AP2014 exam: GC frequency in human genes

How to solve this assignment

In each problem you are asked to write a function. The final file with code that you hand in should only contain function definitions. No code testing the functions should be included in the final file. Comments are not considered in the evaluation.

At the end of the exam you will be allowed to go online and upload your solution to the folder “Afleveringer” on the course page on AULA (http://aula.au.dk/courses/AP2014/). If you are done before the end of the exam you can raise your hand and you will be allowed to upload your solution under supervision.

The file you upload must be named StudentID_FirstName_LastName.py (with underscores in between).

Happy coding.

DISCLAIMER:

THIS EXAM ASSIGNMENT IS AN OLD EXAM ASSIGNMENT AND IS ONLY SUPPLIED TO GIVE YOU SOME ROUGH IDEA ABOUT WHAT AN EXAM SET COULD LOOK LIKE.

THE ACTUAL EXAM WILL USE DIGITAL EXAMEN AND NOT BLACKBOARD.

THIS EXAM DID NOT USE A TESTING FRAMEWORK. THE ACTUAL EXAM DOES.

THE PROGRAMMING PROJECTS YOU COMPLETED DURING THE COURSE MUCH MORE ACCURATELY REFLECT HOW PROGRAMMING ASSIGNMENTS IN THE ACTUAL EXAM ARE FORMULATED.

The assignment

A frequent use of programming is parsing and mining of large data files. In this exam assignment you will work on a file with annotation for genes in the human genome. The name of this file is `Thomas.txt` and is found in the folder "AP2014exam" that you downloaded before the exam. If you do not have this file please raise your hand to get help. To make sure that the Python code you produce for this assignment can find and read the input file you must put the file for your Python code in the same directory as the input file. So start by creating a new file to put Python code in, and then save it to the "AP2014exam" folder.

These are the first 10 lines from input file (`"Thomas.txt"`). Do **NOT** open the input file as you could accidentally change it.

Use EksamensOpgaveAP2014_input.txt instead of Thomas.txt

```
SAMD11 1 860260 879955 12996 nucleic
PLEKHN1 1 901877 911245 5869 signaling
C1orf170 1 910579 917497 4685 transferase
HES4 1 934342 935552 899 transcription
ISG15 1 948803 949920 721 nucleic
AGRN 1 955503 991496 23651 receptor
TTLL10 1 1109264 1133315 13167 cytoskeletal
TNFRSF18 1 1138888 1142071 2147 receptor
TNFRSF4 1 1146706 1149518 1979 receptor
B3GALT6 1 1167629 1170421 1792 transferase
```

Each line has the following six fields separated by a space character: gene name, chromosome name, gene start coordinate, gene end coordinate, number of GC bases in the gene, and the protein category that the gene belongs too. For short we will name the columns: `name, chrom, start, end, gcbases, category.`

This assignment is an analysis of the GC frequency of human genes. I.e. what proportion of bases in the gene sequences that are G or C. For each gene, the number of G or C bases is given in column five and the total number of bases (the length of the gene) is found by subtracting the start coordinate from the end coordinate.

Here is a function that you can use to read the input file:

```
def read_data(file_name):
    fh = open(file_name, 'r')
    data = list()
    for line in fh:
        lst = line.split()
        data.append(lst)
    return data
```

You use it like this: `data = read_data('Thomas.txt')`. Start by writing this function as the first in your assignment. `read_data` returns a list of lists where each sublist contains the fields of one line in the input file.

All the remaining functions you are asked to make in this assignment take as argument the list of lists produced by `read_data`. Be sure to use the functions you make along the way inside other functions where this makes sense.

Problem 1:
First you must make a data structure that lets you access the information for each gene using the name of the gene.

