Last Name: ____________ First: ____________ Netid: ____________

CS 1110 Prelim 2 November 21st, 2019

This 90-minute exam has 5 questions worth a total of 100 points. Scan the whole test before starting. Budget your time wisely. Use the back of the pages if you need more space. You may tear the pages apart; we have a stapler at the front of the room.

It is a violation of the Academic Integrity Code to look at any exam other than your own, look at any reference material, or otherwise give or receive unauthorized help.

You will be expected to write Python code on this exam. We recommend that you draw vertical lines to make your indentation clear, as follows:

```
def foo():
|   if something:
|   |   do something
|   |   do more things
|   do something last
```

You should not use while-loops on this exam. Beyond that, you may use any Python feature that you have learned about in class (if-statements, try-except, lists, for-loops, recursion and so on).

Question	Points	Score
1	2	
2	21	
3	25	
4	26	
5	26	
Total:	100	

The Important First Question:

1. [2 points] Write your last name, first name, and netid at the top of each page.

References

String Operations

Operation	Description
`len(s)`	**Returns**: Number of characters in `s`; it can be 0.
`a in s`	**Returns**: True if the substring `a` is in `s`; False otherwise.
`a*n`	**Returns**: The concatenation of `n` copies of `a`: `a`+`a`+ ⋯ +`a`.
`s.find(s1)`	**Returns**: Index of FIRST occurrence of `s1` in `s` (-1 if `s1` is not in s).
`s.count(s1)`	**Returns**: Number of (non-overlapping) occurrences of `s1` in `s`.
`s.islower()`	**Returns**: True if `s` is *has at least one letter* and all letters are lower case; it 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 uppper case; it returns False otherwise (e.g. `'A123'` is True but `'123'` is False).
`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.isalnum()`	**Returns**: True if `s` is *not empty* and its elements are all letters or numbers; it returns False otherwise.

List Operations

Operation	Description
`len(x)`	**Returns**: Number of elements in list `x`; it can be 0.
`y in x`	**Returns**: True if `y` is in list `x`; False otherwise.
`x.index(y)`	**Returns**: Index of FIRST occurrence of `y` in `x` (error if `y` is not in `x`).
`x.count(y)`	**Returns**: the number of times `y` appears in list `x`.
`x.append(y)`	Adds `y` to the end of list `x`.
`x.insert(i,y)`	Inserts `y` at position `i` in `x`. Elements after `i` are shifted to the right.
`x.remove(y)`	Removes first item from the list equal to `y`. (error if `y` is not in `x`).

Dictionary Operations

Function or Method	Description
`len(d)`	**Returns**: number of keys in dictionary `d`; it can be 0.
`y in d`	**Returns**: True if `y` is a key `d`; False otherwise.
`d[k] = v`	Assigns value `v` to the key `k` in `d`.
`del d[k]`	Deletes the key `k` (and its value) from the dictionary `d`.
`d.clear()`	Removes all keys (and values) from the dictionary `d`.

2. [21 points total] **Iteration**.

   Implement the functions below using for-loops. You **do not** need to enforce preconditions.

   (a) [9 points]

```
def skipmult(alist,n,k):
    """MODIFIES alist to multiply every kth element by n (starting at 0).

    Example: Suppose a = [1, 3, 5, 7]. skipmult(a,2,2) makes a == [2, 3, 10, 7]
    (positions 0 and 2). Similarly, skipmult(a,2,3) makes a == [2, 3, 5, 14].

    Precondition: alist is a list of numbers (int or float)
    Precondition: n is number (int or float), k is an int > 1"""
```

   (b) [12 points]

```
def merge(dict1,dict2):
    """Returns a new dictionary merging (joining keys) dict1 and dict2.

    If a key appears in only one of dict1 or dict2, the value is the value
    from that dictionary. If it is in both, the value is the sum of values.
    Example: merge({'a':1,'b':2},{'b':3,'c':4}) returns {'a':1,'b':5,'c':4}

    Precondition: dict1, dict2 are dictionaries with int or float values"""
```

3. [25 points total] **Recursion**.
   Use recursion to implement the following functions. Solutions using for-loops will receive no credit. You **do not** need to enforce the preconditions.

   (a) [12 points]

```
def clamp(alist,min,max):
    """Returns a copy of alist with every element between min and max (inclusive).

    Any number less than min becomes min. Any number greater than max becomes
    max. Any number strictly between min and max is left unchanged.

    Example: clamp([-1, 1, 3, 5],0,4) returns [0,1,3,4]

    Precondition: alist is a (possibly empty) list of numbers (float or int)
    Precondition: min <= max are numbers (int or float)"""
```

(b) [13 points]

```
def disemvowel(text):
    """Returns a tuple splitting text into the pair (consonants,vowels)

    The vowels are 'a','e','i','o','u'.  The order and number of characters is
    preserved.  If text has no consonants, that side of the pair is the empty
    string and likewise for vowels.

    Example: disemvowel('membership') returns ('mmbrshp','eei')
    Example: disemvowel('grr') returns ('grr','')
    Example: disemvowel('aie') returns ('','aie'),

    Precondition: text is a (possibly empty) string of lower case letters"""
```

4. [26 points] **Classes and Subclasses**

In this problem, you will create `Cornellian`, a class representing a person working or studying at Cornell. You will also create the subclass `Student`, which is a `Cornellian` with a GPA.

