1 Practical Bioinformatics { Day 3

1.1 Using \." to \look inside" an object

First, let's make a sequence (multiplication on a string gives tandem repeats)

```
seq = "ATGC"*50
```

```
seq
```

```
'ATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCAT
```

Here I:

- Call **rstrip** on seq, returning a string
- Call **split** on the returned string, returning a list of strings
- Slice the returned list of strings (internally, this is a call to the list's **__getitems__** method), returning a smaller list of strings

```
seq.rstrip("C").split("TG")[:10]
```

```
['A', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA', 'CA']
```

1.2 Functions vs. Classes

As an example, we will try two different ways of implementing reverse complementation for DNA

1.2.1 As a function

```
def complement(dna):
    retval = ""
    for i in dna:
        if(i == "A"):
            c = "T"
        elif(i == "T"):
            c = "A"
        elif(i == "G"):
            c = "C"
        else:
            c = "G"
        retval = c + retval
    return retval
```

```
s = "GATACA"
complement(s)
```

```
'TGTATC'
```

1.2.2 As a class method

```
class DNA:
    def __init__(self, seq):
        # A class constructor is a good place to sanitize/normalize
        # our data. Here, we normalize DNA sequences to uppercase
        # to simplify comparison
        self.seq = seq.upper()

    def complement(self):
        retval = ""
        for i in self.seq:
            if(i == "A"):
                c = "T"
            elif(i == "T"):
                c = "A"
            elif(i == "G"):
                c = "C"
            else:
                c = "G"
            retval = c + retval
        return retval
```

Constructing a DNA sequence:

- Python creates a new, empty DNA object
- Python calls DNA.__init__, passing the new DNA object as *self* and "GATACA" as *seq*
- Python assigns the new DNA object to **dna**

```
dna = DNA("GATACA")
```

Calling a class method:

- Python calls DNA.complement, passing **dna** as *self*

```
dna.complement()
```

```
'TGTATC'
```

Example of using python's built-in help. (I am commenting this out to keep my transcript small). Experiment with this!

```
#help(str)
```

1.3 Examples of pre-allocating vectors and matrices

Multiplication on lists is similar to strings – here I'm creating a list of ten zeros

```
l1 = [0]*10
```

```
l1
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Here's a 5x10 matrix of zeros as a list of lists (I use a for loop to avoid creating 10 linked copies of a single row)

```
l2 = []
for i in xrange(10):
    l2.append([0]*5)
```

```
l2
```

```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

The same matrix as a numpy array:

```
a2 = zeros((10,5))
```

```
a2
```

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Explicitly creating an array of 32-bit signed integers, rather than the default floating-point dtype
This is an example of passing an optional, named parameter to a function. You can give your

```
a3 = zeros((10,5), dtype = "int32")
```

```
a3
```

```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=int32)
```

1.3.1 Using indexing to change the values in a matrix

```
l2
```

```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

```
l2[3][2]
```

```
0
```

```
l2[3][2] = "ham"
```

```
l2
```

```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 'ham', 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

```
a2
```

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

```
a2[3][2] = 5
```

```
a2
```

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  5.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Note that numpy arrays can only contain one type of data:

```
a2[3][3] = "ham"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-4b7f8e330901> in <module>()
----> 1 a2[3][3] = "ham"

ValueError: could not convert string to float: ham
```

1.4 Homework problem: parsing a CDT le

Bryne doing a quick scouting run on the file, to see how the data is structured:

```
data = open("supp2data.cdt").readlines()
data[0]
```

```
'ORF\tNAME\talpha 0\talpha 7\talpha 14\talpha 21\talpha 28\talpha 35\talpha 42\talpha 49\talpha 56\talph
```

Bryne's homework solution as a python function

```
data[1]
```

```
'YBR166C\tTYR1   TYROSINE BIOSYNTHESIS    PREPHENATE DEHYDROGENASE (NADP+\t0.33\t-0.17\t0.04\t-0.07\t-0.
```

```
def sepData(fn):
    geneName=[]
    geneAnn=[]
    num= []

    dat = open(fn).readlines()

    expCond = [i.strip() for i in dat[0].split("\t")[2:]]

    for i in dat[1:]:
        w= i.split("\t")
        geneName.append(w[0])
        geneAnn.append(w[1])
        row = []
        for j in w[2:]:
            try:
                row.append(float(j))
            except ValueError:
                row.append(0.)
        num.append(row)

    return geneName,geneAnn,expCond,num
```

```
a = sepData("supp2data.cdt")
```

Four return values, as expected:

```
len(a)
```

```
4
```

Checking that the parsed data has the expected shape (2467 genes by 79 conditions)

```
len(a[0])
```

```
2467
```

```
len(a[2])
```

```
79
```

Assigning a tuple to a tuple – this is a good trick for unpacking a return value:

```
geneName,geneAnn,expCond,num = sepData("supp2data.cdt")
```

1.4.1 Digression – lists vs. generators

The important point is that lists represent a sequence by actually allocating it in memory, whereas a generator simply emits each element of a sequence one at a time, without allocating the full sequence.

