Introduction to Python
Exercises

Version 2022-12

Licence

Exercise 1: Variables, Operators, Functions, Methods

Mathematical Operations

Molarity Calculator

- Write a script to calculate how much of a compound is needed to make a solution of a given molarity. You will need to create variables to store:
  - The molecular mass of the compound (in g/mol)
  - The volume of solution you want to create (in ml)
  - The desired concentration (molar M)

- The formula will be:
  - Mass (g) = Concentration (mol/L) * Volume (L) * Formula Weight (g/mol)

- Make the script `print` a summary of the input variables and the calculated value by passing all of these as separate arguments to your `print` function.

Interactive Molarity Calculator

- Make a modified version of your initial script in which the different values you need for the calculation are requested interactively using `input` statements.

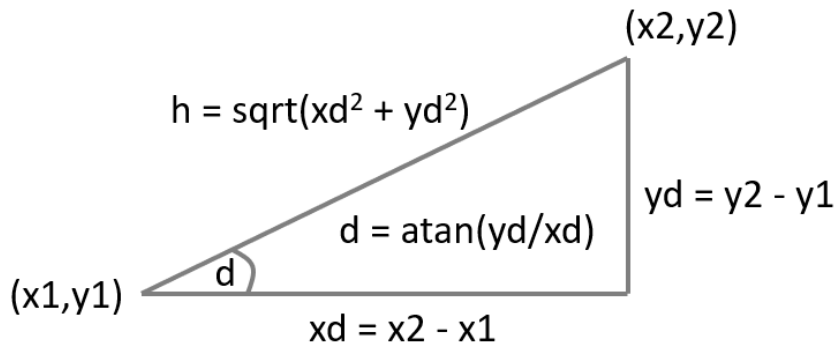
- Remember that the return value from input will be a string (`str`), which you can't use in mathematical calculations. You will need to use `float()` to convert it to a numerical value.

- So that the calculated value doesn't get too specific use the `round` function to limit the precision to 2 decimal places when printing the result.

2D Geometry Calculator

- Make a script which defines two 2D positions (`x1, y1, x2, y2`) and then calculates the distance between them (Pythagorean distance). Just make up some coordinates and put them in your script.

- Also calculate the angle (in degrees) from the first point to the second



- You will need to use the `sqrt` and `atan` functions from the `math` package for this, the angle will be in radians so you can use the `degrees` function from math to convert this. https://docs.python.org/3/library/math.html

String Manipulation

Organism names

- Write a script to correctly format organism scientific names (eg *Homo sapiens*).
    - It should interactively ask for the Genus (eg homo) and Species (eg sapiens) using `input()` statements
    - It should then `print` out the name with the Genus with an uppercase initial letter and the species all in lowercase.
    - Look at the `capitalize` and `lower` methods in the `str` class to do these transformations. https://docs.python.org/3/library/stdtypes.html#textseq

Text alignment

- Write a script which uses 5 `input` statements to ask the user for 5 names and stores them in 5 separate variables
- After collecting the names `print` them out, one per line, using the `center` method to center them within a 40 character block
    - Make sure that the names are all in uppercase (`upper` method), and make sure that any spaces have been removed from the ends of the names (`strip` method)

- At the bottom of the list add the total length of all of the names which were entered.
    - You will need to use the `len` function to get the lengths of the names which you can then add together.

Tricky challenge (if you have time / motivation!)

- Make a modified version of your 2D distance calculation script, except that this time the user supplies the following information
    - First x position
    - First y position
    - Distance to the second point

- Your script should then calculate an x/y position which would fall that distance from the first point. Since there are a large number of points which would fit these criteria you will need to use a method from the `random` package to select a suitable random point.

- Your script should print out the final result and explain its working.

Exercise 2: Data Structures

Lists and Dictionaries

Simple Data Collection

Write a script which will prompt the user to enter 5 numeric values using 5 `input` statements. Convert these to numbers using `float()` and put them into a list. Use the `sort` method to order the list then `print` it.

