CSC108: Assignment 2: DNA Manipulation

Deadline: Monday, March 11, 2019 **before** 4:00pm
Initial results: Wednesday, March 13, 2019
Re-submission with 20% deduction (optional): Friday, March 15, 2019 **before** 4:00pm (no lates accepted)

What is re-submission? The assignment test results will typically be released within 48 hours of the deadline. You may choose to resubmit, fixing any errors detected by our tests, or to submit a late submission without the benefit of tests. These "re-submissions" will be accepted up until the deadline above with a 20% deduction. No late assignment re-submissions will be accepted (the assignment late penalty scheme does not apply to re-submissions). Since a 20% penalty is applied and the highest mark a re-submission can receive is 80%, the re-submission is most commonly used by students whose code had a minor error that resulted in many test cases failing. The majority of students do not and should not resubmit.

Frequently Asked Questions (with Answers)

A list of Frequently Asked Questions is here.

Goals of this assignment

- Continue to practice learning about a problem domain in order to write code about that domain.
- Continue to use the Function Design Recipe to plan, implement, and test functions.
- Design and write function bodies using loops (over strings and lists) and string and list manipulation. (You can do this whole assignment with only the concepts from Weeks 1 through 6 of CSC108.)
- Practice good programming style.

DNA Manipulation

Computer science concepts can be applied to solve problems in many different domains. In this assignment, the problem domain is deoxyribonucleic acid (DNA) manipulation.

Before you read the rest of the handout, please go read about the DNA problem domain.

The rest of this handout assumes that you have read about the problem domain and that you have an undertstanding of the following terms. (You might open both this page and the DNA problem domain page side by side so that it's easy to remind yourself of the terminology.)

- DNA
- base
- base pair
- palindrome
- DNA strand
- DNA sequence
- complement
- complementary strand
- DNA palindrome
- restriction site
- restriction enzyme
- recognition sequence
- DNA mutation
- 1-cutter

What To Do

In this assignment, you will add your work to two different files. The first file is named `palindromes.py` and you will add your code to implement the required functions listed in the table under Task 1 below. The second file is named `dna.py` and you will add your code to implement the required functions listed in the table under Task 2 below.

Files to Download

Please download the Assignment 2 Files and extract the zip archive. In the sections below we describe the functions that you will add to the starter files.

(Students who downloaded the zip file before 10:40am on Sunday, Feb 24th should download the revised a2_simple_checker.py file and replace the original file with the revised version.)

CSC108 A2 Checker

As with Assignment 1, we have provided a checker module, `a2_simple_checker.py`, which you can and should run on your code as you work. You are responsible for doing thorough testing of your own code and making sure it works correctly, but the checker is a good first step. Make sure you use it!

Task 1: Regular Palindromes

To help you get warmed up, the first part of this assignment asks you to write code that deals with regular (not DNA) palindromes. According to the Oxford Dictionary, a *palindrome* is "a word, phrase, or sequence that reads the same backward as

forward." For example, the phrases "*Madam, I'm Adam.*" and "*nurses run*" are palindromes. Notice from these examples that blanks and punctuation are ignored when considering whether or not a phrase is a palindrome. We also ignore digits. Only alphabetic characters are examined when determining whether or not a string is a palindrome. The word "*radar*" is an odd-length palindrome, and "*toot*" is an even-length palindrome.

In file `palindromes.py`, complete the function definitions for the functions listed in the table below. Use the function design recipe that you have been learning. We have included the type contracts in the table. Please read through the table to understand how the functions will be used.

Come up with your own examples. It's part of the process of understanding the domain, and is a skill any computer programmer needs.

Function name: (Parameter types) -> Return type	**Full Description (paraphrase to get a proper docstring description)**
`is_palindrome_word: (str) -> bool`	The parameter is a string consisting only of lowercase alphabetic letters. It may or may not be a palindrome. Return `True` if and only if the parameter is a palindrome. The empty string is considered to be a palindrome.
`is_palindrome_phrase: (str) -> bool`	The parameter is a string that may or may not be a palindrome. Return `True` if and only if the parameter is a palindrome, independent of letter case and ignorning non-alphabetic characters. To be clear, non-alphabetic characters should be ignored (that is, treated as if they are not present), and uppercase letters should be considered to be equal to their lowercase equivalents. **Hint:** Before writing code, write the phrase "Madam I'm Adam." on a piece of paper. This phrase is a palindrome. Think about the steps that you need to go through in order to confirm that it is a palindrome, and then think about how you can express those steps in code.
`get_odd_palindrome_at: (str, int) -> str`	The first parameter is a non-empty string consisting only of lowercase alphabetic characters, and the second parameter is a valid index into the string. Return the longest odd-length palindrome in the string that is centered at the specified index.

