1 Practical Bioinformatics – Day 6

1.1 Using dictionaries for indexing

Load the array data from last week

```
import cdt
```

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

If we know the row we are interested in, we can do lookups by simple list indexing:

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

```
'YNL272C'
```

```
data.geneAnn[10]
```

```
'SEC2   SECRETION        GDP/GTP EXCHANGE FACTOR FOR SEC4P'
```

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

```
[0.31, 0.12, 0.34, 0.61, 0.18, 0.28, 0.14, 0.0, -0.07, -0.01]
```

A python dictionary is a type of *associative array*.
Define python dictionaries using curly braces enclosing a list of colon-delimited key-value pairs:

```
d = {"spam":6,3:"onion","four":"five"}
```

```
d2 = {("eggs","four"):5}
```

The number of keys in a dictionary is its length:

```
len(d)
```

```
3
```

Indexing a dictionary is like indexing a list, but with keys in place of integer indices:

```
d["spam"]
```

```
6
```

```
d[3]
```

```
'onion'
```

Trying to look up a non-existant key raises a **KeyError**

```
d["eggs"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-14-1835664bf5b8> in <module>()
----> 1 d["eggs"]

KeyError: 'eggs'
```

We can also use indexing to assign a key to a new value or to add a new key,value pair:

```
d["spam"] = 7
```

```
d["spam"]
```

```
7
```

```
d["eggs"] = "fried"
```

```
d["eggs"]
```

```
'fried'
```

Use **del** to remove a key,value pair *(it is rare that you'll need this)*

```
del d["spam"]
```

```
d["spam"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-20-eb605e19a614> in <module>()
----> 1 d["spam"]

KeyError: 'spam'
```

Application: Index the microarray data by systematic gene name.

Our mapping will be **name** to **row index**, so that we can use the same dictionary to do lookups on both annotations (**data.geneAnn**) and expression profiles (**data.num**).

```
gene_index = {}
n = 0
for i in data.geneName:
    gene_index[i] = n
    n += 1
```

GAL4, a transcriptional regulator of galactose catabolism in *Saccharomyces*, has the systematic

gene name “YPL248C”.

Use the dictionary to lookup the row index for GAL4:

```
gene_index["YPL248C"]
```

```
2047
```

Use the row index to confirm our name-based lookup:

```
data.geneName[gene_index["YPL248C"]]
```

```
'YPL248C'
```

Look up the corresponding annotation:

```
data.geneAnn[gene_index["YPL248C"]]
```

```
'GAL4   GALACTOSE REGULATION     TRANSCRIPTIONAL ACTIVATOR'
```

And first 10 columns of the corresponding expression profile:

```
data.num[gene_index["YPL248C"]][:10]
```

```
[-0.06, -0.29, -0.29, 0.01, -0.18, -0.29, -0.15, -0.3, -0.18, 0.03]
```

Create the dictionary from **In [22]**, using **enumerate** to simplify the code:

```
gene_index = {}
for (n,i) in enumerate(data.geneName):
    gene_index[i] = n
```

Generate a many-to-many mapping of annotation keywords to row indices.

In order to support multiple rows for a single keyword, we will have our dictionary map from strings (keys) to lists of integers (values).

```
keywords = {}
for (n,i) in enumerate(data.geneAnn):
    # Split annotation on whitespace -> list of words
    for keyword in i.split():
        try:
            # If we've seen this keyword before -> append the current row index to its
                list
            keywords[keyword].append(n)
        except KeyError:
            # The first time we've seen this keyword -> make a new list for it
            keywords[keyword] = [n]
```

The keys in a dictionary are unique. Assigning multiple values to a single key results in a single copy of that key mapped to the last value:

```
d = {"A":3,"A":4,"A":5}
```

```
d
```

```
{'A': 5}
```

The **unique** key property of dictionaries is very useful for creating sets of unique elements.

Python includes a special data type, **set**, expressly for this purpose. A set is like a dictionary with empty values (*i.e.*, just keys).

