Good practices
that help minimize the need for debugging

#1: Use ***Iterative Enhancement:***

Repeat the following until you have a solution to your problem:

1. Find a ***stage*** for your problem that:
   - is a step toward a solution, and
   - is a SMALL step, and
   - can be TESTED.
2. ***Solve the stage and TEST*** your solution to it. Don't proceed until you get this stage working.

#2: ***Break a problem into sub-problems.***

Write AND TEST separate functions for the sub-problems.

#3: ***Keep patterns in mind.***

Don't reinvent the wheel.

#4: ***Maintain intellectual control of your program.***

Techniques to do so include using ***descriptive names***, sparse but well-chosen ***internal comments***, and good use of ***white space***.

The next slides explain each of these practices.

#1: Use ***Iterative Enhancement:***

Repeat the following until you have a solution to your problem:

1. Find a ***stage*** for your problem that:
   - is a step toward a solution, and
   - is a ***SMALL step***, and
   - ***can be TESTED***.
2. ***Solve the stage and TEST*** your solution to it. Don't proceed until you get this stage working.

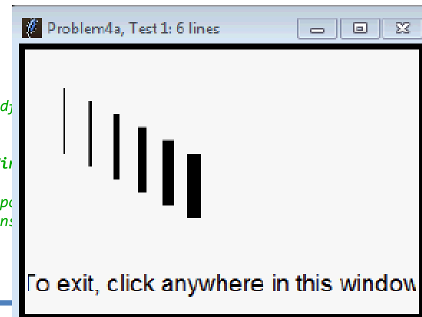
How would you apply ***Iterative Enhancement*** to this Session 7 problem? (Answer on next slide.)

```
def problem4a(window, point, n):
    """
    See   problem4a_picture.pdf   in this project for pictures that may
    help you better understand the following specification:

    Draws a sequence of  n  rg.Lines on the given
    rg.RoseWindow, as follows:
    -- There are the given number (n) of rg.Lines.
    -- Each rg.Line is vertical and has length 50.
          (All units are pixels.)
    -- The top of the first (leftmost) rg.Line
          is at the given rg.Point.
    -- Each successive rg.Line is 20 pixels to the right
          and 10 pixels down from the previous rg.Line.
    -- The first rg.Line has thickness 1.
    -- Each successive rg.Line has thickness 2 greater than
       the zg.Line to its left, but no greater than 13.
       (So once a rg.Line has thickness 13, it and all
        the rg.Lines to its right have thickness 13.)

     Returns the sum of
     the thicknesses of
     the rg.Line's.
     (See  problem4a_picture.pdj

    Preconditions:
        :type window: rg.RoseWi
        :type point: rg.Point
      The third argument is a po
      and the given point is ins
    """
```



#1. Use *Iterative Enhancement:*

Repeat the following until you have a solution to your problem:

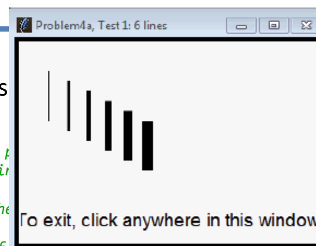
1. Find a ***stage*** for your problem that:
   - is a step toward a solution, and
   - is a SMALL step, and
   - can be TESTED.
2. ***Solve the stage and TEST*** your solution to it. Don't proceed until you get this stage working.

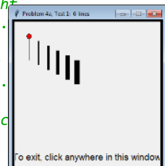
How would you apply
Iterative Enhancement to this Ses

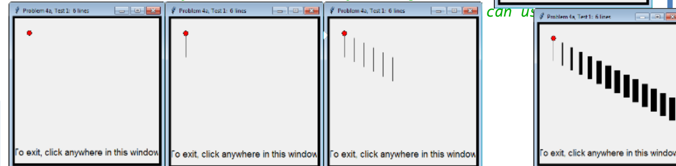
```
def problem4a(window, point, n):
    """
    See   problem4a_picture.pdf   in this p
    help you better understand the followin

    Draws a sequence of  n  rg.Lines on the
    as follows:
      -- There are the given number (n) of
      -- Each rg.Line is vertical and has length 50.
            (All units are pixels.)
      -- The top of the first (leftmost) rg.Line
            is at the given rg.Point.
      -- Each successive rg.Line is 20 pixels to the right
            and 10 pixels down from the previous rg.Line.
      -- The first rg.Line has thickness 1.
      -- Each successive rg.Line has thickness 2 greater
         the zg.Line to its left, but no greater than 13.
           (So once a rg.Line has thickness 13,
           it and all the rg.Lines to its right have thic

    Returns the sum of the thicknesses of the rg.Line's.
                                                can us
```







Answer: Here is one ***Iterative Enhancement Plan***. The key is to get each stage TESTED and WORKING before continuing to the next stage.

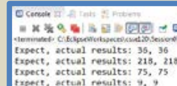
Stage 1: A test window appears, with your test point drawn on the window. (Remove that point when you finish the problem.)

Stage 2: The first line is drawn successfully, at the right place.

Stage 3: N lines are drawn successfully, where your test has (say) N=6 lines. All the same width at this stage.

Stage 4: The line widths increase by 2 per line.

Stage 5: The line widths don't increase past 13.



Stage 6: The sum of the thicknesses is computed and returned.

#2: Break a problem into sub-problems.

