Illustrating Python via Examples from Bioinformatics

Hans Petter Langtangen[1,2]

Geir Kjetil Sandve[2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Mar 12, 2012

1 Basic Bioinformatics Examples in Python

Life is definitely digital. The genetic code of all living organisms are represented by a long sequence of simple molecules called nucleotides, or bases, which makes up the Deoxyribonucleic acid, better known as DNA. There are only four such nucleotides, and the entire genetic code of a human can be seen as a simple, though 3 billion long, string of the letters A, C, G, and T. Analyzing DNA data to gain increased biological understanding is much about searching in (long) strings for certain string patterns involving the letters A, C, G, and T. This is an integral part of *bioinformatics*, a scientific discipline addressing the use of computers to search for, explore, and use information about genes, nucleic acids, and proteins.

Below are some simple examples on DNA analysis that brings together basic building blocks in programming: loops, `if` tests, and functions.

1.1 Counting Letters in DNA Strings

Given some string `dna` containing the letters `A`, `C`, `G`, or `T`, representing the bases that make up DNA, we ask the question: how many times does a certain base occur in the DNA string? For example, if `dna` is `ATGGCATTA` and we ask how many times the base `A` occur in this string, the answer is 3.

A general Python implementation answering this problem can be done in many ways. Some solutions are presented below.

List Iteration. The most straightforward solution is to loop over the characters in the string, test if the current character equals the desired one, and if so, increase a counter. Looping over the characters is obvious if the characters are stored in a list. This is easily done by

```
>>> list('ATGC')
['A', 'T', 'G', 'C']
```

Our first solution becomes

```
def count_v1(dna, base):
    dna = list(dna)  # convert string to list of characters
    i = 0            # counter
    for c in dna:
        if c == base:
            i += 1
    return i
```

String Iteration. Python allows us to iterate directly over a string without converting it to a list:

```
>>> for c in 'ATGC':
...     print c
A
T
G
C
```

In fact, all objects in Python which contains a set of elements in a particular sequence allow a `for` loop construction of the type `for element in object`.

A slight improvement of our solution is therefore to iterate directly over the string:

```
def count_v2(dna, base):
    i = 0 # counter
    for c in dna:
        if c == base:
            i += 1
    return i

dna = 'ATGCGGACCTAT'
base = 'C'
n = count_v2(dna, base)
print '%s appears %d times in %s' % (base, n, dna)
```

Program Flow. For correct programming it is a fundamental importance to be able to simulate the program above by hand, statement by statement. Three tools are effective for helping you reach the required understanding for doing a manual simulation: (i) printing variables, (ii) using a debugger, and (iii) using an online program flow tool.

Inserting `print` statements and examining the about help to demonstrate what is going on:

```
def count_v2_demo(dna, base):
    print 'dna:', dna
    print 'base:', base
    i = 0 # counter
    for c in dna:
        print 'c:', c
        if c == base:
            print 'True if test'
            i += 1
    return i
```

```
n = count_v2_demo('ATGCGGACCTAT', 'C')
```

An efficient way to explore this program is to run it in a debugger where we can step through each statement and see what is printed out. Launch `ipython` and run the program with a debugger: `run -d programname.py`. Use `s` (for step) to step through each statement, or `n` (for next) for stepping without also stepping through functions.

```
ipdb> s
> /some/disk/user/bioinf/src/count_v1.py(2)count_v2_demo()
1     1 def count_v1_demo(dna, base):
----> 2     print 'dna:', dna
      3     print 'base:', base

ipdb> s
dna: ATGCGGACCTAT
> /some/disk/user/bioinf/src/count_v1.py(3)count_v2_demo()
      2     print 'dna:', dna
----> 3     print 'base:', base
      4     i = 0 # counter
```

Observe the output of the `print` statement. One can also print a variable explicitly:

```
ipdb> print base
C
```

Misunderstanding of the program flow is one of the most frequent sources of programming errors, so whenever in doubt about any program flow, enter a debugger to establish confidence.

The Python Online Tutor is, at least for small programs, a spendid alternative to debuggers. Go to the webpage, erase the sample code and paste in your own code. Press *Visual execution*, then *Forward* to execute statements one by one. To the right the status of variables are explained and the text field below the program shows the output. An example is shown in Figure 1.

Index Iteration. Although it is natural in Python to iterate over the characters in a string (or more generally over elements in a sequence), programmers with experience from other languages (Fortran, C and Java are examples) are used to `for` loops with an integer counter running over a all indices in a string or array:

```
def count_v3(dna, base):
    i = 0 # counter
    for j in range(len(dna)):
        if dna[j] == base:
            i += 1
    return i
```

Python indices always start at 0 so the legal indices for our string become `0`, `1`, ..., `len(dna)-1`, where `len(dna)` is the number of characters in the string `dna`. The `range(x)` function returns a list of integers `0`, `1`, ..., `x-1`, implying that `range(len(dna))` generates all the legal indices for `dna`.

