SFU ID: ....
(not your student number!)

CMPT 120—Midterm (Mar. 21st)

SOLUTIONS

BEFORE YOU START DOING THE MIDTERM:
PRINT using CAPITAL LETTERS
YOUR SFU ID AT THE TOP LEFT OF EVERY PAGE.

YOUR SFU ID: is with letters (mine is "jlumbros").

A few instructions to be **read before taking the midterm**:

- **If you are using a crib/cheat sheet**, please hand it in with your exam, for an extra two bonus points. This is for research purposes, and it will not affect your grade in any way (other than the two bonus points). **Please put your SFU ID in printed capital letters on every page of your crib sheets to get the extra points.**
- There are two exercises in this midterm, and they are both mandatory.
- **Notice that the exercises run on several pages**. There are **8 pages** (on **4 sheets**). Count all pages to make sure you have all of them.
- When you are done, **remain seated until the end of the quiz**;
- I know that other faculties ask for your student number (30114762...), but I am asking for your SFU ID (somename...); **ALL QUIZZES THAT HAVE A STUDENT NUMBER INSTEAD OF AN SFU ID WILL GET A -25.00 GRADE; NOT JUST 0.00, -25.00, CONSIDER YOURSELF FOREWARNED!!!!**
- If what I have seen of Assignment 1, and of Course Exercise 7 is any indication, this midterm should be right down your alley. Keep calm: be more concerned with answering questions right, than doing as many as you can. If the midterm seems like it was too longs, grades will be adjusted accordingly.

SFU ID: ....
(not your student number!)

EXERCISE 1: correct code that does not work [20 pts + 2 extra pts]

Recall the CodeWrite exercise for which you had to write a function `matching_parentheses(s)` which took one string parameter `s`, and returned `True` if the parentheses in `s` where matching and `False` if not:

- `"((hello))(a)"` and `"(()(())())"` are examples of strings with matching parentheses (the function would return `True`);
- `")hello("`, `")()("` and `"(()))()"` are examples of strings with non-matching parentheses (the function would return `False`)

Here are two actual submissions that did not work. Your goal is to **make the minimum number of modifications** to make the submissions work. You can:

- remove a line: strike it out
- modify a line: for this, strike out the line in the original, and write in the box to the right the line you want replacing it
- add a line: for this, simply write in the box to the right, with an arrow pointing at zone where you want to add a line

For example:

`def product_of_range(n):`
`  ~~result = 0~~`
`  for i in range(1,n+1):`
`    result = result * i`
	`result = 1`
	
	`return result`

1) [10 pts] For help, try running the function with `")("` and with `")()"`.

```
def matching_parentheses(s):
  exists = False
  for i in range (len(s)):
    if s[i] == "(" or s[i] == ")":
      match = 0
      seen = 0
      exists = True
  if exists:
    for i in range (len(s)):
      if s[i] == "(":
        match = match + 1
        seen = seen + 1
      elif seen > 0 and s[i] == ")":
        match = match - 1
        seen = seen - 1
  if exists:
    if match == 0:
      return True
    else:
      return False
  else:
    return True
```

(`seen <= 0` below, also possible)

```
elif seen == 0 and s[i] == ")":
  return False
```

Other solutions are possible of course: what is important is that if ever the number of closed parentheses becomes larger than the number of open parentheses (that is, whenever `seen` becomes negative), we should return False.

2) [10 pts] In the following code, `"hey) (ho"` is an example that fails.

```python
def matching_parentheses(s):
  openbrackets = 0
  closebrackets = 0

  for k in s:
    if k == "(":
      openbrackets = openbrackets + 1
    elif k == ")":
      closebrackets = closebrackets + 1

  if closebrackets == openbrackets:
    return True

  return False
```

```python
if openbrackets < closebrackets:
  return False
```

Again the issue here is immediately returning False if ever there are more closed parentheses than open.

