Data analysis using Python

Student and Staff IT Introduction

ssiti2013@gmail.com*

26/06/2013

The Art of Programming

Programming is a science and an art. As we are going to do more complex thing with programming, you will soon discover there are many ways to do the same thing. At the beginning one may only want to get the right result, whatever the way. In such case, it is better to use a code as simple as possible, with the functions you know the best.
Good programming is hard to define. Some are looking for speed optimization, some are looking for memory optimization, some care a lot about elegance and style, which are quite subjective. A good program will often use a trade-off between speed and memory, depending on the user's need.

In any case, it is always better to follow good practice while writing scripts, such as:

- Keep your code simple
- Use explicit names for your variables, and file names
- Be consistent in your code, with name, spacing, etc.
- Comment your code. Imagine yourself in a couple weeks, trying to figure out what your script meant.

While coding remember to use the function **help()** as much as possible. It is also a good idea to have a "programming buddy", share your code, share your findings, improve together. Finally the internet is your best friend. There are an incredible amount of tutorials, lectures, help, cheat-sheet, Q&A, etc.. The answer is out there.

Setting up for today

As a reminder, you can program in python using the interpreter in the shell or using scripts. We are going to increase the level of complexity or our programs, in which case, scripts are highly recommended. It can never hurt to keep a shell interpreter open on the side, to check the help for example.
We still need a shell to run our script with the interpreter, so we will start by opening a shell, move to the Desktop and make a new directory called **ssiti3**. This will be our working directory for today. You will also need to copy the file *file1.txt* and *file2.txt* into this directory, which is in the ssiti directory on your Desktop.
Remember, you can do it in a few command lines:

```
$ cd  /Desktop
$ mkdir ssiti3
$ cd ssiti3
$ cp ../ssiti/file1.txt .
$ cp ../ssiti/file2.txt .
```

For each exercise, you should open a new script. There is a correction to each exercise at the end. Take time to look for a solution, that is the only way to learn. Do refer to it only when you succeeded or if you are completely stuck. Read them even if you succeeded, you may have a different solution, which is fine, but we tried to make the simplest ones including a couple of nice tricks.

*Document written by Elsa Guillot (e.guillot@massey.ac.nz), for IFS staff and students, Massey University.

Reading a file

Python has a type of object called **file**, which is what we are going to use. Reminder: to know everything about this type you can use the python interpreter:
`$ python >>> help(file)`

Open a new script *yop.txt*. We are going to start by reading the file *file1.txt*. First we need to open the file:

```
filename='file1.txt'
f=open(filename,'r')
print('the type of f is :')
print(type(f))
f.close()
```

We open the file with option **'r'** which means *read*.Any file open **MUST** be closed. Any operation on the file must be done between the open and close commands.

As a start we can call the function **readline()** on the file object. This will read a line:

```
filename='file1.txt'
f=open(filename,'r')
l=f.readline() # read the first line
print(l) # check in the shell
l=f.readline() # read the second line
print(l) # check in the shell
f.close()
```

Using this method you read one line, starting at the reading cursor and displace the reading cursor to the new line. The reading cursor is at the beginning of the file when you open it.

The easiest way to read a file is to used the function **readlines()** which load each line of the file in a list. Here is an example:

```
filename='file1.txt'
f=open(filename,'r')
ll=f.readlines()
print(ll)
f.close()
```

Can you recognize the type of the elements of the list *ll*?

Looking at *ll*, you will notice unexpected symbols in your variables like **\t** and **\n**: **\t** code for a tabulation, **\n** code for the end of a line. An other signal exists—though it does not appear here—when you read a file called **EOF** which refers to the end of the file. When python receives **EOF** signal, it will stop reading operations.

Using a **for** loop, make a script to read *file1.txt* and output the content of the file in the shell.
You will see that some empty lines appear that were not in the original file. Can you make those empty lines disappear? (Hint : use string concatenation + or the function **split** on string.). Look at the end for a correction.
Reminder: To read the original file, out of python, in command lines, you can use the command *cat* or *less*.

Using the **split()** function can you output only the second column of the file? Look at the end for a correction.
Hint: You already know how to extract each line. What separates the columns? Can you use this to extract each column on each line?