Write AND TEST separate functions for the sub-problems.

```
def keep_integers(list_of_lists):
    """
    Given a list of sub-sequences, returns a list
    that contains only the sub-sequences
    that contain ONLY integers.  For example,
    if the given argument is:
         [(3, 1, 4),
          (10, 'hi', 10),
          [1, 2.5, 3, 4],
          'hello',
          [],
          ['404'],
          [30, -4]
         ]
     then this function returns:
         [(3, 1, 4),
          [],
          [30, -4]
         ]
```

Has non-integers, so don't keep it

Has non-integers, so don't keep it

Has non-integers, so don't keep it

Has non-integers, so don't keep it

What would be a reasonable ***sub-problem*** for the problem to the right? (It is a variation of a problem from Session 16.)

(Answer on next slide.)

#2. Break a problem into sub-problems.

Write AND TEST separate functions for the sub-problems.

```
def keep_integers(list_of_lists):
    """
    Given a list of sub-sequences, returns a list
    that contains only the sub-sequences
    that contain ONLY integers.  For example, if the
    given argument is:
        [(3, 1, 4),
         (10, 'hi', 10),
         [1, 2.5, 3, 4],
         'hello',
         [],
         ['oops'],
         [30, -4]
        ]
     then this function returns:
        [(3, 1, 4),
         [],
         [30, -4]
```

Has non-integers, so don't keep it

What would be a reasonable ***sub-problem*** for the problem to the right? (It is a variation of a problem from Session 16.)

Answer: One way to solve this problem is to:

- loop through the list and, for each sub-sequence,
- ***determine if the sub-sequence has any non-integers in it***.

It would be reasonable to make the latter a sub-problem of its own (in its own function). Here is a full solution, with the ***"helper" function*** on the right.

```
def keep_integers(list_of_lists):
    answer = []
    for k in range(len(list_of_lists)):
        if is_all_integers(list_of_lists[k]):
            answer.append(list_of_lists[k])

    return answer
```

```
def is_all_integers(sequence):
    """ Returns True if the given
        sequence contains
        only integers. """
    for k in range(len(sequence)):
        if type(sequence[k]) != int:
            return False

    return True
```

Using the FIND pattern

#3. Keep ***patterns*** in mind. Don't reinvent the wheel.

- From Session 3 et al: ***Looping through a RANGE with a FOR loop***.
  - ***range(m)*** – goes ***m*** times (from ***0*** to ***m-1***, inclusive)
  - ***range(m, n+1)*** – goes from ***m*** to ***n***, inclusive (does NOT include n+1)
- From Session 3 et al: ***Using objects***.
  - ***Constructing*** an object
  - Applying a ***method***
  - Referencing a ***data attribute***
  - Using the ***DOT trick*** (and what to do when it seems not to work)
- From Session 4 et al: ***Calling functions***, including ***functions defined within*** the module
- From Session 6: The ***Accumulator Pattern***, in:
  - ***Summing***:
    ```
    total = total + number
    ```
  - ***Counting***:
    ```
    count = count + 1
    ```
  - ***Graphics***:
    ```
    x = x + pixels
    ```

- From various sessions: the ***SWAP pattern***:
  ```
  temp = a
  a = b
  b = temp
  ```
- From various sessions: Introducing an ***auxiliary variable*** that works within a FOR or WHILE loop.
- From Session 9: ***Waiting for an Event*** (using a WHILE loop with an IF statement and BREAK)
- From Session 11: ***Accumulating a sequence***.
- From Session 11: ***Patterns for iterating through sequences***, including:
  - Beginning to end
  - Other ranges (e.g., backwards and every-3rd-item)
  - The ***COUNT/SUM/etc*** pattern
  - The ***FIND*** pattern (via LINEAR SEARCH)
  - The ***MAX/MIN*** pattern (in a number of variations)
  - Looking ***two places*** in the sequence ***at once***
  - ***Looking at two sequences in parallel***
- From Session 12: ***Mutating*** a list or object, and ***TESTING whether the mutation worked*** correctly.

#4. ***Maintain intellectual control of your program.***

Techniques to do so include using ***descriptive names***, sparse but well-chosen ***internal comments***, and good use of ***white space***.

For example:

```
def index_of_largest_number(numbers, n):
    """ ... """
    index_of_largest = 0  # using max/min pattern
    for k in range(1, n):
        if numbers[k] > numbers[index_of_largest]:
            index_of_largest = k

    return index_of_largest
```

Short internal comment indicating the pattern used ***(keep these sparse!)***

Single blank line to separate the "chunks" of the function from each other (but ***two*** blank lines between function ***definitions***)

Names:

- Use *plurals* for names of sequences (`numbers`).
  Use *singular* for non-sequence items (**`circle`** or **`circle1`** vs **`circles`**).
- The name might indicate the type of the object (`index_of_largest`, makes it clear that this is an INDEX)
- The name certainly should indicate WHAT it stands for (so `upper_left_corner` instead of just `point`)
- j, k, and i for index variables (this practice goes back over 60 years!)
- m, n for integers (and perhaps x for floats) for which no better name is easily available

Review: ***Good practices***
that help minimize the need for debugging

#1: Use ***Iterative Enhancement:***

Repeat the following until you have a solution to your problem:

1. Find a ***stage*** for your problem that:
   - is a step toward a solution, and
   - is a SMALL step, and
   - can be TESTED.
2. ***Solve the stage and TEST*** your solution to it. Don't proceed until you get this stage working.

#2: ***Break a problem into sub-problems.***

Write AND TEST separate functions for the sub-problems.

#3: ***Keep patterns in mind.***

Don't reinvent the wheel.

#4: ***Maintain intellectual control of your program.***

Techniques to do so include using ***descriptive names***, sparse but well-chosen ***internal comments***, and good use of ***white space***.