Last Name: _____ First: _____ Netid: _____ Section _____

CS 1110 Prelim 2 November 10th, 2016

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	30	
3	26	
4	20	
5	22	
Total:	100	

The Important First Question:

1. [2 points] Write your last name, first name, netid, and *lab section* at the top of each page. If you cannot remember the section number, please write down your lab's time and place.

String Functions and Methods

Function/Method	Description
`len(s)`	**Returns**: Number of characters in `s`; it can be 0.
`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.replace(a,b)`	**Returns**: A *copy* of `s` where all instances of `a` are replaced with `b`.

List Functions and Methods

Function/Method	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.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`).
`x.sort()`	Rearrages the elements of `x` to be in ascending order.

Dictionary Functions and Methods

Function/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.keys()`	**Returns**: List containing all the keys in `d`.
`d.values()`	**Returns**: List containing all the values in `d`. It may have duplicates.

2. [30 points] **Classes and Subclasses**

The next two pages have skeletons of two classes: `Pet` and `ExoticPet`. `ExoticPet` is a subclass of `Pet`, while `Pet` is only a subclass of `object`. You are to

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

Pay close attention specifications. There are no parameters beyond the ones specified. If a parameter has a default value, it is listed in the specification. There are no headers for getters or setters. You are to write those from scratch. However, **you are not expected to write specifications for the getters and setters**.

Important: `ExoticPet` must not access the hidden attributes of `Pet` directly. If you need to access a hidden attribute, use methods from `Pet`.

```
class Pet(          object          ):              # Fill in missing part
    """Instances represent a person's pet.

    MUTABLE ATTRIBUTES
        _name:  The name of the pet [nonempty string]
        _tag:  The license number for this pet [int >= -1]

    We use a _tag value of -1 to indicate a pet without a license."""

    # PUT GETTERS/SETTERS HERE AS APPROPRIATE. SPECIFICATIONS NOT NECESSARY.
    def getName(              self              ):
        """Returns:  The name of this Pet."""
        return self._name


    def setName(         self,     value         ):
        """Sets the name of this pet"""
        assert type(value) == str
        assert len(value) > 0
        self._name = value


    def getTag(              self              ):
        """Returns: The license number of the pet (or -1 if none)."""
        return self._tag


    def setTag(         self,     value         ):
        """Sets the license number of the pet (or -1 if none)"""
        assert type(value) == int
        assert value >= -1
        self._tag = value


    def __init__(  self,     name,     tag = -1  ):       # Fill in parameters
        """Initializer: Creates an pet with the given name and (optional) tag.

        Precondition: Parameter name is a nonempty string
        Precondition: Parameter tag is an int >= -1 (optional, default is -1)"""
        self.setName(name)  # Handles precondition for us.
        self.setTag(tag)    # Handles precondition for us.
```

```
    # Class Pet (CONTINUED).

    def __str__(              self              ):       # Fill in parameters
        """Returns:  A string representation of this pet.
        The string is the pet name with the license number in parentheses.
        A pet with a tag of -1 is "unclaimed".
        Examples: 'Sparky (pet #43)' or 'Rover (unclaimed)'"""
        if tag == -1:
            result = self._name+' (unclaimed)'
        else:
            result = self._name+' (pet #'+str(self._tag)+')'
        return result


class ExoticPet(           Pet           ):            # Fill in missing part
    """Instances represent a pet certified by a wild-life official.
    IMMUTABLE ATTIBUTE (In addition to those from Pet):
        _official :  The official that certified the pet [nonempty string]
    In addition, this class disables the setter setTag in Pets (you cannot
    change the license of an exotic pet).  It does this by overriding setTag
    so that it creates a NotImplementedError (an an class built into Python)
    whenever it is called."""
    # PUT GETTERS/SETTERS HERE AS APPROPRIATE. SPECIFICATIONS NOT NECESSARY.
    # OVERRIDE THE SETTER FOR TAG, BUT NOT THE GETTER.

    def getOfficial(              self              ):
        """Returns: The wild-life official that certified the pet"""
        return self._official


    def setTag(        self,     value        ):
        """Raises a NotImplementedError when called"""
        raise NotImplementedError()
```

```
# Class ExoticPet (CONTINUED).
def __init__( self,  name,  tag,  official ):  # Fill in parameters
    """Initializer: Creates an exotic pet with name, tag, and official.

    In the case of an exotic pet, the tag is not optional and must be a
    value greater than -1.

    Precondition: Parameter name is a nonempty string
    Precondition: Parameter tag is an int >= 0
    Precondition: Parameter official is a nonempty string"""
    Pet.__init__(self,name,tag)
    assert tag >= 0
    assert type(official) == str
    assert len(official) > 0
    self._official = official



def __str__(              self              ):      # Fill in parameters
    """Returns: A string representation of this exotic pet

    The string is the same as for Pet, except that it also includes the
    official who certified the pet.

    Example: 'Tigger (pet #675); authorized by Sgt. O'Malley'"""
    result = Pet.__str__(self)
    result += '; authorized by '+self._official
    return result
```

