Last Name: _____ First: _____ Netid: _____

CS 1110 Final, December 13th, 2025

This 150-minute exam has 8 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
```

Unless you are explicitly directed otherwise, you may use anything you have learned in this course. You may use the backside of each page for extra room for your answers. However, if you do this, **please indicate clearly** on the page of the associated problem.

Question	Points	Score
1	2	
2	12	
3	13	
4	23	
5	14	
6	15	
7	9	
8	12	
Total:	100	

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: _____

References

String Operations

Expression	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.
`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.split(s1)`	**Returns**: Returns the list of substrings of `s` separated by `s1`. `s1` is not included in the list. *Ex*: `'a;b;;c'.split(';')` returns `['a','b','','c']`
`s.lower()`	**Returns**: A copy of`s` with all letters converted to lower case.
`s.upper()`	**Returns**: A copy of`s` with all letters converted to upper case.
`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 digits; it returns False otherwise.
`s.isalnum()`	**Returns**: True if `s` is *not empty* and its elements are all letters or digits; it returns False otherwise.

List Operations

Expression	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.
`del x[k]`	Deletes the element of the list at position k.
`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

Expression	Description
`len(d)`	**Returns**: number of keys in dictionary `d`; it can be 0.
`y in d`	**Returns**: True if `y` is a key in dictionary `d`; False otherwise.
`d[k] = v`	Assigns value `v` to the key `k` in dictionary `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`.

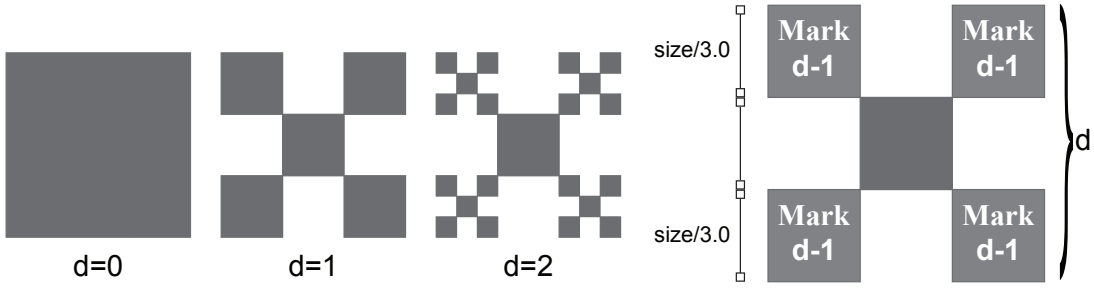
Last Name: _____ First: _____ Netid: _____

2. [12 points] **Recursion**

In Assignment 4, you had a helper function similar to the one below.

```
def fillsquare(w,x,y,size):
    """Draws a solid square with bottom left corner (x,y) and length size.

    Preconditions: w is a window, and x, y, and size are positive floats"""
```

You are going to use this helper to recursively draw the *cantor mark*. This shape is a square `X` where the corners get recursively fancy. The shape is defined as follows:



Define the function below. You do not need to create a `Window`, or worry about color. Simply use the function `fillsquare` above to defined the shape. **Do not assert preconditions.**

```
def mark(w, x, y, size, d):
    """Draws a cantor mark of depth d with bottom left at (x,y) and length size.

    Preconditions: w a window; x, y, and size are floats w/ size > 0; d >= 0 an int"""

    # Handle the base shape
    if d == 0:
        fillsquare(w,x,y,size)
        return

    nsize = size/3
    # Draw the center square
    fillsquare(w,x+nsize,y+nsize,nsize)

    # Do the recursive calls
    mark(w,x,y,nsize,d-1)
    mark(w,x,y+2*nsize,nsize,d-1)
    mark(w,x+2*nsize,y,nsize,d-1)
    mark(w,x+2*nsize,y+2*nsize,nsize,d-1)
```

Last Name: _____ First: _____ Netid: _____

3. [13 points] **Nested Lists**

Implement the function below according to its specification. You do **not** need to enforce the precondition.

