Assignment 3: Grids (text based alternative)

Assignment 3: Grids (text based alternative)

This assignment parallels Assignment 3: Images, but is text based for our friends with visual impairments, or just for another perspective on grids / nested lists (images are, after all, just grids of pixels!).

We are going to use a library we put together for you called TextGrid. In short, a TextGrid is exactly what it sounds like -- a grid (essentially a rectangle, with a width and a height) of letters. You can read all about it in this handout.

Brief biology refresher + terminology

This assignment is DNA-themed, so here's a bit of clarification on some biology terms.

In real life, DNA is composed of nucleotides (letters in the DNA sequence) which are essentially building blocks that can be one of four chemical molecules: adenine (A), cytosine (C), guanine (G), and thymine (T). You can think of DNA as 2 long chains of nucleotides, where each "link" in one chain is paired with its matching "link" in the other. The bases are paired as follows: adenine (A) with thymine (T); cytosine (C) with guanine (G). Conveniently, a pair of nucleotides (one on each chain) is called a base pair. The human genome comprises of around *3 billion* base pairs!

Throughout this assignment, we'll be representing DNA as a TextGrid! To represent DNA, we'll mostly be using the letters "A", "T", "C", and "G" to populate the TextGrid and represent nucleotides. In short:

- "nucleotide": a single letter in the DNA sequence, either "A", "T", "C", or "G", which we will represent as a single character in a TextGrid
- "genome", "sequence", "read": a DNA sequence, such as "ATTAG" or "ACTC", which we will represent as a TextGrid
- TextGrid: rectangular grid of characters which will represent nucleotides

Q1: Gene translation

We're going to start by writing a program in `translate_DNA.py` that uses a given DNA sequence, and translates it to its complement sequence using the base-pair complement rules. Your job is to translate the given DNA sequence, which will be provided as a TextGrid, like this:

- Change A to T
- Change T to A
- Change G to C
- Change C to G

Here is an example. Let's take the following text grid, which represents one sequence in the genome for COVID-19:
ATTAA
AGGTT

After running `python translate_DNA.py` on the TextGrid, your result will be:
TAATT
TCCAA

Q2 & Q3: Sequence assembly

We're going to write a program in `sequence_assembly.py` which will take in 3 different `TextGrid` objects that represent one gene, and output the correct sequence for that gene. Your job is to fill in the functions `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)` and `assemble_sequence()`.

Genomic sequencing, which is a sophisticated way of saying "making expensive machines read DNA", has seen many revolutionary advances in modern times, but is still far from perfect. DNA sequences are read incorrectly in some cases -- once in a while, an A is read as a C, a nucleotide (a letter in the DNA sequence) isn't read at all, etc. One solution scientists have come up with, to get around this inaccuracy, is just to read a given piece of DNA over and over and compare these reads to each other in order to piece together the true sequence and lower the risk that we've misread something. "Piecing back together" the different reads of the same piece of DNA is called sequence assembly.

Your job is to write the code to do sequence assembly, for small genomes which we will represent using TextGrids. You will be given 3 sequences (TextGrids) at a time that correspond to the same gene. Each individual sequence will have random mistakes or missing letters, but you will be able to piece them together in order to generate the true sequence and return the solution TextGrid.

We've split this problem into two milestones -- in the first, you'll write `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)`, and in the second you'll use your code from the first milestone to write `assemble_sequence()`.

Q2: Sequence assembly, Milestone 1

Let's start by writing a `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)` function. `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)` takes in three nucleotides (letters) and returns the nucleotide with the **most common, non-blank value**. That is, if you are given 'A', 'C', and 'A', `get_best_nucleotide(A, C, A)` would return 'A', and if you're given '_' (blank), '_', and 'C', `get_best_nucleotide(blank, blank, C)` would return 'C'. You can assume that you will always be given an unambiguous situation; *you will never be given a case where there is no clear right answer* (for example, finding the best nucleotide of C, G, and A).

We've written a `main()` function that lets you enter 3 nucleotides to input into `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)` , but you can also test your code by clicking the Check button in the bottom right corner to run the DocTests, or by adding your own DocTests! If you're adding your own DocTests, make sure that you're using the variables we've created (`A`, `T`, `C`, `G`, and `blank`) for your tests -- the parameters to `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)` are `Cell` objects which we've created for you in the first few lines of the DocTest, not strings!

Q3: Sequence assembly, Milestone 2

Copy paste your code from the previous milestone into `get_best_nucleotide(nucleotide1, nucleotide2, nucleotide3)` and write `assemble_sequence(sequence1, sequence2, sequence3)` to do the actual sequence assembly! The `main()` function we provide for you, which you should not modify, does the housekeeping of loading the sequences, and then calls `assemble_sequence()` to do the real work. You can assume that all three sequences passed into this function are the same size (have the same height and width).

Your function should create a new (blank) sequence of the same size as the grids passed into the function and then appropriately set the nucleotides in this new sequence to construct the solution sequence. The solution sequence is generated by simply placing the best nucleotide (most common, non-blank nucleotide, at that position across all three sequences) at each position. For example, if you're given 'AA_', 'ATA', and 'AT_', you will be able to discern that the solution sequence is 'ATA'.