The parsing example resumes at **In [64]**

```
range(5)
```

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

```
for i in range(5):
    print i **2
```

```
0
1
4
9
16
```

```
for i in xrange(5):
    print i**2
```

```
0
1
4
9
16
```

```
x = range(5)
y = xrange(5)
```

```
x
```

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

```
y
```

```
xrange(5)
```

```
xi = iter(x)
yi = iter(y)
```

```
xi.next()
```

```
0
```

```
xi.next()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-58-ef459a0034b7> in <module>()
----> 1 xi.next()

StopIteration:
```

```
yi.next()
```

```
0
```

```
yi.next()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-54-b381cd96e979> in <module>()
----> 1 yi.next()

StopIteration:
```

```
x
```

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

```
y
```

```
xrange(5)
```

```
xi = iter(x)
xi.next()
```

```
0
```

```
yi = iter(y)
yi.next()
```

```
0
```

1.4.2 Parsing example continued

We converted Bryne's parsing function to a class (in cdt.py on the website) to which we added an output **write** function

Importing the module:

```python
import cdt
```

Parsing the cdt file by creating an instance of our class, ExpressionProfile, which results in a call to the __init__ function that we wrote:

```python
data1 = cdt.ExpressionProfile("supp2data.cdt")
```

Using dir to look inside of our class instance:

```python
dir(data1)
```

```
['__doc__',
 '__init__',
 '__module__',
 'expCond',
 'geneAnn',
 'geneName',
 'num',
 'write']
```

Trying out the write method:

```python
data1.write("test1.cdt")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-67-5ce7d47be52d> in <module>()
----> 1 data1.write("test1.cdt")

/home/mvoorhie/Projects/Courses/PracticalBioinformatics/python/May2013/cdt.py in write(self, fname)
     29         out = open(fname,"w")
     30         out.write("\t".join(["ORF","NAME"]+self.expCond)+"\n")
---> 31         for (name, anno, data) in (self.geneName, self.geneAnn, self.num):
     32             out.write("\t".join([name,anno]+[coerce_num(i) for i in data])+"\n")
     33         out.close()

ValueError: too many values to unpack
```

Oops! I forgot to use **zip** in my for loop → fixed the bug and saved the file

```python
reload(cdt)
```

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

Still buggy → data1 is still bound to the old version of the class

```
data1.write("test1.cdt")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-69-5ce7d47be52d> in <module>()
----> 1 data1.write("test1.cdt")

/home/mvoorhie/Projects/Courses/PracticalBioinformatics/python/May2013/cdt.py in write(self, fname)
     29         out = open(fname,"w")
     30         out.write("\t".join(["ORF","NAME"]+self.expCond)+"\n")
---> 31         for (name, anno, data) in zip(self.geneName, self.geneAnn, self.num):
     32             out.write("\t".join([name,anno]+[coerce_num(i) for i in data])+"\n")
     33         out.close()

ValueError: too many values to unpack
```

If I create a new class instance, it is bound to the new (reloaded) version

```
data2 = cdt.ExpressionProfile("supp2data.cdt")
```

```
data2.write("test2.cdt")
```

I can dynamically rebind data1 to the new version of the class by directly reassigning its __class__ attribute (note that this *does not* result in a call to __init__)

```
data1.__class__
```

```
cdt.ExpressionProfile
```

```
data1.__class__ = cdt.ExpressionProfile
```

```
data1.write("test1.cdt")
```

A quick check that all rows of the log ratio matrix are the same length.

To do this I am using a **set**, which is an unordered set of unique elements.

I initialize the set with a generator comprehension for the sequence of lengths of the rows in the log ratio matrix.

Because all 2467 values emitted by the generator are identical (they are all the integer 79), **set**'s __init__

```
{79}
```

```
%logstart -o BMS270b.2013.03.log
```

```
Activating auto-logging. Current session state plus future input saved.
Filename       : BMS270b.2013.03.log
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active
```