```
def remove_negative_columns(table):
    """MODIFIES this table to remove any negative columns.

    A negative column is one where ALL values in the column are negative

    Example: If table = [[1,-2,-3,4],[5,6,-7,-8],[9,-10,-11,-12]] then
        remove_negative_columns(table) modfies the table to be
        [[1,-2,4],[5,6,-8],[9,-10,-12]]. That is because only the third
        column has all negative numbers with -3, -7, and -11.

    Precondition: table is a non-empty rectangular 2D list of integers"""

    # Modifying the table this way is best done with a while-loop
    # We need to make the loop variable ahead of time
    col = 0

    while col < len(table[0])
        # Accumulator to check that all values are negative
        allneg = True
        # Loop through rows of this column
        for row in range(len(table)):
            if table[row][col] >= 0:
                allneg = False

        # Now delete this column if negative
        if allneg:
            for row in range(len(table)):
                del table[row][col]
        else:
            # Only increment if did not delete column
            col = col + 1

    # No return value
```

Last Name: _____ First: _____ Netid: _____

4. [23 points total] **Call Frames and Name Resolution**

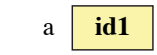
(a) [4 points] Consider the (undocumented) class to the right. **Without drawing call frames**, draw the contents of the heap and global space after the call

```
>>> a = A(5)
```

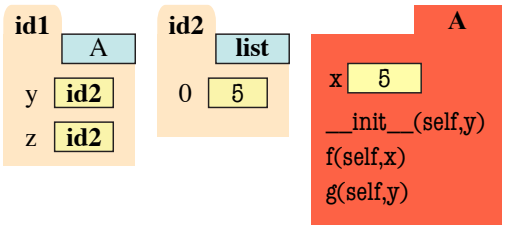
Include the class folder in your answer.

