CS 1110 Prelim 2 Solutions, April 2023

1. [8 points] **Iteration over lists.** Implement this function, making effective use of for-loops.

```
def floor_divide_lists(numers, denoms):
    """Performs floor division across two lists of ints.
    Returns a new list. Each element of this new list is the string of the
    floor division result of dividing each int in the numers (numerators)
    list by its corresponding int in the denoms (denominators) list.

    Recall that floor division -- `//` -- is an integer operation that
    yields the integer that is less than or equal to the standard
    division -- '/' -- result.
    Examples: 1//2 --> 0, 2//2 --> 1, 3//2 --> 1, 4//2 --> 2

    Note:
      - The new list should be a list of STRINGS; each string is the
        string representation of the int resulting from the floor division
      - In case of dividing by 0, put 'E' (for 'Error') at the corresponding
        position in the resulting list.

    Return: a (possibly empty) list of str

              Example 1                Example 2                Example 3
       numers is [0,  12,  4]     numers is [9,   1,   6]    numers is []
       denoms is [7,   3,  3]     denoms is [0,   3,   3]    denoms is []
       return   ['0', '4','1']    return   ['E', '0', '2']   return    []

    Preconditions:
       numers: a (possibly empty) list of int
       denoms: a (possibly empty) list of int
       len(numers) == len(denoms), length can be 0                    """
    # STUDENTS: assume preconditions are met. No need to assert them.

    result = []
    for i in range(len(numers)):
        if denoms[i] != 0:
            result.append(str(numers[i] // denoms[i]))
        else:
            result.append('E')
    return result
```

2. [12 points] **Iteration over dictionaries.** Implement this function, using for-loops effectively.

```
def invert_dict(input_dict):
    """ Given:
          `input_dict` with the following properties:
          - the keys are characters a-z and ' '
          - values are ints

    Returns: the inverse of input_dict
    a new dictionary with the following properties:
    - the keys are the int values found in `input_dict`
    - the values are lists of letters that correspond to that int in `input_dict`
      (the letters in the lists can be in any order)

    EXAMPLES:
    {'a': 2, 'b': 1, 'c': 3, ' ': 5} → {2: ['a'], 1: ['b'], 3: ['c'], 5: [' ']}
    {'a': 2, 'b': 2, 'c': 3, ' ': 5} → {2: ['a', 'b'], 3: ['c'], 5: [' ']}
    {'a': 2, 'b': 2, 'c': 2, ' ': 2} → {2: ['a', 'b', 'c', ' ']}
    {} → {}

    Precondition: input_dict is a possibly empty dictionary that will only have:
    - characters a-z or space as keys
    - ints as values
    """
    # STUDENTS: assume preconditions are met. No need to assert them.

    newdict = {}
    for key in input_dict: # or input_dict.keys() (see reference sheet)
        val = input_dict[key]
        if val not in newdict:
            newdict[val] = [key]
        else:
            newdict[val].append(key)
    return newdict
```

3. **Lists and indexing** The variable `mylist` is successfully initialized as follows:

```
mylist = [ [ [ [ 1 ], 1, 2, 3] ], [2, 5, 6], [7] ]
```

After this initialization, what do the following Python expressions evaluate to? Write ERROR if evaluating the expression causes Python to throw an error. Assume each expression is evaluated independently.

(a) [2 points] `mylist[0][0].index(1)` evaluates to ☐

Correct answer: 1

(b) [2 points] `mylist[1][2]` evaluates to ☐

Correct answer: 6

(c) [2 points] `mylist[1][1][1]` evaluates to ☐

Correct answer: Error

4. [14 points] **Recursion.** Let `Worker` be a class whose objects have the following two attributes:

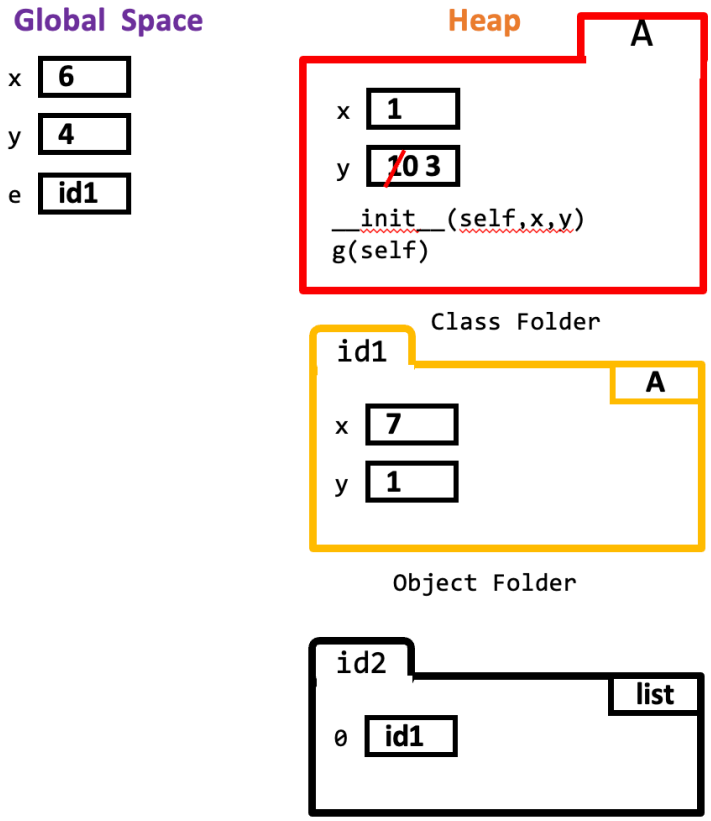
```
name    [str] - unique non-empty name of worker
reports [list of Worker] - (possibly empty) workers who report directly
    to this worker instance, who acts as the manager for these reports.
```

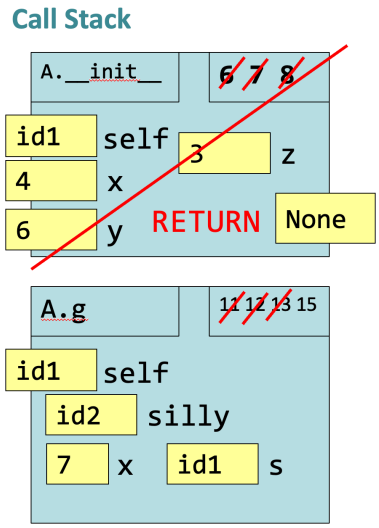
A Implement the following **function** (not a class method), making effective use of recursion. For-loops are allowed as long as your solution is fundamentally recursive.

```
def calculate_salary(w):
    """Returns: the salary of worker `w` where workers make $50,000 by default,
    and managers make $10,000 more than the best paid worker that reports to them.

    Parameter w: the worker whose salary is to be calculated
    Precondition: w is an Worker object

    Example:                                Carlos
                                            /   \
    w1 = Worker("Ricky", [])              Rene  Draco
    w2 = Worker("Johnny", [w1])            |
    w3 = Worker("Rene", [w2])           Johnny
    w4 = Worker("Draco", [])               |
    w5 = Worker("Carlos", [w3, w4])     Ricky

    calculate_salary(w1) and calculate_salary(w4) Returns 50000
        because these 2 workers do not manage anyone (they have no reports)
    calculate_salary(w2) Returns 60000
    calculate_salary(w3) Returns 70000
    calculate_salary(w5) Returns 80000
        because Rene is the best paid of Carlos' 2 reports,
        so Carlos makes 10000 more than Rene                              """
    # STUDENTS: assume preconditions are met. No need to assert them.

    if len(w.reports) == 0: # or if not w.reports:
        return 50000
    most = 0
    for r in w.reports:
        r_sal = calculate_salary(r)
        if r_sal > most:
            most = r_sal
    return most + 10000
```

5. [20 points] **Simulating Python Execution.** Diagram the execution of each line of code until Python reaches the comment `# STOP HERE` which would stop the code in the middle of the first iteration of the for loop. Draw the memory diagram as seen in class and Assignment 5. (As usual, do not draw the objects/folders for imported modules or for function definitions.) Include global variables, class folders, object folders and call frames. Pay attention to what goes in the **Global Space** / **Call Stack** / **Heap**. If a value changes, cross out the old value so that it remains legible. (Do not erase the old value.) This code runs without error. We have provided a skeleton of a Class and Object folder for you to use. Please use them.

```
1  class A:
2      x = 1
3      y = 10
4
5      def __init__(self, x, y):
6          self.x = y + 1
7          A.y = x - 1
8          z = self.y
9
10     def g(self):
11         silly = [self]
12         for s in silly:
13             x = s.x
14             # STOP HERE
15             s.x = y
16         return s.y
17 x = 6
18 y = 4
19 e = A(y,x)
20 e.y = A.x
21 w = e.g()
```





6. **Classes.** Consider the following class, defined as follows:

```
class LapTime:
    """Represents a runner's time for a single lap around a track.
    Each runner has at most one LapTime object for each lap they complete.

    Attributes:  name: [string] the name of the runner
                 lap:  [int] >= 0, representing the lap number
                 time: [float] >= 0.0, time taken to complete the lap in seconds
    """
    def __init__(self, name, lap_num, time):
        """Initializes a LapTime object
           Preconditions: name is a string,
                          lap_num is an int >= 0
                          time is a float >= 0.0                              """
        self.name = name
        self.lap = lap_num
        self.time = time
```