Each `Cornellian` must have a unique Cornell id. The class attribute `NEXTID` is used to automatically assign Cornell ids (a person cannot pick their id). When a `Cornellian` object is created, it gets `NEXTID` as its id and then `NEXTID` is incremented by one.

In summary, the attributes of these two classes are as follows:

`Cornellian`

Attribute	**Invariant**	**Category**
`NEXTID`	int > 0	Class attribute
`_cuid`	int > 0	**Immutable** instance attribute
`_name`	nonempty string	**Mutable** instance attribute

`Student`

Attribute	**Invariant**	**Category**
`_gpa`	float in 0.0 to 4.3	**Mutable** instance attribute

On the next three pages, you are to do the following:

1. Fill in the missing information in each class header.
2. Add getters and setters as appropriate for the instance attributes
3. Fill in the parameters of each method (beyond the getters and setters).
4. Implement each method according to the specification.
5. Enforce any preconditions in these methods using asserts

We have not added headers for any of the getters and setters. You are to write (and name) these yourself. **You are not expected to write specifications for the getters and setters**. For the other methods, pay attention to the provided specifications. The only parameters are those indicated by the preconditions.

Important: `Student` is not allowed to access any hidden attributes of `Cornellian`. As an additional restriction, `Student` may not access any getters and setters in `Cornellian`.

Last Name: ________________ First: ________________ Netid: ________________

```
class Cornellian___________________________________          # Fill in missing part
    """A class representing a student at Cornell"""

    Attribute NEXTID: A CLASS ATTRIBUTE that is an int > 0.  Its initial value is 1."""
    # Attribute _cuid: The Cornell id. An int > 0 (IMMUTABLE)
    # Attribute _name: The full name of the person. A non-empty string (MUTABLE)

    # INITIALIZE THE CLASS ATTRIBUTE



    # DEFINE GETTERS/SETTERS/HELPERS AS APPROPRIATE. SPECIFICATIONS NOT NEEDED.



    # THE INTIALIZER
    def __init_____________________________________          # Fill in missing part
        """Initializes a Cornellian with name n.

        The initializer assigns the NEXTID value as the Cornell ID.
        It it also increments the class attribute NEXTID by one.

        Precondition: n is a nonempty string"""
```

```
    # Class Cornellian (CONTINUED).
    def __str__________________________________              # Fill in missing part
        """Returns a description of this Cornellian

        The description has form 'name [cuid]'
        Example:  'Walker White [1160491]' """



    def __eq__________________________________               # Fill in missing part
        """Returns True other is a Cornellian object equal to this one.

        Two Cornellians are equal if they have the same Cornell ID EVEN
        THOUGH their names may be different (people can change names).

        Precondition: NONE (other can be anything)"""



class Student__________________________________              # Fill in missing part
    """A class representing a student at Cornell"""
    # Attribute _gpa: Student's grade point average. Float between 0.0 and 4.3 (MUTABLE)

    # DEFINE GETTERS/SETTERS AS APPROPRIATE. SPECIFICATIONS NOT NEEDED.
```

```
# Class Student (CONTINUED).
def __init________________________________          # Fill in missing part
    """Initializes a Student with name n and gpa g.

    Precondition: n is a nonempty string
    Precondition: g is a float between 0.0 and 4.3 (DEFAULT value 0.0)"""




def onDeansList______________________________        # Fill in missing part
    """Return True if the student's GPA >= 3.5; False otherwise"""




def __str_______________________________             # Fill in missing part
    """Returns a description of this Student

    The description is the same as Cornellian, except that it adds
    "Dean's List" (after a comma) if the Student is on the Dean's List.

    Examples: 'Bob Roberts [234781]' or "Emma Towns [492886], Dean's List" """
```

Last Name: ________________ First: ________________ Netid: ________________

5. [26 points total] **Folders and Name Resolution**

Consider the two (undocumented) classes below, together with their line numbers.

```
1  class A(object):
2      x = [1]
3      y = 5
4
5      def __init__(self,x,y):
6          self.x.append(x)
7          self.y = y
8
9      def f(self,y):
10         return 1-self.g(y)
11
12     def g(self,x):
13         return x*self.y
14
```

```
15 class B(A):
16     x = [2]
17     z = 3
18
19     def __init__(self,x):
20         super().__init__(x,x+1)
21         self.x = x
22
23     def f(self,y):
24         return super().f(y)+2
25
26     def g(self,x):
27         self.x = x
28         return 3+self.y
```

(a) [10 points] Execute (do **not** diagram) the constructor call

```
>>> b = B(3)
```

Draw **all** of the contents of the heap (including class folders and any objects) after this call is completed. You do not need to draw global space or any call frames. Assign `b` the folder identifier `id1` (even if other objects are present) so we can clearly identify it.

(b) [16 points] On the next two pages, diagram the method call

```
>>> a = b.f(4)
```

Diagram the state of the entire call stack for the method when it starts, for each line executed, and when the frame is erased. If any other methods are called, you should do this for them as well (at the appropriate time). This will require a total of **eight diagrams**.

You should draw also the state of global space and the heap at each step. You can ignore everything in the heap except for the folder for `b` (so no need to draw class folders). You may also write "unchanged" if no changes were made to either global space or the heap.

Last Name: ______________ First: ______________ Netid: ______________

Call Frames	Global Space	The Heap
①		
②		
③		
④		

Last Name: ______________ First: ______________ Netid: ______________

Call Frames	Global Space	The Heap
5		
6		
7		
8		