```
1  class A(object):
2      x = 5
3
4      def __init__(self,y):
5          self.y = [y]
6          self.z = self.y
7
8      def f(self,x):
9          if self.x == 0:
10             return self.y[-1]
11         else:
12             return self.g(x)+1
13
14     def g(self,y):
15         self.y = self.y+[self.x]
16         self.x = y
17         return self.f(y-1)+1
18
```

Global Space



The Heap

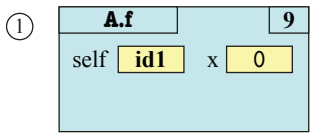


(b) [19 points] Using the object `a` from above, diagram the call

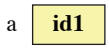
```
>>> b = a.f(0)
```

Use the space below and the next two pages. You will need **eleven diagrams**. Draw the call stack, global space and the heap. If the contents of any space are unchanged between diagrams, you may write *unchanged*. You **you do not need to draw the class folder**, but you should include any other folders from part (a).

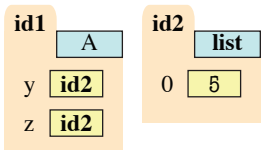
Call Stack



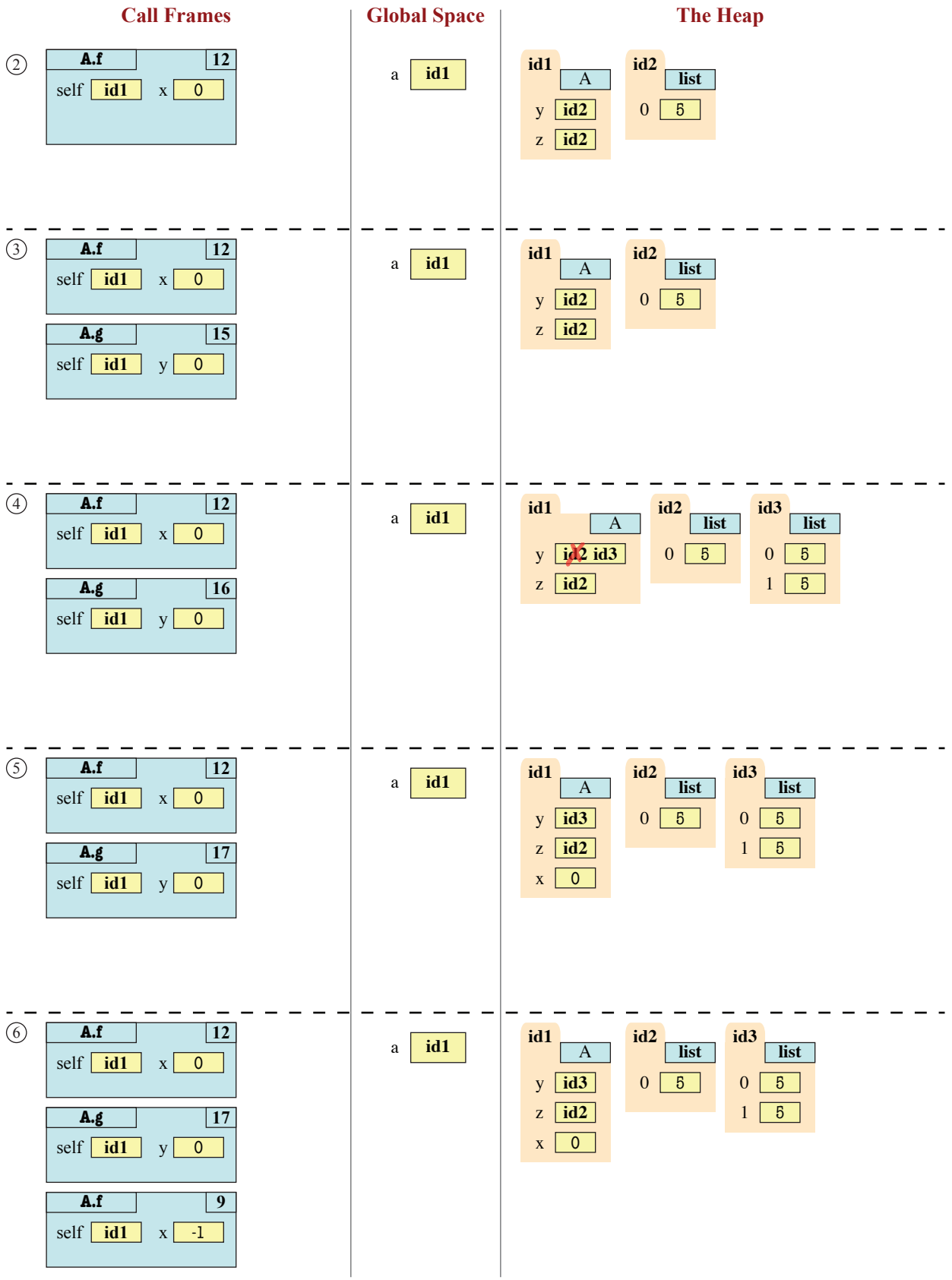
Global Space



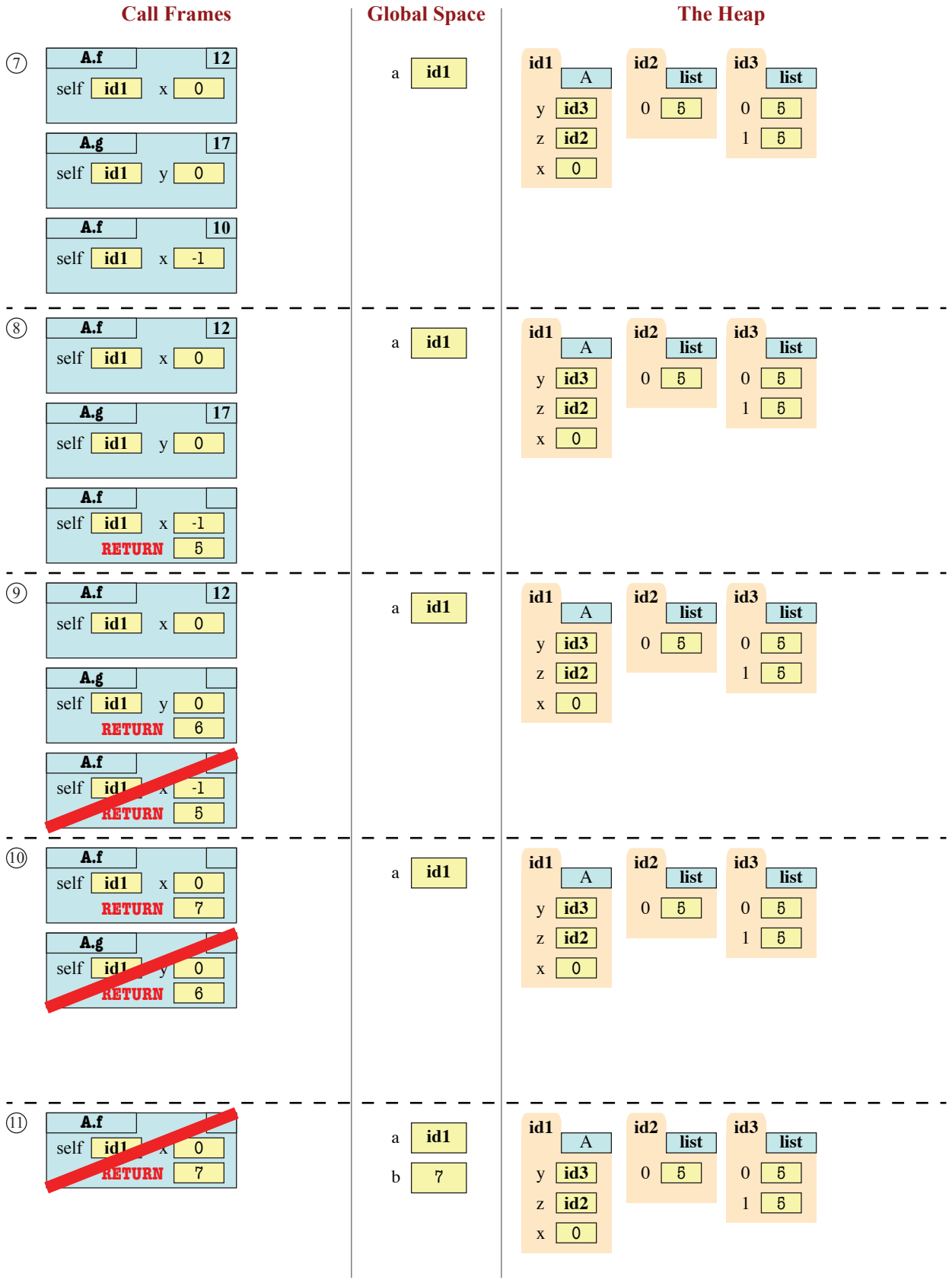
The Heap



Last Name: _____ First: _____ Netid: _____



Last Name: _____ First: _____ Netid: _____



Last Name: _____ First: _____ Netid: _____

5. [14 points total] **Generators**

Implement the generators below using anything from class. However, note the precondition of `moving_avg`. Iterables that are not lists or strings cannot be sliced and have no length. **They also cannot be converted to lists**, as they may be infinite. Do **not** enforce the preconditions.

(a) [6 points]

