CIS 211
Spring 2019 Final Exam

Your name: Chill ScerpensKEY

Student ID#: 0o377010

Please do not write your name on other pages.

Unsolicited advice: Read the whole exam once before you start. Work the easy ones first. Think before you write. Draft and revise code on scratch paper before copying to the exam. If you get stuck on a problem, move on to others, then return to it. Take a deep slow breath.

Total: _____

1. [10 points] Consider the following program.

```
class Listenable:
    def __init__(self):
        self.listeners = []

    def add_listener(self, listener: "Listener"):
        self.listeners.append(listener)

    def notify_all(self, msg: str):
        for listener in self.listeners:
            listener.notify(self, msg)

class Tile(Listenable):
    def __init__(self, label: str):
        super().__init__()
        self.label = label

    def relabel(self, label: str):
        self.notify_all(f"New label {label}")
        self.label = label

class Listener:
    def notify(self, tile: Tile, msg: str):
        print(f"Tile {tile.label}: {msg}")

row = [ Tile("alpha"), Tile("beta"), Tile("gamma") ]
row[1].add_listener(Listener())

for tile in row:
    tile.relabel("Modified")
```

What does it print?

```
Tile beta: New label Modified
```

Note that this is the *only* thing the program prints. The point is that listeners are attached dynamically — since we attach the listener to only one of the tiles, the `notify_all` call causes the messsage to be printed only for that tile. Note also that `notify_all` is called *before* the value of the label is changed, and that the listener obtains the old label from the tile.

I took off only 1 point if you had the call on the wrong tile (but just one), 2 points if you treated the change of the label before the notification, but 6 points if your answer looked like a listener got called on all three tiles.

(score)

2. [10 points] Recall that in Python regular expressions, we use \(and \) to treat treat parentheses as ordinary chracters rather than forming groups. Consider the following program:

```
import re

pat = re.compile(r"""
 \s* for \s+
 (?P<iter_var> [a-zA-Z]\w*)
 \s+ in \s+
 (  ( range\(len\( (?P<seq_i> [a-zA-Z]\w*) \)\) )
  | (  (?P<seq_e> [a-zA-Z]\w*) )
 ) \s*
 : \s*
""", re.VERBOSE)

def check(s: str):
    match = pat.fullmatch(s)
    if match:
        parts = match.groupdict()
        if parts["seq_i"] is not None:
            print(f"{parts['iter_var']} indexes through {parts['seq_i']}")
        else:
            print(f"{parts['iter_var']} loops directly through {parts['seq_e']}")
    else:
        print(f"for loop pattern not recognized in {s}")

check("for i in li:")
check("for i in range(len(li)):")
check ("x += li[17]")
```

What does it print?

```
i loops directly through li
i indexes through li
for loop pattern not recognized in x += li[17]
```

I was surprised by the number of students who seemed to have trouble understanding how string interpolation works in f-strings. I allocated 4 points for each of the first two checks and 2 points for the last. There are too many variations on mistakes to list them all here. A few common errors with how much I took off:

- -3: Wrong capture on the second pattern
- -7: Had the write matches, but in the dark on on the groupdict and/or string interpolation
- -4: Got the captures right, but seems not to understand string interpolation (f-strings).
- -7: Just one line of output, but it's correct.
- -4: Two lines correct, incorrect match on the third
- -6: Printing the groupdict instead of the output of the actual print statement.

(score)

3. [10 points] Programmers often use hexadecimal or "hex" (base 16) to represent binary values because each hex digit represents exactly four binary digits. But hex is not the only reasonable choice. Some Unix programs have traditionally used octal (base 8), in which each octal digit represents three binary digits. Finish the following functions for converting between strings of octal digits and positive integers. For full credit, do not use integer addition, multiplication, division, or exponentiation.

```
OCTAL_DIGITS = "01234567"    # indexable, e.g., OCTAL_DIGITS[3] == "3"
OCTAL_VAL = { "0": 0, "1": 1, "2": 2, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7 }

def octal_to_int(oct: str) -> int:
    """Convert string of octal digits to positive integer.
    Examples: octal_to_int("377") == 255, octal_to_int("027") == 23
    Assume all characters in 'oct' are octal digits.
    """
    result = 0
    for digit in oct:
        result = (result << 3) | OCTAL_VAL[digit]
    return result


def int_to_octal(w: int) -> str:
    """Concert positive integer w to string of octal digits.
    Examples:   int_to_octal(255) == "377", int_to_octal(23) == "27"
    """
    result = ""
    while w > 0:
        low = w & 0b111
        low_dig = OCTAL_DIGITS[low]
        result = low_dig + result
        w = w >> 3
    return result


assert octal_to_int("027") == 23
assert int_to_octal(255) == "377"
```

Note that there is no multiplication or addition in this solution. It is almost identical to the hexadecimal conversions we worked out in lecture, except that it works 3 bits at a time instead of 4 bits at a time. You can get up to 6 points (out of 10) for solutions that do use integer arithmetic. I allocated 5 points to each function (up to 3 each for using arithmetic, or for approaches limited to three-digit octal numbers).