Use a list selection using `[ ]` to pull out and `print` just the first (lowest) value in the list.

Next `reverse` the sorted list and `print` it again so you can see the numbers from highest to lowest. Have a look at the documentation for the `sort` method to see how you could have done the sorting this way initially.

Finally `print` out a statement which uses the `count` method to say how many times the number 2 was present in the data.

Animal Sets

Write a script which asks the user to `input` the names of 5 animals. Put each of these into a `set`. Then ask them for a final `input` and `print` out a message saying whether the animal they entered last has been seen before by using the `in` test. Make the query case insensitive by converting everything to `lower` case.

For the output you can just print `True` or `False` – we'll be able to do cleverer things with the data after next week's session.

Random Data

Make an empty list and then populate it with 10 numbers drawn from a normal distribution with a mean (mu) of 10 and a standard deviation (sigma) of 3

Calculate the mean, standard deviation and standard error of the mean of the simulated data.

You can `import random` then use the `random.normalvariate` function to get the random values.

You can `import statistics` then use `statistics.mean` and `statistics.stdev` to calculate those values. For the SEM you need to divide the SD by the square root (`math.sqrt`) of the number of data points (`len` of the list) minus one.

Random Genomic Position

Write a script which will generate a random genomic position of the form: `chr3:+:10200353`, ie `chr:strand:position`

Create a list of chromosome names and use the `random.choice` function from the random package to select a random item from the list.

Create a dictionary which has chromosome names as keys and their lengths as the values. Use this to get the length of the randomly selected chromosome, then use the `random.randint` function from the `random` package to select a random integer smaller than that value.

Finally create a tuple of + and - and use `random.choice` to randomly select one of those two values.

Print all of the randomly selected values you created.

Multi Level Data

Amino Acid Properties

Make up a data structure of the properties of amino acids. The top level should be a dictionary where the keys are the one letter amino acid codes, and the values are another dictionary with keys of "long name" and "molecular weight" containing the appropriate values. Details of the values can be found here.

Allow the user to enter a 1 letter amino acid code and have the script print the long name and weight for that amino acid.

Gene Model

Put the information below into a suitable multi-level data structure. It probably makes sense to build the top level gene information in one go, and then add the individual transcripts afterwards.

Gene				
Name	Nanog			
Description	Nanog homeobox			
Location	**Chromsome**	12		
	Start	7,787,794		
	End	7,799,146		
	Strand	Forward		
Transcripts	**Name**	**ID**	**Length**	**Amino Acids**
	NANOG-201	ENST00000229307.9	5182	305
	NANOG-202	ENST00000526286.1	870	289
	NANOG-204	ENST00000541267.5	836	186
	NANOG-203	ENST00000526434.2	558	0

The top level structure will be a dictionary. The Location value will be a second dictionary containing the different pieces of location information.

The Transcripts value will be a list, where each value in the list will be a dictionary containing the different transcript details.

Use this data structure to pull out the length of the last transcript.

Reconstruct a location string by querying the structure.

Tricky challenge (if you have time / motivation!)

Create a program to simulate a random sequence based on the composition of a reference sequence. The inputs to the program should be

- A reference DNA sequence
- The length of a random sequence to simulate

For reasons we'll come on to you can treat a string such as `"GATATCG"` as if it was a list for many operations.

Make your script count and report the number of occurrences of each of the four letters. Print out a count for the number of letters which aren't accounted for by the 4 expected bases.

Use the `random.choices` function to make a selection of the requested size from a population which reflects the composition of the original sequence. Since your original sequence may have non-GATC letters in it, construct a list of weights to use for `choices`.

Your script should cope with the input sequence being uppercase, lowercase or a mix of the two. It should force the length to be an integer.

