CS 1110 Prelim 2 Solutions, April 2022

1. [8 points] **What's the point?** Imagine a word-based game like Scrabble where:

• Global variable `points` is a dictionary whose keys are letters and values are the points earned for using that letter:

```
points = {'a':1, 'b':3, 'c':3, 'd':2, 'e':1, ... , 'w':4, 'x':8, 'y':4, 'z':10}
```

• Words get placed on a board such that some of the word's individual letters might lie on places that earn a bonus of double or triple points.

For a given word, bonus multipliers for the word's letters are stored in a list `mults`, each entry of which is either 1 (no change), 2 (double the score), or 3 (triple the score).

• A word's score is the sum of each of its individual letter's scores after any bonus multipliers.

Examples: From dictionary `points`, we know the following point values: 'e': 1 and 'w': 4.

word	mults	score
"eww"	[1, 1, 1]	$1 \times 1 + 4 \times 1 + 4 \times 1 = 9$
"eww"	[1, 2, 3]	$1\times1 + 4\times2 + 4\times3 = 21$
"we"	[2, 3]	$4\times2 + 1\times3 = 11$

Implement the following function.

```
def score_word(word, mults):
    """ Given `word` & its letter multipliers `mults`, returns word's score, an int

    Precondition (no need to assert):
        word [str]:  contains only lowercase letters, length >= 1.
        mults:  list of ints with same length as `word`.
                Each entry is either 1, 2, or 3.

        `points` is a dictionary in **global space** (not a parameter of this
        function) as described in the problem text. """

    score = 0
    for i in range(len(word)):
        pt = points[word[i]]
        score += (pt * mults[i])
    return score
```

2. [12 points] **We need a holiday!** Implement the following function, using for-loops effectively.

```
def num_holidays(holiday_list):
    """Returns the number of days off, given a non-empty list of holidays, holiday_list
       that has no duplicate holidays and no overlapping holidays

    A holiday is a list of 2-3 items:
        * a non-empty string, the name of the holiday
        * a start date
        * an optional end date (if the holiday lasts longer than 1 day).
          This is the last day the holiday is celebrated.
    A date is a list with 2 items:
        * a non-empty string, the month
        * an int, the day of the month (assume valid number for the month)

    You may assume that all holidays start and end in the same month.

    Examples:
        SU22 = [["Juneteenth",  ["Jun", 20]]]                # 1 day holiday
        num_holidays(SU22) --> returns 1

        FA21 = [["Labor Day",  ["Sep", 6]],                  # 1 day holiday
                ["Fall Break", ["Oct", 9], ["Oct", 12]],     # 4 day holiday
                ["Thanksgiving", ["Nov", 24], ["Nov", 28]]   # 5 day holiday
        num_holidays(FA21) --> returns 10
    """

    n_holidays = 0
    for holiday in holiday_list:
        if len(holiday) == 2:
            n_holidays +=1
        else:
            start = holiday[1][1]
            stop = holiday[2][1]
            n_holidays += stop - start + 1
    return n_holidays
```

3. **Class it up!** In the previous question, dates were represented as lists. Now let's represent them using classes.

   (a) [2 points] In the code below, insert python code that creates the class attribute MAX_DAYS.

   (b) [6 points] In the code below, insert python code that completes Date's `__init__()` method.

```
class Date:
    """Objects represent an instance of a Date.

    Class attributes:
         MAX_DAYS: 31, the maximum number of days that any month can have

    Instance attributes:
          month [str]: 3-character, uppercase abbreviation of the month
          day [int]: the day of the month, 0 < day <= MAX_DAYS for a Date   """

    MAX_DAYS = 31

    def __init__(self, m, d):
        """ Creates a new Date with attributes set as follows:
                month: the first 3 characters of m, uppercase
                day: set to d, **OR** the max legal value if d is too large

        Preconditions:  (STUDENTS: don't assert them)
            m: a str with len >= 3
            d: an int, 0 < d        """

        self.month = m[0:3].upper()

        if d > Date.MAX_DAYS:
            d = Date.MAX_DAYS
        self.day = d
        # alternate version of the above:
        # self.day = min(d, Date.MAX_DAYS)
```