There are other methods to read files :
f.read(x) will read (at most) x bytes in your file. You can iterate on read until the end of the file.
f.seek(position, offset) will put the reading cursor at a specific position. Position must indicate the number of octet

after (or before if negative) the offset. The offset is either the beginning (0), the end (2), where the reading cursor is currently placed (1).

Write in a file

Now it is time to create file with our script. As before we need to start by opening a file, but with the option **'w'**, for *write*.

```
newfilename='newfile1.txt'
f=open(newfilename,'w')
f.close()
```

As before, we must close the file at the end! This will create a new file called *newfile1.txt*. As we do no operation on it the file will be empty. If a file already existed with the same name, the previous file will be deleted and replaced by the new one.
If you want to write a text at the end of an existing file you must use the option **'a'** instead of **'w'**.
To write in the file, use the function f.write(). Here is an example:

```
newfilename='newfile1.txt'
f=open(newfilename,'w')
f.write('Hello world')
f.close()
```

Can you write in a file the list of the first ten numbers (from 0 to 9) displayed on one line?
Can you write in a file the list of numbers from 10 to 100 displayed each on a different line?

If speeds really matter, you should know that it is faster to concatenate your output in a string, and then apply the function **write** in the whole string, instead of applying **write** many times. The same is true for the **print** function to display in the shell.

Reading a file, writing in another

It is possible to read and write in the same file. However it is not a good practice, if something goes wrong when you test your code, you will erase your data file. Therefore we will look into the safer method which is reading a file and writing in an other. You already know all the commands, you just have to be careful dealing with the different file without mixing up. Keep your variable names clear, and it should be easy. Here is the skeleton for a code to do that:

```
filenameToRead='file1.txt'
fToRead=open(filenameToRead,'r')

filenameToWrite='newfile2.txt'
fToWrite=open(filenameToWrite,'w')

# do something

fToWrite.close()
fToRead.close()
```

Can you read *file1.txt* and copy it into a new file called *file1_copy.txt*?
Can you read *file1.txt* and copy it into a new file called *file1_copy_with_dash.txt* replacing the tabulation in the file by a dash **-**?

Importing a library

Python's programming is heavily based on library. All the command we have been using so far belongs to the python language. They will be available on any computer where python is installed (with a similar version). In theory you could do any programming just using these command lines. However you would reinvent the wheel. You will want to

use functions that someone probably already coded before you, probably even better (i.e. faster) than you. There will also be functions that you may not know how to write yourself at all, but fortunately someone already did it and you can stay naive to the way it works (to a certain point). A library often needs to be installed on your system first, which means that your code loses portability.
Libraries will give you access to numerical manipulations, statistical analyses, GUI programming and many more. There are here to make your coding easier, use them!
Today we will look into a very popular library : **NumPy**[1] (Numerical Python). *NumPy is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.* (Wikipedia). Basically this library permits you to do operation similar to MatLab in Python (for free...) while optimizing the speed. It is open source and well maintained.

It is already installed on the computers of the physics lab. To use it in a script (or in the shell) you must load it with the function **import**.

```
import numpy
# do something
```

You can then use the help to look at the description of the library. Any function called that refers to **numpy**, will start with **numpy.yourFunction()**. To avoid writing numpy too many times we use a trick to make it shorter :

```
import numpy as np
# do something
```

If for example we want to access the value of Π which is in the library:

```
import numpy as np
mypi=np.pi
print(mypi)
```

Numpy provides function such as **mean** and **var** to provide the mean and the variance of a set of numbers. Have a look at their description in the help. Can you compute the mean of 3, 5, 7, 9, 11, 13, 17, 19 and 23? Their variance?

NumPy provides some very handy object such as multi-dimensional arrays which have a new type **numpy.ndarray**. To create a 1 dimension array full of zeros simply use **np.zeros(nbCell)**. You can then access and modify the cells as you did with a list. However the object is now of type **numpy.ndarray**.

```
import numpy as np
my1Darray=np.zeros(10)
firstElement=my1Darray[0]
my1Darray[3]=42 # modify the 4th cell
print(my1Darray) # output the full, now modified, array
print(type(myIntegerArray))
```

By default it will create an array of float. You can modify that using the option dtype within the function zeros. Here is an example. For more information look at the help.

```
import numpy as np
myIntegerArray=np.zeros(10,dtype='i') # creates an array of 0 of the type integer
print(myIntegerArray)
```

To create a 2 dimensional array, full of zeros use **np.zeros((nbCol,nbLines))**. As before default type is float but can be modified.

```
import numpy as np
myArray=np.zeros((5,4)) # create a 5x4 array
firstLine = myArray[0]
firstElementInTheFirstLine = myArray[0][0]
```

[1] http://www.numpy.org/

Based on the previous example, can you create a 10x2 array and access the first element of the second line? The last element of the first line?
np.zeros can create an array of any dimension the same way as 1D or 2D as long as insert a tuple of the dimension of your array : **np.zeros((3,4,2,6))** will have four dimensions for example.