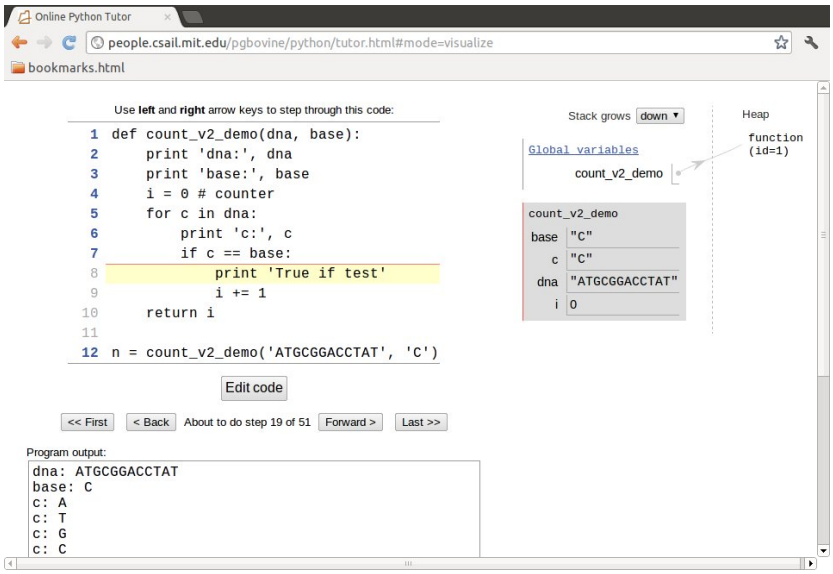

Figure 1: Visual execution of a program using the Python Online Tutor.

While Loops. The `while` loop equivalent to the last function reads

```
def count_v4(dna, base):
    i = 0 # counter
    j = 0 # string index
    while j < len(dna):
        if dna[j] == base:
            i += 1
        j += 1
    return i
```

Correct indentation is here crucial: a common error is to fail indenting the `j += 1` line correctly.

Summing a Boolean List. The idea now is to create a list `m` where `m[i]` is `True` if `dna[i]` equals the character we search for (`base`). The number of `True` values in `m` is then the number of `base` characters in `dna`. We can use the `sum` function to find this number because doing arithmetics with boolean lists automatically interprets `True` as `1` and `False` as `0`. That is, `sum(m)` returns the number of `True` elements in `m`. A possible function doing this is

```
def count_v5(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        if c == base:
            m.append(True)
        else:
            m.append(False)
    return sum(m)
```

Inline If Test. Shorter, more compact code is often a goal if the compactness enhances readability. The four-line `if` test in the previous function can be condensed to one line using the inline `if` construction: `if condition value1 else value2`.

```
def count_v6(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        m.append(True if c == base else False)
    return sum(m)
```

Using Boolean Values Directly. The inline `if` test is in fact redundant in the previous function because the value of the condition `c == base` can be used directly: it has the value `True` or `False`. This saves some typing and adds clarity, at least to Python programmers:

```
def count_v7(dna, base):
    m = []   # matches for base in dna: m[i]=True if dna[i]==base
    for c in dna:
        m.append(c == base)
    return sum(m)
```

List Comprehensions. Building a list via a `for` loop can often be condensed to one line by using list comprehensions: `[expr for e in sequence]`, here `expr` is some expression normally involving the iteration variable `e`. In our example, we can introduce a list comprehension

```
def count_v8(dna, base):
    m = [c == base for c in dna]
    return sum(m)
```

Here it is tempting to reduce the function body to a single line:

```
def count_v9(dna, base):
    return sum([c == base for c in dna])
```

Using a Sum Iterator. The DNA string is usually huge - 3 billion characters for the human specie. Making a boolean array with `True` and `False` values therefore increases the memory usage by a factor of two in our sample functions `count_v5` to `count_v9`. Summing without actually storing an extra list is desireable. Fortunately, `sum([x for x in s])` can be replaced `sum(x for x in s)`, where the latter sums the elements (`x`) in `s` as `x` visits the elements of `s` one by one. Removing the brackets therefore avoids first making a list and then applying `sum` on that list. This is a minor modification of the `count_v9` function:

```
def count_v10(dna, base):
    return sum(c == base for c in dna)
```

Generating Random DNA Strings. It is obvious that function `count_v9` doubles the memory requirements compared with `count_v10`, since storage for both `dna` and `m` is required when using `count_v9`. But which one is the fastest? To answer the question we need some test data, which should be a huge string `dna`. We could write

```
N = 1000000
dna = 'A'*N
```

to make a string `'AAA...A` that is `N` characters long, and this would be sufficient for testing efficiency. Nevertheless, it is more exciting to work with a DNA string with characters from the whole alphabet A, C, G, and T. To make a DNA string with a random composition of the characters we can first make a list of random characters and then join all those characters to a string:

```
import random
alphabet = list('ATGC')
dna = [random.choice(alphabet) for i in range(N)]
dna = ''.join(dna)  # join the character elements to a string
```

The `random.choice(x)` function selects an element in the list `x` at random.

Note that `N` is very often a large number. In Python version 2.x, `range(N)` generates a list of `N` integers. We can avoid this by using `xrange` which generates an integer at a time and not the whole list. In Python version 3.x, the `range` function is actually the `xrange` function in version 2.x. Using `xrange`, combining the statements, and wrapping the construction of a random DNA string in a function, gives

```
import random

def generate_string(N, alphabet='ATCG'):
    return ''.join([random.choice(alphabet) for i in xrange(N)])

dna = generate_string(6000000)
```

The call `generate_string(10)` may generate something like `AATGGCAGAA`.

Measuring Efficiency. Our next goal is to generate a very long string `dna` and see how much time the `count_v9` and `count_v10` functions spend on counting letters in that string. Measuring the time spent in a program can be done by the `time` module as follows:

```
import time
...
t0 = time.clock()
```

```
# do stuff
t1 = time.clock()
cpu_time = t1 - t0
```

The `time.clock` function returns the CPU time spent in the program since its start.

Running through a set of functions and recording timings can be done by

```
import time
timings = []
functions = [count_v1, count_v2, count_v3, count_v4,
             count_v5, count_v6, count_v7, count_v8,
             count_v9, count_v10, count_v11]
for function in functions:
    t0 = time.clock()
    function(dna, 'A')
    t1 = time.clock()
    cpu_time = t1 - t0
    timings.append(cpu_time)
```

In Python, functions are ordinary objects so making a list of functions is no more special than making a list of strings.