```
def clamp(nlist,min,max):
    """Generates the elements of nlist clamped to the range [min,max]

    Numbers < min are replaced with min; numbers > max are replaced with max.
    Ex: clamp([0,1,2,3],1,2) generates 1, 1, 2, 2.
    Precond: nlist is a list of numbers, min <= max are numbers"

    for x in nlist:
        if x < min:
            yield min
        elif x > max:
            yield max
        else:
            yield x
```

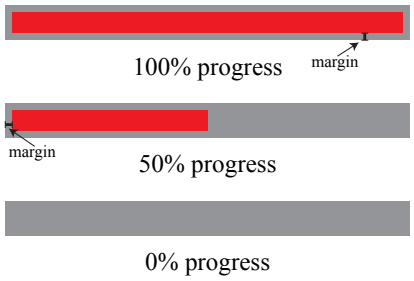
(b) [8 points]

```
def moving_avg(stream,n):
    """Generates the moving average of a stream of numbers

    A moving average uses the most recent n numbers, including the current one.
    Ex: moving_avg([1,2,3,7,9],3) generates 1 (avg of 1), 1.5 (avg of 1, 2),
    2 (avg of 1, 2, 3), 4 (avg of 2, 3, 7) and so on.

    Precond: stream is an iterable of numbers; n is an int > 0"""

    # Create window of the n most recent numbers
    window = []

    for x in stream:
        # See if we need to make room in window
        if len(window) == n:
            del window[0]
        window.append(x)

        # Compute the average of the window
        sum = 0
        for y in window:
            sum = sum + y

        yield sum/len(window)
```

Last Name: _____ First: _____ Netid: _____

6. [15 points] **Classes and Subclasses**

For this question, you will use the classes of Assignment 7 to make a `GProgressBar`. Shown to the right, this is one rectangle nested inside of another to indicate progress towards a goal. The progress is a value between 0 and 1. The size of the progress bar is determined by the outer (grey) rectangle. The inner (red) rectangle, when progress is 1 or 100%, is smaller than the outer one by `margin` on all sides. If the progress is less than one, the width of the inner bar is its maximum width multiplied by progress. But it is always drawn so it is margin away from left side of the outer rectangle.



One easy way to make a GUI element composed of two different objects is to make the class a subclass of one and have the other as an attribute. That is what we have done on the next page. `GProgressBar` is a subclass of `GRectangle` (representing the outer rectangle) but it has a `_bar` attribute for the interior `GRectangle`.

As a reminder, the `GRectangle` class comes with the following attributes.

Attribute	Invariant	Description
`left`	`int or float`	x-coordinate of the left edge of the rectangle.
`bottom`	`int or float`	y-coordinate of the bottom edge of the rectangle.
`width`	`int or float` $\geq$ `0`	The width along the horizontal axis.
`height`	`int or float` $\geq$ `0`	The height along the vertical axis.
`fillcolor`	`str`	The interior color (represented as the name, e.g. `'blue'`).
`linecolor`	`str`	The border color (represented as the name, e.g. `'green'`).

These are the only attributes you need for this problem.

With this in mind, implement this class on the next page. We have provided the specifications for the methods `__init__` and `draw`. You should also add getters and setters (where appropriate) for the new attributes. Those setters must have preconditions to enforce the attribute invariants. Furthermore, all methods (not just the setters) must enforce the preconditions for any value other than `self`.

One of the attributes – `_bar` – is **inaccessible**. This is a stronger restriction than immutable. It means the user can *never* access the attribute, even with a getter. If user could access the attribute with a getter, that user could then change the dimensions of the interior rectangle. This should never happen. The dimensions of the interior rectangle should always be a function of (1) the outer rectangle, (2) the progress amount, and (3) the margin.

Instead, you have the pseudo setter/getter `getBarColor` and `setBarColor`, which are specified for you. These allow you to change the color of the interior rectangle without accessing the interior rectangle directly.

Hint: The attributes in `GRectangle` work like they do in Assignment 7, and have invisible setters and getters. Therefore, you never have to enforce the invariants for these attributes. You only need to worry about your new attributes: `_progress`, `_margin`, and `_bar`. Also, remember that their constructors use keyword arguments (e.g. `GRectangle(fillcolor='red')`).

