Bioinformatics Programming Class 5

Aim of the exercise

In this class we will develop a codon translation program. We will do that in a two steps, first will be a program that reads the codon table (for each codon, which amino acid goes with it) from file, and prints out all codons for each amino acid. This we will call “transpose.py”, and it will work as follows:

```
./transpose.py my_codon_table.txt↵
A ['GCA', 'GCC', 'GCG', 'GCT']
C ['TGT', 'TGC']
E ['GAG', 'GAA']
……
```

The second script will also read the codon table, and an input DNA or RNA sequence and print the corresponding protein sequences (more, because of reading frames). This we will call “translate.py”, and it will work as follows:

```
./translate.py my_codon_table.txt
ATGATAACAAGGACCTGACTACTAGCTAGCTAGCTAGCATTAGGATCGCAGAGCTACGAT↵
MITRT*LLAS*LALGSQSYD
```

But before we start this we will first need to introduce file operations, dictionaries and some more string parsing.

File operations

In previous classes you got to work with command line arguments and text input. In many cases it is much more convenient to put (all) input in a file. In python, a few lines of code will get you started with reading from file. Download '`codon_table.txt`' from the wiki and try the following in the python shell:

```
>>> file=open('codon_table.txt')
>>> print file
```

Q1: what does python return? Can you understand what that might mean?

Now try the following:

```
>>> line = file.readline()
>>> print line
```

What do you get? Hint: open the same file in a text editor.
Can you explain why you get the empty line?

Q2: what type of object is `file.readline()`?

Now reading again:

```
>>> line = file.readline()
```

```
>>> print line
```

Q3: What do you get? Is this what you expect? (Note, the comment in previous lesson on functions that may have side effects when used multiple times – this is an example of that.)

Now we read only one line at a time, but we may want to go through all lines in a file one by one. Previously, you used `for` to loop through items in a list. In python, files act much the same as lists, in fact it acts exactly like a list of lines, like this:

```
>>> for line in file:
...     print line
...
```

We could also use a `while` loop (from previous class) to do exactly the same, but that turns out to be much more cumbersome.

Finally, we can also write to a file. First we need to open a file for writing (careful! If the file already exists, it will be emptied!):

```
>>> out_file = open('out.txt', 'w')
```

Now we can print to this file:

```
>>> print >>out_file, 'Hello world'
>>> out_file.close()
```

Open `out.txt` in a text editor. Note, the `out_file.close()` is needed, because otherwise the output may still be 'on the way' to the file, but not yet there.

We could now read our input file again, and write it to the output file:

```
>>> file=open('codon_table.txt')
>>> out_file = open('out.txt', 'w')
>>> for line in file:
...     print >>out_file, line
...
>>> out_file.close()
```

If you look into `out.txt` in a text editor (re-read or revert if you still have it open), you will see the empty lines again that we saw before. That is because `file.readline()` considers the 'new line' character as part of the line, and reads it in. But `print` always adds another 'new line' – your empty line is there. To suppress that, write '`print >>out_file, line,`' instead (note the trailing comma!). Try it.

Dictionaries – storing objects and finding it back

We can now read a file, but we also want to store the information in a practical way. We could simply make a list of lines, but depending on what we need it for, that is not practical. In our case, we want to be able to look up codons (translation, remember?), so it would be nice if we had a list where we could do something like:

```
>>> print codon_table['ATG']
Methionine
```

It turns out, python has something much like that; they are called dictionaries. Much like a real one, you can put stuff in, and get it out using a keyword. Try the following:

```
>>> dictionary = {}
```
This creates an empty dictionary.
```
>>> dictionary[0]='Apple'
```
This still lokes a lot like a list.
```
>>> dictionary['Mac']='Apple'
```

Q4: What happens if you do that with a list?

You can get it out again as well:
```
>>> print dictionary[0]
>>> print dictionary['Mac']
```
You can't get stuff out that isn't there:
```
>>> print dictionary['PC']
```

Now have a look at how this is stored:
```
>>> print dictionary
```
You'll see that both 0 and 'Mac' are keys.

We will try to get a bit more familiar with dictionaries, because you will get to find them so useful you will probably start using them anywhere!
```
>>> dictionary.keys()
>>> dictionary.values()
```

Q5: do you understand what you are getting here?

Just as with files and lists, you can loop through the items in it:
```
>>> for thing in dictionary:
...     print 'thing:', thing, 'dictionary:', dictionary[thing]
...
```

Finally, remember you can put anything in a dictionary. The example used strings, but numbers work to. So do lists or other dictionaries! (In fact, any object, so even files or functions – don't worry about that – but could you see how you could rewrite your calculator script of Class 2?)

Now: storing stuff from file

Combining file input and dictionaries (and a tiny bit of string parsing), we now have the tools we need to put our colon table in a dictionary. Later, we can look up triplets from our input sequence in the dictionary to translate (sounds logical, doesn't it?).

So, we need to go through three steps:

1) go through the file, line by line
2) for each line, get the words (tab separated – a tab in python is `'\t'`)
3) use the first word (codon) as key, and the second word (single letter abbreviated amino acid) as value in your codon_table dictionary.

You should be able to piece this together from what you just learned in the lessons above.
Make a function out of it that gets the file as argument and returns a translation dictionary.
(Go back to Class 4, sect. 2.1 for a short introduction on how to write functions, if you need.)

Still with me? Because this is only the first step.

Next, we want to do something useful with our data, that is in the following section.

Using the stored stuff

The final bit is to read an input DNA sequence string from an input file, and print a translated amino acid sequence string to an output file. You will want to write another function that gets a DNA sequence and a translation dictionary as input, and returns an amino acid sequence. Since codons are three nucleotides long, you will want to step through your input sequence by three at a time.

Try this:

```
>>> seq='ATGATAACAAGGACC'
>>> print len(seq)
>>> print range(len(seq))
>>> print range(0, len(seq))
>>> print range(0, len(seq), 3)
```

Q6: Do you see how you can make steps of three? Go back to Class 2, section 2.1 for an explanation on range(), if you need.

Now we use slices to get isolated triplets out of our sequence:

```
>>> for i in range( <fill this in> ):
...     print seq[i:i+3]
```

Q7: Can you explain what happens here? Slices were introduced in Class 3, sect. 2.1, for lists – but you should know that python treats a string as a list of characters as well.

Now try again, but with `seq='ATGATAACAAGGAC'`
Q8: what is the difference? How could you avoid this error?

Translator

You are now ready to combine all the pieces and write your calculator. Possible extensions:

- translate for all six (why six?) possible reading frames
- recognize start and stop codons; put start codon at start of line and stop codon at end of line
- checking if input from file is correct

Please do hand in your script in the next class: please hand it in on paper, so that we can give you some feedback.