CSC108H Assignment 1: Connect-N

Deadline: 4 February 2019 **before** 4:00pm
Initial results: 6 February 2019
Re-submission with 20% deduction (optional): 8 February 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.

Preamble: what is a problem domain?

A *problem domain* is an area of expertise that a programmer needs to understand in order to write a program related that domain. For example, if a programmer is hired by a non-profit organisation to help with their financial database, that programmer might have to learn how taxes apply to non-profit organisations, and then represent that knowledge in a working computer program.

Every assignment in this course will have a problem domain that you will need to learn about, and the bulk of each assignment handout will describe that domain.

The problem domain for this assignment is a Connect-N game. Most of this handout describes what that means, including how the problem domain concepts will be represented in your solution. Please ask for help if something doesn't make sense to you!

Goals of this assignment

- Practice learning about a problem domain in order to write code about that domain.
- Use the Function Design Recipe to plan, implement, and test functions.
- Write function bodies that exercise the concepts of: variables; numeric, bool, and string types; operators/functions for numeric, bool, and string types; and conditional statements (if-statements). (You can do this whole assignment using only the concepts from Weeks 1, 2, and 3 of CSC108.)
- Learn to use Python 3 (version 3.7), Wing101, provided starter code, a type-check module, and other tools.

Connect-N

Connect-N is a generalised version of a game called Connect Four. Each player has a symbol to play with (usually represented as `R` and `B`) and the goal is to be the first player to place N of their symbols in a straight line on the game board (either across a row, down a column, or diagonally). The Wikipedia page for Connect-Four (have a look!) provides a good description of the game and the rules.

In this assignment, you are to complete some functions that make up part of a larger program for playing a simplified version of Connect-N. The game will be played with an M-by-M board, where M is an integer between 1 and 9, inclusive. A player will win when they place N symbols consecutively in a row, column or along a diagonal of the game board. Note that N may be smaller than M but may not be larger than M. Our game boards will **always be square**. When you have completed your functions for this Assignment, you will be able to play games of Connect-N on your computer! The graphics won't be fancy, but we think that's okay, since the purpose of the assignment is to give you some practice with the parts of Python that you will learn in the first few weeks of the course.

Preliminary details and terminology

So far in the course, you have learned how to use variables to refer to objects that hold a single numerical or logical value, or a string of characters. For this assignment, we'll use what you've learned so far to write code that works with variables that refer to objects that represent game boards. We can think of our game boards as looking like this:

```
 R | B | R
---+---+---
   | B | R
---+---+---
 B | R |
```

To get started, we will need some terminology so that we can discuss Connect-N game boards and their parts.

Let's describe the game board given above as being a 3-by-3 grid of **cell**s. We will say that the **game board size** is 3, since the number of cells along each side of the board is 3. Rows run horizontally across the board, and columns run vertically. There are two diagonals of length 3 on the board: the DOWN_RIGHT diagonal that runs from the upper-left corner to the bottom-right corner, and the DOWN_LEFT diagonal that runs from the upper-right corner to the bottom-left corner. The board also contains DOWN_RIGHT diagonals of length 2, and DOWN_LEFT diagonals of length 2.

We will need a way to describe the position of each cell on the board. This can be done by using an ordered pair of numbers, representing row and column indices. In the example board above, the **first row** (the row with row index 1) has a cell with an R (in the column with column index 1), a cell with a B (column index 2), and another cell with an R (column index 3). The **second column** has two cells containing B's (row indices 1 and 2), and a cell containing an R (row index 3). The DOWN_RIGHT diagonal of length 3 contains, in order, an R, a B and an empty cell. One of the two DOWN_LEFT diagonals of length 2 contains two Rs. For each cell in our example 3-by-3 board, the row index will be either 1, 2, or 3, and the column index will also be either 1, 2 or 3. The cell in position (1, 2) (row 1 and column 2) contains a B, while the cell in position (3, 3) is empty. **If at this point you are confused by the terminology, ask a question in class or on Piazza!** You will have a hard time completing the assignment if you don't understand the terms that have been used!

Representing the game board

When representing a game board like the one above in a program, we only need to store the contents of each cell, not the divider symbols (i.e., `|`, `-`, and `+`) that are used to separate cells when game boards are displayed. This leaves us with a game board that looks like this:

```
RBR
 BR
BR
```

It can be difficult to interpret boards that contain a lot of empty cells, so let's use the hyphen symbol `-` to fill empty cells, as shown below:

```
RBR
-BR
BR-
```

We have not yet studied how to store 2-dimensional information in Python programs, so we will need a different way to represent our game board. The approach that we will take for this Assignment is to store the rows one after another, starting with the first row. For the example board above, we would get:

```
RBR-BRBR-
```

And how might we store this information? We will use a Python `str`. With these choices, the Python statement:

```
game_board = 'R-BR'
```

creates a variable named `game_board` that refers to the `str` object with length 4 `'R-BR'`, which represents a 2-by-2 game board that looks like this:

```
 R | -
---+---
 B | R
```

Accessing cells

We have used row **and** column indices to describe the position of each cell in the grid representation of a game board. But each character in a Python `str` is in a position that is described by a single index. How is the Python program going to translate between row and column indices and `str` indices? To answer this question, we will have to determine a formula!

