Last Name: _____ First: _____ Netid: _____

CS 1110 Prelim 2 November 8th, 2018

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	20	
3	22	
4	25	
5	31	
Total:	100	

Last Name: _____ First: _____ Netid: _____

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`+$\cdots$+`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`.

The Important First Question:

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

Last Name: _____ First: _____ Netid: _____

2. [20 points total] **Iteration**.

   One of the oldest ways of hiding messages was to use an alphabetic cipher. Each letter would be swapped with a new letter, and swapping the letters back would give the original message. We can keep track of a cipher as a dictionary; for each key in the dictionary, we swap that key with its value. So the cipher {`'a':'o','b':'z'`} converts `'bat'` to `'zot'`. We do not need a key for each letter (letters with no key are unchanged), but each value must appear only once. So {`'a':'o','b':'z'`} is a good cipher, but {`'a':'o','b':'o'`} is bad.

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

   (a) [9 points]

```
def encode(cipher,text):
    """Returns an encoded copy of text using given cipher dictionary
    Example: encode({'a':'o','z':'b'},'razzle') returns 'robble'
    Precondition: cipher is good with lowercase letters as keys and values
    Precondition: text is a (possibly empty) string of lowercase letters"""
```

   (b) [11 points] **Hint**. Break this into two steps: copy and invert the `cipher`, then update it.

```
def invert(cipher):
    """MODIFIES cipher to swap the keys and values
    Example: If d = {'a':'o','z':'b'}, then invert(d) modifies the cipher d
    to be {'o':'a','b':'z'} instead.
    Precondition: cipher is good with lowercase letters as keys and values"""
```

Last Name: _____ First: _____ Netid: _____

3. [22 points total] **Recursion**.
Some types of encoding are not used to hide messages, but instead help the transfer of messages accross the Internet. Use recursion to implement the following functions. Solutions using for-loops will receive no credit. You **do not** need to enforce the preconditions.

(a) [9 points] **Hint**: Remember that you can multiply strings by ints. `'ab'*3` is `'ababab'`.

```
def decode(nlist):
    """Returns a string that represents the decoded nlist
    The nlist is a list of lists, where each element is a character and
    a number. The number is the number of times to repeat the character.
    Example: decode([['a',3],['h',1],['a',1]]) is 'aaaha'
    Example: decode([]) is ''
    Precondition: nlist is a (possibly empty) nested list of two-element lists,
    where each list inside is a pair of a character and an integer"""
```

(b) [13 points]

```
def encode(text):
    """Returns a nested list encoding the duplication of each character
    The returned list is a (possibly empty) nested list of two-element lists,
    where each list inside is a pair of a character and an integer.
    Example: encode('aaaha') is [['a',3],['h',1],['a',1]]
    Example: encode('') is []
    Precondition: text is a (possibly empty) string"""
```

Last Name: _____ First: _____ Netid: _____

4. [25 points total] **Folders and Name Resolution**

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

```
1  class A(object):
2      x = 5
3      y = 10
4
5      def __init__(self,x):
6          self.z = x
7
8      def f(self,x):
9          if x > 0:
10             return 2+self.f(x-1)
11         else:
12             return x+self.x
```

```
14 class B(A):
15     x = 3
16
17     def __init__(self,x):
18         self.w = self.f(x-3)
19
20     def g(self,x):
21         if x > 0:
22             return self.f(x-1)
23         else:
24             return x+self.y
25
```

(a) [5 points] Draw the class folders in heap space for these two classes.

(b) [20 points] On the next two pages, diagram the call

```
>>> z = B(4)
```

You will need **ten diagrams**. Draw the call stack, global space and heap space. If the contents of any space are unchanged between diagrams, you may write *unchanged*. You do not need to draw the class folders from part (a).

When diagramming a constructor, you should follow the rules from Assignment 5. Remember that `__init__` is a helper to a constructor but it is not the same as the constructor. In particular, there is an important first step before you create the call frame.