We can now iterate over `timings` and `functions` simultaneously via `zip` to make a nice printout of the results:

```
for cpu_time, function in zip(timings, functions):
    print '%s: %.1f s' % (function.func_name, cpu_time)
```

Timings on a MacBook Air 11" running Ubuntu showed that the functions using `list.append` required almost the double of the time of the functions that worked with list comprehensions.

Extracting Indices. Instead of making a boolean list with elements expressing whether a character matches the given `base` or not, we may collect all the indices of the matches. This can be done by adding an `if` test to the list comprehension:

```
def count_v11(dna, base):
    return len([i for i in range(len(dna)) if dna[i] == base])
```

A debugger or the Python Online Tutorial do not help so much to understand this compact code. A better approach is to examine the list comprehension in an interactive Python shell:

```
>>> dna = 'AATGCTTA'
>>> base = 'A'
>>> indices = [i for i in range(len(dna)) if dna[i] == base]
>>> indices
[0, 1, 7]
>>> print dna[0], dna[1], dna[7]  # check
A A A
```

The element `i` in the list comprehension is only made when the corresponding character in `dna` equals `base`.

Using Python's Library. Very often when you set out to do a task in Python, there is already functionality for the task in the object itself, in the Python libraries, or in third-party libraries found on the Internet. Counting how many times a character (or substring) `base` appears in a string `dna` is simply done by `dna.count(base)`:

```
def count_v12(dna, base):
    return dna.count(base)
```

A lesson learned is: google around before you start out to implement what seems to be a quite simple task. Others have probably already done it for you. And better: `dna.count(base)` runs over 30 times faster than the best of our handwritten Python functions! The reason is that the `for` loop needed to count in `dna.count(base)` is implemented in C and runs very much faster than loops in Python.

1.2 Computing Frequencies

Your genetic code is essentially the same from you are born until you die, and the same in your blood and your brain. Which genes that are turned on and off make the difference between the cells. This regulation of genes is orchestrated by an immensely complex mechanism, which we have only started to understand. A central part of this mechanism consists of molecules called transcription factors that float around in the cell and attach to DNA, and in doing so turn nearby genes on or off. These molecules bind preferentially to specific DNA sequences, and this binding preference pattern can be represented by a table of frequencies of given symbols at each position of the pattern. More precisely, each row in the table corresponds to the bases A, C, G, and T, while column j reflects how many times the base appears in position j in the DNA sequence.

For example, if our set of DNA sequences are TAG, GGT, and GGG, the table becomes

base	0	1	2
A	0	1	0
C	0	0	0
G	2	2	2
T	1	0	1

From this table we can read base A appears once in index 1 in the DNA strings, base C does not appear at all, base G appears twice in all positions, and base T appears once in the beginning and end of the strings.

In the following we shall present different data structures to hold such a table and different ways of computing them. The table is known as a *frequency matrix* in bioinformatics, and this is the term used below.

Separate Frequency Lists. Since we know that there are only four rows in the frequency matrix, an obvious data structure would be four lists, each holding a row. A function computing these lists may look like

```
def freq_lists(dna_list):
    n = len(dna_list[0])
    A = [0]*n
    T = [0]*n
    G = [0]*n
    C = [0]*n
    for dna in dna_list:
        for index, base in enumerate(dna):
            if base == 'A':
                A[index] += 1
            elif base == 'T':
                T[index] += 1
            elif base == 'G':
                G[index] += 1
            elif base == 'C':
                C[index] += 1
    return A, T, G, C
```

We need to initialize the lists with the right length and a zero for each element, since each list element is to be used as a counter. Creating a list of length `n` with object `x` in all positions is done by `[x]*n`. Finding the proper length is here done by inspecting the length of the first element in `dna_list`, assuming that all elements have the same length.

In the `for` loop we use the `enumerate` function which is used to extract both the element value and the element index when iterating over a sequence. For example,

```
>>> for index, base in enumerate(['t', 'e', 's', 't']):
...   print index, base
...
0 t
1 e
2 s
3 t
```

Here is a call and printout of the results:

```
dna_list = ['GGTAG', 'GGTAC', 'GGTGC']
A, T, G, C = freq_lists(dna_list)
print A
print T
print G
print C
```

with output

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

Nested List. The frequency matrix can also be represented as a nested list `M` such that `M[i][j]` is the frequency of base `i` in position `j` in a DNA string. Here `i` is an integer, where 0 corresponds to A, 1 to C, 2 to G, and 3 to T. The frequency is the number of times base `i` appears in position `j` in a set of DNA

strings. Sometimes this number is divided by the number of DNA strings in the set so that the frequency is between 0 and 1. Note all the DNA strings must have the same length.

The simplest way to make a nested list is to insert the `A`, `T`, `C`, and `G` lists into another list:

```
>>> frequency_matrix = [A, T, G, C]
>>> frequency_matrix[2][3]
2
>>> G[3]  # same element
2
```

Nevertheless, we can illustrate how to compute this type of nested list directly:

```
def freq_list_of_lists_v1(dna_list):
    # Create empty frequency_matrix[i][j] = 0
    # i=0,1,2,3 corresponds to A,T,G,C
    # j=0,...,length of dna_list[0]
```

As in the case with individual lists we need to initialize all elements in the nested list to zero.