One good way to figure out the formula is to play with a concrete example. Suppose a game board was 4-by-4, and that each cell was filled with a different letter from A through P. (Yes, in a real Connect-N game, only two symbols would be used, but for this example, it helps to put a different letter in each cell.) The grid representation could look like this:

```
ABCD
EFGH
IJKL
```

```
MNOP
```

and then the `str` representation would look like this:

```
'ABCDEFGHIJKLMNOP'
```

Consider the same grid representation as above, with index labels added:

	column indices			
row indices	**1**	**2**	**3**	**4**
1	A	B	C	D
2	E	F	G	H
3	I	J	K	L
4	M	N	O	P

Game board grid with indices

... and the string representation from above, again with index labels added, as well as the row and column information from the previous table:

row, col	1, 1	1, 2	1, 3	1, 4	2, 1	2, 2	2, 3	2, 4	3, 1	3, 2	3, 3	3, 4	4, 1	4, 2	4, 3	4, 4
str_index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

(row, col) indices vs string indices

We now explore how `str_index` changes as we move across a row and down a column. This will help us determine a formula that relates row and column indices and `str` indices.

In the example above, we see that the character in the cell with position `(2,1)` (cell with position row 2 and column 1) has `str_index` 4. The other cells in that row have `str` indices 5, 6 and 7. We conclude that when moving one cell across a row, the `str_index` increases by 1. The cells in the column with column index 3 have `str` indices 2, 6, 10 and 14. Moving one cell down a column increases the `str_index` by 4, the size of the game board. We conclude that when moving one cell down a column, the `str_index` increases by the game board size.

We now introduce variables that will allow us to express the formula explicitly.

Let a variable named `str_index` refer to the position of a cell in the `str` representation of a game board with size `board_size`, and let variables `row_index` and `col_index` refer to the position of the same cell in the grid representation of a game board. From what we have seen, we can conclude that `str_index` depends on `row_index`, `col_index` and `board_size`.

Plugging in a few examples will show that, given a `row_index`, `col_index` and `board_size`, the following formula will compute `str_index`:

```
(row_index - 1) * board_size + (col_index - 1)
```

(The minus `1`'s are needed for the arithmetic to work out. This is because we chose to index our rows and columns starting at 1. As a thought exercise, what would the formula be if we chose to index the rows and columns starting at 0?)

Now that we have described how our game boards can be stored in a Python program, we move to a discussion of what files to download, and then a specification of the functions that you will need to write to complete your Assignment solution.

Files to Download

Please download the Assignment 1 Files and extract the zip archive. The following paragraphs explain the files you have been given.

Starter code: `connect_n_functions.py`

This file contains some constants, the header and the complete docstring (but not body!) for the first function you are to write. You will update this file to include the complete functions that you write.

Starter code: `connect_n_program.py`

This file contains the main program and when it is run, the functions that you wrote and put in the `connect_n_functions.py` file will be called. **Do not** make any changes to the `connect_n_program.py` file. You will be able to understand most of this code by week six.

Checker: `a1_simple_checker.py`

We have provided a checker program that you should use to check your code. See below for more information about `a1_simple_checker.py`

Constants

Constants are special variables whose values do not change once assigned. A different naming convention (uppercase pothole) is used for constants, so that programmers know to not change their values. For example, in the starter code, the constant `EMPTY` is assigned the value `'-'` at the beginning of the module and the value of `EMPTY` should never change in your code. When writing your code, if you need to refer to the symbol used to show that a cell has not yet been chosen, you should use `EMPTY`, rather than '-'. The same principle must be used when the value of the other constants are used in your code.

Using constants simplifies code modifications and improves readability. If we later decide to use a different symbol in a non-selected cell, we would only have to change the symbol in one place (the `EMPTY` assignment statement), rather than making changes throughout the program.

The starter code provides the following constants for you to use:

Constant	`EMPTY`	`DOWN`	`ACROSS`	`DOWN_RIGHT`	`DOWN_LEFT`	`MAX_BOARD_SIZE`
Value	`'-'`	`'down'`	`'across'`	`'down_right'`	`'down_left'`	`9`

Constants provided in the starter code

What to do

In the starter code file `connect_n_functions.py`, complete the following function definitions. We expect you to use the Function Design Recipe from the Week 2 PCRS Prepare videos, and to write complete docstrings for each function.

We have included the type contracts in the table below. We will be evaluating your docstrings in addition to your code. You must include two examples in your docstrings. You will need to paraphrase the full descriptions of the functions to get an appropriate docstring description; copying and pasting what you see in the table below is not sufficient. Check the steps of the Function Design Recipe to make sure you have properly paraphrased.