Last Name: _____ First: _____ Netid: _____

Call Frames	Global Space	Heap Space
①		
②		
③		
④		
⑤		

Last Name: _____ First: _____ Netid: _____

	Call Frames	Global Space	Heap Space
⑥			
⑦			
⑧			
⑨			
⑩			

5. [31 points] **Classes and Subclasses**

In this problem, you will create a class representing a license plate in a small state. License plates in this state are three (upper case) letters followed by a number 0..999. This number is padded with leading 0s to make it three digits. Examples of licenses are `ABC-001` or `XYZ-093`.

One of the most important properties of a license plate is that there can only be one of them with a given value. So we cannot have two different objects for the same license `ABC-001`. To model this propery, the class `License` has a class attribute list named `USED`. Every time a new license plate is created, the value is added to this list so that it cannot be used again. In addition, the license plate value is immutable (since allowing a user to change it would mean that the user could create two plates with the same value).

In addition to normal license plates, some people like to have vanity plates. A common vanity plate is one that is attached to a specific university, showing that the owner is an alum. Again, we cannot have a vanity plate with the same number as an existing plate. But since `Vanity` is a subclass of `License`, this should not be a problem if we initialize it properly.

On the next four 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 these from scratch. However, **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: `Vanity` is not allowed to access any hidden attributes of `License`. We are also adding the additional restriction that `Vanity` may not access any getters and setters in `License`.

```
class License_____                                  # Fill in missing part
    """A class representing a license plate

    CLASS ATTRIBUTES (NO GETTERS/SETTERS):
        USED:  The license plates used so far [list of [prefix,suffix] pairs]
    The initial value is the empty list.

    MUTABLE ATTRIBUTES:
        _owner: The name of the owner [nonempty string or None]

    IMMUTABLE ATTRIBUTES:
        _prefix: The first half of the licence  [str of 3 upper case letters]
        _suffix: The first half of the licence  [int 0..999 inclusive]"""
    # CLASS ATTRIBUTES.




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

```
# Class License (CONTINUED).
def __init__(_____):                                    # Fill in missing part
    """Initializes a license plate with the given prefix and suffix.

    No license plate can be created if it has the same prefix and suffix as an
    existing plate.  On creation, the pair (prefix,suffix) is added to the
    class attribute USED to ensure that they cannot be reused.

    Precondition: prefix is a string of 3 upper case letters
    Precondition: suffix is an int in 0..999, inclusive
    Precondition: owner is a nonempty string or None (Optional; default None)
    Additional precondition: No other license plate has this prefix,suffix"""
```

```
def __str__(_____):                                     # Fill in missing part
    """Returns a string representation of this license plate.

    The string is of the form prefix-suffix. The suffix is padded with leading 0s
    to have three characters.  If the plate has an owner, the owner follows
    the string in parentheses.  Otherwise, nothing is added to the string.
    Example: 'ABC-001' if no owner, or 'XYZ-093 (Bob)' """
```

```
def __eq__(_____):                                      # Fill in missing part
    """Returns True if other is equal to this license plate; otherwise False

    Two license plates are equal if they have the same prefix and suffix.
    They do NOT need to have the same owner.
    Precondition: other is a License object"""
```

Last Name: _____ First: _____ Netid: _____

```
class Vanity_____                                   # Fill in missing part
    """A class representing a vanity license plate

    MUTABLE ATTIBUTE (In addition to those from File):
        _university : The university displayed on the plate [a nonempty string]"""
    # DEFINE GETTERS/SETTERS AS APPROPRIATE. SPECIFICATIONS NOT NEEDED.

    def __init__(self,prefix,suffix,owner,university)_____   # Fill in missing part
        """Initializes a vanity license plate with the given values.

        Vanity license plates must have an (initial) owner.  No arguments are optional.
        Precondition: prefix is a string of 3 upper case letters
        Precondition: suffix is an int in 0..999, inclusive
        Precondition: owner is a nonempty string
        Precondition: university is a nonempty string
        Additional precondition: No other license plate has this prefix,suffix"""

    def __str__(self)_____                          # Fill in missing part
        """Returns a string representation of this vanity plate

        The format is 'prefix-suffix (Owner, University)'  If owner is None (the
        setter allows this to happen), the format is 'prefix-suffix (University)'
        Example: 'ABC-001 (Cornell)' if no owner, or 'XYZ-093 (Bob, Syracuse)' """
```