CIS 211

Winter 2019 Final Exam (Key)

Your name: Ayma Kee

Student ID#: 0xF34A97

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: Lots!

1. [10 points] Consider the following program.

```
from typing import List

class SummaryHook:
    """Abstract base class"""
    def __init__(self, initial: int = 0):
        self.summary = initial

    def get(self) -> int:
        return self.summary

    def visit(self, cell: int):
        raise NotImplementedError("Oops")


class Summarizable:
    def __init__(self, elements: List[int] = []):
        self.elements = elements

    def foreach(self, hook: SummaryHook) -> int:
        for cell in self.elements:
            hook.visit(cell)
        return hook.get()

class Sm(SummaryHook):
    def visit(self, cell: int):
        self.summary += cell

class Mx(SummaryHook):
    def visit(self, cell: int):
        if cell > self.summary:
            self.summary = cell

s = Summarizable([10, 30, 20])

summary = s.foreach(Sm())
print(f"Summarized by Sm to {summary}")

summary = s.foreach(Mx())
print(f"Summarized by Mx to {summary}")
```

What does it print?

```
Summarized by Sm to 60
Summarized by Mx to 30
```


(score)

2. [10 points] Recall that in Python regular expressions, "." is the wild-card that matches any character and "\s" matches a whitespace character. We use \(and \) to treat treat parentheses as ordinary chracters rather than forming groups. Consider the following program:

```
import re
from typing import Pattern

pat = re.compile(r"""
    \s* def \s*
    (?P<f> [a-zA-Z_]+)  \s*
    \(  (?P<a> .* )  \) \s*
    ( -> \s*  (?P<r> .+) )?   \s*
    : \s*
    """, re.VERBOSE)

def check(p: Pattern, s: str):
    """Check how p matches a string"""
    match = pat.fullmatch(s)
    if not match:
        print(f"No match of '{pat.pattern}' on {s}")
    else:
        d = match.groupdict()
        print(f"Matched {d}")

check(pat, "def add(x: int, y: int) -> int:")
check(pat, "def shout():")
```

What does it print?

```
Matched {'f': 'add', 'a': 'x: int, y: int', 'r': 'int'}
Matched {'f': 'shout', 'a': '', 'r': None}
```


(score)

3. [10 points] Sometimes we compactly encode a path such as movement of a pen or movement of a monster in a video game as a series of small moves in a grid. In this problem, we will pack one of 8 compass directions into bits 5..7 of a byte and use the remaining bits 0..4 to record a distance of up to 31 units. Finish the three functions so that they correctly pack and extract the information. Recall that you can convert a Direction `d` to an integer as `d.value` and you can convert an integer `i` in the range 0..7 to a Direction as `Direction(i)`. You may assume that values provided to `to_dir` and `to_dist` were produced by `encode`.

```
from enum import Enum

class Direction(Enum):
    North = 0
    NorthEast = 1
    East = 2
    SouthEast = 3
    South = 4
    SouthWest = 5
    West = 6
    NorthWest = 7

def encode(dir: Direction, steps: int) -> int:
    """Return 8-bit encoding of move"""
    return (dir.value << 5) | (steps & 0b11111)

def to_dir(word: int) -> Direction:
    """Extract direction value from encoded mvement"""
    return Direction(word >> 5)

def to_steps(word: int) -> int:
    """Extract number of steps from movement"""
    return word & 0b11111

east13 = encode(Direction.East, 13)
assert to_dir(east13) == Direction.East
assert to_steps(east13) == 13
```


(score)

