1 Practical Bioinformatics – Day 4

1.1 Introspecting classes

How can we figure out how to get at the data in our ExpressionProfile class?

Create an instance of the class:

```
import cdt
```

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

IPython's ? tells us:

- The type and class of an object
- Where to find the corresponding source code
- The signatures for any member functions, etc.
- Any available docstrings *(but we didn't provide any in the original version of our class)*

```
data?
```

?? tells us more, including the source code for the class, which is enough to remember the names of the member variables

```
data??
```

So, *e.g.*, we can get at the gene list like this:

```
data.geneName[:10]
```

```
['YBR166C',
 'YOR357C',
 'YLR292C',
 'YGL112C',
 'YIL118W',
 'YDL120W',
 'YHL025W',
 'YGL248W',
 'YIL146C',
 'YJR106W']
```

We can use **dir** to get a list of an object's attributes:

```
dir(data)
```

```
['__doc__',
 '__init__',
 '__module__',
 'expCond',
```

```
 'geneAnn',
 'geneName',
 'num',
 'write']
```

dir() with no arguments lists all top-level objects. For IPython in {pylab mode, this is a very long list. More useful is the IPython magic **%who**, which lists all top-level objects that we explicitly created.

```
%who
```

```
cdt data
```

We can also use **help** to view an object's docstrings (again, we haven't de ned any for ExpressionPro le, so its **help** output is terse).

```
help(data)
```

```
Help on instance of ExpressionProfile in module cdt:

class ExpressionProfile
 |  Methods defined here:
 |
 |  __init__(self, fn)
 |
 |  write(self, fname)
```

1.1.1 After updating cdt.py with docstrings

```
reload(cdt)
```

```
<module 'cdt' from 'cdt.py'>
```

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

```
help(data)
```

```
Help on instance of ExpressionProfile in module cdt:

class ExpressionProfile
 |  Annotated gene expression matrix.  Isomorphic to a CDT file.
 |
 |  Methods defined here:
 |
 |  __init__(self, fn)
 |      Initialize from the name of a CDT file.
 |
 |  write(self, fname)
 |      Save to fname in CDT format.
```

1.1.2 Digression

After de ning ExpressionPro le.num with a **@property** decorator, so that we could add a docstring to it.

```
reload(cdt)
```

```
<module 'cdt' from 'cdt.py'>
```

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

```
help(data)
```

```
Help on instance of ExpressionProfile in module cdt:

class ExpressionProfile
 |  Annotated gene expression matrix.  Isomorphic to a CDT file.
 |
 |  Methods defined here:
 |
 |  __init__(self, fn)
 |      Initialize from the name of a CDT file.
 |
 |  write(self, fname)
 |      Save to fname in CDT format.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  num
 |      Two dimensional array of log ratios for genes vs. conditions.
```

```
data.num[5][5:10]
```

```
[-0.12, 0.01, -0.36, -0.01, -0.17]
```

At this point, we rolled back the **@property** change to cdt.py to avoid confusion *(too late?)*

1.1.3 End digression

1.2 Calculating Pearson correlations for all pairs of genes

Load the **pearson** function from the example on the website

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

Our rst try at a function to calculate the pairwise correlations.
Rather than de ning it explicitly in terms of **pearson**, we ask for a **distance** parameter to be

passed to the function. We assume that **distance** is a mapping from two equal length vectors to a scalar, and that it is symetric ($D(x, y) = D(y, x)$)

We take advantage of the symetry to only calculate the upper triangle of the matrix, filling in the lower triangle by copying previously calculated values.

```
def distmatrix(data, distance):
    D = []
    for i in xrange(len(data.geneName)):
        row = []
        for j in xrange(len(data.geneName)):
            if(j >= i):
                row.append(distance(data.num[i],data.num[j]))
            else:
                row.append(D[j][i])
        D.append(row)
    return D
```

As defined, we have to remember to call **distmatrix** with both an ExpressionPro file *and* a distance function

```
dists = distmatrix(data)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-42361735885d> in <module>()
----> 1 dists = distmatrix(data)

TypeError: distmatrix() takes exactly 2 arguments (1 given)
```

Here, we redefine our function with **pearson** as the default distance function (to be used if this parameter is not supplied).

We also define a second optional parameter, **N**. If given, the calculation is run for only the first **N** genes – this is useful for quick debugging of our function, and for figuring out its run time as a function of problem size (*i.e.*, its computational complexity).

```
def distmatrix(data, distance = pearson, N = None):
    D = []
    if(N is None):
        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
```

Timing our function for the first 10 genes

```
%time dist = distmatrix(data, N = 10)
```

```
CPU times: user 8 ms, sys: 0 ns, total: 8 ms
Wall time: 6.32 ms
```

If we only supply two parameters, without explicitly stating which optional parameter we are supplying, python assumes that we are supplying the first optional parameter (**distance**).

```
dist = distmatrix(data, 10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-3098912401ab> in <module>()
----> 1 dist = distmatrix(data, 10)

<ipython-input-23-747ed186cb1e> in distmatrix(data, distance, N)
      7         for j in xrange(N):
      8             if(j >= i):
----> 9                 row.append(distance(data.num[i],data.num[j]))
     10             else:
     11                 row.append(D[j][i])

TypeError: 'int' object is not callable
```

Supplying all three parameters works (they're in the right order, so we don't need to be explicit about who's who)

```
%time dist = distmatrix(data, pearson, 10)
```

```
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 2.7 ms
```

Scaling up to the first 100 genes

```
%time dist = distmatrix(data, N = 100)
```

```
CPU times: user 244 ms, sys: 36 ms, total: 280 ms
Wall time: 262 ms
```

A quick redefinition of our function to handle **N** larger than the number of genes:

```
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
```

Scaling up to 1000 genes:

```
%time dist = distmatrix(data, N = 1000)
```

```
CPU times: user 20.6 s, sys: 2.44 s, total: 23.1 s
Wall time: 22.9 s
```

And the full calculation:

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

```
CPU times: user 2min 11s, sys: 6.8 s, total: 2min 18s
Wall time: 2min 17s
```

{$>$ run time of $\sim$2 minutes for a single core of a hyper-threaded 2.5 GHz dual core i5-3210M processor running in 64bit mode.

For a similar architecture ($e.g.$, another i5), single-core speed should depend linearly on the clock speed ($e.g.$, a 5 GHz processor would run twice as fast).

(Without explicitly setting up our problem for parallel processing, adding additional processor cores will give us no speed up).

For other architectures, run times are problem dependent (in this case, a oating point benchmark would be a good predictor of the relative run time).

See if you can gure out the time scaling for your computer.