A call and printout,

```
dna_list = ['GGTAG', 'GGTAC', 'GGTGC']
frequency_matrix = freq_list_of_lists_v1(dna_list)
print frequency_matrix
```

results in

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

Dictionary for More Convenient Indexing. The `if` tests in the `freq_list_of_lists_v1` are somewhat cumbersone, especially if want to extend the code to other bioinformatics problems where the alphabet is larger. What we want is a mapping from `base`, which is a character, to the corresponding index 0, 1, 2, or 3. A Python dictionary represents such mappings:

```
>>> base2index = {'A': 0, 'T': 1, 'G': 2, 'C': 3}
>>> base2index['G']
2
```

With the `base2index` dictionary we do not need the series of `if` tests and the alphabet `'ATGC'` could be much larger without affecting the length of the code.

```
def freq_list_of_lists_v2(dna_list):
    frequency_matrix = [[0 for v in dna_list[0]] for x in 'ATGC']
    base2index = {'A': 0, 'T': 1, 'G': 2, 'C': 3}
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base2index[base]][index] += 1

    return frequency_matrix
```

Numerical Python Array. As long as each sublist in a list of lists has the same length, a list of list can be replaced by a Numerical Python (`numpy`) array. Processing of such arrays is often much more efficient than processing of the nested list data structure. To initialize a two-dimensional `numpy` array we need to know its size, here 4 times `len(dna_list[0])`. Only the first line in the function `freq_list_of_lists_v2` needs to be changed in order to utilize a `numpy` array:

```
import numpy as np

def freq_numpy(dna_list):
    frequency_matrix = np.zeros((4, len(dna_list[0])), dtype=np.int)
    base2index = {'A': 0, 'T': 1, 'G': 2, 'C': 3}
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base2index[base]][index] += 1

    return frequency_matrix
```

The resulting `frequency_matrix` object can be indexed as `[b][i]` or `[b,i]`, with integers `b` and `i`. A typical indexing is `frequency_matrix[base2index['C'],i]`.

Dictionary of Lists. Instead of going from a character to an integer index via `base2index`, we may prefer to index `frequency_matrix` by, e.g., `['C'][14]`. This is the most natural syntax for a user of the frequency matrix. The relevant Python data structure is then a dictionary of lists. That is, `frequency_matrix` is a dictionary with keys `'A'`, `'T'`, `'C'`, and `'G'`. The value for each key is a list. Let us now also extend the flexibility such that `dna_list` can have DNA strings of different lengths. The lists in `frequency_list` will have lengths equal to the longest DNA string. A relevant function is

```
def freq_dict_of_lists_v1(dna_list):
    n = max([len(dna) for dna in dna_list])
    frequency_matrix = {
        'A': [0]*n,
        'T': [0]*n,
        'G': [0]*n,
        'C': [0]*n,
        }
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base][index] += 1

    return frequency_matrix
```

Running

```
frequency_matrix = freq_dict_of_lists_v1(dna_list)
import pprint   # for nice printout of nested data structures
pprint.pprint(frequency_matrix)
```

results in the output

```
{'A': [0, 0, 0, 2, 0],
 'C': [0, 0, 0, 0, 2],
 'G': [3, 3, 0, 1, 1],
 'T': [0, 0, 3, 0, 0]}
```

The initialization of `frequency_matrix` in the above code can be made more compact by using a dictionary comprehension:

```
dict = {key: value for key in some_sequence}
```

Here,

```
frequency_matrix = {base: [0]*n for base in 'ATGC'}
```

Adopting this construction in the `freq_dict_of_lists_v1` function leads to

```
def freq_dict_of_lists_v2(dna_list):
    n = max([len(dna) for dna in dna_list])
    frequency_matrix = {base: [0]*n for base in 'ATGC'}
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base][index] += 1

    return frequency_matrix
```

Dictionary of Dictionaries. The dictionary of lists data structure can alternatively be replaced by a dictionary of dictionaries object, often just called a dict of dicts object. That is, `frequency_matrix[base]` is a dictionary with the index `i` as key and the added number of occurences of `base` in `dna[i]` for all `dna` strings in the list `dna_list`. The indexing `frequency_matrix['C'][i]` and the value are exactly as before; the only difference is whether `frequency_matrix['C']` is a list or dictionary.

Our function working with `frequency_matrix` as a dict of dicts is written

```
def freq_dict_of_dicts_v1(dna_list):
    n = max([len(dna) for dna in dna_list])
    frequency_matrix = {base: {index: 0 for index in range(n)}
                        for base in 'ATGC'}
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base][index] += 1

    return frequency_matrix
```

Using Dictionaries with Default Values. The manual initialization of each subdictionary to zero,

```
    frequency_matrix = {base: {index: 0 for index in range(n)}
                        for base in 'ATGC'}
```

can be simplified by using a dictionary with default values for any key. The construction `defaultdict(lambda: obj)` makes a dictionary with `obj` as default value. This construction simplifies the previous function a bit:

```
from collections import defaultdict

def freq_dict_of_dicts_v2(dna_list):
    n = max([len(dna) for dna in dna_list])
    frequency_matrix = {base: defaultdict(lambda: 0)
                        for base in 'ATGC'}
    for dna in dna_list:
        for index, base in enumerate(dna):
            frequency_matrix[base][index] += 1

    return frequency_matrix

frequency_matrix = freq_list_of_lists_v2(dna_list)
pprint.pprint(frequency_matrix)
frequency_matrix = freq_dict_of_dicts_v1(dna_list)
pprint.pprint(frequency_matrix)
frequency_matrix = freq_dict_of_dicts_v2(dna_list)
pprint.pprint(frequency_matrix)
```

Remark. Dictionary comprehensions were new in Python 2.7 and 3.1, but can be simulated in earlier versions by making (key, value) tuples via list comprehensions. A dictionary comprehension

```
d = {key: value for key in sequence}
```

is then constructed as

```
d = dict([(key, value) for key in sequence])
```

1.3 Analyzing the Frequency Matrix

Having built a frequency matrix out of a collection of DNA strings, it is time to use it for analysis. A typical question is: for a given position in the DNA string, which of A, T, G, or C has the highest frequency (highest count)? We can then build a new DNA string with the most frequent character for each position. This is in bioinformatics known as finding consensus from a frequency matrix.