Exercise 3: Iteration and Conditions

Iteration and Looping

Gene Set Intersection

Make two lists (or tuples if you prefer) of gene names containing:

List 1: `"Npepl1", "Rab13", "Reg4", "Asb17", "Clcnka", "Nup62", "Upf3a", "Kcnn1", "Ccdc151", "Arg1", "Tmem98", "Mtx3", "Isl1", "Fam53c"`

List 2: `"Kcnj2", "Rab13", "Reg4","Nol6", "Masp2", "Clcnka", "Upf3a", "Kcnn1", "Arg1", "Krt75", "Smpd3", "Mtx3", "Trim8", "Fam53c"`

Write a script which will build a new list containing the genes in List 1 which are also present in List 2. Create an empty list which will store the genes you select. Write a `for` loop to go through the genes in list1. For each one use an `in` test to see `if` it's in list2 and `append` it to the new list if it is.

When writing out the results `sort` the intersection genes alphabetically and number them (using `enumerate`) as you print them out, ie:

1. AAC
2. BCL

etc.

Angle Ranges

For the set of integer angles from 0-1800 degrees find which ones have a `math.sin` value of zero. Use a `range` statement to create a `for` loop through the angles.

Because you're going to be calculating small fractional values then an `==` equality test is not reliable, so test for the absolute (`abs` function) sin value being less than 0.01.

Variant Counting

You are running an experiment looking for a series of protein variants, specifically

`E23A, P12S, W88Y and R32N`

Write a script which repeatedly prompts you to enter a variant name and keep count of the number of times the ones in the list come up. If you get a variant which isn't in the list then print out a warning and move on.

If they enter a blank variant then stop asking for more input and print out the counts of the variants you saw.

To do this:

- Create a dictionary with the variant names as keys and an integer (initially 0) as values
- Start an infinite loop with `while True:`

- Ask for the name of the variant in an `input` statement.
- If nothing was entered then `break` to exit the loop
- Look up the variant in the dictionary and increase the value by using += 1
- Print out the dictionary after you exit the loop.

Molecular Weights

Use the amino acid properties data structure you made in Exercise 2 and calculate the total mass of the antigenic peptide:

```
'T','E','N','K','Y','S','Q','L','D','E','E','Q','P'
```

Log transformation

Create a random 2D dataset of 10 rows and 10 columns using nested lists. Populate it with random values between 1 and 1,000,000.

Start with an empty list then use two nested `for` loops using `range(10)` to `append` a new list and `append` `math.randint()` values into it.

Log transform the data by iterating over the original data structure and building a new one. Again this will be two `for` loops, one to go through the 10 lists, and the second to go through the 10 values in each list. Reduce the precision of the log transformed values to 1 decimal place using `round`.

Print out the transformed data.

All hexamers

Write a program to collect and print out all 6-mer combinations of the DNA bases `GATC`.

There are potentially multiple ways to do this. One would be to make a list of the bases then over 6 rounds add each base to each position and replace the original list. To append a letter to a string you use the + operator, so `"GA" + "TC" = "GATC"`

Tricky challenge (if you have time / motivation!)

Create a data structure to hold the following genomic positions taken from the human GRCh38 genome.

```
Chr1:90435481-90535480
Chr4:121080701-121180700
Chr5:58203396-58303395
Chr6:24011285-24111284
Chr7:27397324-27497323
Chr9:63677076-63777075
Chr12:57831538-57931537
Chr13:80438618-80538617
Chr16:86177236-86277235
Chr18:39459388-39559387
```

Use the code you wrote in last week's exercise to generate random genomic positions, only this time use a loop to generate positions and check them to see if they fall into any of the list of regions above. Count how many times you hit each region. Stop when any of the regions has 10 hits in it.

Finish by printing the number of hits you got to each region and the total number of random positions you had to generate to achieve this.

Exercise 4: String Manipulation

String manipulation

Mappability Presentation

Below is some data from calculating the mappability of some regions of the genome for sequencing reads of different sizes.

From the data calculate the percentage mappability (mapped fragments as a percentage of total fragments), and then write out the results as nicely formatted text using f-strings.