4. [10 points] Professor X gives several quizzes in a term. If a student misses a quiz, a score of 0.0 is recorded. However, if a student misses up to three quizzes, those scores will be replaced with the average of the other quiz scores. If a student misses more than 3 quizzes, *none* of the missing quiz scores will be replaced. Canvas doesn't support this grading method ... help Professor X by writing a Python function to replace the missing scores.

```
from typing import List

def replace_missing_quizzes(quizzes: List[float]):
    """A missing quiz is recorded as 0.0 (and
    no other quizzes have score 0.0).
    If more than 3 quizzes are missing, no
    scores are replaced.  If 3 or fewer quizzes
    are missing, those quiz scores are replaced by
    the average of the other quiz scores.
    """
    sum = 0.0
    present_count = 0
    missing_count = 0
    for quiz in quizzes:
        if quiz == 0.0:
            missing_count += 1
        else:
            sum += quiz
            present_count += 1
    if missing_count > 3:
        return
    average = sum / present_count
    for i in range(len((quizzes))):
        if quizzes[i] == 0.0:
            quizzes[i] = average
    return

scores = [15.0, 0.0, 3.0, 12.0]
replace_missing_quizzes(scores)
assert scores == [15.0, 10.0, 3.0, 12.0 ]

scores = [15.0, 0.0, 0.0, 3.0, 12.0, 0.0, 7.0, 0.0]
replace_missing_quizzes(scores)
assert scores == [15.0, 0.0, 0.0, 3.0, 12.0, 0.0, 7.0, 0.0]
```


(score)

5. [10 points] Finish the Inner and Leaf classes below so that `t.sum_in_range(min, max)` returns the sum of the leaves whose values are between *min* and *max* inclusive.

```
class Tree:

    def sum_in_range(self, min_val: int, max_val: int) -> int:
        """Sum of leaf values in range min_val .. max_val"""
        raise NotImplementedError("Hey!  You forgot!")


class Leaf(Tree):

    def __init__(self, v: int):
        self.value = v

    def sum_in_range(self, min_val: int, max_val: int) -> int:
        """Sum of leaf values in range min_val .. max_val"""
        if self.value < min_val or self.value > max_val:
            return 0
        else:
            return self.value


class Inner(Tree):

    def __init__(self, left: Tree, right: Tree):
        self.left = left
        self.right = right

    def sum_in_range(self, min_val: int, max_val: int) -> int:
        """Sum of leaf values in range min_val .. max_val"""
        return self.left.sum_in_range(min_val, max_val) + self.right.sum_in_range(min_val, m



t = Inner(Leaf(5), Inner( Inner(Leaf(8), Leaf(3)),  Leaf(6)))

assert t.sum_in_range(4,7) == 11
assert t.sum_in_range(-3, -5) == 0
assert t.sum_in_range(0,10) == 22
```


(score)

6. [10 points] Finish the following function:

```
def some_col_all_pos(m: List[List[int]]) -> bool:
    """There is some column in m, a rectangular matrix of integers,
    in which all the entries are greater than zero.
    Examples: some_col_all_pos([]) == False  (no columns)
              some_col_all_pos([[]]) == False (no columns)
              some_col_all_pos([[42]]) == True ([42] is all positive)
              some_col_all_pos([[0]])) == False
              some_col_all_pos([[1, 0], [2, -3]]) == True ([1,2] is all positive)
              some_col_all_pos([[0, 0, 1, 1],
                                [1, 2, 3, 4],
                                [8, 7, 1, 0]]) == True ([1, 3, 1] is all positive)
    """
    if len(m) == 0:
        return False
    n_cols = len(m[0])
    for col_index in range(n_cols):
        all_pos = True
        for row in m:
            if row[col_index] <= 0:
                all_pos = False
        if all_pos:
            return True
    return False

assert not some_column_increasing([])
assert not some_col_all_pos([[]])
assert some_col_all_pos([[42]])
assert not some_col_all_pos([[0]])
assert some_col_all_pos([[1, 0], [2, -3]])
assert some_col_all_pos([[0, 0, 1, 1],
                         [1, 2, 3, 4],
                         [8, 7, 1, 0]])

assert not some_col_all_pos([[0, 0, 1, 1],
                             [1, 2, 0, 4],
                             [8, 7, 1, 0]])
```


(score)