Exercise 6

NOTE only Python standard library packages are allowed i.e., no Pandas, NumPy, BioPython etc.

Questions

1. triangle.py (25 points)

   Write a script that prints a triangle of text. The script should take two command line inputs: the character to print, and the number of characters tall the triangle should be.

   Within your script, use functions to handle any calculations and printing. The body of your script should only contain a function call which passes the commandline input as arguments.

   You must include type hints and docstrings for you functions. Your docstrings should describe the function purpose as well as any arguments and returned values.

   i.e., your script should look something like this (you can name your functions whatever you like)

   ```
   #!/usr/bin/env python3

   import sys

   def func1(arg1: type, arg2: type) -> return_type:
       """Docstring"""
       ...

   def func2(more_stuff: type) -> return_type:
       """Docstring"""
       ...

   # The only line of code not in a function is calling a function
   func1(sys.argv[1], sys.argv[2])
   ```

   and the usage should be `triangle.py <character> <height>`

   And the output should look like this when run in a terminal

   ```
   $ ./triangle.py X 5
   X
   XX
   XXX
   XX
   X
   $ ./triangle.py @ 8
   @
   @@
   @@@
   @@@@
   ```

```
@@@@
@@@
@@
@
```

2. pair_parens.py (25 points)

   Earlier in the course, we worked briefly with Newick format trees. Part of the specification of Newick format is the each node of a tree must be enclosed within parentheses. i.e., each open parenthesis must have a corresponding closing parenthesis. Additionally, there can not be a close parenthesis that does not follow an open parenthesis.

   If you were to write a script to validate Newick format, you would need to check the correct pairings of parentheses as well as the other format rules. This task is to write a script that checks whether parentheses are paired.

   Write a script that uses one or more functions to assess whether or not all parentheses in an input string are paired.

   As with question 1, the body of your script should only have a single line (except `import sys` and the shebang) that is not within a function. That line should be calling a function and passing command line input as arguments. Your functions should use type hints and include useful docstrings.

   The usage of your script should be `pair_parens.py <test string>`

   Your script should print a result of either "PAIRED" or "NOT PAIRED" to the terminal according to the result of your analysis.

   Note that simply counting parentheses is not sufficient as a closing parenthesis followed by an opening parenthesis is not valid in Newick format.

   Some example data showing inputs and expected output (Note, you will need to put the parentheses in quotes)

```
$ ./q2_script.py "()"
PAIRED
$ ./q2_script.py "()()()"
PAIRED
$ ./q2_script.py "(()())"
PAIRED
$ ./q2_script.py "((()))"
PAIRED
$ ./q2_script.py "(()())"
PAIRED
$ ./q2_script.py "((()))())"
PAIRED
$ ./q2_script.py ")"
NOT PAIRED
$ ./q2_script.py ")("
NOT PAIRED
$ ./q2_script.py "()()("
```

```
NOT PAIRED
$ ./q2_script.py ")(((())"
NOT PAIRED
$ ./q2_script.py "))(("
NOT PAIRED
$ ./q2_script.py "())("
NOT PAIRED
```

3. get_homolog_seqs.py (50 points)

   Some of the previous exercises have required that you reuse code you have already written previously. It is common, when starting a new coding project, to have relevant code from a different project which does something like what you need in the new project. Perhaps you previously wrote code to read in a certain file type, or a function to perform a relevant calculation. In other cases you may join a project that someone else has already started and have to familiarize yourself with an existing codebase. In both cases, the ability to read an understand both how code works to achieve its current goal, and how that code could be modified for new purposes, is an important skill to develop. This question requires that you further develop the find_homologs script. However, this task may also require that you change some of the code you had previously written, depending on what you had written before.

   I have included in the canvas folder a script that I wrote (find_homologs.py) which can serve as the foundation for this question if you did not complete the extra credit portion of last weeks assignment, or if you would prefer to use my script. The script you use as the foundation of this question's solution will not be considered in your grade.

