# Bootstrappers: Python 2, Session 4

February 12, 2015 by Chris MacKay with Mike Purcaro & the GSBS Bootstrappers

contact: christopher.mackay@umassmed.edu

## Outline

This lesson we will be using bioinformatics problems from the Rosalind website to illustrate some concepts in *object-oriented* programming

- Rosalind Problems:
  - more on classes and methods
  - briefly go over module and package basics Link
  - using basic command line arguments Link

## Rosalind Problems:

these are the problems we will go over, in this order:

1. DNA Counting DNA Nucleotides
2. RNA Transcribing DNA into RNA
3. REVC Complementing a Strand of DNA
4. GC Computing GC Content
5. HAMM Counting Point Mutations
6. PROT Translating RNA into Protein
7. SPLC RNA Splicing
8. SUBS Finding a Motif in DNA
9. PRTM Calculating Protein Mass
10. REVP Locating Restriction Sites

## Problem 1: DNA

- *Given:* A DNA string s of length at most 1000 nt.
- *Return:* Four integers (separated by spaces) counting the respective number of times that the symbols ‘A’, ‘C’, ‘G’, and ’T' occur in s.

**Try this:**

- solve the problem with a `DNASequence` class, with a method that returns a count of all bases
- what if your DNA sequence has `"N"` s?

## Possible Solution:

```
from collections import defaultdict

class DNASequence(object):
    def __init__(self, sequence):
        self.sequence = sequence

    def countBases(self):
        count_dict = defaultdict(int)
        for base in self.sequence:
            count_dict[base] += 1
        return count_dict


# put sequence from rosalind here:
raw_sequence = 'AGCTAGCTAGATCG'

my_DNASeq_object = DNASequence(raw_sequence)

my_count = my_DNASeq_object.countBases()

## to get the output Rosalind asks for:

print ' '.join([my_count['A'], [my_count['C'], [my_count['G'], [my_count['T']
```

## Notes on the above solution:

### `defaultdict`

This is a subclass of the standard python dictionary that does not give you an error if your key does not already exist in the dictionary. `defaultdict` can be found in the `collections` package, which comes standard

with every python installation. For this reason we need to import it form collections using `from collections import defaultdict`.

When using a `defaultdict` dictionary, if you ask python for `my_dict[my_key]` and it is not in the dictionary, `defaultdict` returns the *zero* value for the type the dictionary was created for:

- a `defaultdict(int)` returns a `0`
- a `defaultdict(str)` returns an empty string `''`
- a `deafultdict(list)` returns an empty list `[]`
- etc…

### why are we using a class?

You might be asking, why are we using a class here, when you could more quickly get an answer without it…

Once the class is set up, you can see that it takes two simple and very straight forward lines of code to go from the raw sequence to having a count of all the bases in that sequences.

Hopefully the benefits of classes and objects will become more obvious as we continue…

---

# Problem 2: RNA

- *Given:* A DNA string *t* having length at most 1000 nt.
- *Return:* The transcribed RNA string of *t*.

**Try this:**

- how about adding a new method to your `DNASequence` class that returns an RNA version of the DNA sequence?

## Possible Solution:

appending our code from above:

```
class DNASequence(object):
    def __init__(self, sequence):
        self.sequence = sequence

    def countBases(self):
        count_dict = defaultdict(int)
        for base in self.sequence:
            count_dict[base] += 1
        return count_dict

    def transcribe(self):
        rna = self.sequence.replace('T','U')
        return rna

# put sequence from rosalind here:
raw_sequence = 'AGCTAGCTAGATCG'

my_DNASeq_object = DNASequence(raw_sequence)

my_RNA_string = my_DNASeq_object.transcribe()

print my_RNA_string
```

### Notes on the above solution:

`string.replace(old_substring, new_substring)`

Here we use a built-in function of strings called `replace()`. A simple google search *python string replace* can help you to discover this technique: here you go

---

# Problem 3: REVC

- *Given:* A DNA string $s$ of length at most 1000 bp.
- *Return:* The reverse complement $s^c$ of $s$.