For example, if the frequency matrix looks like this (list of lists, with rows corresponding to A, T, G, and C),

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

we see that for position 0, which corresponds to column 0 in the table, T has the highest frequency (3). The maximum frequencies for the other positions are seen to be T for position 1, A for position 2, and G for position 3. The consensus string is therefore TTAG.

(**hpl**: *This can be more precisely and better explained?*)

List of Lists Frequency Matrix. Let `frequency_matrix` be a list of lists. For each position `i` we run through the "rows" in the frequency matrix and find and keep track of the maximum frequency value and the corresponding character. If two or more characters have the same frequency value we use a dash to indicate that this position in the consensus string is undetermined.

The following function computes the consensus string:

```
def find_consensus_v1(frequency_matrix):
    base2index = {'A': 0, 'T': 1, 'G': 2, 'C': 3}
    consensus = ''
    dna_length = len(frequency_matrix[0])

    for i in range(dna_length):  # loop over positions in string
        max_freq = -1            # holds the max freq. for this i
        max_freq_base = None     # holds the corresponding base

        for base in 'ATGC':
            if frequency_matrix[base2index[base]][i] > max_freq:
                max_freq = frequency_matrix[base2index[base]][i]
                max_freq_base = base
            elif frequency_matrix[base2index[base]][i] == max_freq:
                max_freq_base = '-' # more than one base as max

        consensus += max_freq_base  # add new base with max freq
    return consensus
```

Since this code requires `frequency_matrix` to be a list of list we should insert a test on this and raise an exception if the type is wrong:

```
def find_consensus_v1(frequency_matrix):
    if isinstance(frequency_matrix, list) and       isinstance(frequency_matrix[0], list):
        pass # right type
    else:
        raise TypeError('frequency_matrix must be list of lists')
...
```

Dict of Dicts Frequency Matrix. How must the `find_consensus_v1` function be altered if `frequency_matrix` is a dict of dicts?

1. The `base2index` dict is no longer needed.
2. Access of sublist, `frequency_matrix[0]`, to test for type and length of the strings, must be replaced by `frequency_matrix['A']`.

The updated function looks like

```
def find_consensus_v3(frequency_matrix):
    if isinstance(frequency_matrix, dict) and \
       isinstance(frequency_matrix['A'], dict):
        pass # right type
    else:
        raise TypeError('frequency_matrix must be dict of dicts')

    consensus = ''
    dna_length = len(frequency_matrix['A'])

    for i in range(dna_length):  # loop over positions in string
        max_freq = -1            # holds the max freq. for this i
        max_freq_base = None     # holds the corresponding base

        for base in 'ATGC':
            if frequency_matrix[base][i] > max_freq:
                max_freq = frequency_matrix[base][i]
                max_freq_base = base
            elif frequency_matrix[base][i] == max_freq:
                max_freq_base = '-' # more than one base as max

        consensus += max_freq_base  # add new base with max freq
    return consensus
```

Here is a test:

```
frequency_matrix = freq_dict_of_dicts_v1(dna_list)
pprint.pprint(frequency_matrix)
print find_consensus_v3(frequency_matrix)
```

with output

```
{'A': {0: 0, 1: 0, 2: 0, 3: 2, 4: 0},
 'C': {0: 0, 1: 0, 2: 0, 3: 0, 4: 2},
 'G': {0: 3, 1: 3, 2: 0, 3: 1, 4: 1},
 'T': {0: 0, 1: 0, 2: 3, 3: 0, 4: 0}}
GGTAC
```

Let us try `find_consensus_v3` with the dict of defaultdicts as input (`freq_dicts_of_dicts_v2`). The code runs fine, but the output string is just `G`! This implies that `dna_length` is 1, and that the length of the `A` dict in `frequency_matrix` is 1. Printing out `frequency_matrix` yields

```
{'A': defaultdict(X, {3: 2}),
 'C': defaultdict(X, {4: 2}),
 'G': defaultdict(X, {0: 3, 1: 3, 3: 1, 4: 1}),
 'T': defaultdict(X, {2: 3})}
```

where our `X` is a short form for text like

```
`<function <lambda> at 0xfaede8>`
```

We see that the length of a defaultdict will only count the nonzero entries. Hence our function must get the length of the DNA string to build as extra argument:

```
def find_consensus_v4(frequency_matrix, dna_length):
    ...
```

(**hpl**: *Make a unified function that can handle lists of lists, dict of lists, dict of dicts, and dicts of defaultdicts? Could be cool.*)

1.4 Probability Matrix

UNFINISHED!