- The percentage values should be shown to one decimal place
- The different columns should line up (be the same width in all lines)

Read Length	Fragments	Mappable
30	1725	697
40	1713	794
50	1689	912
70	1677	1103
85	1660	1187

To load this data initially copy it into a triple quoted string in your script so you can have multiple lines. Use the `split` method to split on newlines (`\n`) so you can get a list of lines. Use a `for` loop to iterate through the lines so you can `split` out the different values and put them into a suitable data structure.

When printing you can use f-strings to format the lines as you print them.

Date Interpretation

You have been given a set of dates which are in inconsistent formats

```
2018-04-16 10:25:29 AM MDT
2018-01-05 12:05:00 PM CST
Tue Jan 02 2018 08:12:10 GMT-0600 (CST)
2018/01/01 4:50 PM HST
01/01/2018
12/02/2018
```

Write a script which can parse these and produce a tuple of (year,month,day) as integer values. There are ways to achieve this with both standard string parsing and regular expressions. One process would be:

- Check the first letter of the date. If it's numeric parse it as a delimited number, otherwise parse it as text

- For the delimited numbers, extract everything up to the first space. Check for the presence of minus or slash to confirm the delimiter and split the date into sections. Check the length of the first section to see if you've got day-month-year or year-month-day

- For the text dates split into sections based on spaces and then extract the day month and year from those. Use a dictionary to map text months to numbers

When printing out results use a format string. To get consistent formatting of days and months you can use the `str.zfill` method to add leading zeroes to a value.

Regular Expressions

Cleaning Sample Names

You have been given some files with embedded sample names

```
lane1_NoCode_L001_R1.fastq.gz
lane1_NoIndex_L001_R1.fastq.gz
lane1_NoIndex_L001_R2.fastq.gz
pipeline_processing_output.log
lane7127_ACTGAT_JH25_L001_R1.fastq.gz
lane7127_ACTTGA_E30_1_2_Hap4_24h_L001_R1.fastq.gz
lane7127_AGTTCC_JH14_L001_R1.fastq.gz
lane7127_CGGAAT_JH37_L001_R1.fastq.gz
lane7127_GCCAAT_E30_1_2l_Hap4_log_L001_R1.fastq.gz
lane7127_GGCTAC_E30_1_4_Hap4_48h_L001_R1.fastq.gz
```

We want to extract the sample name from these files. The structure is:

1. Written lane number
2. Barcode
3. Sample name
4. Numeric lane number (starting with L)
5. Read number (R1/2/3/4)
6. File suffix (always .fastq.gz)

So for `lane7127_GCCAAT_E30_1_2l_Hap4_log_L001_R1.fastq.gz` the sample name would be `E30_1_2l_Hap4_log`

Extract the sample names from the file names, ignoring any files which do not match the expected pattern.

You will need to use a combination of basic string processing to do some simple checks on the files, with more complexes regular expression substitutions to remove the unwanted parts of the filename from the start and end of the string to leave just the sample name in the middle.

Tricky challenge (if you have time / motivation!)

Transcription Factor Binding Sites

Write a program to search a DNA Sequence for the presence of one of the following transcription factor binding sites, which utilise ambiguity codes. In each case write a regular expression to represent the binding site and then search for all of the positions of that site in the sequence being searched.

Code	Represents
A	Adenine
G	Guanine
C	Cytosine
T	Thymine
Y	Pyrimidine (C or T)
R	Purine (A or G)
W	weak (A or T)
S	strong (G or C)
K	keto (T or G)
M	amino (C or A)
D	A, G, T (not C)
V	A, C, G (not T)
H	A, C, T (not G)
B	C, G, T (not A)

Transcription Factor	Consensus Sequence
RUNX1	BHTGTGGTYW
TGIF1	WGACAGB
IKZF1	BTGGGARD