3. [26 points total] **Folders and Name Resolution**

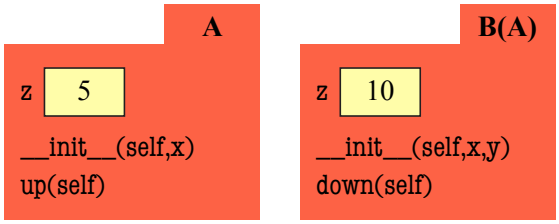
   Consider the two (undocumented) classes below, together with their line numbers.

```
1   class A(object):
2       z = 5
3
4       def __init__(self,x):
5           self.x = x+self.z
6
7       def up(self):
8           shift = self.x-self.z
9           return B(shift,shift)
10
```

```
11  class B(A):
12      z = 10
13
14      def __init__(self,x,y):
15          A.__init__(self,x)
16          self.y = y
17
18      def down(self):
19          shift = self.x-self.z
20          return A(shift)
```

Last Name: _____ First: _____ Netid: _____ Section _____

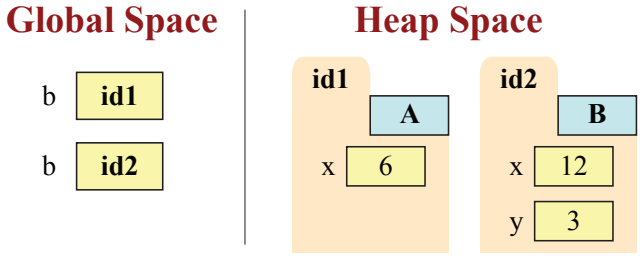
(a) [5 points] Draw the class folders in heap space for these two classes.



(b) [4 points] Execute (but do not diagram) the following statements

```
>>> a = A(1)
>>> b = B(2,3)
```

Draw the resulting variables and objects below.