There are other tools to create array of the type **numpy.ndarray**:
np.ones() creates array full of ones, it is called as the **np.zeros()** function with dimensions and eventually type.
np.empty() is also called the same way. It creates an "empty" array, i.e. it does not initialize the value in each cell. In such case the number within the cell could be anything.
np.identity(x) where **x** is an integer, creates the identity matrix of size **x*x**
np.arange() works like **range** but creates arrays of type **np.ndarray**. It can also creates arrays of float instead of integer if the numbers given are float themselves, i.e. **np.arange(5.0)** will create an array with 0,1,2,3 and 4 as floats.

You can also create directly your array with the numbers you want using **np.array**:

```
import numpy as np
myarray=np.array([[[1,2],[3,4]],[[5,6],[7,8]]]) # this creates a 3d array
print(myarray)
```

You can also use **np.array** to transform list into the **np.ndarray** type.

Can you create an array of 2 dimension, 10 lines, 10 column, displaying the 100 first non-zeros numbers (from 1 to 100)? Hint: to avoid entering 100 numbers remember the function **range**.

Matlab's user will be familiar the function linspace. NumPy provides the same one called **np.linspace(a,b,c)** with a,b,c integers. This function creates an array with **c** numbers from **a** to **b** uniformly distributed. Try :

```
import numpy as np
myarray=np.linspace(0,10,4)
print(myarray)
```

Data analysis

We have now imported NumPy which provides us with new mathematical tools and a very handy array type to manipulate data. NumPy is full of mathematical functions. We have seen **np.mean()** and **np.var()**, but there are also **np.exp()**, **np.sqrt()**,**np.square()**, **np.sin()**, **np.cos()**, **np.log()**,**np.sum()**,**np.min()**,**np.max()** for which the names are quite explicit. Look at the help and try them on simple numbers.

It is possible to apply mathematical operation on the **numpy.ndarray** type. For example:

```
import numpy as np
myarray=np.arange(1.0,11.0)
mynewarray=10*myarray
print(mynewarray)
mynewarray=myarray/3
print(mynewarray)
mynewarray=myarray+7
print(mynewarray)
```

It is also possible to apply **np.mean()** and **np.var()**, **np.exp()**, **np.sin()**, **np.cos()**, **np.log()**,**np.sum()**,**np.min()**,**np.max()** to those array.
Can you compute the sum of the square of the 10 first non zero integers (from 1 to 10) using NumPy functions?

We will now work on analyzing data from a file. For that you will work on the file *file1.txt* which (let's imagine[2]) contain three types of data, the name of a gene, the number of species expressing this gene, and the frequency of G bases in the

[2]This is a fake data file

gene sequence. You must load the data from your file into python.
Which data type should the name, the species and the G frequency should be represented by?

Save the names (first column) in a list, and the species number (second column) and the G frequency (third column) in a **numpy.ndarray** (one for each).

What is the maximum number of species that express by the same gene? the minimum? the mean? the variance?
What is the maximum frequency of G? the minimum?

Can you select the genes which have less than 10 percent of G? Output them in a file called *lowG.txt*.
Hint : it is easier to select while reading through the lines.

Python and command lines

Imagine you write a script to analyze data, such as previously, but you want to apply it on several files. Wouldn't it be convenient to precise the file in the command lines of python when you call the script? Remember we are launching our script like that:
`$python myscript.py`
We would like something like:
`$python myscript.py mydatafile.txt`
That is exactly what we are going to do here.

In order to handle the command line we will import an other library **sys** which is present on every python distribution, and its function **sys.argv**.
Using the normal command line '`python myscript.py`' try (Please replace myscript by the name of your script)

```
import sys
something=sys.argv
print(something)
```

Now try the same thing with '`python myscript.py file1.txt`'.
Can you figure out what sys.argv does?