The sequence to search against is shown below. Enter this directly as shown and extract a single string of just the DNA bases from the structure you were initially given (ie remove the header and the line breaks).

```
>search_seq
GACACCTCAGTACTAGGATGNNNNNNTATCAGCCTGAACTAGCAGGCCTGGTTCCAAATT
TTTTTATCAACACTCGTAGGGGGATTATCCTAGAGGGGGTCTGGGATTTCTTTGACATCA
GAGTATTTTTGCCTTGCTCCTTCACAATTTGGGAACAAATAATTTAGTGGTTATTAACCC
TGGCTACGCACTGGAAACTTTAAAAATAATGCTGGTATGAAATTTACACAGAGTATCGTG
AAAATTTTCACTGAGTACCATGTGGTTATACATTGGATAAGGCTCCAGGAAGCAGCTACT
GGAAGACAGCCATGCCAAGAGTGGTTAGTGGTTGGAATTTTGGCAAGTCAGTTTTAGTCT
GCCTTATCAAATACATGGGCATACAGATAAATCCTTAGATGGCTCTCCTACTTACTGAAA
CATTTTCTATCTATCTATCTATCTATCTATCTATTTGGGAAGCTATCTATCTATCTATCA
TTTATTTAAGGTAGTCTCTATCTGCCTCTGTCTCTGTCTGTCTCTGTGTCTCTGTGTCTG
TCTGCTCTCTCTCTCTCTCTGTGGGAATCTCTCTCTGTGTGTGTGTGTGTATGTGTGTGT
GTGTGTGTGTGGTGTGCATGAACATGAGTAAAATCCATAAGGAAACTTTCAGAGTTGGTC
CTCTCCTTATATCAAATGGATCCAGGAATTAAACTCAGGTTCAATTCTTGGTGCCTTTAC
TAGTTGAGCCATCTCACTGGCTCTTCATCATCTTTAGAATAAACTCACTTTATTACACAC
ACACACACACACACAACCTGGGAGTACACACACACACACAACCAAAGCCCCAACGGAAAA
CTACAATATTATAATGAATACACAGGTTCTCAACATAGTCTCTGCCACGCTTGCAGACAA
AGATGAGTAGAAGTAGAAAGAACCAGGGAAACGTGGAGCAAGTCAGAAGGAATAACAGTC
AGAAGGAATAACAGTCAGAAGGAATAACAGTCAGAAGGAGTAACAGTCAGAAGGAATAGC
AGTCAGAAGGAATAACAGTCAGAAGACAGCACAGTCAGAAGGAATAACAGTCAGAAGGAA
TAACAGTCAGAAGGAATAACAGTCAGAAGGAATAACAGTCAGAAGGAATAGCAGTCAGAA
GGAATAACAGTCAGAAGGAATAACAGTCAGAAGGAATAACAGTCAAAGAAATAGCAGTCA
GAAGGAATAGCAGTCAGAAGGAATAACAGTCAAAGGAGCAGTCAGAAGGAGTAACAGTCA
GAAGGAATAACAGTCAGAAGGAATAACAGTCAAAGGAATAGCAGTCAGAAGGAGTAACAG
TCAGAGCAAACACAGAGATGACAAAGGCAATGGGGTCAGAGACTTCACCACTCTCCAAGA
```

You will need to use the `re.finditer` function to search for multiple hits and return an iterator of `re.match` objects, on which you can use the `span()` function to get a tuple of the start and end points for each match.

Exercise 5: Reading and Writing Files

For these exercises you will need to download and uncompress the zip file of data files we have provided.

Filtering Data Files

Simple Filtering

Read the data in `the cancer_stats.csv` file. This is a datasets of case and fatality numbers for different types of cancer, split between males and females.

We want to extract from this file the subset where the survival rates for females `((cases-deaths)/cases)` was higher than for males. We will exclude any cancer types which only one sex can get (values are NA for the other sex).

This is a csv file so you can read the file and just split on comma. For a more complete solution you could switch to using the `csv` package to do the reading.

Initially you can just print the results to the screen, but you could also try saving the results to an output file, preserving the headers from the original data.

Matched Filtering

You've been given two files:

1. `statistical_hits.txt` is a set of p-values and FDRs (corrected p-values) for a set of genes
2. `genome_annotation.txt` is a list of genome annotations

Write a script which will add the genome annotation to the statistical hits and write the results out to a new file.

