Last Name: ______________ First Name: ______________ Cornell Netid: ________

CS 1110 Prelim 2 November 6th, 2012

This 90-minute exam has **??** questions worth a total of **??** 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, to look at any other reference material, or to 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 may not use while-loops. Beyond that, you may use any Python feature that you have learned about in class (particularly for-loops and recursion).

Run LaTeX again to produce the table

The Important First Question:

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

2. [26 points] **Classes and Subclasses** On the next two pages are the skeletons of two classes. For Species complete the two properties and a constructor (not __str__). For Animal complete everything. **Enforce invariants in setters and enforce preconditions in both constructors**. Do not write specifications (e.g. for the properties) beyond what is given.

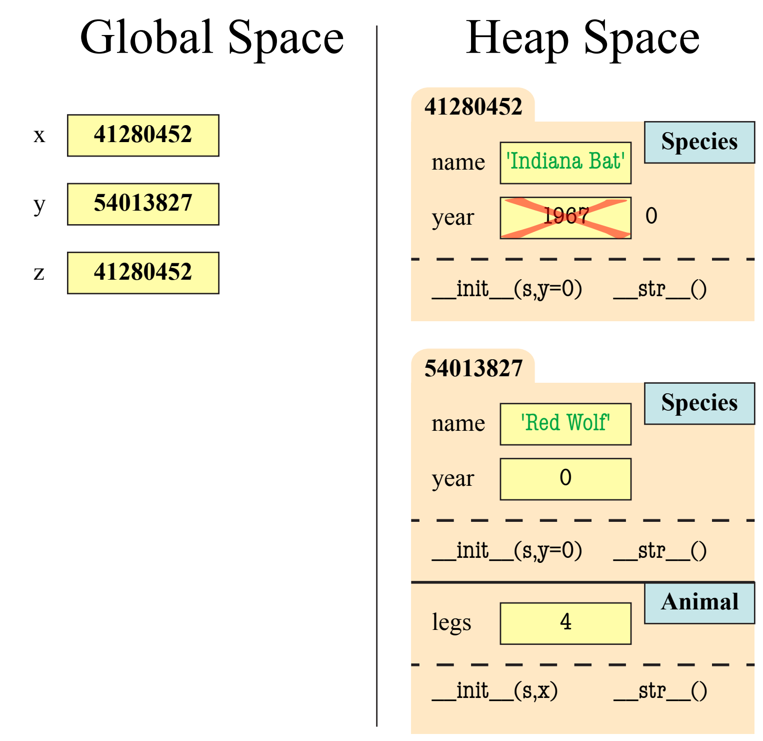
```
class Species(object):
    """Instance is an endangered species"""
    # IMMUTABLE ATTRIBUTES
    _name = '' # Species name; must be a string
    # MUTABLE ATTRIBUTES
    _year = 0 # Year on endangered list; int >= 1900 (0 if not on list)

    # DEFINE PROPERTY name for FIELD _name (specification not necessary)
    @property
    def name(self):
        return self._name


    # DEFINE PROPERTY year for FIELD _year (specification not necessary)
    @property
    def year(self):
        return self._year


    @year.setter
    def year(self,value):
        assert type(value) == int
        assert value >= 1900 or value == 0
        self._year = value


    def __init__( self, s, y=0 ):          # Fill in parameters

        """Constructor: species with name s, put on list in year y.
        Precondition: s a string, y an int >= 1900 or == 0.
        y has default of 0."""
        assert type(s) == str
        self._name = s
        self.year = y # property handles asserts

    def __str__(self):
        """Returns: Description of this species
        We have completed this method for you.  Do not change it."""
        suffix = ''
        if self.year != 0:
            suffix = ' endangered since ' + str(self.year)

        return self.name+suffix
```

```
class Animal(Species):
    """Instance is a species of Animal"""
    # IMMUTABLE ATTRIBUTES
    _legs = 0 # Number of legs; int >= 0

    # DEFINE PROPERTY legs for FIELD _legs (specification not necessary)
    @property
    def legs(self):
        return self._legs


    def __init__(self,s,x):
        """Constructor: an x-legged animal of species s that is not endangered.
        Precondition: s a string, x an int >= 0."""
        # YOU MUST USE SUPER()
        assert type(x) == int and x >= 0
        super(Animal,self).__init__(s) # Handles other asserts
        self._legs = 0


    def __str__(self):
        """Returns: Description of this species
        Format is __str__ from the superclass followed by ' with <legs> legs', where
        you fill in the value for legs."""
        # YOU MUST USE SUPER()
        prefix = super(Animal,self).__str__()
        return prefix+' with '+str(self.legs)+' legs'
```