Write a function

```
def create_data_struture(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a dictionary of dictionaries where keys for the outer dictionary are gene names and keys for each inner dictionary are `chrom`, `start`, `end`, `gcbases`, and `category`. The values of each inner dictionary must be the corresponding values from the input file. Values for `start`, `end`, and `gcbases`, must be integers.

Example usage:
```
ds = create_data_struture(data)
print(ds['SAMD11'])
```

should print (possibly not in that order):

```
{'category': 'nucleic', 'end': 879955, 'name': 'SAMD11', 'gcbases':
12996, 'start': 860260, 'chrom': '1'}
```

Problem 2:
Now you must compute the frequency of GC bases in each gene. This can be computed by dividing the number of GC bases by the length of the gene (end - start).

Write a function

```
def compute_gc_frequencies(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a dictionary that has gene names as keys and GC frequencies (between 0 and 1) as values.

Example usage:
```
gcdict = compute_gc_frequencies(data)
print(gcdict['SAMD11'])
```

should print
```
0.659862909368
```

Problem 3:
Now you must find the gene with the highest GC frequency.

Write a function:

```
def find_gene_with_largest_gc_frequency(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a string with the name of the gene with the highest GC frequency.

Example usage:
```
print(find_gene_with_largest_gc_frequency(data))
```

should print
```
BHLHA9
```

Problem 4:
Now you must compute average GC frequency across all genes

Write a function:

```
def compute_avg_gc_frequency(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a float representing the average GC frequency across all genes.

Example usage:
```
print(compute_avg_gc_frequency(data))
```

should print
```
0.468535905483
```

Problem 5:
Now compute the number of genes that have a GC frequency in each of the following bins: 0 - 0.2, 0.2 - 0.4, 0.4 - 0.6, 0.6 - 0.8, 0.8 - 1. The genes assigned to the first bin should be those with a GC frequency from zero up to, but not including, 0.2. The genes assigned to the second bin should be those with a GC frequency from 0.2 up to, but not including, 0.4. Assignment to the remaining bins are made the same way, with the exception that genes with a GC frequency of 1 should be assigned to the last bin.

Write a function:

```
def count_genes_in_gc_frequency_range(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a list of tuples where the first element in each tuple is the start of a bin and the second is the number of genes in the corresponding bin.

Example usage:
```
print(count_genes_in_gc_frequency_range(data))
```

should print
```
[(0.0, 0), (0.2, 2887), (0.4, 8243), (0.6, 791), (0.8, 0)]
```

Problem 6:
Now make a histogram of GC frequency among genes with the five bins used in `count_genes_in_gc_frequency_range`.

Write a function:

```
def print_histogram_of_cg_frequency(data):
```

that takes as argument the list of lists returned by `read_data`. The function must print (not return) a histogram *exactly* as the one shown below where the number of '#' represents the number of times a gene count can be divided by 200 (`count // 200`).

Example usage:
```
print_histogram_of_cg_frequency(data)
```

should print
```
0.0 - 0.2: 
0.2 - 0.4: ##############
0.4 - 0.6: #########################################
0.6 - 0.8: ###
0.8 - 1.0: 
```

Problem 7:
Now find the all the different protein categories. The protein category is given column six in the input files.

Write a function:

```
def find_all_categories(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a a list of all the different protein categories. Each name must occur only once in the list.

Example usage:
```
print(find_all_categories(data))
```

should print (not necessarily in that order)
```
['calcium-binding', 'storage', 'transfer/carrier', 'cytoskeletal',
'receptor', 'viral', 'transferase', 'transcription', 'transmembrane',
'cell', 'defense/immunity', 'enzyme', 'lyase', 'phosphatase',
'membrane', 'extracellular', 'hydrolase', 'surfactant', 'isomerase',
'signaling', 'transporter', 'structural', 'nucleic', 'chaperone',
'ligase', 'oxidoreductase']
```

Problem 8:
Now count the number of genes in each protein category. The protein category is given column six in the input files.

