H2 Computing Practical Worksheet Review – T1W4 (Part 2)

1d Assume that you are to store the score information for multiple subjects. Initially you are told to store these in a multi-dimensional array, such that it follows the form:

`scores[student_id][sub1_score][sub2_score]…[subn_score]`

However, it is then pointed out that the primary purpose of this score data is to determine all the students that do not satisfy a certain threshold score.

As such, you instead propose the creation of n Binary Search Trees. Each node in each tree is to hold:

- `student_ID` in the form "YXXX", where Y is an English letter and X is a digit
- `score` for Subject k (where 1 ≤ k ≤ n) and where score ranges from 0.0 to 100.0

Each tree should contain the following methods:

- **initialisation(STRING)**
  - o reads and stores the data from a text file (name given as the STRING input) – i.e., the student_ids and scores corresponding to the 1 subject linked to this tree
- **insert(STRING, FLOAT)**
  - o stores the given student_id (STRING) and score (FLOAT) in the tree
  - o the initialisation method should utilised this method
- **inorder_traversal(FLOAT): ARRAY<(STRING, FLOAT)>**
  - o performs a special inorder where only scores below the input value (FLOAT) are returned as an array in the form
- **get_all_data(): ARRAY<(STRING, FLOAT)>**
  - o returns an array of all the student_id and score data for the subject in question using inorder traversal
  - o this array should be sorted in ascending order of score (i.e., lowest to highest that are under the threshold)
  - o this should correspond to an array of student_id (STRING) and score (FLOAT) tuples this method should utilise **inorder_traversal(FLOAT): ARRAY<(STRING, FLOAT)>**
- **get_weak_scores(FLOAT): ARRAY<(STRING, FLOAT)>**
  - o prints all the student_ids and scores that are under the given threshold (FLOAT)
  - o this array should be sorted in ascending order of score (i.e., lowest to highest that are under the threshold)
  - o this should correspond to an array of student_id (STRING) and score (FLOAT) tuples this method should utilise **inorder_traversal(FLOAT): ARRAY<(STRING, FLOAT)>**

A. Discuss the following implementation and its design. In particular, please comment on the use of globals, and the methods used to perform the traversals.

```
class BSTNode():
    def __init__(self, student_ID, score):
        self._student_ID = student_ID
        self._score = score
        self._left = None
        self._right = None

    def get_student_ID(self):
        return self._student_ID

    def set_student_ID(self, new_student_ID):
        self._student_ID = new_student_ID

    def get_score(self):
        return self._score

    def set_score(self, new_score):
        self._score = new_score
    def get_left(self):
        return self._left

    def set_left(self, new_left):
        self._left = new_left

    def get_right(self):
        return self._right

    def set_right(self, new_right):
        self._right = new_right

class BST():
    def __init__(self, file):
        self._root = None
        f = open(file, "r")
        for line in f:
            to_insert = line.strip().split(",")
            self.insert(to_insert[0], float(to_insert[1]))
        f.close()

    def insert(self, student_ID, score):
        if self._root == None:
            self._root = BSTNode(student_ID, score)
        else:
            cur = self._root
            while True:
                if score < cur.get_score():
                    if cur.get_left() == None:
                        cur.set_left(BSTNode(student_ID, score))
                        break
                    cur = cur.get_left()
                else:
                    if cur.get_right() == None:
                        cur.set_right(BSTNode(student_ID, score))
                        break
                    cur = cur.get_right()

    def inorder_traversal(self, score = 101):
        global result
        result = []
        if self._root == None:
            return result
        elif self._root.get_score() >= score:
            if self._root.get_left() == None:
                return []
            self._inorder_helper(self._root.get_left(), score)
        else:
            self._inorder_helper(self._root, score)
        temp = []
        for node in result:
            temp.append([node.get_student_ID(), node.get_score()])
        return temp

    def _inorder_helper(self, bstnode, score):
        global result
        if bstnode != None:
            self._inorder_helper(bstnode.get_left(), score)
            if bstnode.get_score() < score:
                result.append(bstnode)
            self._inorder_helper(bstnode.get_right(), score)

    def get_all_data(self):
        return self.inorder_traversal()

    def get_weak_scores(self, score):
        return self.inorder_traversal(score)
```

B. Only 1 of the 4 sets code submitted perform tree traversals via the Node class, the 3 others all perform the traversals via a helper method in the BST class. Comment on which is more appropriate and why you believe this is the case.