(score)

4. [10 points] Finish function `only_multiples` consistent with its docstring and the assertions at the bottom of this page.

```
from typing import List

def only_multiples(li: List[int]) -> List[int]:
    """Returns a list consisting of items in li
    that appear more than once, in the same order
    as they appeared in li.
    Example: only_multiples([1, 2, 3, 2, 1, 2, 4]) == [1, 2, 2, 1, 2]
             only_multiples([1, 2, 3, 4, 5]) == [ ]
    """
    counts = { }
    for i in li:
        if i not in counts:
            counts[i] = 0
        counts[i] += 1
    result = [ ]
    for i in li:
        if counts[i] > 1:
            result.append(i)
    return result

assert only_multiples([1, 2, 3, 2, 1, 2, 4]) == [1, 2, 2, 1, 2]
sample = [1, 2, 3, 4, 5]
assert only_multiples(sample) == [ ]
assert sample == [1, 2, 3, 4, 5]
```

Note that `only_multiples` does not change li. When we write a function that returns a value, that is an implicit specification that it does not change its inputs unless its docstring explicitly says it does.

There are other ways to keep track of which values appear multiple times, but I think all good solutions will be two-pass algorithms.

It's possible to write a rather bad solution that uses nested loops, with performance that is quadratic rather than linear in the size of the list. In retrospect I should have specified that only linear time algorithms are acceptable.

There were lots of variations (there always are), but among the most common errors were failing to return the first or last occurrence of an item (-5), nested loops that multiply duplicates (-7), using a dict in a way that will not preserve order (-5), finding only adjacent pairs (-7), or a good first pass followed by a second pass that makes no sense (-7).

(score)

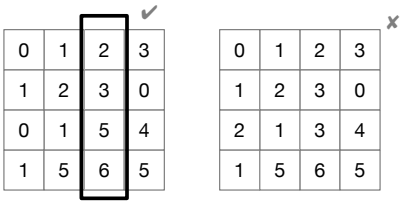
5. [10 points] Finish the `free_of` method of classes `Composite_Food` and `Simple_Food`.

```
from typing import List

class Food:
    def free_of(self, allergens: List[str]):
        """True if it does NOT contain any of the allergens"""
        raise NotImplementedError

class Composite_Food(Food):
    def __init__(self, ingredients: List[str]):
        self.ingredients = ingredients

    def free_of(self, allergens: List[str]) -> bool:
        for ingredient in self.ingredients:
            if not ingredient.free_of(allergens):
                return False
        return True

class Simple_Food(Food):
    def __init__(self, name: str):
        self.name = name

    def free_of(self, allergens: List[Food]) -> bool:
        return not self.name in allergens


# Example
crust = Composite_Food([Simple_Food("wheat"), Simple_Food("yeast"), Simple_Food("water")])
sauce = Composite_Food([Simple_Food("tomato"), Simple_Food("garlic"), Simple_Food("onion")])
topping = Simple_Food("cheese")
pizza = Composite_Food([crust, sauce, topping])

assert not pizza.free_of(["wheat", "barley"])
assert pizza.free_of(["peanuts", "cashews", "walnuts"])
```

I allocated 6 points to the recursive case (`Composite_Food`) and 4 to the base case (`Simple_Food`). Several students got full credit for one and none for the other. Common issues included not using the result from the recursive call (-4), not passing the right arguments in the recursive call (-4), "any" logic where "all" logic is needed in the recursive case (-3), missing the 'else' case in the base case (-2), or reversing sense in the base case (-2).

(score)

6. [10 points] Finish function `some_column_increasing`. For full credit, do not construct additional lists.



0	1	2	3
1	2	3	0
0	1	5	4
1	5	6	5

✔

0	1	2	3
1	2	3	0
2	1	3	4
1	5	6	5

✘

```
"""A sequence is *increasing* if each item is
greater than the item before it.  We find whether there
is at least one increasining column in a matrix,
represented as a list of lists.
"""
from typing import List

def some_col_increasing(m: List[List[int]]) -> bool:
    """True iff at least one column of the matrix
    is strictly increasing, e.g., [[1, 3], [0, 4], [8, 5]] has
    one column (the second) increasing.
    """
    if len(m) == 0:
        return False
    n_cols = len(m[0])
    for col in range(n_cols):
        prior = m[0][col] - 1
        increasing = True
        for row in m:
            if row[col] <= prior:
                increasing = False
            prior = row[col]
        if increasing:
            return True
    return False

assert some_col_increasing([[0, 1, 2, 3], [1, 2, 3, 0], [0, 1, 5, 4],[1, 5, 6, 5]])
assert not some_col_increasing([[0, 1, 2, 3],[1, 2, 3, 0],[2, 1, 3, 4],[1, 5, 6, 5]])
assert not some_col_increasing([])  # Vacuous case: No rows or columns, so none are increasing
assert not some_col_increasing([[]]) # Vacuous case: No columns, so none are increasing
```

I think the trickiest bit of this is the initialization of 'prior'. An alternative would be to set prior to `m[0][col]` and then run the loop starting with the second element. Another would be to compare pairs of adjacent element without keeping a `prior` value, being careful not to index an item before the first or after the last. You could break the inner loop out as a separate function.

I have a long, long list of mistakes, so I'll just list some of those that were both pretty severe and common. I took off 4 points for assuming the matrix was 4x4, 3 for forgetting to set the prior value variable in each loop iteration, 6 for prematurely returning False before all the columns have been checked, 3 for a break that returns True from the inner loop before it should, 3 for indexing [col][row] (which effectively makes it check rows rather than columns).

(score)