Can you write the previous script (select the genes with less than 10% of G) such that the name of the file is given in the command lines instead of the script?

If you have done it correctly you should be able to apply now the same script on *file2.txt* which contain the same data for an other study[3].
Your script so far outputs a file called *lowG.txt*. Therefore it erased the previous results when writing the new ones. Can you modify the name of the output file depending on the name of the input file?

You are now ready for the world of programming, anything becomes possible. Good luck!

Corrections

Reading *file1.txt* with a nice output:
Using string concatenation:

```
filename='file1.txt' # filename to read
f=open(filename,'r') # open the file
ll=f.readlines() # load the lines
out='' # we create an empty string
```

[3]Those are not real data either

```
for l in ll: # for every line in the file
    out+=l # concatenate the strings
print(out)
f.close()
```

Or using **split**:

```
filename='file1.txt' # filename to read
f=open(filename,'r') # open the file
ll=f.readlines() # load the lines
for l in ll: # for every line in the file
    tmp=l.split('\n') #  split the string where there is a '\n'
    tmp2=tmp[0] # take the first part of this list, i.e. everything before the '\n'
    print(tmp2)
print(out)
f.close()
```

Reading only the second column of *file1.txt*:

```
filename='file1.txt' # filename to read
f=open(filename,'r') # open the file
ll=f.readlines() # load the lines
for l in ll: # for every line in the file
    tmp=l.split('\t') #  split the string where there is a '\t'
    tmp2=tmp[1] # take the second part of this list,
#i.e. everything between the first '\t' and the second one
    print(tmp2)
print(out)
f.close()
```

Writing the numbers from 0 to 1 on one line:

```
filename='oneToTen.txt'
f=open(filename,'w')
for i in range(10): # i takes the value form 0 to 9
    f.write('%d '%i) # each number is follow by a space
f.write('\n') # send the end of a line signal to create an empty line at the end
f.close()
```

Writing the numbers from 10 to 100 each on a line:

```
filename='tenToHundred.txt'
f=open(filename,'w')
for i in range(10,110,10): # i takes the value form 10 to 10 with a step of 10
    f.write('%d\n'%i) # each number is follow by a end of a line
f.close()
```

A faster version could be:

```
filename='tenToHundred.txt'
f=open(filename,'w')
output=''
for i in range(10,110,10): # i takes the value form 10 to 10 with a step of 10
    output+='%d\n'%i # using string concatenation
f.write(output)
f.close()
```

Copy *file1.txt* into *file1_copy.txt*

```
filename='tenToHundred.txt'
f=open(filename,'w')
output=''
for i in range(10,110,10): # i takes the value form 10 to 10 with a step of 10
    output+='%d\n'%i # using string concatenation
f.write(output)
f.close()
```

Compute the mean of 3, 5, 7, 9, 11, 13, 17, 19 and 23 :

```
import numpy as np
premNumbers=[3,5,7,9,11,13,17,19,23]
mymean=np.mean(premNumbers)
myvar=np.var(premNumbers)
print('the mean is %f'%mymean) # be careful here, the mean will be a float,
#therefore you should use %f
print('the variance is %f'%myvar)
```

Array of 2 dimension, 10 lines, 10 column, displaying the 100 first non-zeros numbers (from 1 to 100)

```
import numpy as np
myArray=np.zeros((10,10),dtype='i') #create the array
number=1
for i in range(10): # for each column
    for j in range(10): # for each line
        myArray[i][j]=number #change the number
        number+=1 #increase by 1
print(myArray)
```

Something a little bit more elegant:

```
import numpy as np
myArray=np.zeros((10,10),dtype='i')
for i in range(10):
    for j in range(10):
        myArray[i][j]=10*i+j
print(myArray)
```

Import the data from *file1.txt*:

```
import numpy as np
f=open('file1.txt','r') #open the file

ll=f.readlines()

nbGene=len(ll) # gives the size of the list ll, meaning the number of lines in the file,
# meaning the number of genes
names=[] # the names will be store in a list
S=np.zeros(nbGene,dtype='i') # the species will be in an array of integer,
#now filled with zeros
G=zeros(nbGene) # the G frequency will be in a an array of float,
#now filled with zeros

whichline=0 # count the line where we are
for l in ll: #for each line
    tmp=l.split('\t') #separate the column (we already know it is separated by tab
    name=tmp[0] # is already a string
```

```
    names.append(name)
    S[whichline]=int(tmp[1]) # must convert the string to an int
    G[whichline]=float(tmp[2].split('\n')[0])  #must be converted to a float,
    #careful do not forget to take off the end of a line signal from your file
    whichline+=1

# we check what we did by printing the variables
print(names)
print(S)
print(G)
```