Write a function:

```
def count_genes_in_each_category(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a dictionary where keys are protein categories and values are integers representing the number of genes in that category.

Example usage:
```
print(count_genes_in_each_category(data))
```

should print (not necessarily in that order)
```
{'calcium-binding': 89, 'storage': 22, 'transfer/carrier': 337,
'cytoskeletal': 692, 'receptor': 1049, 'viral': 4, 'transferase': 1044,
'transcription': 1470, 'transmembrane': 55, 'cell': 291, 'defense/
immunity': 62, 'enzyme': 859, 'lyase': 85, 'phosphatase': 176,
'membrane': 269, 'extracellular': 171, 'hydrolase': 631, 'surfactant':
8, 'isomerase': 145, 'signaling': 967, 'transporter': 644, 'structural':
87, 'nucleic': 1845, 'chaperone': 174, 'ligase': 331, 'oxidoreductase':
414}
```

Problem 9:
Now compute the average GC frequency for each protein category.

Write a function:

```
def compute_avg_gc_frequency_by_category(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a dictionary where keys are categories and values are GC frequencies.

Example usage:
```
print(compute_avg_gc_frequency_by_category(data))
```

should print (not necessarily in that order)
```
{'calcium-binding': 0.4838503503262077, 'transmembrane':
0.47197371580746317, 'transfer/carrier': 0.46036917962180823,
'cytoskeletal': 0.4763794910619796, 'receptor': 0.47412210835678475,
'viral': 0.4519219268240308, 'defense/immunity': 0.485254596741273,
'transcription': 0.47494649425757857, 'storage': 0.4431222636965003,
'cell': 0.4655849748782357, 'transferase': 0.4667055820636091, 'enzyme':
0.46257085424548494, 'lyase': 0.4681742585413936, 'phosphatase':
0.4585012390909853, 'membrane': 0.46715818380436647, 'extracellular':
0.4866632619407623, 'hydrolase': 0.470646526460335, 'surfactant':
0.4482903807562709, 'isomerase': 0.47476827410403527, 'signaling':
0.4726463636968355, 'transporter': 0.46822697610221886, 'structural':
0.4837599466884884, 'nucleic': 0.4642468127890071, 'chaperone':
0.4550591443641406, 'ligase': 0.4434753161655074, 'oxidoreductase':
0.4647160873497887}
```

Problem 10:
Now print nicely the results obtained by `compute_avg_gc_frequency_by_category`.

Write a function:

```
def print_avg_gc_frequency_by_category(data):
```

that takes as argument the list of lists returned by `read_data`. The function must print (not return) the results from `compute_avg_gc_frequency_by_category` as shown below (although not necessarily in that order):

Example usage:
```
print_avg_gc_frequency_by_category(data)
```

should print
```
calcium-binding: 0.483850350326
transmembrane: 0.471973715807
transfer/carrier: 0.460369179622
cytoskeletal: 0.476379491062
receptor: 0.474122108357
viral: 0.451921926824
defense/immunity: 0.485254596741
transcription: 0.474946494258
storage: 0.443122263697
cell: 0.465584974878
transferase: 0.466705582064
enzyme: 0.462570854245
lyase: 0.468174258541
phosphatase: 0.458501239091
membrane: 0.467158183804
extracellular: 0.486663261941
hydrolase: 0.47064652646
surfactant: 0.448290380756
isomerase: 0.474768274104
signaling: 0.472646363697
transporter: 0.468226976102
structural: 0.483759946688
nucleic: 0.464246812789
chaperone: 0.455059144364
ligase: 0.443475316166
oxidoreductase: 0.46471608735
```

Problem 11:
Now you must create a data structure that holds the average GC frequency for each category on each chromosome.

Write a function:

```
def compute_avg_gc_frequency_by_chrom_and_category(data):
```

that takes as argument the list of lists returned by `read_data`. The function must return a dictionary of dictionaries where keys for the outer dictionary are chromosome names and keys for each inner dictionary are names of protein categories. All combinations of chromosomes and categories must be represented in the data structure. In the cases where a category is not represented on a chromosome (so that a GC frequency can not be calculated) the value must be `None`.

Example usage:
```
freqs = compute_avg_gc_frequency_by_chrom_and_category(data)
print freqs['1']['nucleic']
print freqs['20']['calcium-binding']
```

should print
```
0.461360894057
None
```