List of functions to implement for Assignment 2 Task 1

Task 2: DNA palindromes and restriction enzymes

This part of the assignment deals with DNA palindromes and restriction enzymes. Complete the function definitions for the functions listed in the table below in `dna.py`.

The tasks performed by these functions overlap — you should identify and call some of these functions from other functions as helpers.

The `dna.py` file that was provided for download contains a helper function named `get_complementary_base()` and some constants. You are encouraged to use them in your code.

Function name: (Parameter types) -> Return type	**Full Description (paraphrase to get a proper docstring description)**
`is_base_pair: (str, str) -> bool`	Each parameter is a single character representing a base (`'A'`, `'T'`, `'C'`, or `'G'`). Return `True` if and only if the two parameters form a base pair.
`are_complementary: (str, str) -> bool`	Both parameters are non-empty strings representing DNA strands. The two strands are of equal length and only contain characters from the four characters that represent bases: `'A'`, `'T'`, `'C'`, and `'G'`. Return `True` if and only if the two strands are complementary. **Hint:** on a piece of paper, write down `'GGATC'`. What string would be complementary to this string in a DNA sequence?
For the rest of the functions, keep using this technique where your first step is to construct an example of the parameters. Some functions are intricate, and you'll need to take notes.	
`is_dna_palindrome: (str, str) -> bool`	Both parameters are non-empty strings representing DNA strands. The two strands form DNA: `are_complementary` would return `True` if called on the two inputs. Return `True` if and only if the DNA strands represented by the two parameters form a DNA palindrome. **Hint:** re-read the last paragraph in the DNA Palindromes description. Can you call a function from `palindromes.py` to do most of the work?
`restriction_sites: (str, str) -> List[int]`	The first parameter represents a strand of DNA. The second parameter is a recognition sequence. Return a list of all the indices where the recognition sequence appears in the DNA strand. (These are the restriction sites.) For example, if the recognition sequence appears at the beginning of the DNA strand, then `0` would be the first item in the returned list. Only look from left to right; don't reverse either parameter to look backwards. The indices in the list that is returned are the indices of the start of each occurrence of the recognition sequence in the

	DNA strand. **Hint:** `str.find` will probably be helpful.

`match_enzymes: (str, List[str], List[str]) -> List[list]`
 	
`The return type is a list of two-item [str, List[int]] lists`
	The first parameter represents a strand of DNA. The last two parameters are parallel lists: the second parameter is a list of restriction enzyme names, and the third is the corresponding list of recognition sequences. (For example, if the first item in the second parameter is `'BamHI'`, then the first item in the third parameter would be `'GGATCC'`, since the restriction enzyme named `BamHI` has the recognition sequence `GGATCC` — you can refer to the table of restriction enzymes to see more examples of restriction enzymes and their recognition sequences.) Return a list of two-item lists where the first item of each two-item list is the name of a restriction enzyme and the second item is the list of indices (in the DNA strand) of the restriction sites that the enzyme cuts.
	
	The indices in the sublist that is returned are the indices of the start of each occurrence of the corresponding recognition sequence in the DNA strand.
	
	Hint: how can you use function `restriction_sites`?
	
	From the FAQ:
	Question: In match_enzymes, should an enzyme be included in the returned list if its recognition sequence does not appear in the strand?
	Answer: Yes it should, but the list of indices associated with the enzyme should be an empty list. The length of the list returned by match_enzymes must be the same as the length of the two parallel lists (enzyme names, corresponding recognition sequences).

`one_cutters: (str, List[str], List[str]) -> List[list]`
 	
`The return type is a list of two-item [str, int] lists`
	The parameters are the same as for `match_enzymes`. Return a list of two-item lists representing the 1-cutters for the DNA strand that is the first parameter. The first item of each two-item list is the name of a restriction enzyme and the second item is the index (in the DNA strand) of the one restriction site that the enzyme cuts.
	