3) [2 extra pts] Of these two submissions, which do you find easier to understand, and why? Which would you recommend as a preferred coding style?

- **The first submission contains a lot of redundant code: for instance, both the variables seen and match always contain the exact same thing (so you don't need two variables; testing the existence in the beginning of parentheses is not necessary. This first submission is very confusing.**

- **<u>More importantly</u>, the variable names do not help understand their function. So it is hard to follow how the code works.**

- **On the other hand, the second submission has clearly named variables, which aid in the understanding of the code. Its structure is very straightforward.**

The second submission is a preferred way of coding.

SFU ID: ....
(not your student number!)

EXERCISE 2: Tic-Tac-Toe, Connect 4... with lists of lists [28 pts + 5 extra pts]

1) [4 pts] Fill in the following table, knowing that `L = [2, 3, 5, 7, 11]`.

Expression	Value	Explanation
`L[1]`	3	`L[1]` returns the second element of list `L`
`L[-1]`	**11**	**L[-1] returns the last element of list L**
`L[0:3]`	**[2, 3, 5]**	**L[0:3] returns the sublist of elements in L, starting in position 0 and ending in position 3 (which is <u>not</u> included)**
`L[:]`	**[2,3,4,7,11]**	**L[:] returns a copy of the entire list**

We consider that we have a variable `boxes` containing a list of list. You may for instance consider that

```
boxes = [ [ "O", "O", "X", "X" ],
          [ "X", "X", "O", "X" ],
          [ "O", "O", "X", "O" ],
          [ "X", "X", "O", "O" ] ]
```

2) [2 pts] How can you access the list of the row containing the **bold** element (in other words, the row `[ "X", "X", "O", "X" ]`) from the variable `boxes`?

boxes[1]

3) [2 pts] How can you access the element that is in **bold**, from variable `boxes`?

boxes[1][1]

Grading: -1 point from what the grade would be for both questions if index is [2] and [2][2] instead of the correct one (instead of 0 points for the question).

For instance, if answers are "boxes[2]" and "boxes[2][2]", 3 points for both questions (4 - 1)

SFU ID: ....
(not your student number!)

4) [10 pts] Write a function `check_lines(grid)` that takes a list parameter `grid`, which is a list of list of strings that are either `"X"` or `"O"` or `""`. The function returns `"X"` if there are **three** `"X"` aligned **on the same line**, `"O"` if there are three `"O"` aligned on the same line, and `False` if nothing is aligned. *Grids always have* ***at most*** *one alignment.*

(Don't worry if you do not fill up all the space, there is way more than needed to make sure you are not cramped!)

(This is only two possible answers, but there are plenty.)

These answers all assumes that `grid` has the same number of lines as columns. But you could easily forget this condition, by always having the second loop, if it uses `range(len(grid))` do instead `range(len(grid[0]))`

Grading note: these answers assume that grid has some size and the same number of lines and colums. It is fine if the student has assumed that the grid is a 4x4 (as in the example above). If the student has considered that the grid is a 3x3 like the original Tic-Tac-Toe, grade normally and -3 points (per question).

Cool answer suggested by one of the students in this class. It doesn't use flags.

```
def check_lines(grid):
    for line in grid:
      for i in range(len(grid)):
        next_three = line[i:i+3]
        if next_three == ["X"]*3:
          return "X"
        if next_three == ["O"]*3:
          return "O"
    return False
```

Expected answer, which uses flags.

```
def check_lines(grid):
    for line in grid:
      symbol = ""
      num_in_line = 0
      for ch in line:
        if ch == symbol:
          num_in_line = num_in_line + 1
        else:
          symbol = ch
          num_in_line = 0  #(*)
        if num_in_line == 3:
          if symbol == "X" or symbol == "O":
            return symbol
    return False
```

If this scheme is globally adopted (even if not correct at all): 6 points
If closer to correct, common errors and their costs:

- forget to check num_in_line inside the loop (checking it after would not find the "X" aligned in ["X", "X", "X", "O"], because the counter would be reset by the "O") --> - 1.5 points
- forget to check if symbol is "X" or "O" (could return "" or " "): - 0.5 point
- forget to reset (num_in_line = 0 where there is the star): -2 points

Another answer, which makes it easier to do the next question, same as previous answer, but iterate over positions instead of elements.

```
def check_lines(grid):
    for i in range(len(grid)):
      symbol = ""
      num_in_line = 0
      for j in range(len(grid)):
        if grid[i][j] == symbol:
          num_in_line = num_in_line + 1
        else:
          symbol = grid[i][j]
          num_in_line = 0  #(*)
        if num_in_line == 3:
          if symbol == "X" or symbol == "O":
            return symbol
    return False
```

Same grading as before.
If this scheme is globally adopted (even if not correct at all): 6 points
If closer to correct, common errors and their costs:

- forget to check num_in_line inside the second loop (checking it after would not find the "X" aligned in ["X", "X", "X", "O"], because the counter would be reset by the "O") --> - 1.5 points
- forget to check if symbol is "X" or "O" (could return "" or " "): - 0.5 point
- forget to reset (num_in_line = 0 where there is the star): -2 points

Other possible answer, uses if i < len(grid) to check if index is out of bound.

SFU ID: ....

(not your student number!)

5) [10 pts] Same as the previous question, except this new function `check_columns` is now checking **columns** instead of lines.

(This is only three possible answers, but there are plenty.)

```
def check_lines(grid):
    for i in range(len(grid)):
      for j in range(len(grid)-3+1)
        next_thr = [grid[j][i],grid[j+1][i],grid[j+2][i]]
        if next_thr == ["X"]*3:
          return "X"
        if next_thr == ["O"]*3:
          return "O"
    return False
```

Iterating over symbols is **NOT** practical, so there is not equivalent to the second answer in the previous question. (Grading: if the student used iterating over elements and it does not work, give them 3 points for trying.)

Iterating over positions is a breeze, just two characters to swap compared with previous answer:

```
def check_lines(grid):
    for j in range(len(grid)):      # swapped i with j
      symbol = ""
      num_in_line = 0
      for i in range(len(grid)):    # swapped j with i
        if grid[i][j] == symbol:
          num_in_line = num_in_line + 1
        else:
          symbol = grid[i][j]
          num_in_line = 0  #(*)
        if num_in_line == 3:
          if symbol == "X" or symbol == "O":
            return symbol
    return False
```

Same grading as before, except: +1 point (bonus if the student has full points on everything), if the student saw that there just needs to swap i and j

6) [5 extra pts] **THIS QUESTION IS A HARDER BONUS QUESTION THAT MUST ONLY BE DONE IF YOU HAVE BEEN ABLE TO CONFIDENTLY COMPLETE THIS ENTIRE EXERCISE.**

Same as the previous question, except this new function `check_diagonals` is now checking **diagonals** instead of columns or lines.

These all assumes that `grid` has the <u>same number of lines as columns</u>. But you could easily forget this condition, by always having the second loop, if it uses `range(len(grid))` do instead `range(len(grid[0]))`

```
def check_diagonals(grid):
    for i in range(len(grid)):
      for j in range(len(grid)):
        # (i,j) is going to be the starting point
        # we have to check (i,j) (i+1,j+1) (i+2, j+2)
        symbol = grid[i][j]
        num_aligned = 0
        for k in range(3):
          if i+k < len(grid) or j+k < len(grid):
            break
          if symbol != grid[i+k][j+k]:
            break
          num_aligned = num_aligned + 1
        if num_aligned == 3:
          if symbol == "X" or symbol == "O":
            return symbol
    return False
```

Same grading as before, except all divided by two (since this question is only on five points).

2 points to students who made an *honest* effort to answer the question regardless of whether it is cogent.