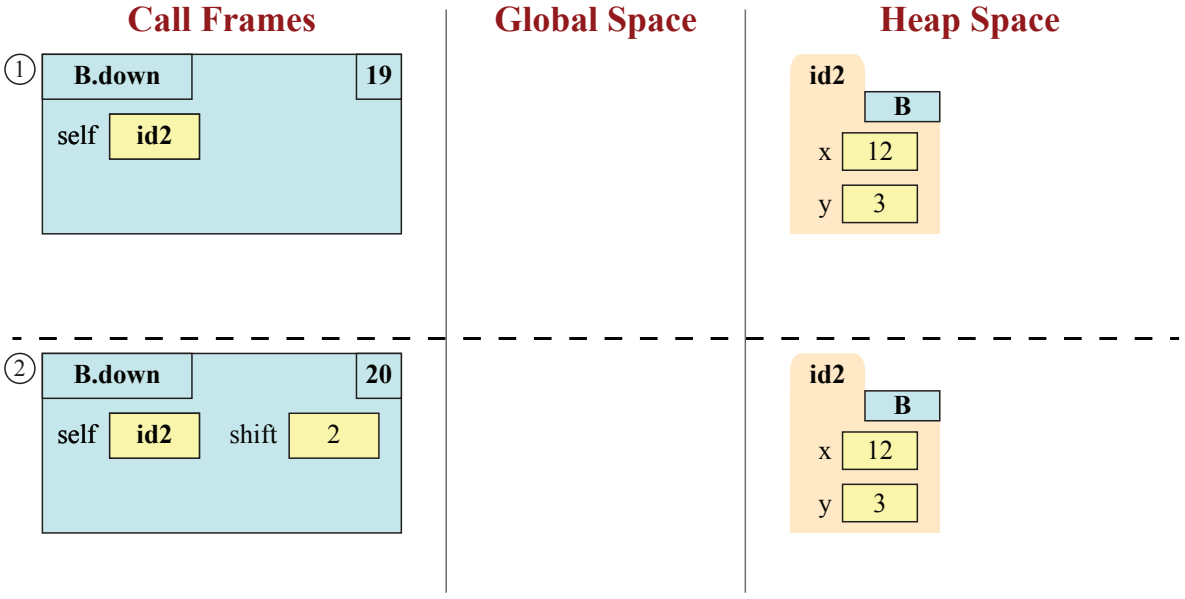


(c) [17 points] On this page and the next, diagram the call

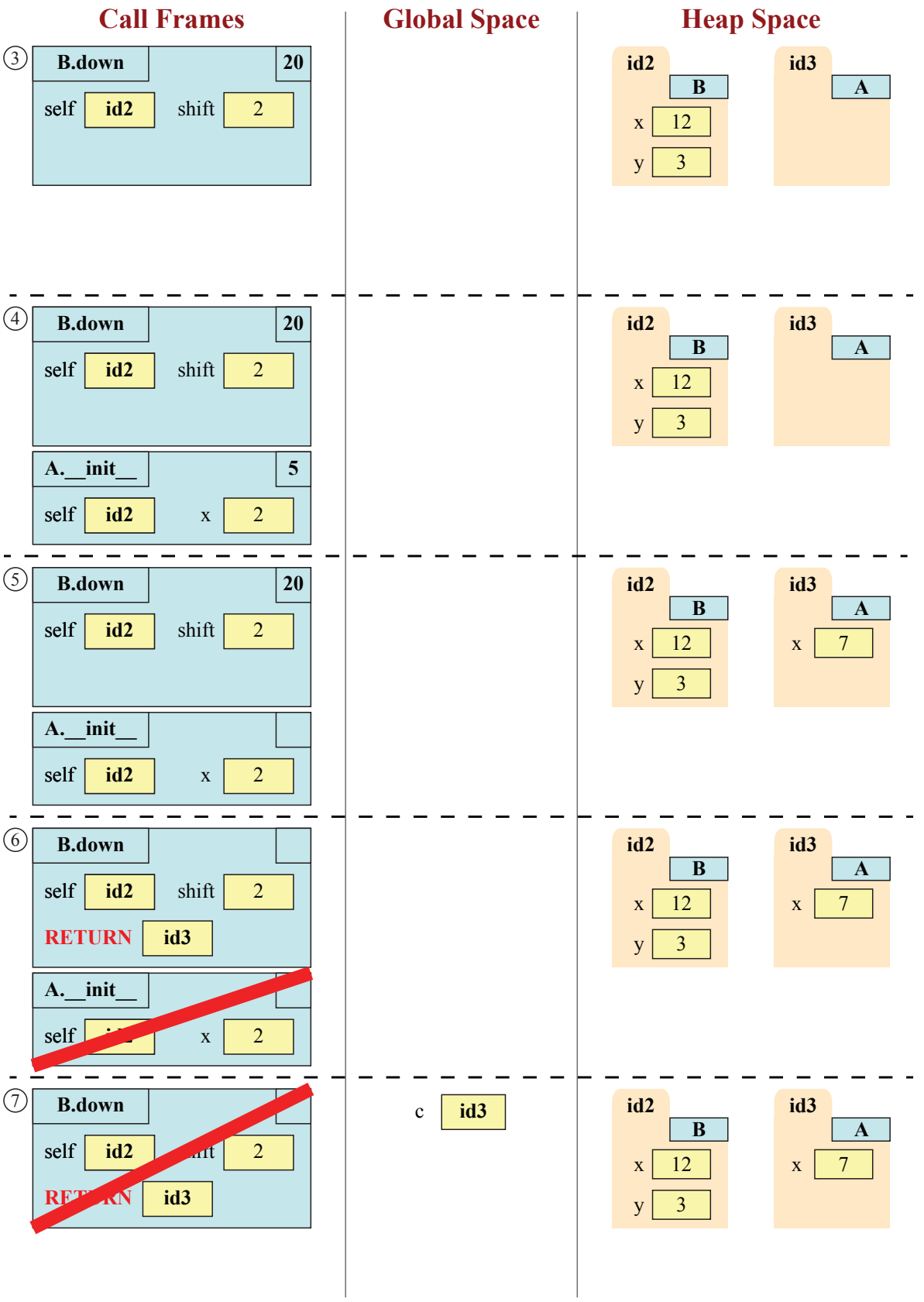
```
>>> c = b.down()
```

You will need **seven 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) or the object folders from part (b).

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.



Last Name: _____ First: _____ Netid: _____ Section _____



4. [20 points total] **Iteration**.

   The following functions take (possibly empty) strings as their input. Use for-loops to implement them according to their specification. You **do not** need to enforce the preconditions.