You may care to read through `connect_n_program.py` to see how the functions that you will write can be used. The code in the `connect_n_program.py` file uses Python statements that you may not have learned yet, so don't worry if you do not understand it fully.

The `connect_n_functions.py` starter code file starts with the definition of a few constant variables (see above) and a complete docstring for the `between` function. In your Assignment 1 solution, **do not** change the code that we have provided. You are to **add your Python statements** to this starter code file and fulfil the requirements listed in the table below.

Function name: (Parameter types) -> Return type	**Full Description (paraphrase to get a proper docstring description)**
`between:` `(int, int, int) -> bool`	The first parameter refers to an `int` value, the second to the minimum value for a range of values, and the third to the maximum value for a range of values. Assume that the value of the second parameter is less than or equal to the value of the third parameter. (That means that, when you write your code, you don't need to worry about other situations. As always, the assumptions should be described in a precondition.) This function is to return `True` if and only if the value of the first parameter is not less than the second parameter and not more than the third parameter. In other words, `True` is returned if and only if the first parameter is between the second and third parameters, or equal to one or both of them.
`game_board_full:` `(str) -> bool`	The parameter refers to a valid game board. This function is to return `True` if and only if all of the cells in the game board have been chosen. That is, `True` is returned if and only if there are no `EMPTY` characters in the game board. (See above for discussion of the `EMPTY` character.)
`get_board_size:` `(str) -> int`	The parameter refers to a valid game board. Given that our game boards are square, this means that the length of the parameter is a perfect square. This function is to return the length of each side of the given game board. NOTE: Make sure that your `get_board_size` function's return type is correct.
`create_empty_board:` `(int) -> str`	The parameter refers to the size of a valid game board. You may assume that its value is ≥ 1 and ≤ `MAX_BOARD_SIZE`. This

	function is to return a string for storing information about a game board whose size is given by the parameter. Each character in the returned string is to have been set to the `EMPTY` character, to indicate that no cells have been chosen yet.
`get_str_index: (int, int, int) -> int`	The first and second parameters refer to the row and column indices, respectively, of a cell in a valid game board whose size is given by the third parameter. You may assume that the parameters refer to valid values. This function is to return the index in the string representation of the game board corresponding to the given row and column indices.
`make_move: (str, int, int, str) -> str`	The first parameter refers to a one character string containing a symbol (usually, but not necessarily, an `'R'` or `'B'`). The second and third parameters refer to row and column indices, respectively, of a cell in the valid game board referred to by the fourth parameter. This function is to return the game board that results when the given symbol is placed at the given cell position in the given game board. You may assume that the cell at the specified row and column in the game board is initially `EMPTY`.
`get_increment: (str, int) -> int`	The first parameter refers to a string that describes one of the four directions: `DOWN`, `ACROSS`, `DOWN_RIGHT`, or `DOWN_LEFT`. The second parameter refers to a valid game board size. This function is to return the difference between the `str` indices of two adjacent cells on a line that goes in the direction specified by the first parameter. You may assume that the given direction is one of the four directions listed above.
`get_last_index: (int, int, str, int) -> int`	The first and second parameters refer to the row and column indices, respectively, of a cell in a valid game board whose size is given by the last parameter. The third parameter refers to a string that describes one of the directions: `DOWN`, `ACROSS`, `DOWN_RIGHT`, or `DOWN_LEFT`. This function is to return the `str` index of the last cell in a line that begins at the specified location and goes in the specified direction all the way to the game board boundary, on a game board of the specified size. You may assume that the row or column number is valid for the game board. You may also assume that, when the direction is `DOWN_LEFT`, the row and column are either on the top row or the rightmost column of the game board, and when the direction is `DOWN_RIGHT`, the row and column are either on the top row or the leftmost column of the game board.

List of functions to implement for Assignment 1

No Input or Output!

Your `connect_n_functions.py` file should contain the starter code, plus the function definitions specified above. `connect_n_functions.py` 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 other code outside of the function definitions.

How should you test whether your code works?

First, run the checker and review ALL output — you may need to scroll. You should also test each function individually by writing code to verify your functions in the Python shell. For example, after defining function `game_board_full`, you might call it from the shell (e.g., `game_board_full('BRRRBRRBB')`) to check whether it returns the right value (`True`). One call usually isn't enough to thoroughly test the function — for example, we should also test `game_board_full('B-BR-RB-B')` where it should return `False`.

Once you've checked each function individually, play the game by running the `connect_n_program.py` starter code to see whether it works as expected. If not, go back to testing the functions individually. It is part of the assignment for you to decide which tests to use.

CSC108 A1 Checker

We are providing a checker module (`a1_simple_checker.py`) that tests two things:

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

To run the checker, open `a1_simple_checker.py` and run it. Note: the checker file should be in the **same** directory as your `connect_n_functions.py`, as provided in the starter code zip file. Be sure to scroll through and read all messages.

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 A1:

- **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 `connect_n_functions.py` on MarkUs by following the instructions on the course website. Remember that spelling of filenames, including case, counts: your file must be named exactly as above.