(**hpl**: *I didn't understand this example, i.e., I see that a probability matrix is given for strings of a certain length, then we pick out every substring of this length of DNA, and computes the probability of the sequence of characters in the substring. But what is it good for?*)

```
DNA='ATCTGATCAA'
probabilityMatrix={'A': {0: 0.2, 1: 0.2, 2: 0.6, 3: 0.2, 4: 0.2},
                   'C': {0: 0.2, 1: 0.4, 2: 0.0, 3: 0.0, 4: 0.8},
                   'T': {0: 0.2, 1: 0.0, 2: 0.2, 3: 0.6, 4: 0.0},
                   'G': {0: 0.4, 1: 0.4, 2: 0.2, 3: 0.2, 4: 0.0}}

len_window=len(probabilityMatrix['A'])

probabilitiesList=[]
for num in range(len(DNA)-len_window+1):
    substring=DNA[num:num+len_window]

    prob_value=1
    for index, value in enumerate(substring):
       prob_value *=probabilityMatrix[value][index]

    probabilitiesList.append(prob_value)

print probabilitiesList
```

1.5 Dot Plots from Pair of DNA Sequences

(**hpl**: *terminology: DNA strings or sequences? Maybe sequences is better from a pedagogical point of view since it implicitly says that strings are sequences, and if sequences are interpreted as Python sequences, it helps the programming :-)*)

Dot plots are commonly used to visualize the similarity between two protein or nucleic acid sequences. They compare two sequences, say `d1` and `d2`, by organizing `d1` along the x-axis and `d2` along the y-axis of a plot. When `d1[i] == d2[j]` we mark this by drawing a dot at location `i,j` in the plot. The coordinates along the axes are the integers 0, 1, 2, and so forth.

(**hpl**: *Insert example here! Show plot.*)

(**hpl**: *The "y string" goes downards while the "x string" goes to the right... when we print it, since we print row by row...But we can join the rows in reverse order to what I would expect - both strings starting in the origin and then increasing to the right and upwards.*)

In our forthcoming examples, a dot is represented by 1. No presence at a given location is represented by 0. A dot plot can be manually read to find common patterns between two sequences that has undergone several insertions and deletions, and it serves as a conceptual basis for algorithms that align two sequences in order to find evolutionary origin or shared functional parts. Such alignment of biological sequences is a particular variant of finding the edit distance between strings, which is a general technique, also used for, e.g., spell correction in search engines.

The dot plot data structure must mimic a table. The "x" direction is along rows, while the "y" direction is along columns. First we need to initialize the whole data structure with zeros. Then, for each for each position in the "x string" we run through all positions in the "y string" and mark those where the characters match with 1. The algorithm will be clear when presented with specific Python code.

Using Lists of Lists. Since the plot is essentially a table, a list of lists is therefore a natural data structure. The following function creates the list of lists:

```
def dotplot_list_of_lists(dna_x, dna_y):
    dotplot_matrix = [['0' for x in dna_x] for y in dna_y]
    for x_index, x_value in enumerate(dna_x):
        for y_index, y_value in enumerate(dna_y):
            if x_value == y_value:
                dotplot_matrix[y_index][x_index] = '1'
    return dotplot_matrix
```

To view the dot plot we need to print out the list of lists. Here is a possible way:

```
dna_x = 'TAATGCCTGAAT'
dna_y = 'CTCTATGCC'

M = dotplot_list_of_lists(dna_x, dna_x)
for row in M:
    for column in row:
        print column,
    print
```

The output becomes

```
1 0 0 1 0 0 0 1 0 0 0 1
0 1 1 0 0 0 0 0 0 1 1 0
0 1 1 0 0 0 0 0 0 1 1 0
1 0 0 1 0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 0 1 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0
1 0 0 1 0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 0 1 0 0 0
0 1 1 0 0 0 0 0 0 1 1 0
0 1 1 0 0 0 0 0 0 1 1 0
1 0 0 1 0 0 0 1 0 0 0 1
```

Note that the "y string" `dna_y` has its first index at the top, with indices growing downward, while the "x string" `dna_x` has its indices growing to the right.

One can, alternatively, translate the list of lists to a multi-line string containing the whole plot as a string object. This implies joining all the characters in each row and then joining all the rows:

```
rows = [' '.join(row) for row in dotplot_matrix]
plot = '\n'.join(rows)
# or combined
plot = '\n'.join([' '.join(row) for row in dotplot_matrix])
```

The construction `'d'.join(l)` joints all the string elements of the list `l` and inserts `d` as delimiter: `'x'.join(['A','B','C']` becomes `'AxBxC'`. We use a space as delimiter among the characters in a row since this gives a nice layout when printing the string. All rows are joined with newline as delimiter such that the rows appear on separate lines when printing the string. To really understand what is going on, a more comprehensive code could be made so that each step can be examined:

```
def make_string_expanded(dotplot_matrix):
    rows = []
    for row in dotplot_matrix:
        row_string = ' '.join(row)
        rows.append(row_string)
    plot = '\n'.join(rows)
    return plot

M2 = [['1', '1', '0', '1'],
     ['1', '1', '1', '1'],
     ['0', '0', '1', '0'],
     ['0', '0', '0', '1']]

s = make_string_expanded(M2)
```

Unless the join operation as used here is well understood, it is highly recommended to paste the above ode into the Python Online Tutor, step through the code, and watch how variables change their content. Figure 2 shows a snapshot of this type of code investigation.