   (a) [8 points] In class, we saw how to implement the function below with recursion. This time *do not* use recursion; use a for-loop instead.

```
def deblank(s):
    """Returns: A copy of the string s with spaces removed
    Example: deblank('a b cd') is 'abcd'
    Precondition: Parameter s is a (possibly empty) string"""
    # Accumulator is a new string
    result = ''

    # Loop over the letters in the string
    for letter in s:
        if letter == ' ':
            result += letter

    return result
```

   (b) [12 points] An *inverted string* is a dictionary whose keys are characters and whose values are lists of positions. For example, the string `'hello'` is inverted as the dictionary

```
{ 'h':[0], 'e':[1], 'l':[2,3], 'o':[4] }
```

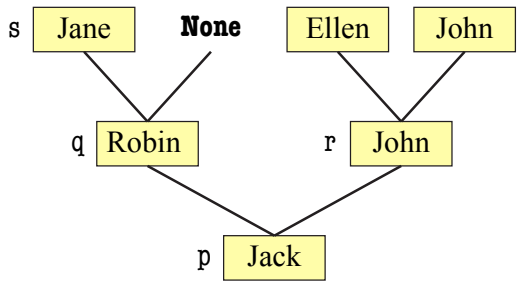
   You are free to use anything about lists or dictionaries to create an inverted string.

```
def invert(s):
    """Returns: An inverted string representing s
    Example: invert('abcac') is {'a':[0,3], 'b':[1], 'c':[2,4]}.
             invert('') is {}.
    Precondition: Parameter s is a (possibly empty) string"""
    # Accumulator is a dictionary
    result = {}

    # Much easier to loop over positions
    for x in range(len(s)):
        value = s[x]

        # Only create a list if we do not have one already
        if value in result:
            result[value].append(x)
        else:
            result[value] = [x]

    return result
```

5. [22 points total] **Recursion**.

In lecture, we introduced the class `Person`, whose specification is below. You create a person with the constructor `Person(name,mom,dad)`. To make the person s in the picture to the right, you use the assignment `s = Person('Jane',None,None)`. To make the person `q`, you use the assignment `q = Person('Robin',s,None)`. Finally, to make the person `p`, you use the assignment `p = Person('Jack',q,r)`.

```
class Person(object):
    """Instance is a person/family tree
    INSTANCE ATTRIBUTES:
       name: First name [nonempty str]
       mom: Mom's side [Person or None]
       dad: Dad's side [Person or None]
    """
    ...
```



Use recursion to implement the functions specified below; **do not use loops**. You do not need to enforce preconditions.

(a) [11 points]

```
def ancestors(p):
    """Returns: The list of names of all ancestors of p
    The name of p should not be in the list (unless another ancestor has this
    name).  Duplicates names (e.g.  different ancestors with the same name) are
    okay.  However, the list returned should be sorted alphabetically.
    Example: ancestors(p) is ['Ellen','Jane','John','John','Robin'], if p
    is the person shown above.
    Precondition: p is a Person and not None"""

    # Process small data
    if p.mom is None and p.dad is None:
        return []


    # Break it up and solve on both halves
    left = []
    if not p.mom is None:
        left = [p.mom.name]+ancestors(p.mom)

    right = []
    if not p.dad is None:
        right = [p.dad.name]+ancestors(p.dad)


    # Combine the answer
    result = left+right
    result.sort()
    return result
```

(b) [11 points] As we have seen, making a `Person` is a complicated affair. You have to create each ancestor object individually. One way to create Person objects quickly is to use a *geneaology list*. A *geneaology list* is defined recursively as follows:

- It is a nonempty list with exactly three elements.
- The first element is a nonempty string, respresenting the person's name.
- The last two elements are either None or genealogy lists.

Looking at the `Person` objects on the previous page, `s` is `['Jane',None,None]` while `q` is `['Robin',['Jane',None,None],None]`. We can even represent `p` as the list

```
['Jack',['Robin',['Jane',None,None],None],
        ['John',['Ellen',None,None],['John',None,None]]]
```

Use this recursive definition to implement the function below.

Hint: This problem is actually very simple. Just read the definition above.

```
def list2person(glist):
    """Return: A person object equivalent to the given genealogy list.
    Example: list2person(['Fred', None, None]) is Person('Fred',None,None)
    See the description above for more examples.
    Precondition: glist is a genealogy list."""

    # Compute the mother's side
    mom = None
    if not glist[1] is None:
        mom = list2person(glist[1])

    # Compute the father's side
    dad = None
    if not glist[2] is None:
        dad = list2person(glist[2])

    # Create the person
    return Person(glist[0],mom,dad)
```