(c) [2 points] Given the Date class as it is defined on the previous page, what is the value of x after executing the following code?

```
d1 = Date("August", 12)
d2 = Date("August", 12)
x = (d1 == d2)
```

Circle One: True False Neither*

*because an Error occurs before x is given a value

Correct Answer: **False**

(d) [4 points] Override the following special method of class Date according to its specification.

```
def __eq__(self, other):
    """ Returns: True if the month and day of the Dates are equal,
    False o.w.
    Precondition (no need to assert):   other is a Date.      """

    return self.month == other.month and self.day == other.day
```

(e) [2 points] The precondition above does not state that self needs to be a Date. Does asking Python to evaluate the expression "annoying string" == Date("Feb", 29) cause the Date __eq__() method to be called with a value of self or other not being a Date? Explain your answer. (Credit given only for correct explanation — an answer of just "Yes" or "No" will not receive points.)

Given what we have learned in CS1110 so far, how we would reason is as follows. Date's __eq__() is only called when you have an expression like d1 == d2 where d1 — the item on the left-hand side of the double-equals — is a Date. If d1 isn't a Date, then a different __eq__() method is called. (In the given example, it would be the __eq__() method for strings.[1])

Notes:

- The instance attributes day and month are not relevant to the question.
- An __eq__() method can be invoked even when the items on the two sides of the == are of different types.
- What Date's __eq__() evaluates to is not relevant to the question.
- Python does not require the value of self in a method call to be an object of the method's class; it just typically doesn't cause that to happen.

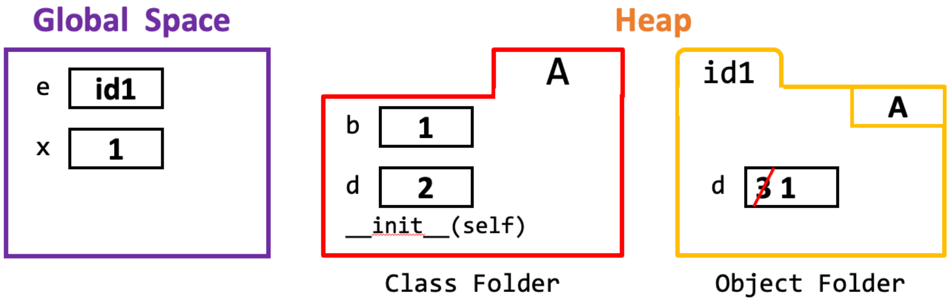
[1] Extremely technical details which are completely beyond the scope of CS1110: the str __eq__()__ will return NotImplemented because the writers of the string class decided that's what happens when the "other" object is a non-string. This NotImplemented value causes Python to then try using the __eq__() method for the right-hand object applied to the left-hand object, and then an error will occur, because strings don't have month or day attributes. This complex situation is why we are only grading this question on reasoning, not on the yes/no answer a student supplies. See https://stackoverflow.com/questions/2281222/why-when-in-python-does-x-y-call-y-eq-x.

4. [20 points] **A Picture is worth a thousand words ...and ~~not 4~~ 5 points.** Diagram the execution of each of the following code snippets. Include global variables, object folders and class folders, but **omit** call frames.

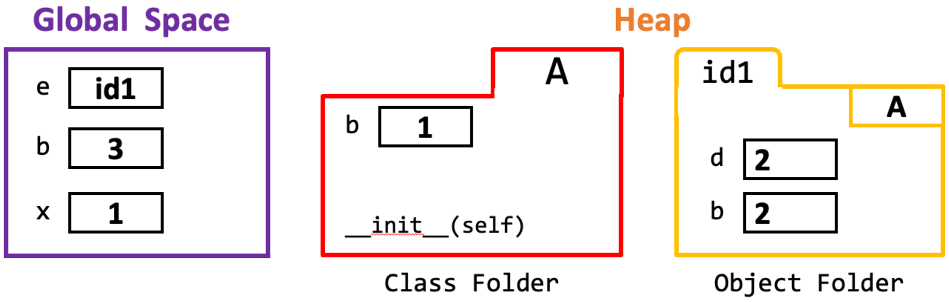
If the code changes a value, write in the old value and then cross it out. (Don't just erase.)