3. [18 points] **Drawing Folders** Suppose you are executing the following sequence of commands at the interactive prompt (and hence in global space).

```
>>> x = Species('Indiana Bat', 1967)
>>> y = Animal('Red Wolf', 4)
>>> z = x
>>> x.year = 0
```

On the next page (or the back), create two columns: one for global space and another for heap space. Clearly show what is created in each, drawing folders for objects and boxes for variables. If the value of a variable or attribute changes, cross the old one out and write the new value out beside it.



4. [14 points] **Iteration.**
Complete procedure `clamp(seq,vmin,vmax)` below according to the specification. You should use a for-loop. Do not use recursion or a while-loop. Note that it is a procedure that modifies the list. **It does not return a new value**. You do not need to assert function preconditions.

Hint: The function `len` gives the length of a list.

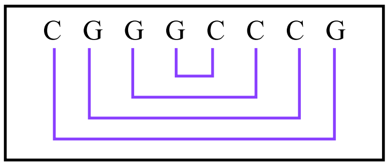
```
def clamp(seq,vmin,vmax):
    """Clamp the values in list seq (modifies seq, does not return a copy).

    Values less than vmin become vmin; values greater than vmax become vmax
    Example:  if a = [2, -5, 7], then clamp(a,-4,4) modifies the list a so
    that it is now [2, -4, 4].

    Precondition:  seq is a list of ints.  vmax > vmin are ints."""
    for k in range(len(seq)):
        if seq[k] > vmax:
            seq[k] = vmax

        if seq[k] < vmin:
            seq[k] = vmin

    # Procedure; nothing else to do
```

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

(a) [14 points]

m-RNA (messenger RNA) is a chain or string of "nucleotides." These nucleotides are typically denoted by the four symbols `'C'` (cytosine), `'G'` (guanine), `'A'` (adenine), and `'U'` (uracil). When writing programs to process m-RNA, we typically represent them as a string of these four letters. For example, `'AUUGC'` is an m-RNA sequence.

C G G G left half
G C C C right half reversed



Each nucleotide has a complementary nucleotide with which it bonds; **`'C'` and `'G'` are a complementary pair, as are `'A'` and `'U'`**. We call the RNA sequence `'CGGGCCCG'` a perfect hinge because, if the right half is reversed and placed under the left half, the corresponding characters are complementary. This is shown above. From this definition of a perfect hinge, we can deduce that **an empty sequence is a perfect hinge and a sequence with an odd number of elements is not a perfect hinge**.

Structures similar to perfect hinges play an important role in biology. Hence, we often want to detect whether an m-RNA sequence has this property. Implement the function below with the given specification. Use recursion, not a loop.

Hint: You may find this problem easier if you write a helper function to check if two symbols are complimentary pairs. If you do, include a specification docstring. *Do not* bother to assert the preconditions, for either `ishinge` or any helper function.

```
def ishinge(seq):
    """Returns: True if the m-RNA sequence is a perfect hinge.  Returns
    False otherwise (e.g. not even, does not match).

    Precondition: seq is a string with characters 'C', 'G', 'A', 'U'."""
    # Special cases mentioned above in bold
    if len(seq) == 0:
        return True
    elif len(seq) % 2 != 0:
        return False

    # Recursive case:  even and non-empty
    # USES HELPER ON NEXT PAGE
    return complement(seq[0],seq[-1]) and ishinge(s[1:-1])
```