1e Write the code to:

- Generate 4 text files containing the score data of the same 50 students (i.e., the students generated for each of the 4 files should have the same student_ids). Use a uniform distribution to generate the individual scores.
- Initialise the required Binary Search Trees (based on your implementation in 1d) to store the data in those text files
- Using thresholds of 40, generate output (to screen – i.e., print) for each Binary Search Tree based on:
  - **get_all_data(): ARRAY<(STRING, FLOAT)>**
  - **get_weak_scores(FLOAT): ARRAY<(STRING, FLOAT)>**

A. What is the issue with the following?

```
def gen_ID():
    f = open("STUDENT_ID.TXT", "w")
    student_id_list = []
    for i in range(50):
        while True:
            student_id = chr(random.randint(65, 90))
            student_id += str(random.randint(0,100)).zfill(3)
            if student_id not in student_id_list:
                student_id_list.append(student_id)
                break
    for id_entry in student_id_list:
        f.write(id_entry + "\n")
    f.close()

def gen_score(file):
    f1 = open("STUDENT_ID.TXT", "r")
    f2 = open(file, "w")
    for id_entry in f1:
        f2.write(id_entry.strip() + ", " + str(random.randint(0, 1000) / 10) + "\n")
    f1.close()
    f2.close()
```

1f Write code to print out the list of students who did not satisfy their respective subject thresholds, but this time sorted by the number of subjects whose thresholds they failed to satisfy.

A. Review the following code and discuss its design and applicability.

```
physics = BST("PHYSICS.TXT")
print(physics.get_all_data(), end = "\n\n")
physics_failures = physics.get_weak_scores(40)
print(physics_failures, end = "\n\n")

math = BST("MATH.TXT")
print(math.get_all_data(), end = "\n\n")
math_failures = math.get_weak_scores(40)
print(math_failures, end = "\n\n")

econs = BST("ECONS.TXT")
print(econs.get_all_data(), end = "\n\n")
econs_failures = econs.get_weak_scores(40)
print(econs_failures, end = "\n\n")

gp = BST("GP.TXT")
print(gp.get_all_data(), end = "\n\n")
gp_failures = gp.get_weak_scores(40)
print(gp_failures, end = "\n\n")

def compile_failures(subject_failures, subject_id, compiled_list = []):
    #subject_id: 1 for physics, 2 for math, 3 for econs, 4 for gp
    for student in subject_failures:
        for i in range(len(compiled_list)):
            if student[0] == compiled_list[i][0]:
                compiled_list[i][subject_id] = student[1]
                break
        else:
            compiled_list.append([student[0], None, None, None, None])
            compiled_list[-1][subject_id] = student[1]
    return compiled_list

p = compile_failures(physics_failures, 1)
pm = compile_failures(math_failures, 2, p)
pme = compile_failures(econs_failures, 3, pm)
pmeg = compile_failures(gp_failures, 4, pme)

def num_fails(compiled_list):
    for i in range(len(compiled_list)):
        fails = 0
        for j in range(1, 5):
            if compiled_list[i][j] != None:
                fails += 1
        if fails == 0:
            print('ERROR')
            break
        compiled_list[i].append(fails)

num_fails(pmeg)

def sort_num_fails(compiled_list):
    for i in range(1, len(compiled_list)):
        while i >= 1 and compiled_list[i-1][-1] < compiled_list[i][-1]:
            compiled_list[i-1], compiled_list[i] = compiled_list[i],
compiled_list[i-1]
            i -= 1

def subject_num_fails(subject_id, compiled_list):
    sort_num_fails(compiled_list)
    to_return = []
    for student in compiled_list:
        if student[subject_id] != None:
            to_return.append([student[0], student[subject_id]])
    return to_return

for i in range(1,5):
  print(subject_num_fails(i, pmeg), end = "\n\n")
```

Other general programming practice issues:

A. Review the following code and determine if there are any issues?

```
def write_file(filename,student_ids):
      task_a()
      file = open('SCORES.TXT')
      data = []
      for i in file:
            data.append(i.strip())
      file.close()
      file = open(filename,'w')
      for i in range(50):
            file.write(student_ids[i] + ',' + data[i] + '\n')
      file.close()
```

B. Comment on the following methods in terms of encapsulation.

```
def inorder_traversal(self, cutoff):
    return list(self._inorder(self._root, cutoff))

def _inorder(self, tree, cutoff):
    if tree == None:
        pass
    else:
        for x in self._inorder(tree.get_left(), cutoff):
            yield x
        if tree.get_score() < cutoff:
            yield (tree.get_id(), tree.get_score())
        for x in self._inorder(tree.get_right(), cutoff):
            yield x
```