**Try this:**

- how about adding a new method to your `DNASequence` class?

### Possible Solution:

appending our code from above by adding a `reverseComplement` function to the `DNASequence` class…

```
class DNASequence(object):
    def __init__(self, sequence):
        self.sequence = sequence

    def countBases(self):
        count_dict = defaultdict(int)
        for base in self.sequence:
            count_dict[base] += 1
        return count_dict

    def transcribe(self):
        rna = self.sequence.replace('T','U')
        return rna

    def reverseComplement(self):
        watson_crick = {'A':'T', 'C':'G', 'G':'C', 'T':'A'}
        complement = ''
        for base in self.sequence:
            complement += watson_crick[base]
        rev_complement = complement[::-1]
        return rev_complement

# put sequence from rosalind here:
raw_sequence = 'AGCTAGCTAGATCG'

my_DNASeq_object = DNASequence(raw_sequence)

my_rev_comp = my_DNASeq_object.reverseComplement()

print my_rev_comp
```

## Notes on the above solution:

Here we create a dictionary based on Waston-Crick basepairing, allowing us to use one base as a key and return its complement as the value.

Additionally we use the python idiom `list[::-1]` to reverse the list.

# Problem 4: GC

- *Given:* At most 10 DNA strings in FASTA format (of length at most 1 kbp each).
- *Return:* The ID of the string having the highest GC-content, followed by the GC-content of that string on the next line.

**Try this:**

- try creating a `FASTAFile` class with a method that returns a `DNASequence` object for each sequence in the file
- how about adding a new method to your `DNASequence` class that calculates the GC content?

## Possible Solution

Here we add a `FASTAFile` class to our code and a function to the `DNASequence` class that returns the GC percentage…

```
from collections import defaultdict

class DNASequence(object):
    def __init__(self, sequence, id):
        self.sequence = sequence

    def countBases(self):
        count_dict = defaultdict(int)
        for base in self.sequence:
            count_dict[base] += 1
        return count_dict

    def transcribe(self):
        rna = self.sequence.replace('T','U')
        return rna

    def reverseComplement(self):
        watson_crick = {'A':'T', 'C':'G', 'G':'C', 'T':'A'}
        complement = ''
        for base in self.sequence:
            complement += watson_crick[base]
        rev_complement = complement[::-1]
        return rev_complement

    def GCContent(self):
```

```
        counts = self.count()
        gc = counts['G']+counts['C']
        gc_percent = float(gc)/len(self.sequence)
        return gc_percent

class FASTAFile(object):
    def __init__(self, path):
        self.path = path

    def sequences(self):
        found_sequences = []
        with open(self.path, 'r') as input:
            sequence = ''
            for i, line in enumerate(input):
                if i == 0:
                    id = line.rstrip('\n').lstrip('>')
                else:
                    if line.startswith('>'):
                        new_sequence_object = DNASequence(sequence, id)
                        found_sequences.append(new_sequence_object)
                        sequence = ''
                        id = line.rstrip('\n').lstrip('>')
                    else:
                        sequence += line.rstrip('\n')
            else:
                new_sequence_object = DNASequence(sequence, id)
                found_sequences.append(new_sequence_object)

        return found_sequences

path = '/path/to/your/fasta/file.fasta'

my_sequences = FASTAFile(path).sequences()

my_sequences.sort(key = lambda x: x.GCContent(), reverse = True)

top_GC = my_sequences[0]

print top_GC.id
print top_GC.GCContent()
```

## Notes on the above solution:

### Parsing the FASTA file

FASTA files are annoying because each sequence starts with a `>`, but then can continue for a unspecified number of lines before ending.

There are a number of ways you could address this and the approach above, is just one (somewhat convoluted) example.