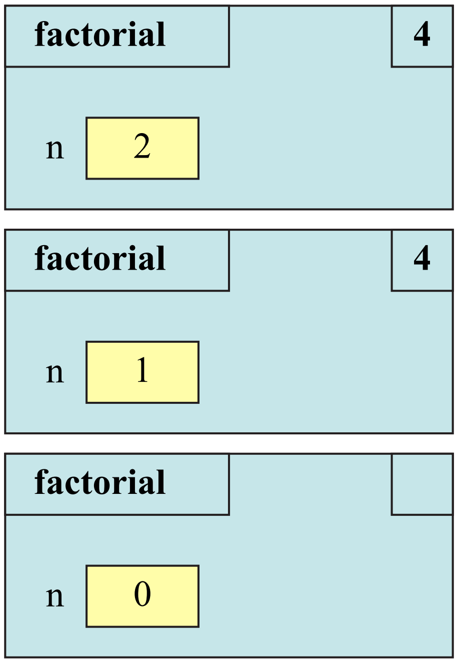
(more space for part a)

```
def complement(c1,c2):
    """Returns:  True if c1 and c2 are complementatary, False otherwise
    Precondition:  c1, c2 are each either 'G', 'G', 'A', 'U'"""
    return ( (c1 == 'G' and c2 == 'C') or
             (c1 == 'C' and c2 == 'G') or
             (c1 == 'A' and c2 == 'U') or
             (c1 == 'U' and c2 == 'A'))
```

(b) [8 points] One recursive function that we saw in class was `factorial`:

```
  def factorial(n):
      """Returns: n!
      Precondition: n a nonnegative integer"""
1     if n==0:  # Base case
2         return 1
3     # Recursive case
4     return n*factorial(n-1)
```

Evaluate the recursive call `factorial(2)` until the base case completes its return-statement at line 2 (but before the frame is erased and the value is returned). Draw the call stack at this point in time. The stack will have 3-4 call frames.



6. [**??** points total] **Short Answer**.
Answer the following questions. Each answer will require multiple sentences, but should not require more than a paragraph.

(a) [4 points] What is the difference between `is` and `==`? Give an example of when you would want to use each of these operators.

The operator `is` compares objects by "folder name."

The operator `==` invokes the `__eq__` method. This generally, but not always, compares objects by contents.

In general, you want to use `==` if you want to compare objects by contents (e.g. check if two distinct Vector objects have different x and y attributes). You always need to use `is` to compare an object to `None`, as `None` has no contents.

(b) [4 points] What is *dispatch-on-type*? How does it apply to error handling in Python?

Dispatch-on-type is the concept that the behavior of a Python command (such as a function or method call, or a `try-except` block) depends upon the type of the objects in involved. Different types create different behaviors.

In error handling, the `except` part of a `try-except` block can specify the type of an error object. It will only recover if the error received matches that type; it will not recover for other errors.

(c) [4 points] What is the *bottom-up* rule? How does it relate to *overriding*?

The bottom-up rule says that, when we call a method on an object, Python works from the bottom partition to find the method definition. It searches the partitions in the object folder starting from the main class, and progressively through each super class until it finds a method of that name.

Overridding is the act of the replacing a method that is defined in a parent class with a new definition. The bottom-up rule guarantees that the new definition will always be used.

(d) [4 points] Explain the difference between *interface* and *implementation*.

Interface is everything any other programmer can see: function/method headers and specifications (for methods and attributes). Implementation is everything else: hidden functions/methods/fields and function/method bodies.

Interface is difficult to change because other people depend on it. Implementation is easy to change because it is hidden

(e) [2 points] Is the following function definition legal? Why or why not?

```
def absmax(x,y=0,z):
    """Returns: the maximum absolute value of x, y, and z.
    Precondition: x, y, and z are numbers (int or float)"""
    return max(abs(x),abs(y),abs(z))
```

It is illegal. You cannot have parameters without defaults (e.g. `z`) after a parameter with a default (e.g. `y=0`).