Iterating over files

Mapping data extraction

In the Mapping Stats folder you've been given a set of 126 mapping results files from a large sequencing dataset where each file has a name like:

```
ERR2588244_Mbd3FLAG_Chd4_R1_GRCm38_bowtie2_stats.txt
```

..and has contents which look like:

```
211539334 reads; of these:
  211539334 (100.00%) were unpaired; of these:
    5252917 (2.48%) aligned 0 times
    151453128 (71.60%) aligned exactly 1 time
    54833289 (25.92%) aligned >1 times
97.52% overall alignment rate
```

Write a program which will iterate though these files and collate the percentage of reads which aligned exactly once (so 71.60% in the above example). Print out the file name with everything from `_GRCm38` onwards removed and the extracted mapping percentage.

Finding Files

Write a script which finds all of the Excel files (`.xlsx`) files under a specified starting directory. Have it search through the files in your home directory (or wherever you might have some excel files). For each file print its location and its size.

Tricky challenge (if you have time / motivation!)

In the Bacteria fold of the course data are a series of gzipped GTF files. These are the genome annotations for a series of different bacterial species. The GTF format is described here https://www.ensembl.org/info/website/upload/gff.html

Iterate through these files and read them using the gzip package. For each species count the number of genes present in the organism and the length of the largest gene along with its gene name.

Exercise 6: Writing a full application

Only one exercise this week, but it's a larger task than the previous exercises you've been given.

We want you to write a complete end-user application called `tf_search.py`

This program will take in the location of a DNA sequence in fasta format and the name of an output file and will search for transcription factor binding sites in the input sequence file and will report them into the output file. By default the program should search with all of the transcription factors it knows about, but the user should also be able to specify a specific sub-list of transcription factors to use.

We have provided you with a list of human transcription factors which contains their (ambiguous) consensus sequence in the file `Transcription Factors/human_tf_consensus.tsv`. You can either read this from wherever you have your data files unzipped, or you can put it alongside the script file and use the `__file__` special variable to find the location of the current script so you can find the associated data file.

You will need to split your program into functions and call them sequentially from an initial main function. The script you write should also be able to be called as a library. One suggested order of operations would be something like:

```
def main():
      options = read_options()
      sequence = read_sequence(options.sequence)
      tf_list = read_tf_list(options.tfs)
      hits = search_for_tfs(sequence,tf_list)
      print_hits(hits,options.outfile)
```

You may also want to write other functions to handle tasks within one of the top level functions (for example generating a regex from a consensus sequence). Use whatever structure you think best.

Each of your functions should have some basic documentation which will allow the help function to work with them.

- You should use the `argparse` package to manage the command line options. It's up to you how you choose to implement the options which are required.

- The sequence will be in standard fasta format and you should record both the sequence name and the contents of the sequence

- The TF list is a TSV file so you can split each line based on tabs to get the individual pieces of data. You only need the name of the factor and the consensus sequence from the file.

- Your output file should contain the following information for each hit.

    1. The name of the sequence
    2. The name of the transcription factor
    3. The start and end position for the match
    4. The consensus sequence for the transcription factor
    5. The actual sequence matched during the search

- Your output file can be either a CSV file or a TSV (tab separated value) file

We have given you a sequence for the scyl3 promoter which you can use to test the program. You should see hits in this sequence, including hits to FOXK1 and ISL1 if you want to test specific factors.

Tricky challenge if you have time

Some extensions to the basic exercise if you want to try some more advanced options. For these we have provided a gzipped version of all of the promoters on human chromosome 1 to give you a larger dataset to work with.:

- Write a test suite to go with your program and validate the basic functionality of the program
- Make your script also support reading from gzip compressed multi-sequence fasta files
- Make your script support searching on the reverse complement strand
- Add an option to ignore TFs with more than *n* matches, where *n* is an option
- Add progress messages so you can see what the program is doing, but add a `--quiet` option to suppress them.

Exercise 7: Writing an application using external resources

For your final exercise you're going to write another user-facing application, but this one will also make use of external API resources.