While going line by line through a FASTA file, if a `>` is found, that line is stripped of the `>` and `'\n'` and stored as the current `id`. The following lines are then sequentially added to the growing `sequence` string.

If a new `>` is found, this indicates that the previous sequence has ended, and a new sequence is starting. Therefore the previous sequence and id is recorded in the form of a `DNASequence` object which is then appended onto the `found_sequences` list. Once the previous sequence is added to `found_sequences` the id for the next sequences is saved as `id`, and the `sequence` variable is reset to an empty string (`''`) so that it is ready to receive the subsequent lines of sequence.

Using this approach there are two scenarios when encountering a `>` cannot be used to trigger creating and storing the previous sequence as a `DNASequence`:

#### 1. at the beginning of the file

At the beginning of the file there is no previous sequence, so trying to store the previous sequence results in an empty `DNASequence` object.

In order to address this, I set up a special case where on the first line of the file, I don't trigger the creating and storing of a `DNASequence` object. Instead i just save the first line as an `id` and move on.

To do this I used `for i, line in enumerate(file)`. When you use a `for` loop over an iterable item (list, string, etc.) you get each element in that item returned to you during each loop.

By using `enumerate(thing)` you get each item, but you also get the index of each item. See the example below:

```
>>> test = ['this', 'is', 'a', 'test', 'list']
>>> for element in enumerate(test):
...     print element
...
(0, 'this')
(1, 'is')
(2, 'a')
(3, 'test')
(4, 'list')

# each element is a tuple containing the index (position) of the item in
# the list and then the item itself

# you can *unpack* these two elements by giving python two variable names
# (i and item) to assign to each tuple. Python can figure it out from there.

>>> for i, item in enumerate(test):
...     print i
...     print item
...
0
this
1
is
2
a
3
test
4
list
```

**2. at the end of the file**

At the end of the file there is no next `>` to trigger creating and storing the last sequence. For this reason we have added a `else` clause that is connected to our `for` loop.

`for...else` is a way of executing code once the `for` loop iteration is over. Once the `for` loop is complete, whatever is in the `else` statement is then executed, and then the loop is over.

This method of linking an action to the termination of the `for` loop with `else` helps to keep clear (to you and anyone else who might read your code) that these two pieces of code are linked together, and are considered part of the same loop structure.

## Sorting Lists

At the end of the above solution we have our list of `DNASequence` objects (stored as `my_sequences`) and we then need to find which sequence in this list has the highest GC content, so that we can return the correct answer to *Rosalind*.

One way to do this is to sort this list, from highest to lowest, and then use the zeroth sequence.

There are at least two ways to sort a list:

### 1. Sorting a list with `sorted(list)`

`sorted(list)` creates a newly sorted list and leaves the old list intact.

```
my_list = [67, 81, 24, 100]
test = sorted(my_list)
print test  # [24, 67, 81, 100]

# you can set reverse equal to True, to reverse the order of the list
test = sorted(my_list, reverse = True)
print test # [100, 81, 67, 24]

# you can even define your own function to arrive at a key for each item in the list
# here I have a function that uses the last digit of each item as the key

# function to return the last digit:
def getKey(item):
    string_item = str(item)
    return int(string_item[-1])

#using the getKey function
test = sorted(my_list, key = getKey)
print test # [100, 81, 24, 67]

# using an *anonymous* (ie lambda) function to do the same thing,
# but with one line of code:

# (here x is equivalent to item)
test = sorted(my_list, key = lambda x: int(str(x)[-1]))
print test # [100, 81, 24, 67]
```

### 2. Sorting a list with `list.sort()`

`list.sort()` sorts the list in place with out creating a new list (*NO NEW LIST IS MADE*).

```
my_list = [67, 81, 24, 100]
my_list.sort()
print my_list # [24, 67, 81, 100]

my_list.sort(reverse = True)
print my_list # [100, 81, 67, 24]

def getKey(item):
    string_item = str(item)
    return int(string_item[-1])

my_list.sort(key = getKey)
print my_list # [100, 81, 24, 67]

my_list.sort(key = lambda x: int(str(x)[-1]))
print my_list # [100, 81, 24, 67]
```

**TO BE CONTINUED…**

# Problem 5: HAMM

- *Given:* Two DNA strings $s$ and $t$ of equal length (not exceeding 1 kbp).
- *Return:* The Hamming distance $d_H(s, t)$.