```
class GProgessBar(GRectangle):
    """A class representing a simple progress bar

    The progress bar is drawn as two rectangles, one inside of the other.
    At progress 1, the interior rectangle is the same size as the outer, minus margin on all
    sides. At progress < 1, the width of the interior rectangle is multipled by progress,
    and the interior rectangle is margin away from the left edge of the outer rectangle.
    The interior rectangle has a single color for both fill and line."""
    # MUTABLE ATTRIBUTE:
    # _progress: the amount of progress; an int or float between 0 and 1 (inclusive)
    # IMMUTABLE ATTRIBUTE:
    # _margin: the distance between the inner and outer bar; an int or float >= 0
    # INACCESSIBLE ATTRIBUTE:
    # _bar: the interior progress bar; a GRectangle with one color for fill and line
    # Because the outer bar can be resized, _bar must be rescaled each time it is drawn

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

    def getProgress(self):
        """Returns amount of progress""
        return self._progress

    def setProgress(self,value):
        """Sets amount of progress""
        assert isinstance(value,int) or isinstance(value,float)
        assert 0 <= value <= 1
        self._progress        = value

    def getMargin(self):
        """Returns the margin spacing"""
        return self._margin



    def getBarColor( self ):                                                    # Fill in
        """Returns the color (fill and line) of the interior bar"""
        return self._bar.fillcolor




    def setBarColor( self,  value ):                                            # Fill in
        """Sets the color (fill and line) of the interior bar

        Parameter value: a string, and not the fillcolor of the outer bar"""
        assert isinstance(value) == str
        assert value != self.fillcolor
        self._bar.fillcolor = value
        self._bar.linecolor = value
```

Last Name: _____ First: _____ Netid: _____

```
# Class GProgessBar (CONTINUED).

def __init__( self,  left,  bottom,  width,  height,  bcolor, margin = 0):     # Fill in
    """Initializes a progress bar with the given parameters.

    The values left, bottom, width, and height define the size of the
    outer rectangle. Both the fill color and line color of the outer
    rectangle is 'grey'. The initial progress value is 0.

    Parameter left:   left edge of outer rectangle; an int or float
    Parameter bottom: bottom edge of outer rectangle; an int or float
    Parameter width:  width of outer rectangle; an int or float >= 0
    Parameter height: height of outer rectangle; an int or float >= 0
    Parameter bcolor: color of interior bar; a string and NOT 'grey'
    Parameter margin: margin of interior bar; an int or float >= 0
    The argument margin is OPTIONAL with default value 0."""
    assert isinstance(bcolor, str) and bcolor != 'grey'
    assert isinstance(margin,int) or isinstance(margin,float)
    assert margin >= 0

    # super().__init__ enforces the other preconditions for us
    super().__init__(left=left,bottom=bottom,width=width,height=height)
    self.linecolor = 'grey'
    self.fillcolor = 'grey'
    self._margin   = margin
    self._progress = 0     # No parameter, so no preconditions to check
    self._bar = GRectangle(left=left+margin,bottom=bottom+margin,
                           width=0,height=height-2*margin)
    self._bar.linecolor = bcolor
    self._bar.fillcolor = bcolor


 def draw( self,  view ):                                                  # Fill in
    """Draws this object to the given view.

    The interior bar is scaled to the progress, and hidden if progress is 0.
    Parameter view: the view to draw to, which is a GView object"""
    # HINT: The user may have altered the position and dimensions of the outer bar.
    # You must recompute the position and dimensions of the inner bar before drawing.

    super().draw(view)                                  # Enforces precondition for us

    # Recompute the size/position of the interior bar
    w = (self.width-2*self._margin)*self._progress
    self._bar.left = self.left+self._margin
    self._bar.bottom = self.bottom+self._margin
    self._bar.width  = w
    self._bar.height = self.height-2*self._margin

