CS 1110 Prelim 2, April 2022

This 90-minute closed-book, closed-notes exam has 6 questions worth a total of roughly 80 points (some point-total adjustment may occur during grading).
You may separate the pages while working on the exam; we have a stapler available.

It is a violation of the Academic Integrity Code to look at any exam other than your own, to look at any reference material besides the reference provided in the exam itself, or to otherwise give or receive unauthorized help.
We also ask that you not discuss this exam with students who are scheduled to take a later makeup.
Academic Integrity is expected of all students of Cornell University at all times, whether in the presence or absence of members of the faculty. Understanding this, I declare I shall not give, use or receive unauthorized aid in this examination.

Signature: ______________________________ Date ____________

First Name: ______________________________

Last Name: ______________________________

Cornell NetID, all caps: [ ]

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. """
```

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
    """
```

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   """





    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        """
```

(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

(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.      """
```

(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.)

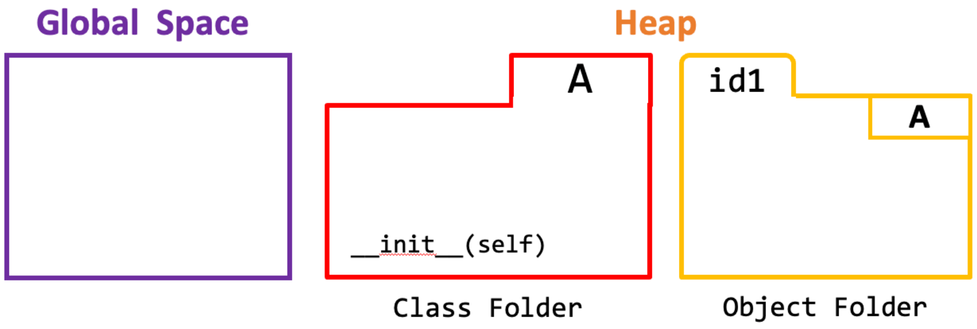
4. [20 points] **A Picture is worth a thousand words ...and ~~not 4~~ 5 points.** Diagram the execution of each of the following code snippets. Include global variables, object folders and class folders, but **omit** call frames.

   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.

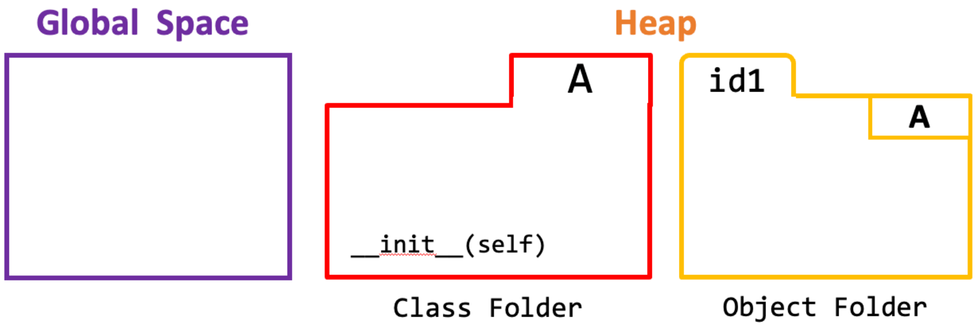
```
class A:
    b = 1
    d = 2
    def __init__(self):
        self.d = 3

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



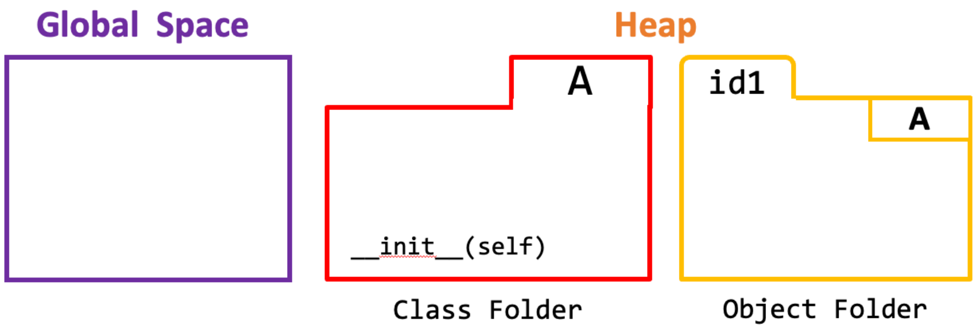
```
class A:
    b = 1
    def __init__(self):
        self.d = 2

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



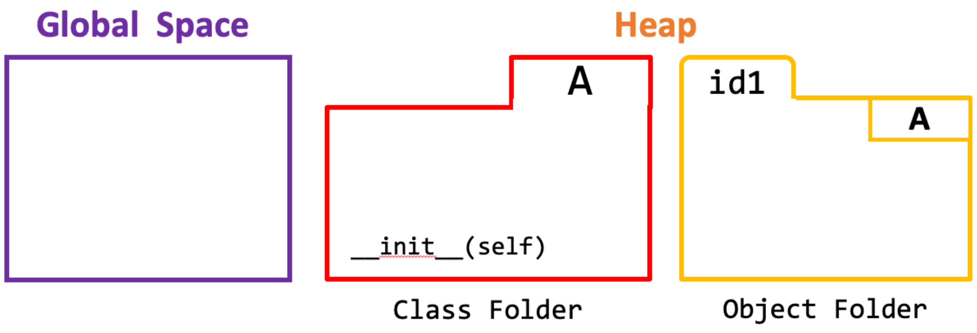
```
class A:
    b = 1
    def __init__(self):
        self.d = 2

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



```
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.

```python
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
    found = False
    for parent in p.parents:
        found = find_waldo_broken(parent)
    return found

```

(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

(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.)