If an error occurs, diagram all variable/attribute changes that occur before the error occurs, and then write "ERROR" in large letters in the box containing the code.

```
class A:
    b = 1
    d = 2
    def __init__(self):
        self.d = 3

e = A()
e.d = e.b
x = e.d
```

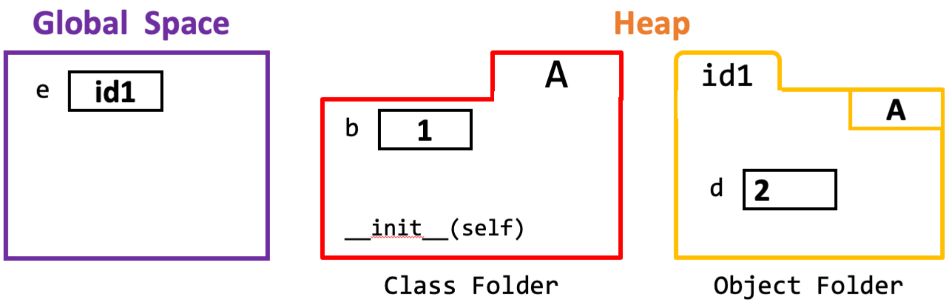


```
class A:
    b = 1
    def __init__(self):
        self.d = 2

e = A()
b = 3
e.b = e.d
x = A.b
```

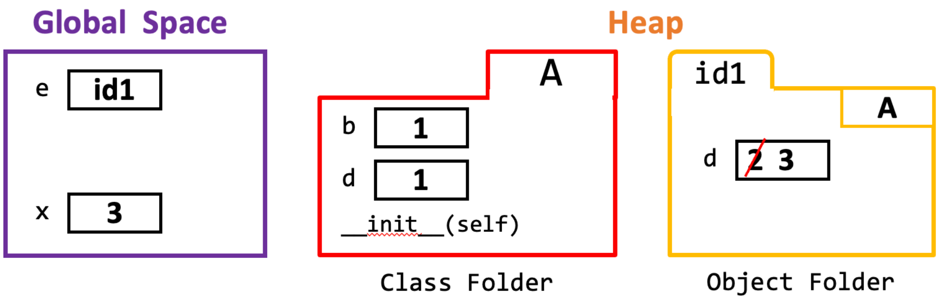


```
class A:
    b = 1
    def __init__(self):
        self.d = 2

e = A()
A.b = A.d
A.d = 3
x = A.b
```

An error occurs for line
A.b = A.d



```
class A:
    b = 1
    def __init__(self):
        self.d = 2

e = A()
e.d = 3
A.d = A.b
x = e.d
```



5. **Where's Waldo?** This question involves a `Person` class with 2 instance attributes:

    - `name [str]`
    - `parents [list of Persons]`, possibly empty

    You may assume that no person appears twice in a family tree.

    The function below is buggy. It does **not** accomplish the task in its specification.

```
def find_waldo_broken(p):
    """  Returns:
            True  if any ancestor of p (including p) has the name "Waldo"
            False if no ancestor of p (including p) has the name "Waldo"
        Precondition (no need to assert): p is a person
    """
    if p.name == "Waldo":
        return True
```

```
9       found = False
10      for parent in p.parents:
11          found = find_waldo_broken(parent)
12      return found
13
```

(a) [2 points] **Identify the problem.** Describe the problem with the above implementation:

(A) Y'all are wrong. This function works according to its specification!