   Your task is the modify your find_homologs.py script so that does the following:

   Takes the following inputs as command line input:

   1. The path to a BLAST output (same as last exercise)
   2. The path to a BED file (same as last exercise)
   3. The path to an assembly in FASTA format
   4. The path to an output file that your script will write

   Performs the following operations:

   1. Processes the BLAST output to keep only hits with >30% identity and 90% length (same as last exercise)
   2. Identify BED features (i.e., genes) that have a BLAST hit entirely within the boundaries of the feature start and end (same as last exercise)
   3. Extract the sequence of identified homologous genes from the assembly sequence
   4. If the gene is encoded on the - strand (indicated in the BED file), reverse complement the sequence
   5. Write the sequences of the homologous genes to the specified output file (specified using command line input) in FASTA format (gene name as header).

   If you correctly check that BLAST hits are on the same sequence as BED features and within the boundaries of the feature, you should get 34 homologs for the *Vibrio cholerae* assembly.

If you correctly extract gene sequences, you should get the below sequence for the *V. cholerae glnL* gene:

```
>glnL
GTGAGTGCAGAATTAAGCCAAACCATCATTAATAATCAGGTCACATCAGTGCTCATTTTGGACGAGTCACTGATGATT
CGCTACGCCAACCCTGCCGCTGAACAGCTGTTTTCACAAAGTGCTAAACGCTTGATGCATCAAAGCTTAAATCATTTA
GTGCAACACTCCTCTCTCGATTTACAACTGCTCACGCAGCCACTCCAGAGCGGACAAAGCATTACTGACAGCGATGTC
ACCTTGGTGATCGATAGCAAACCCTTAATGCTCGAAGTCACCGTCAGCCCGATTTCTTGGCACAAAGAGCTGCTGTTA
CTGGCCGAAATGCGCACGATTGGTCAACAACGCCGGCTAACCCAAGAACTCAATCAACACGCTCAGCAACAAGCGGCT
AAGTTATTGGTCAGAGGCTTGGCTCATGAAATCAAAAATCCTTTGGGTGGTTTAAGAGGTGCGGCCCAGCTCTTAGAG
CGTATGCTTCCCGATCCGGCCCTGATGGAATATACCCAAATCATCATCGAACAGGCAGATCGCTTGCGGGGATTGGTT
GATCGCTTACTCGGCCCGCAACGTCCGGGGGGAGAAAAAATGGGAAAACCTTCACCTGATTTTGGAGAAGGTGCGTCAG
TTGGTCGAGCTAGAAGCGGGCGCGAATTTGGTCTTTGAGCGCGATTATGACCCAAGTCTGCCGAATATTTTGATGGAC
ACTGATCAAATCGAACAAGCCTTACTGAACATTGTCAGTAATGCGGCGCAAATTTTGACTAACCAAACGCACGGCGTG
ATCACCTTGCGCACCAGAACAGTGCATCAAGCCAATATCCATGGTCAACGTCATAAGCTCGTCGCCAGCATCGAGATT
ATCGATAACGGCCCCGGCATCCCTCCTGAGCTGCAAGATACGCTGTTTTATCCCATGGTGAGTGGCCGCGAAGGAGGC
ACTGGCTTGGGGTTATCCATTTCACAAAACCTGATCGATCAACATCAGGGAAAAATAGAGGTGCAAAGCTGGCCAGGA
CGCACCGTGTTTACCATTTATTTGCCAATTTTGAATTGCTGT
```

Note All processing must be done in Python except for running BLAST. You must run BLAST using a command like that below (with filepaths pointing to wherever the files are on your system). Don't use `-task tblastn-fast`. Your submission will be assessed using BLAST output created using a command like the one below.

```
tblastn -query data/HK_domain.faa -subject data/Vibrio_cholerae_N16961.fna -
outfmt '6 std qlen' > Vc_blastout.txt
```

Usage of your script should be

`get_homolog_seqs.py <blast file> <bed file> <assembly file> <output file>`

As with the other questions in this assignment all code except imports, the shebang, and a single function call should be within functions and those functions should include type hints and docstrings. Try to identify chunks of code that fit together nicely as a conceptual unit (e.g., code to open and read the contents of the BED file into an object). Each conceptual unit of code can be put in its own function. As you work on writing the script, don't be afraid to go back and edit code to break it up into functions - "refactoring" like that is often an iterative approach and is an essential part of developing anything more than the simplest scripts. Identifying what code constitutes a unit and deciding how units should behave is an important skill to develop when learning to code.