    # Draw the interior bar
    self._bar.draw(view)
```

7. [9 points total] **Short Answer**

   (a) [4 points] Describe how we write a function specification in this class. We are looking for you to identify four important parts of the specification (though not every function specification has all of these parts).

   A specification is written as a docstring at the start of the body with the following:

   - This docstring starts with a single line summary (includin the value returned).
   - This is followed by one or more paragraphs giving more detail about the function.
   - Each parameter is identified and described in the specification.
   - There is a precondition for each parameter, identifying what values it can take.

   (b) [3 points] What is a watch? What is a trace? What are they used for?

   A *watch* is a print statement used to display the contents of a variable.
   A *trace* is a print statement used to visualize program flow.
   They are used in debugging (on the next page!) to help us visualize code.

   (c) [2 points] What is the key difference between the quicksort algorithm and the insertion sort algorithm that allows quicksort to be so much faster? You should not need to say more than a sentence here.

   Quick sort uses divide-and-conquer with recursion to speed up sorting.

8. [12 points] **Debugging**

   On the next page is the code for a function `wrap_text`. This function is takes really long text and wraps it around multiple lines. It does this by adding `'\n'` characters for line breaks. Also, when it breaks up lines, any spaces on the end are removed. So if our line width is 3 `'one two'` becomes `'one\ntwo'`. Finally, even if words are not broken up, multiple spaces are collapsed into one to make room. So the string `'a  tv  show'` at width 4 becomes `'a tv\nshow'`. Note that `'a tv'` is four characters including the space.

   Examine the code for `wrap_text` and its helpers on the next page. There are ***at least* four bugs**. To help you find the bugs, we have added several print statements, and show the results on the next few pages. Using this information as a guide, identify and fix the four bugs. **Your fixes may include more than one line of code**. You should explain your fixes.

```
def wrap_text(text, width):
    """Returns text wrapped to fit in width

    Ex: wrap_text('a b c',3) returns 'a b\nc'
    Precond: text a str, width an int > 0"""
    # Remove bad spaces and simplify string
    print('start wrap_text')          # Trace
    simple = collapse(text)
    print('simple: '+repr(simple))    # Watch
    if simple == '':
        return ''

    # Split into words at spaces
    words = text.split(' ')
    print('words: '+repr(words))      # Watch
    curr  = ''
    lines = []

    # Add WATCHES at each if
    for w in words:
        if curr == '':
            curr = w
            print('first word: '+repr(curr))
        elif word_fits(w, curr, width):
            curr = curr + ' ' + w
            print('fits: '+repr(curr))
        elif len(w) > width:
            # Large words by themselves
            lines.append(curr)
            lines.append(w)
            curr = ''
            print('long word: '+repr(w))
        else:
            # Start a new line
            lines.append(curr)
            curr = w
            print('next line: '+repr(curr))

    print('lines: '+repr(lines))     # Watch
    # Add the last line
    lines.append(curr)
    print('lines: '+repr(lines))     # Watch

    result = combine(lines)
    print('end wrap_text')           # Trace
    return result


def combine(lines):
    """Returns string formed from lines.

    Precond: lines a list of nonempty str"""
    print('start combine')           # Trace
    result = ''
    for l in lines:
        if result != '':
            result += '\n'
        result += l

    print('result: '+repr(result))   # Watch
    print('end combine')             # Trace
    return result


def collapse(text):
    """Returns a copy w/ spaces collapsed

    Ex: collapse('a   b  ') return 'a b'
    Precond: text is a str"""
    print('start collapse')          # Trace
    result = ''
    in_space = True

    # Look for spaces in the string
    for ch in text:
        if ch == ' ':
            # Keep the first space we see
            if not in_space:
                result += ' '
                in_space = True
        else:
            result += ch
            in_space = False
    print('result: '+repr(result))   # Watch

    # Take care of any spaces at the end
    if result[-1] == ' ':
        print('trimming end')        # Trace
        result = result[:-1]

    print('end collapse')            # Trace
    return result


def word_fits(word, curr, width):
    """Returns True if word + space fits in line.