We've given you a couple folders with three sequences each that will be used to test your code when you press the Mark button. You can view the sequences in these folders by clicking the upper left icon in your file workspace (directly to the left of the tab for the file `sequence_assembly.py` your code is in!). To change the folder you run your code on, change the value of the `SEQUENCES_FOLDER` constant at the top of the file to one of the other folder names, either `'tiny'`, `'no_blanks'`, `'long'`, or `'large'`.

Q4 (optional): Mystery Patches DNA

Your job is to generate 1 row of 4 DNA TextGrid patches. In the first patch, replace all A's with a '?.' In the second patch, replace all T's with a '?'. In the third patch, replace all C's with a '?'. Finally, in the fourth patch, replace all G's with a '?'. Print the resulting grid.

Here is an example. Let's take the following TextGrid (which corresponds to the patch `patches/patch1.txt` in your workspace):
GA
TC

After running your program on on the TextGrid, your program should print:

G?GAGA?A
TC?CT?TC

Mystery Patches DNA Code

We strongly recommend implementing a function called `edit_patch(to_change)` This function will take in a letter called `to_change` , which represents the letter in the patch that should be marked with '?'. `edit_patch(to_change)` reads in a TextGrid from a file and changes every instance of the `to_change` letter to '?'.

Here is an example run of the function `edit_patch(to_change)` . Let's take the following text grid:
GA
TC

After running `edit_patch('A')` on the TextGrid, this will be the result:
G?
TC

After running `edit_patch('G')` on the TextGrid, this will be the result:
?A
TC

You are also welcome to implement any helper functions that you might find useful. In the main function, you will read in a TextGrid from a file and generate the 4 patches pieced together in one row.

Type `python mystery_patches.py` to run your program. We've provided you three patches: `patch1.txt`, `patch2.txt`, and `patch3.txt`, all of which you can test your code on by changing the value of the constant `PATCH_NAME` to the name of the file you'd like to run the program on (e.g.

`PATCH_NAME = 'patch3.txt'`).

Mystery Patches Milestones

To help you take the Mystery Patches problem one step at a time, we've written some structured milestones for you to follow.

Pseudocode / roadmap

When coding a complicated task, it's frequently helpful to think of the problem in your native language first, instead of diving straight into Python. How would you explain how to solve the problem to a child? How might you split the task into smaller subtasks? Here's our pseudocode for this problem, with the corresponding milestones:

- The Mystery Patches effect is characterized by editing the same patch 4 times, replacing a different letter (A, T, C or G) with '?' each time. This sounds like a good place to decompose a helper function with parameters, since we're doing the same task (recoloring a patch) over and over, but need to be able to customize *how* we're doing the task (which letter to change). Milestone 1, `edit_patch(to_change)` , will implement this.
- Once we know how to edit the patches appropriately, we'll need to figure out how to add them to our final TextGrid (`final_grid`) at a given position. Milestone 2, `add_patch(patch, patch_index, final_grid)` , will implement this.
- Finally, we can finish up by writing the `main()` function to put everything together -- we'll want to use `edit_patch(to_change)` to generate 4 different patches, and then call `add_patch(patch, col, final_grid)` with the right parameters to place the patches in a row. Milestone 3 will implement this.

Hopefully this thought process makes some sense to you -- happy coding! Below are the milestones in more detail.

Milestone 1: `edit_patch(to_change)`

We recommend you start by filling in the `edit_patch(to_change)` function in the starter code which should use `PATCH_NAME` to create a new `TextGrid` and then check every element in the new patch to see if its value (if you're using a `for cell in patch` loop, you can access the value of a cell with `cell.value`) matches `to_change` . If its value matches, you should change its value to '?'. Finally, return the finished patch.

A quick way to test that your code works for this function is to call it in `main()` with a letter of your choice as the parameter `to_change` and print the result. You'll want to eventually remove this print statement, though, because you'll eventually want to only print `final_grid` !

Milestone 2: `add_patch(patch, patch_index, final_grid)`

This helper function should take in a `TextGrid` patch (`patch`) of dimensions `PATCH_SIZE` x

`PATCH_SIZE`, a `patch_index` integer which represents the index of the patch relative to the final grid (patch 0 is the leftmost patch, patch 1 is one to the right of that, patch 2 is one to the right of that, etc.), and `final_grid`, the `TextGrid` which is initialized for you in the starter code to be wide enough to accommodate exactly `N_COLS` patches. You should write code which applies `patch` to `final_grid` by changing one element in `final_grid` at a time, using `patch_index` to calculate where in `final_grid` to start applying the patch. You can optionally return `final_grid`, but even if you don't, your function will modify it!

As a note, `final_grid` is initialized to also be tall enough to accommodate `N_ROWS` rows, though for this problem we're only asking you to make it work for one row of 4 patches. As an extension, consider how you'd make this program work for multiple rows (how would the `add_patch(patch, patch_index, final_grid)` function need to change?).

Milestone 3: `main()`

We've got all the pieces we need, now to use the tools we've built! Fill out the `main()` function to add 4 patches in a row. How will you correctly increment the `patch_index` parameter to `add_patch(patch, patch_index, final_grid)`? How will you make sure to change the correct letter (`to_change`) to a '?' in each patch?