	The index in the sublist that is returned is the index of the start of the corresponding recognition sequence in the DNA strand.

	The first parameter represents a list of mutated strands of DNA. The second parameter represents a clean strand of DNA. The third and fourth parameters are the same as the second and third parameters for `one_cutters` and `match_enzymes`. The function modifies the list of mutated strands that share a

	1-cutter with the clean strand by replacing all bases starting at the 1-cutter in the mutated strand with all bases starting at the 1-cutter in the clean strand, up to and including the end of the strand.

`replace_mutations: (List[str], str, List[str], List[str]) -> None`	Assume that the clean strand contains exactly one 1-cutter from the enzyme names and recognition sequences provided.
	
	Here is an example of a call to `replace_mutations`. You should use this example to help you understand what the function does, and then write two more of your own examples to include in your docstring.
	
	`>>> strands = ['ACGTGGCCTAGCT', 'CAGCTGATCG']`
	`>>> clean = 'ACGGCCTT'`
	`>>> names = ['HaeIII', 'HgaI', 'AluI']`
	`>>> sequences = ['GGCC', 'GACGC', 'AGCT']`
	`>>> replace_mutations(strands, clean, names, sequences)`
	`>>> strands`
	`['ACGTGGCCTT', 'CAGCTGATCG']`

List of functions to implement for Assignment 2 Task 2

When you get stuck

If you get stuck, it's likely that the terminology is confusing you. Refer to the DNA manipulation problem domain page often. Whenever you encounter a term you don't know, use "find" to search for it in this page.

Use the function design recipe process that we've been practicing. *In a new problem domain with tricky functions this is particularly important.* You need to be confident that you know what you are implementing before you start writing code. Come up with multiple examples — with expected return values — before you start coding. If you're not sure whether an example you've created is correct, you're welcome to post it on the discussion board to get feedback.

Want to earn a good mark? Test your work!

You should carefully verify your code *before submitting* to determine whether it works: the Test step of the Function Design Recipe is particularly important for assignments. Once the deadline has passed, we will run our own set of tests on your submission.

To test your work, you should call on each function with a variety of different arguments and check that the function returns the correct value in each case. This can be done in the shell or using another `.py` file, but must not be done in your submitted `.py` files.

No Input or Output!

Your `palindromes.py` and `dna.py` files should contain the starter code, plus the function

definitions specified above. These files must *not* include any calls to the `print` and `input` functions. Do *not* add any `import` statements. Also, do *not* include any function calls or any other code outside of the function definitions.

CSC108 A2 Checker

The purpose of the simple checker is to help you make sure that we will be able to test your code, and to check your code for style errors. To use the checker, make sure that `a2_simple_checker.py`, `palindromes.py`, `dna.py`, and the `pyta` directory are all in the same directory, **exactly as they were in the zip file we provide**, and run `a2_simple_checker.py`. Be sure to scroll through and read all messages.

The checker tests two things:

- whether your functions have the correct parameter and return types, and
- whether your code follows the Python and CSC108 style guidelines.

If the checker passes for both style and types:

- Your function parameters and return types match the assignment specification. **This does not mean that your code works correctly in all situations.** We will run additional tests on your code once you hand it in, so be sure to thoroughly test your code yourself before submitting.
- Your code follows the style guidelines.

If the checker fails, carefully read the message provided:

- It may have failed because one or more of your parameter or return types does not match the assignment specification, or because a function is misnamed. Read the error message to identify the problematic function, review the function specification in the handout, and fix your code.
- It may have failed because your code did not follow the style guidelines. Review the error description(s) and fix the code style. Please see the PyTA documentation for more information about errors.

Make sure the checker passes before submitting.

Marking

These are the aspects of your work that may be marked for A2:

- **Correctness (80%):** Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.
- **Coding style (20%):** Make sure that you follow Python style guidelines that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we don't provide an exhaustive list of

style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with one exception: docstrings may be evaluated separately. For each occurrence of a PyTA error, one mark (out of 20) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.

What to Hand In

The very last thing you do before submitting should be to run the checker program one last time.

Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit `palindromes.py` and `dna.py` on MarkUs by following the instructions on the course website. Remember that spelling of filenames, including case, counts: your files must be named exactly as above.