    Precond: word, curr are str
    Precond: word is nonempty
    Precond: width is an int > 0"""
    print('start word_fits')         # Trace

    # We are starting a new line
    if curr == '':
        print('end word_fits (1)')   # Trace
        return True

    # Verify width of the word and space
    newlen = len(curr) + len(word)

    print('newlen: '+repr(newlen))
    if newlen <= width:
        print('end word_fits (2)')   # Trace
        return True

    # If we got here, it is too long
    print('end word_fits (3)')       # Trace
    return False
```

Last Name: _____ First: _____ Netid: _____

First Bug:

```
# Expect: 'one two\nfive six'
>>> wrap_text('one two five six',11)
start wrap_text
start collapse
result: 'one two five six'
end collapse
simple: 'one two five six'
words: ['one', 'two', 'five', 'six']
first word: 'one'
start word_fits
newlen: 6
end word_fits (2)
fits: 'one two'
start word_fits
newlen: 11
end word_fits (2)
fits: 'one two five'
start word_fits
newlen: 15
end word_fits (3)
next line: 'six'
lines: ['one two five']
lines: ['one two five', 'six']
start combine
result: 'one two five\nsix'
end combine
end wrap_text
'one two five\nsix'
```

Second Bug:

```
# Expect: 'one\ntwo\nthree'
>>> wrap_text('one two three ',3)
start wrap_text
start collapse
result: 'one two three '
trimming end
end collapse
simple: 'one two three'
words: ['one', 'two', 'three']
first word: 'one'
start word_fits
newlen: 6
end word_fits (3)
next line: 'two'
start word_fits
newlen: 8
end word_fits (3)
long word: 'three'
lines: ['one', 'two', 'three']
lines: ['one', 'two', 'three', '']
start combine
result: 'one\ntwo\nthree\n'
end combine
end wrap_text
'one\ntwo\nthree\n'
```

Explanation and Fix:

What is happening here is that the function `word_fits` thinks that `'one two five'` is 11 characters or less. But it is not with the two spaces. The problem is that, despite the comment on line 106, line 107 does not take the space into consideration in its calculation. You should change line 107 to

```
newlen = len(curr) + len(word) + 1
```

Explanation and Fix:

You can see that the second watch for `lines` on line 42 is adding an empty string to the list. This violates the precondtion of `combine` and so an extra `'\n'` is added to the end. However, the error is in `wrap_text`, not `combine`, so `combine` should not be modified.

There are multiple solutions. One is just to remove lines 27-32 as they are not necessary (`word_fits` already handles long words). But the easiest solution is to change line 41 to

```
if curr != '':
    lines.append(curr)
```

Last Name: _____ First: _____ Netid: _____

Third Bug:

```
# Expect: 'a bb cc'
>>> wrap_text('a   bb cc',7)
start wrap_text
start collapse
result: 'a bb cc'
end collapse
simple: 'a bb cc'
words: ['a', '', '', 'bb', 'cc']
first word: 'a'
start word_fits
newlen: 1
end word_fits (2)
fits: 'a '
start word_fits
newlen: 2
end word_fits (2)
fits: 'a  '
start word_fits
newlen: 5
end word_fits (2)
fits: 'a   bb'
start word_fits
newlen: 8
end word_fits (3)
next line: 'cc'
lines: ['a   bb']
lines: ['a   bb', 'cc']
start combine
result: 'a   bb\ncc'
end combine
end wrap_text
'a   bb\ncc'
```

Explanation and Fix:

You can see from the traces that the problerm is that extra spaces are being added between `'a'` and `'bb'`. The function `collapse` appears to work correctly, but the watch for `words` shows several empty strings (which is what happens when you split on adjacent spaces).
The problem is that we called `collapse` and then forgot to use the result of that function. You should change line 14 to

```
word = simple.split(' ')
```

Fourth Bug:

```
# Expect: ''
>>> wrap_text('  ',4)
start wrap_text
start collapse
result: ''
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exam.py", line 8, in wrap_text
    simple = collapse(text)
             ^^^^^^^^^^^^^^
  File "exam.py", line 85, in collapse
    if result[-1] == '  ':
       ~~~~~~~~~~
IndexError: string index out of range
```

Explanation and Fix:

There is a crash on line 85. This is a problem with `collapse`. The string satisfied the precondition, but it was all spaces. The loop removed all the spaces, producing the empty string. So when we try to check if there is a space at the end, we crash. The solution is to change line 85 to

```
if result != '' and result[-1] = ' '
```