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 is a comprehensive reference sheet that might include functions or methods not needed for your prelim.

String methods	
`s[i:j]`	Returns: if `i` and `j` are non-negative indices and `i` $\leq$ `j-1`, a new string containing the characters in `s` from index `i` to index `j-1`, or the substring of `s` starting at `i` if `j` $\geq$ `len(s)`
`s.count(s1)`	Returns: the number of times `s1` occurs in string `s`
`s.find(s1)`	Returns: index of first occurrence of string `s1` in string `s` (-1 if not found)
`s.find(s1,n)`	Returns: index of first occurrence of string `s1` in string `s` STARTING at position n. (-1 if `s1` not found in `s` from this position)
`s.index(s1)`	Returns: index of first occurrence of string `s1` in string `s`; raises an error if `s1` is not found in `s`.
`s.index(s1,n)`	Returns: index of first occurrence of string `s1` in string `s` STARTING at position n; raises an error if `s1` is not found in `s` from this position
`s.isalpha()`	Returns: `True` if `s` is *not empty* and its elements are all letters; it returns `False` otherwise.
`s.isdigit()`	Returns: `True` if `s` is *not empty* and its elements are all numbers; it returns `False` otherwise.
`s.islower()`	Returns: `True` if `s` is has at least one letter and all letters are lower case; returns `False` otherwise (*e.g.*, 'a123' is `True` but '123' is `False`).
`s.isupper()`	Returns: `True` if `s` is has at least one letter and all letters are upper case; returns `False` otherwise (*e.g.*, 'A123' is `True` but '123' is `False`).
`s.lower()`	Returns: a copy of `s`, all letters converted to lower case.
`s.join(slist)`	Returns: a string that is the concatenation of the strings in list `slist` separated by string `s`
`s.replace(a,b)`	Returns: a *copy* of `s` where all instances of `a` are replaced with `b`
`s.split(sep)`	Returns: a list of the "words" in string `s`, using `sep` as the word delimiter (whitespace if `sep` not given)
`s.strip()`	Returns: copy of string `s` where all whitespace has been removed from the beginning and the end of `s`. Whitespace not at the ends is preserved.
`s.upper()`	Returns: a copy of `s`, all letters converted to upper case.

List methods	
`lt[i:j]`	Returns: if `i` and `j` are non-negative indices and `i` $\leq$ `j-1`, a new list containing the elements in `lt` from index `i` to index `j-1`, or the sublist of `lt` starting at `i` if `j` $\geq$ `len(s)`
`lt.append(item)`	Adds `item` to the end of list `lt`
`lt.count(item)`	Returns: count of how many times `item` occurs in list `lt`
`lt.index(item)`	Returns: index of first occurrence of `item` in list `lt`; raises an error if `item` is not found. (There's no "find()" for lists.)
`lt.index(y, n)`	Returns: index of first occurrence of `item` in list `lt` STARTING at position `n`; raises an error if `item` does not occur in `lt`.
`lt.insert(i,item)`	Insert `item` into list `lt` at position `i`
`lt.pop(i)`	Returns: element of list `lt` at index `i` **and also removes that element from the list** `lt`. Raises an error if `i` is an invalid index.
`lt.remove(item)`	Removes the first occurrence of `item` from list `lt`; raises an error if `item` not found.
`lt.reverse()`	Reverses the list `lt` in place (so, `lt` is modified)
`lt.sort()`	Rearranges the elements of `x` to be in ascending order.

Dictionary Operations	
`d[k] = v`	Assigns value `v` to the key `k` in `d`.
`d[k]`	If value `v` was assigned to the key `k` in `d`, `d[k]` evaluates to `v`.
`del d[k]`	Deletes the key `k` (and its value) from the dictionary `d`.

Other useful functions	
`s1 in s`	Returns: `True` if the substring `s1` is in string `s`; `False` otherwise.
`elem in lt`	Returns: `True` if the element `elem` is in list `lt`; `False` otherwise.
`y in d`	Returns: `True` if `y` is a key in dictionary `d`; `False` otherwise.
`input(s)`	prompts user for a response using string `s`; returns the user's response as a string.
`isinstance(o, c)`	Returns: `True` if `o` is an instance of class `c`; `False` otherwise.
`len(s)`	Returns: number of characters in string `s`; it can be 0.
`len(lt)`	Returns: number of items in list `lt`; it can be 0.
`len(d)`	Returns: number of keys in dictionary `d`; it can be 0.
`list(range(n))`	Returns: the list `[0 .. n-1]`