The application will take in one or more human gene names and will write out a file containing the details of the transcripts which exist for those genes. The information for this will come from the Ensembl REST APIs (https://rest.ensembl.org)

To connect to Ensembl you're going to use the requests package, which is not part of the standard library, so you will need to install it using pip.

Later on you are going to use the biopython package to parse a fasta format sequence file. For this you will need to install biopython, which is also available from PyPi.

Inputs

Your program should be able to run in either an interactive or non interactive manner. In the non-interactive mode you should be able to specify the list of genes on the command line, as well as the name of an output file to write to, eg:

```
python3 gene_query.py --genes Nanog Sox2 Brca2 --outfile gene_results.txt
```

The program should then run without further user interaction and write the results to the output file. You can emit progress messages but these should be sent to `sys.stderr` and you should be able to suppress these by adding a --quiet option to the command line. For the genes option you will need to add `nargs="+"` to the argument parser `add_argument` call to allow multiple values to be specified.

In the interactive version of the program you would be able to omit the gene names:

```
python3 gene_query.py --outfile gene_results.txt
```

..and you would be prompted to enter as many gene names as you liked in the terminal (one gene at a time).

Eg:

```
python3 gene_query.py --outfile gene_results.txt
Which gene? Nanog
Which gene? Sox2
Which gene? Brca2
Which gene?
```

..after entering a blank gene name the program should continue.

API queries

You will need to use two different API queries to get the results you want. The first will take in a gene name and return the Ensembl ID of that gene. If multiple genes match the same name take the first one. The API you need is this one: https://rest.ensembl.org/documentation/info/xref_external

You can query against the HGNC database which defines the gene names.

Once you have the gene ID you can then get the details of the transcripts for that gene. You are going to need to get the transcript ID and the sequence. To do this you can use this API https://rest.ensembl.org/documentation/info/sequence_id. The type of query will be `cdna` and you will also need to set `multiple_sequences=1` to allow multiple transcript sequences to be returned for a single gene ID.

Sequence Parsing

Once you retrieve the sequences you can get them as a string from the request object by using `r.text`. The string will contain multiple sequences in fasta format. Rather than parse this yourself you can use the SeqIO package from biopython https://biopython.org/wiki/SeqIO. Since the input to SeqIO need to be a stream (ie filehandle) rather than a string you need to use the `io.StringIO` package to create a stream from the string you get back https://docs.python.org/3/library/io.html.

Calculated Values

Once you have extracted the sequence from the parsed fasta file you need to calculate both its length and it's GC content.

In the output file the fields you need to record are:

1. Gene name (whatever the user provided initially)
2. Gene ID (the Ensembl ID you retrieved from the API)
3. Transcript ID (from the second API query)
4. Transcript length
5. Transcript GC content

A query for Nanog, Sox2 and Brac2 should look like:

```
gene_name   gene_id     transcript_name   length      gc
Nanog ENSG00000111704   ENST00000541267.5 836   49.8
Nanog ENSG00000111704   ENST00000229307.9 5182  44.7
Nanog ENSG00000111704   ENST00000526434.2 558   45.3
Nanog ENSG00000111704   ENST00000526286.1 870   49.0
Sox2  ENSG00000181449   ENST00000325404.3 2512  50.7
Brca2 ENSG00000139618   ENST00000544455.6 11854 35.8
Brca2 ENSG00000139618   ENST00000530893.6 2011  38.2
Brca2 ENSG00000139618   ENST00000380152.8 11954 36.2
Brca2 ENSG00000139618   ENST00000680887.1 11880 36.0
Brca2 ENSG00000139618   ENST00000614259.2 11763 35.8
Brca2 ENSG00000139618   ENST00000665585.1 2598  40.7
Brca2 ENSG00000139618   ENST00000528762.1 495   41.0
Brca2 ENSG00000139618   ENST00000470094.1 842   41.4
```

```
Brca2 ENSG00000139618   ENST00000666593.1 523   39.6
Brca2 ENSG00000139618   ENST00000533776.1 523   40.0
```