(a) [5 points] Override the following special method of `LapTime` according to its specification.

```
    def __lt__(self, other):
        """Compares two LapTimes of the same lap around the track
        (LapTimes of the same lap have the same value for their lap attribute)

        Returns True if this LapTime is less than (i.e., faster than) the other
        LapTime or if the other LapTime is None. Otherwise returns False.

        Precondition: other is either a LapTime object representing the same
        lap around the track as as self or None                              """
        # STUDENTS: assume preconditions are met. No need to assert them.


        if other is None:
            return True
        return self.time < other.time:
```

(b) [5 points] Complete the following test which asserts a case for which your implementation of `__lt__` should return `True`.

```
t1 = LapTime(_____________________________________________)

t2 = LapTime(_____________________________________________)

expected = True
assert_equals(expected, ___________________________________)
t1 = LapTime("Zola", 8, 10.4)
t2 = LapTime("Mary", 8, 12.9)
expected = True
assert_equals(expected, t1 < t2 )
```

(Classes, cont'd) Now we introduce a second class, defined as follows:

```
class Tracker:
    """Records the best lap times of runners in a training session.
    Attributes:
    runner_times: dictionary whose keys are names of runners (string) and
                  values are lists of LapTime objects. Each LapTime object
                  corresponds to the runner's performance for a certain lap.
    best_times: a list with one entry per lap. The element at index i contains
                either None or the LapTime object of the best performance for
                Lap i (across all runners) that have been added to the Tracker.
    """
    def __init__(self, max_laps):
        """Initializes Tracker object self.
              runner_times: set to an empty dictionary
              best_times: a list of length max_laps, containing all Nones

           Precondition: max_laps is an int > 0                              """
        self.runner_times = {}
        self.best_times = []
        for i in range(max_laps):
            self.best_times.append(None)
```

(c) [12 points] Override the following class method of `Tracker` according to its specification.

```
    def add_laptime(self, t):
        """Modifies this Tracker given a LapTime object t

        Adds t to the end of the corresponding runner's list in
        runner_times, or creates a new entry and list if necessary.
        If t represents a faster performance for its lap than the current
        entry for that lap in best_times, t should be added to best_times
        at the appropriate index

        Precondition: t is a LapTime object                                  """
        # STUDENTS: assume preconditions are met. No need to assert them.

        if t.name in self.runner_times:
            self.runner_times[t.name].append(t)
        else:
            self.runner_times[t.name] = [t]
        if t < self.best_times[t.lap]:
            self.best_times[t.lap] = t
```

7. **Debugging.** Consider the following two classes and 3 lines of code that use them:

```
class Course:
    def __init__(self, name, n_credit):
        name = name
        n_credit = n_credit

class Student:
    max_credit = 20

    def __init__(self, netID, courses=None):
        self.netID = netID
        self.courses = courses if courses is not None else []
        # ^^ this line has no error and is also not important to the question
        for one_course in self.courses:
            self.n_credit += one_course.n_credit # Add up all the credits

    def enroll(new_course):
        if new_course.n_credit + self.max_credit <= Student.n_credit:
            self.courses.append(new_course)
            self.n_credit += new_course.n_credit

c1 = Course("CS 1110", 4)
s1 = Student("mep1")
s1.enroll(c1)
```

When the given code is run in Python, the following error is reported:

```
Traceback (most recent call last):
  File "college.py", line 23, in <module>
    s1.enroll(c1)
TypeError: enroll() takes 1 positional argument but 2 were given
```

(a) [2 points] What are the 2 positional arguments that were given?

first argument:

second argument:

first arg: s1, or the id of the Student

second arg: c1, or the id of the course

(b) [2 points] Fix the code to remove only the above error. **Fix only the code that directly leads to the above error message.** Mark your fix(es) with the label **FIX1**.

Line 16 should be changed to `def enroll(self, new_course)`

Now that you have fixed the error, you rerun the code and now a new error is reported:

```
Traceback (most recent call last):
  File "college.py", line 23, in <module>
    s1.enroll(c1)
  File "college.py", line 17, in enroll
    if new_course.n_credit + self.max_credit <= Student.n_credit:
AttributeError: 'Course' object has no attribute 'n_credit'
```

(c) [2 points] Fix the above code to remove only this new error. **Fix only the code that directly leads to the new error message.** Mark your fix(es) with the label **FIX2**.

Line 4 should be changed to `self.n_credit = n_credit`

(d) [2 points] Will your fixed code now run to completion without any reported Python errors?

Circle One: Yes No

If you answered No, circle or underline the source(s) of the next error that Python will report in the code. Label this as **ERR3**.

No, the next error will be from `Student.n_credit` in the same line (line 17).