(B) This function always returns `False`.

(C) This function always returns `True`.

(D) This function sometimes returns `True` when `p` has no a family member named "Waldo".

(E) This function sometimes returns `False` when `p` has a family member named "Waldo".

(F) The function code could throw an error, even when the preconditions are met.

(G) The function could run forever.

Circle One: A B C D E F G

Correct Answer: **E**

(b) [6 points] **Modify the code above** so that it accomplishes the task in its specification. (Your answer should be edits to the original code.)

Change lines 9-end in the original with lines 3-end of the following.

```
1       if p.name == "Waldo":
2           return True
3       for parent in p.parents:
4           if find_waldo_broken(parent):
5               return True
6       return False
```

Alternate solution: replace line 11 in the original with lines 5-6 in the following.

```
1       if p.name == "Waldo":
2           return True
3       found = False
4       for parent in p.parents:
5           if find_waldo_broken(parent):
6               found = True
7       return found
```

Alternate solution, suggested by a student: replace line 11 in the original with:

```
        found = found or find_waldo_broken(parent)
```

Note that in this alternate solution, once found is set to True, found is never set to False and the recursion to the right of the "or" isn't even run, due to short-circuit evaluation.

6. [16 points] **Let's talk about Bruno!** This question involves a Person class with 3 instance attributes:

- name [str]
- birthyear [int], must be > 0 and < 2023 (there is no time travel)
- parents [list of Persons], possibly empty

You may assume that no Person appears twice in a family tree. You may also assume that everyone is born later than their parents.

Implement the following function, making effective use of recursion.

```
def earliest_bruno(p):
    """
    Returns: the birthyear of the earliest born ancestor named "Bruno"
             None if there is no ancestor named "Bruno"
             this includes p

    Example:  if there are two ancestors named "Bruno" born in 2000 and 1909,
                     --> returns 1909

    Precondition (no need to assert): p is a person                  """
```

This solution uses the idea of initializing a variable small to a non-None, but impossible final answer.

```
    small = 2023 # earliest Bruno birth so far, or 2023 for none found
    for parent in p.parents:
        year = earliest_bruno(parent)
        if year != None and year < small: # found an earlier Bruno birth
            small = year
    if small == 2023: # no Bruno among parents and above
        if p.name == "Bruno":
            return p.birthyear
        else:
            return None
    else:
        return small # Bruno among parents must be earlier than p
```

Similar solution — main difference is in logic after the for-loop.

```
    small = 2023
    for parent in p.parents:
        parent_result = earliest_bruno(parent)
        if parent_result is not None and parent_result < small:
            small = parent_result
    if  small == 2023 and p.name == "Bruno": # p is earliest Bruno
        small = p.birthyear
    elif small == 2023:
        return None
    return small # Bruno among parents must be earlier than p
```

Another alternate solution, inspired by student answer, that does not involve using an “impossible” value for the eventual result variable. Short-circuit evaluation is used in line 7 to ensure that `small` is not None before comparing it to `parent_result`.

```
1 if p.name == "Bruno":
2     small = p.birthyear
3 else:
4     small = None
5 for parent in p.parents:
6     parent_result = earliest_bruno(parent)
7     if parent_result is not None and (small is None or small > parent_result):
8         small = parent_result
9 return small
```

Another alternate solution, inspired by student answer: instead of keeping a single minimum-found-so-far variable, keep a list of earliest Bruno birthyears found for each parent, plus perhaps p’s birthyear if it is named Bruno.

```
    bruno_births = [] # keep a list of earliest bruno birthyears
    if p.name == "Bruno":
        bruno_births.append(p.birthyear)
    for parent in p.parents:
        bruno_birth_from_parent = earliest_bruno(parent)
        if bruno_birth_from_parent is not None:
            bruno_births.append(bruno_birth_from_parent)
    if bruno_births == []:
        return None
    else:
        return sorted(bruno_births)[0] # the year at position 0 when bruno_births is sorted
```