1 Practical Bioinformatics – Day 5

1.1 Practicing file I/O: writing the distance matrix in CDT format

1.1.1 Load our data and repeat yesterday's distance calculation

```
import cdt
```

```
data = cdt.ExpressionProfile("supp2data.cdt")
```

(This import stanza is overkill, but useful if I expect to edit and reload the **pearson** function)

```
import stats
reload(stats)
from stats import pearson
```

```
def distmatrix(data, distance = pearson, N = None):
    D = []
    if(N is None):
        N = len(data.geneName)
    else:
        N = min(N, len(data.geneName))
    for i in xrange(N):
        row = []
        for j in xrange(N):
            if(j >= i):
                row.append(distance(data.num[i],data.num[j]))
            else:
                row.append(D[j][i])
        D.append(row)
    return D
```

Check that the calculation works for the first 100 genes

```
d = distmatrix(data, N = 100)
```

Scale up to the full matrix

```
%time D = distmatrix(data)
```

```
CPU times: user 2min 9s, sys: 36 ms, total: 2min 9s
Wall time: 2min 8s
```

–> ~Same runtime as yesterday – hurray for reproducibility

Intelly asked if this version of **pearson** handles missing data –> using IPython's **??** to look at the source code

```
pearson??
```

Checking that the matrix is the expected size:

```
len(D), len(D[0])
```

```
(2467, 2467)
```

Example 1: write the matrix as a simple tab delimited text file (Excel compatable)

```
out = open("mat1.txt","w")
for row in D:
    for value in row:
        s = str(value)
        out.write(s+"\t")
    out.write("\n")
out.close()
```

Example 2: Use **join** to place tabs only *between* elements *(removes a trailing tab relative to the fist example)*

```
out = open("mat2.txt","w")
for row in D:
    r = []
    for value in row:
        s = str(value)
        r.append(s)
    out.write("\t".join(r)+"\n")
out.close()
```

Example 3: Replace the explicit for loop with a list comprehension

```
out = open("mat2.txt","w")
for row in D:
    out.write("\t".join([str(value) for value in row])+"\n")
out.close()
```

Example 4: Replace the list comprehension with a generator comprehension *(compared to example 3, this avoids creating a temporary list in memory)*

```
out = open("mat2.txt","w")
for row in D:
    out.write("\t".join(str(value) for value in row)+"\n")
out.close()
```

If examples 3 and 4 are confusing – that's okay. Example 2 is a perfectly good way to do the problem, so it is enough if you understand that one.

Example 5: Writing a JavaTreeView-compatable CDT file

- Add a header line with two special columns (*ORF* and *NAME*) and the full set of gene names
- In each row add the appropriate gene name and annotation

We use a few tricks to do this:

- **zip** to iterate over three lists of equal length (gene names, annotations, and rows of the distance matrix)
- List concatenation (+) to add two new values to the list comprehension from example 3 *(note that the generator comprehension from example 4 won't work, because we can't concatenate a generator with a list)*

```
out = open("mat.cdt","w")
out.write("\t".join(["ORF","NAME"]+data.geneName)+"\n")
for (gene,anno,row) in zip(data.geneName, data.geneAnn, D):
    out.write("\t".join([gene,anno]+[str(value) for value in row])+"\n")
out.close()
```

A few (bash) shell commands to check the size of the generated file
In bytes:

```
!ls -l mat.cdt
```

```
-rw-rw-r-- 1 mvoorhie mvoorhie 95426749 May 24 16:07 mat.cdt
```

In human-friendly format:

```
!ls -lh mat.cdt
```

```
-rw-rw-r-- 1 mvoorhie mvoorhie 92M May 24 16:07 mat.cdt
```

As part of debugging memory errors, we tried writing just the upper-left 79x79 corner of the distance matrix:

```
out = open("mat_small.cdt","w")
out.write("\t".join(["ORF","NAME"]+data.geneName)+"\n")
for (gene,anno,row) in zip(data.geneName, data.geneAnn, D[:79]):
    out.write("\t".join([gene,anno]+[str(value) for value in row][:79])+"\n")
out.close()
```

Telling python to de-allocate the distance matrix:

```
del D
```

Note: this de-allocation will free up memory *inside* of python, but will not generally return memory from python to the operating system. In an operating system with good memory management (*e.g.*, Linux), this is okay, because the memory that python is not actively using will be swapped out in response to demands from elsewhere in the system.

For the JavaTreeView memory issues that some people ran into in class, memory use by python was probably *not* a factor. The core problem was that the default memory allocation in java was too small to handle the large distance matrix. We solved this by explicitly allocating 1 gigabyte of memory to java via:

```
java -jar -Xmx1024M TreeView.jar
```

We could save this as a bash script via:

```
    # Use cat to create the script from the command-line
    cat > javatreeview.sh << EOF
    #!/bin/bash
    java -jar -Xmx1024M TreeView.jar
    EOF
    # Make the script executable
    chmod "a+x" javatreeview.sh
```

```
%logstart -o BMS270b.2013.05.log
```

```
Activating auto-logging. Current session state plus future input saved.
Filename       : BMS270b.2013.05.log
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active
```