Here, we for a set on each keyword list so that we can iterate over just the unique keywords in each annotation:

```
keywords = {}
for (n,i) in enumerate(data.geneAnn):
    for keyword in set(i.split()):
        try:
            keywords[keyword].append(n)
        except KeyError:
            keywords[keyword] = [n]
```

Demonstrating the uniqueness of **set** elements:

```
s = set("AAAAABBBCCDCCDAD")
```

```
s
```

```
{'A', 'C', 'B', 'D'}
```

```
s2 = set("AAABBDDDDEEE")
```

```
s2
```

```
{'A', 'B', 'E', 'D'}
```

sets define many useful operations from set theory – very useful for "Venn diagram"-type calculations (*c.f.* **help(set)** for a full list)

```
s2.difference(s)
```

```
{'E'}
```

```
s.difference(s2)
```

```
{'C'}
```

```
s.intersection(s2)
```

```
{'A', 'B', 'D'}
```

Number of unique keywords in the annotation column:

```
len(keywords)
```

```
4312
```

Number of genes with "GALACTOSE" in their annotation:

```
len(keywords["GALACTOSE"])
```

```
4
```

Information for all four of those genes:

```
for i in keywords["GALACTOSE"]:
    print data.geneName[i], data.geneAnn[i], data.num[i][:10]
```

```
YML051W GAL80  GALACTOSE REGULATION     TRANSCRIPTIONAL REPRESSOR [0.0, -0.17, -0.22, 0.06, -0.1, 0.0, 0
YJL168C SET2   GALACTOSE REGULATION     TRANSCRIPTIONAL REPRESSOR OF GAL4 [-0.34, -0.03, -0.07, -0.1, -0
YPL248C GAL4   GALACTOSE REGULATION     TRANSCRIPTIONAL ACTIVATOR [-0.06, -0.29, -0.29, 0.01, -0.18, -0.
YLR081W GAL2   TRANSPORT        GLUCOSE AND GALACTOSE PERMEASE [-0.03, -0.03, 0.03, -0.27, 0.07, -0.27,
```

1.1.1 Some useful dictionary methods

keys(): a list of the dictionary's keys in undefined but deterministic order:

```
keywords.keys()[:10]
```

```
['FUSION;',
 'L42B',
 'HDF1',
 'SILENCED',
 'L22A',
 'TELOMERE',
 'PRP6',
 'PRP4',
 'L42A',
 'PRP2']
```

values(): the values, in the same order as **keys()**

```
keywords.values()[:10]
```

```
[[151, 178],
 [1595],
 [1052],
 [2119, 2213],
 [1599],
 [78, 96, 150, 299, 1011, 1969, 2024],
```

```
[1405],
[414],
[1603],
[1952]]
```

items(): (key,value) pairs, in the same order as **keys()** – this is usually what you want when writing a **for** loop over a dictionary:

```
keywords.items()[:10]
```

```
[('FUSION;', [151, 178]),
 ('L42B', [1595]),
 ('HDF1', [1052]),
 ('SILENCED', [2119, 2213]),
 ('L22A', [1599]),
 ('TELOMERE', [78, 96, 150, 299, 1011, 1969, 2024]),
 ('PRP6', [1405]),
 ('PRP4', [414]),
 ('L42A', [1603]),
 ('PRP2', [1952])]
```

1.2 Using dictionaries to transform nucleotide sequences

A dictionary mapping the Watson-Crick base-pairing rules:

```
nuc = {"A":"T","T":"A","G":"C","C":"G","N":"N"}
```

A test sequence to play with:

```
seq = "GATACA"*20
```

```
print seq
```

```
GATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGA
```

1.2.1 Exercise: Given a $5' \to 3'$ sequence, return its $3' \to 5'$ antisense

First try: returns the correct sequence, but as a list:

```
anti = []
for i in seq:
    anti.append(nuc[i])
print anti
```

```
['C', 'T', 'A', 'T', 'G', 'T', 'C', 'T', 'A', 'T', 'G', 'T', 'C', 'T', 'A', 'T', 'G', 'T', 'C', 'T', 'A
```