Using Numerical Python Arrays. A Numerical Python array, with integer elements that equal 0 or 1, is well suited as data structure to hold a dot plot.

```
def dotplot_numpy(dna_x, dna_y):
    dotplot_matrix = np.zeros((len(dna_y), len(dna_x)), np.int)
    for x_index, x_value in enumerate(dna_x):
        for y_index, y_value in enumerate(dna_y):
            if x_value == y_value:
                dotplot_matrix[y_index,x_index] = 1
    return dotplot_matrix

print dotplot_numpy(dna_x, dna_y)
```

(**hpl**: *we should have a real plot with matplotlib here for a somewhat large string.*)

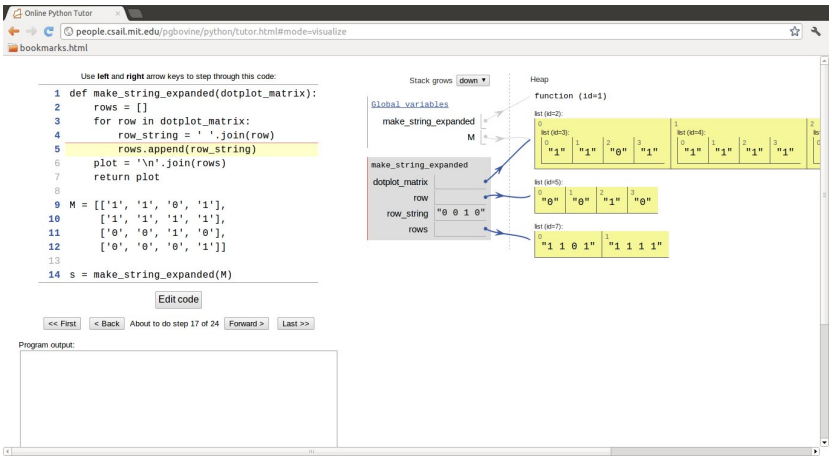

Figure 2: Illustration of how join operations work (Python Online Tutor).

1.6 Finding Base Frequencies

DNA consists of four molecules called nucleotides, or bases, and can be represented as a string of the letters A, C, G, and T. But this does not mean that all four nucleotides need to be similarly frequent. Are some nucleotides more frequent than others, say in yeast, as represented by the first chromosome of yeast? Also, DNA is really not a single thread, but two threads wound together. This wounding is based on an A from one thread binding to a T of the other thread, and C binding to G (that is, A will only bind with T, not with C or G). Could this fact force groups of the four symbol frequencies to be equal? The answer is that the A-T and G-C binding does not in principle force certain frequencies to be equal, but in practice they usually become so because of evolutionary factors related to this pairing.

The first task is to compute the frequencies of the bases A, C, G, and T. That is, the number of times each base occurs in the DNA string, divided by the length of the string. For example, if the DNA string is ACGGAAA, the length is 7, A appears 4 times with frequency 4/7, C appears once with frequency 1/7, G appears twice with frequency 2/7, and T does not appear so the frequency is 0.

From a coding perspective we may create a function for counting how many times A, C, G, and T appears in the string and then another function for computing the frequencies. In both cases we want dictionaries such that we can index with the character and get the count or the frequency out. Counting is done by

```
def get_base_counts(dna):
    counts = {'A': 0, 'T': 0, 'G': 0, 'C': 0}
    for base in dna:
        counts[base] += 1
    return counts
```

This function can then be used to compute the base frequencies:

```
def get_base_frequencies(dna):
    counts = get_base_counts(dna)
    return {base: count*1.0/len(dna)
            for base, count in counts.items()}

frequencies = get_base_frequencies(dna)

def print_frequencies(frequencies):
    return ', '.join(['%s: %.2f' % (base, frequencies[base])
                      for base in frequencies])

print "Base frequencies of sequence '%s':\n%s" % \
      (dna, print_frequencies(frequencies))

# Real data
import urllib, os
yeast_file = 'yeast.txt'
if not os.path.isfile(yeast_file):
    url = \
'http://hplgit.github.com/bioinf-py/doc/src/data/yeast_chr1.txt'
    urllib.urlretrieve(url, filename=yeast_file)

def read_dnafile_v1(filename):
    lines = open(filename, 'r').readlines()
    # Remove newlines in each line and join
    dna = ''.join([line.strip() for line in lines])
    return dna

def read_dnafile_v2(filename):
    dna = ''
    for line in open(filename, 'r'):
        dna += line.strip()
    return dna

dna = read_dnafile_v2(yeast_file)
yeast_freq = get_base_frequencies(dna)
print "Base frequencies of yeast DNA (length %d):\n%s" % \
      (len(dna), print_frequencies(yeast_freq))

assert get_base_frequencies(read_dnafile_v1(yeast_file)) == \
       get_base_frequencies(read_dnafile_v2(yeast_file))
```

A little test,

```
dna = 'ACCAGAGT'
frequencies = get_base_frequencies(dna)

def print_frequencies(frequencies):
    return ', '.join(['%s: %.2f' % (base, frequencies[base])
                      for base in frequencies])

print "Base frequencies of sequence '%s':\n%s" % \
      (dna, print_frequencies(frequencies))
```

gives the result

```
Base frequencies of sequence 'ACCAGAGT':
A: 0.38, C: 0.25, T: 0.12, G: 0.25
```

The `print_frequencies` function was made for nice printout of the frequencies with 2 decimals. The one-line code is an effective combination of a dictionary, list comprehension, and the `join` functionality. The latter is used to get a comma correctly inserted between the items in the result. Lazy programmers would probably just do a `print frequencies` and live with the curly braces in the output and (in general) 16 decimals.

We can try the frequency computation on real data. The file

`http://hplgit.github.com/bioinf-py/doc/src/data/yeast_chr1.txt`

contains the DNA for yeast. We can download this file from the Internet by

```
urllib.urlretrieve(url, filename=name_of_local_file)
```

where `url` is the Internet address of the file and `name_of_local_file` is a string containing the name of the file on the computer where the file is downloaded. To avoid repeated downloads when the program is run multiple times, we insert a test on whether the local file exists or not. The call `os.path.isfile(f)` returns `True` if a file with name `f` exists in the current working folder.

