text data (i.e. one you can load and read in text editor). The principle is the reverse of what you do with writing a file. Firstly, the file must exist if you are to read data from it. Also make sure that your file is in the same folder as the python code or that you have specified the full path fully in the open statement.

Once the file is open you read the data line by line into string variable.

What you do after you have read a line into your programme depends on what is in the data file and what you wish to do with it. Common things you may find, are lines of text that describe the data held in the file. This could be names of datasets, numbers of data points or any other relevant information. If these are not needed in your programme you ignore the line and read in the next line until you find the first line with the data you need. This may be as simple as skipping the first _n_ lines and then reading the data.

Another common form is for the data to be written with 'comment' lines using, for example, the # symbol as the first character. In this case, after you have read a line you check to see if the first chracter is a # and if so, you ignore the line and move on.

If you decide the line has valid data then you need to process the string file and convert the data to integer, float etc. variables in your code.

The example below may help you understand the process for the array *xydata* we saved above.

In [ ]:

```
infile = open("xydata.dat")
temp_data=[]
for dummy in infile:
    x, y = dummy.split(' ')
    temp_data += [(float(x),float(y))]
infile.close()
print (temp_data)
```

10.5 Summary

In this notebook you have learnt about the difference between text and binary (archive) files and you should be able to:

- write a programme that requests input from the user,
- write a programme that can write Numpy arrays to simple text or binary format files,
- write a programme that can read Numpy arrays that have been saved in files with the Numpy library,
- write a programme the can write data into a text file as strings,
- write a programme that can read arbitrary data from a textfile, line by line into strings for further processing.