Data analysis on *file1.txt*. What is the maximum number of species that express by the same gene? the minimum? the mean? the variance? What is the maximum frequency of G? the minimum?

```
import numpy as np
f=open('file1.txt','r') #open the file

ll=f.readlines()

nbGene=len(ll) # gives the size of the list ll, meaning the number of lines in the file,
#meaning the number of genes
names=[] # the names will be store in a list
S=np.zeros(nbGene,dtype='i') # the species will be in an array of integer,
#now filled with zeros
G=zeros(nbGene) # the G frequency will be in a an array of float,
#now filled with zeros

whichline=0 # count the line where we are
for l in ll: #for each line
    tmp=l.split('\t') #separate the column (we already know it is separated by tab
    name=tmp[0] # is already a string
    names.append(name)
    S[whichline]=int(tmp[1]) # must convert the string to an int
    G[whichline]=float(tmp[2].split('\n')[0])  #must be converted to a float,
    #careful do not forget to take off the end of a line signal from your file
    whichline+=1

# we check what we did by printing the variables
print(names)
print(S)
print(G)

maxNbSpecies=S.max()
minNbSpecies=S.min()
meanNbSpecies=S.mean()
varNbSpecies=S.var()
maxGfreq=G.max()
minGfreq=G.min()

print('At most %d express the same gene while at least only %d do'
%(maxNbSpecies,minNbSpecies)) # min and max will be integers
#as the number of species is an integer
print('The mean number of species expressing the same gene is %f and the var is %f'
%(maxNbSpecies,minNbSpecies)) # be careful mean, and var are float
print('The maximum frequency of G is %f and the minimum is %f
```

```
'%(maxGfreq,minGfreq))
```

Select the genes in *file1.txt* that have less than 10% of G and output them in *lowG.txt*.

```
import numpy as np
f=open('file1.txt','r') #open the file
fout=open('lowG.txt','w') # open the second file to write in

ll=f.readlines()

nbGene=len(ll) # gives the size of the list ll,
#meaning the number of lines in the file, meaning the number of genes

whichline=0 # count the line where we are
for l in ll: #for each line
    tmp=l.split('\t') #separate the column (we already know it is separated by tabulation
    name=tmp[0] # is already a string

    species=int(tmp[1]) # must convert the string to an int
    gfreq=float(tmp[2].split('\n')[0])  #must be converted to a float,
#careful do not forget to take off the end of a line signal from your file
    whichline+=1
    if(gfreq<0.1): # if low G frequency
        fout.write(l) # write the complete line into the second file
f.close()
fout.close()
```

Same script that reads the input file in command line:

```
import numpy as np
import sys # always good to import everything at the beginning,
#then someone reading your file will see the dependencies immediately

filename=sys.argv[1]
f=open(f,'r')
fout=open('lowG.txt,'w') # open the second file to write in

ll=f.readlines()

nbGene=len(ll) # gives the size of the list ll, meaning the number of lines in the file,
#meaning the number of genes

whichline=0 # count the line where we are
for l in ll: #for each line
    tmp=l.split('\t) #separate the column (we already know it is separated by tabulation
    name=tmp[0] # is already a string

    species=int(tmp[1]) # must convert the string to an int
    gfreq=float(tmp[2].split('\n')[0])  #must be converted to a float,
    #careful do not forget to take off the end of a line signal from your file
    whichline+=1
    if(gfreq<0.1): # if low G frequency
        fout.write(l) # write the complete line into the second file
f.close()
fout.close()
```

To modify the name of the output file:

```
***/ same /***

# fout=open('lowG.txt','w') # open the second file to write in
# comment to the old (or remove it)

newname=filename.split('.txt')[0]
newname='lowG_'+newname+'.txt'
fout=open(newname,'w')

***/ same /***
```

How to install NumPy

To install numpy on Linux you can simply do :
`sudo apt-get install python_numpy`
On macOS: `sudo port install py27-numpy`
On windows some python distribution already contain NumPy. Otherwise, and on any OS you can install from the sources from `http://www.numpy.org`.