**Try this:**

- add another method to your `DNASequence` class

# Adding default attributes to classes and using keyword arguments

```
class Gene(DNASequence):
    def __init__(self, sequence, id = None)
        self.sequence = sequence
        self.id = id

# this sequence will have a self.id equal to None
new_sequence = DNASequence('ATCGCTAGAGCT')

# if you don't want to keep remembering which order the arguments need to be in,
# just use the argument keywords and equal signs:

next_sequence = DNASequence(id = 'seq_35452',
                            sequence = 'TGCTAGCTGAATCA')
```

# Problem 6: PROT

- *Given:* An RNA string $s$ corresponding to a strand of mRNA (of length at most 10 kbp).
- *Return:* The protein string encoded by $s$. of that string on the next line.
- **HELP**: codon table

**Try this:**

- create a new `RNASequence` class that has a method `translate()`

# Problem 7: SPLC

- *Given:* A DNA string $s$ (of length at most 1 kbp) and a collection of substrings of $s$ acting as introns. All strings are given in FASTA format.
- *Return:* A protein string resulting from transcribing and translating the exons of $s$. (Note: Only one solution will exist for the dataset provided.)

**Try this:**

- use your `FASTAFile` class
- could you create a method of the `DNASequence` class called `splice(introns)` that takes a list of `DNASequence` objects as an argument?

    - eg: `def splice(self, introns):...`

- could you slightly modify your `RNASequence.translate()` method from **Problem 6** and apply it to this `DNASequence`?

---

# Problem 8: SUBS

- *Given:* Two DNA strings $s$ and $t$ (each of length at most 1 kbp).
- *Return:* All locations of $t$ as a substring of $s$.

**NOTE:**

- python uses 0-based counting, but is that what *Rosalind* is looking for?

**Try this:**

- adding a new method to your `DNASequence` class

---

# Problem 9: PRTM

- *Given:* A protein string $P$ of length at most 1000 aa.
- *Return:* The total weight of $P$. Consult the monoisotopic mass table.

**Try This:**

- try creating a protein class, and a `mass()` method.

---

# tangent on Modules...

Say you have a function called `reverseComplement` in a `myCode.py` file which is in the same directory as this script.

You could import your code from `myCode.py`, and access each *class* or *function* or *global variable* in `myCode.py` by using the convention `myCode.name_of_item_you_want`

Here is an example:

---

```
import myCode
new_rev_comp = myCode.reverseComplement(sequence)
```

Additionally, if `myCode` is too long or cumbersome for you, you can reassign it a new value like so:

```
import myCode as my
new_rev_comp = my.reverseComplement(sequence)
```

If you don't need *every* function, class, and variable found in `myCode.py` you can explicitly import the thing you want, in which case you don't need to put `myCode` in front of it to use it. This is useful in that it tells readers of your code explicitly which parts of other documents you will be using:

```
from myCode import reverseComplement
new_rev_comp = reverseComplement(sequence)
```

Finally, you can combine the direct call of a piece of code from a file, with the name reassignment to get a streamlined name:

```
from myCode import reverseComplement as rc
new_rev_comp = rc(sequence)
```

to read more on modules and packages go here

---

# Problem 10: REVP

- *Given:* Given: A DNA string of length at most 1 kbp in FASTA format.
- *Return:* The position and length of every reverse palindrome in the string having length between 4 and 12. You may return these pairs in any order.

```
4 6
5 4
6 6 ...
```

**Try this:**

- try to call in some of your previously written classes and functions from another `.py` file…

---