The appropriate download code then becomes

```
import urllib, os
yeast_file = 'yeast.txt'
if not os.path.isfile(yeast_file):
    url = \
'http://hplgit.github.com/bioinf-py/doc/src/data/yeast_chr1.txt'
    urllib.urlretrieve(url, filename=yeast_file)

def read_dnafile_v1(filename):
    lines = open(filename, 'r').readlines()
    # Remove newlines in each line and join
    dna = ''.join([line.strip() for line in lines])
    return dna

def read_dnafile_v2(filename):
    dna = ''
    for line in open(filename, 'r'):
        dna += line.strip()
    return dna

dna = read_dnafile_v2(yeast_file)
yeast_freq = get_base_frequencies(dna)
print "Base frequencies of yeast DNA (length %d):\n%s" % \
      (len(dna), print_frequencies(yeast_freq))

assert get_base_frequencies(read_dnafile_v1(yeast_file)) == \
       get_base_frequencies(read_dnafile_v2(yeast_file))
```

A copy of the file on the Internet is now in the current working folder under the name `yeast.txt`. None

The `yeast.txt` files contains the DNA string split over many lines. We therefore need to read the lines in this file, strip each line to remove the trailing newline, and join all the stripped lines to recover the DNA string:

```
def read_dnafile_v1(filename):
    lines = open(filename, 'r').readlines()
    # Remove newlines in each line and join
    dna = ''.join([line.strip() for line in lines])
    return dna
```

As usual, an alternative programming solution can be devised as well:

```
def read_dnafile_v2(filename):
    dna = ''
    for line in open(filename, 'r'):
        dna += line.strip()
    return dna

dna = read_dnafile_v2(yeast_file)
yeast_freq = get_base_frequencies(dna)
print "Base frequencies of yeast DNA (length %d):\n%s" % \
      (len(dna), print_frequencies(yeast_freq))
```

The output becomes

```
Base frequencies of yeast DNA (length 230208):
A: 0.30, C: 0.19, T: 0.30, G: 0.20
```

This shows that A and T appears in the yeast DNA with 50 percent higher probability than C and G. (**hpl**: *GK, can we say this, or more precisely, does the observation have any interpretation of significance?*)

1.7 Translating Genes into Proteins

An important usage of DNA is for cells to store information on their arsenal of proteins. These proteins are what makes up the possible functional properties of a cell. Proteins are made based on the recipe found in genes. Genes are, in essence, only regions of the DNA. These are divided into exons, which are the coding regions of the gene, and introns, the regions in between. In order for a protein to be created, the exon regions are copied out of the DNA, joined together and then transcribed into mRNA. mRNA is messenger RNA, which is a small DNA-like sequence that is sent to the protein-creating machinery in the ribosomes, containing the recipe for a protein. One difference between the mRNA and the original DNA is that all T-bases (Thymine) are exchanged with U-bases (Uridine). In the ribosome, the mRNA is translated into proteins. Here, a genetic code is used to translate triplet of bases, or codons, into an amino acid. A protein is made up of a series of amino acids. Interestingly, the genetic code, which is the same for most forms of life, contains redundancy, i.e., that several codons code for the same aminoacids, as the 64 possible codons are used to code for only 20 amino acids.

Here is an example of using the genetic code to create the amino acid sequence of the Lactase protein (LPH), using the DNA sequence of the Lactase gene (LCT) as template. An important functional property of LPH is as a restriction enzyme to cleave the disaccaaride Lactose into two monsaccarides. Lactose is most notably found in milk. Organisms lacking the functionality of

LPH will get digestive problems including elevated osmotic pressure in the intestine leading to diarea and referred to as lactose intolerance. Most mammals and humans lose their expression of LCT and therefore their ability to digest milk when they stop recieving breast milk.

The file

```
http://hplgit.github.com/bioinf-py/doc/src/data/genetic_code.tsv
```

contains information on genetic codes. The file format takes the form

```
UUU     F       Phe     Phenylalanine
UUC     F       Phe     Phenylalanine
UUA     L       Leu     Leucine
UUG     L       Leu     Leucine
CUU     L       Leu     Leucine
CUC     L       Leu     Leucine
CUA     L       Leu     Leucine
CUG     L       Leu     Leucine
AUU     I       Ile     Isoleucine
AUC     I       Ile     Isoleucine
AUA     I       Ile     Isoleucine
AUG     M       Met     Methionine (Start)
```

The four columns describe .... (**hpl**: *Need to write a bit what the significance of the info is*)

[[[

```
0       651
3990    4070
7504    7588
13177   13280
15082   15161
```

2 Exercises

2.1 Find pairs of characters

Write a function `count_pairs(dna, pair)` that returns the number of occurences of a pair of characters (`pair`) in a DNA string (`dna`). For example, `count_pairs('ACTGCTATCCATT', 'AT')` should return 2.
Filename: `count_pairs.py`

2.2 Count substrings

This is an extension of Exercise 2.2: count how many times a certain string appears in another string. For example, `count_substr('ACGTTACGGAACG', 'ACG')` should return 2.

Hint 1. For each match of the first character of the substring in the main string, check if the next `n` characters in the main string matches the substring, where `n` is the length of the substring. Use slices like `s[3:9]` to pick out a substring of `s`.
Filename: `count_substr.py`

2.3 Make a function more robust

Consider the function `get_base_counts(dna)` which counts how many times `A`, `C`, `G`, and `T` appears in the string `dna`:

```
def get_base_counts(dna):
    counts = {'A': 0, 'T': 0, 'G': 0, 'C': 0}
    for base in dna:
        counts[base] += 1
    return counts
```

Unfortunately, this function crashes if other symbols appear in `dna`. Write an enhanced function `get_base_counts2` which solves this problem.
Filename: `get_base_counts2.py`

3 2DO

1. gh-pages index.html
2. make.sh for sphinx, html, pdflatex
3. more ex

Index