We can use **join** to put the list together into a string (we call **join** from an empty string in order to join the elements with no separator)

```
anti = []
for i in seq:
    anti.append(nuc[i])
print "".join(anti)
```

```
CTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCT
```

Alternatively, we can append to a string directly, rather than using a list:

```
anti = ""
for i in seq:
    anti += nuc[i]
print anti
```

```
CTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCT
```

Here's the second solution as a function:

```
def antisense(seq):
    anti = ""
    for i in seq:
        anti += nuc[i]
    return anti
```

Printing the input and antisense sequences on adjacent lines is a good way to check the output. We'll use this trick a lot when working with sequence alignments (note that IPython, most terminals, and most programmer-oriented text editors use a fixed-width font, to aid in this sort of comparison).

```
print seq
print antisense(seq)
```

```
GATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGATACAGA
CTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCTATGTCT
```

1.2.2 Exercise: Given a $5' \rightarrow 3'$ sequence, return its $5' \rightarrow 3'$ reverse complement

We came up with a lot of ways to walk backwards over the sequence:

Via explicit indexing math:

```
anti = ""
for i in xrange(len(seq)):
    anti += nuc[seq[len(seq)-i-1]]
print anti
```

```
TGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTC
```

By including start, stop, and step parameters when calling xrange:

```
anti = ""
for i in xrange(len(seq)-1,-1,-1):
```

```
    anti += nuc[seq[i]]
print anti
```

```
TGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTC
```

By using the **reversed** generator:

```
anti = ""
for i in reversed(seq):
    anti += nuc[i]
print anti
```

```
TGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTC
```

By using special splice syntax directly on the string (this works for lists as well):

```
anti = ""
for i in seq[::-1]:
    anti += nuc[i]
print anti
```

```
TGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTC
```

Alternatively, we could walk forward over the sequence but *prepend* the new characters to the output string:

```
anti = ""
for i in seq:
    anti = nuc[i] + anti
print anti
```

```
TGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTC
```

All of the above approaches are reasonable solutions.
Here, I choose **In [60]** because:

- appending to a string is more efficient than prepending
- using string's slice method directly is the richest way for me to communicate my intent to the string object, providing maximum opportunity for optimization by the string class *(this doesn't guarantee increased efficiency, but it leaves the door open)*

```
def revcomp(seq):
    anti = ""
    for i in seq[::-1]:
        anti += nuc[i]
    return anti
```

```
revcomp(seq)
```

```
'TGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCTGTATCT
```

Our dictionary isn't defined for lower case sequences:

```
revcomp("atgc")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-64-9fc2deb7f315> in <module>()
----> 1 revcomp("atgc")

<ipython-input-62-c7690cdbafe7> in revcomp(seq)
      2     anti = ""
      3     for i in seq[::-1]:
----> 4         anti += nuc[i]
      5     return anti

KeyError: 'c'
```

Rather than augmenting the dictionary, it is better to normalize all inputs to upper-case (this way, we only have to deal with the ambiguity at one place in our program):

```
revcomp("atgc".upper())
```

```
'GCAT'
```

```
s = "atgc"
s = s.upper()
revcomp(s)
```

```
'GCAT'
```

Generating a simple scoring matrix:

```
scores = {}
for i in "ATGC":
    scores[i] = {}
    for j in "ATGC":
        if(i == j):
            scores[i][j] = 1
        else:
            scores[i][j] = -1
```

```
scores
```

```
{'A': {'A': 1, 'C': -1, 'G': -1, 'T': -1},
 'C': {'A': -1, 'C': 1, 'G': -1, 'T': -1},
 'G': {'A': -1, 'C': -1, 'G': 1, 'T': -1},
 'T': {'A': -1, 'C': -1, 'G': -1, 'T': 1}}
```

Some example sequence pairs for the homework problems:

```
s1 = "ATCGAAA"
s2 = "ATGCAAA"
```

```
s3 = "AT-GCAA-"
s4 = "ATCGCAAA"
```

```
%logstart -o BMS270b.2013.06.log
```

```
Activating auto-logging. Current session state plus future input saved.
Filename       : BMS270b.2013.06.log
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active
```