Today: Debugging 1-2-3, debug printing, string upper/lower, string case-sensitive, reversed, movie example, grid, grid testing, demo HW3

Will CS106A Get Harder and Harder Each Week?

Is this course going to get harder and harder each week? Mercifully, no! We're actually going to settle down a little from here on out.

After You Find The Bug, It Looks Obvious

Hang in there! You can spend hours fiddling with some line to get it to work. That's what everybody is doing. You are building up computer-code skills that will be useful for all sorts of things. We do not know a quicker way to do this!

We're going to weave together two things below - show some string algorithms, but also show some debugging techniques.

Debugging 1-2-3

To write code is to see a lot of bugs. We'll mention the 3 debug techniques here, and do some concrete examples of two of these below.

For more details, see the Python Guide Debugging chapter

Debug Technique-1. Look at the exception

Read the exception message and line number. Read the exception text (which can be quite cryptic looking), from the bottom up. Look for the first line of code that is your code and read the error message which can be quite helpful. Go look at the line of your code. Many bugs can be fixed right there, just knowing which line caused the error.

Debug Technique-2. Look at the output/got + code

Don't ask "why is this not working?". Ask "why is the code producing this output?".

The code and the output are not moving around - they are static, just sitting there. Look at the first part of the output which is wrong. What line of the code produced that?

This can work well with Doctests. Run the Doctest and you have the code, the input, and the output all to work with. It can also be handy to write a **small** Doctest.

We talked about writing a simple, obvious test case as a first Doctest. e.g. for the alpha_only() function, an input like `'@Ax4**y'`, looking for output `'Axy'`. That's fine. For debugging, sometimes it's nice to add a small test that still shows the bug, maybe `'@A'` - the loops and everything run so few times, there's less chaos to see through.

Look at the output. Look at the code that produced it, tracing through through the code in your mind. Sometimes "trace through it with your mind" is too hard! In that case try print() below.

Debug-3.
Add print() in the code

Instead of tracking the code in your head, add print() to see the state of the variables. It's nice to just have the computer *show* the state of the variables in the loop or whatever. This works on the experimental server and in PyCharm - demo below.

Note that **return** is the formal way for a function to produce a result. The print() function does not change that. Print() is a sort of side-channel of text, alongside the formal result. We'll study this in more detail when we do files.

The experimental server shows the function result first, then the print output below. This can also work in a Doctest. Be sure to remove the print() lines when done - they are temporary scaffolding while building.

Upper vs. Lower ,`'A'` vs. `'a'`

The chars `'A'` and `'a'` are two different characters.

```
>>> 'A' == 'a'
False
>>> s = 'red'
>>> s == 'red'    # two equal signs
True
>>> s == 'Red'    # must match exactly
False
```

Case Sensitive

Treating upper/lower as different is called "case sensitive". Case-sensitive is the default behavior within computers. If we ask you to write some string code, and don't say anything about upper/lower case, assume it's supposed to be case-sensitive.

String Functions

`s.upper() s.lower() s.isupper() s.islower()`

- In the Roman A-Z alphabet, each alpha char has lower and upper forms:
  `'a'` is the lowercase form of `'A'`
  `'A'` is the uppercase form of `'a'`
- `s.upper()` - returns uppercase form of s
- `s.lower()` - returns lowercase form of s
- **Immutable**: `s.upper()` returns a new, converted string
  The original string s is unchanged
- `s.isupper()` - True if made of uppercase chars
- `s.islower()` - True if made of lowercase chars
- A char with no upper/lower difference, e.g. `'@'` or `'2'`
  Returned unchanged by upper()/lower()
  isupper()/islower() return False

```
>>> 'Kitten123'.upper()  # return with all
chars in upper form
'KITTEN123'
>>> 'Kitten123'.lower()
'kitten123'
>>>
>>> 'a'.islower()
True
>>> 'A'.islower()
False
>>> 'A'.isupper()
True
>>> '@'.islower()
False
>>> 'a'.upper()
'A'
>>> 'A'.upper()
'A'
>>> '@'.upper()
'@'
```

Immutable String - Not Changed By Functions

Functions like `s.upper()` **return** a new answer string, but the original string is always left unchanged.

```
>>> s = 'Hello'
>>> s.upper()     # Returns uppercase form of s
'HELLO'
>>>
>>> s             # Original s unchanged
'Hello'
>>>
>>> s + '!!!'     # Returns + form
```

```
'Hello!!!'
>>>
>>> s             # Original s unchanged
'Hello'
>>>
```

Working With Immutable String: `x = change(x)`

This is easy enough to solve. Each time we call a function to compute a changed string, use = to change the variable, say `s`, to point to the **new** string.

Say we have a string s, and want to change it to be uppercase and have `'!!!'` at its end. Here is code that works to change s:

```
>>> s = 'Kitten'
>>> s = s.upper()  # compute upper, assign
back to s
>>> s
'KITTEN'
>>> s = s + '!!!'
>>> s
'KITTEN!!!'
```

Not Case Sensitive Logic

Suppose I want to compute something not case-sensitive.

- Also known as "case insensitive" logic
- Doing logic on strings, treating upper/lower chars the same
- e.g. Find `'dog'` in a string, case insensitive
  So `'Dog'` or `'dog'` or `'DOG'` all count
  e.g. When you search in a web page for `'dog'`
  You expect `'Dog'` to count as a match
- There is a straightforward way to do this:

- Convert the strings to lowercase first
- Then do the logic on the lowercase form
- By default, computer code is case sensitive
- If we want you to write case insensitive code
  We will say so explicitly in the problem statement

Example: only_ab()

only_ab(): Given string s. Return a string made of the 'a' and 'b' chars in s. The char comparisons should not be case sensitive, so 'a' and 'A' and 'b' and 'B' all count. Use the string .lower() function.

> only_ab()

Strategy: write the code using `s[i].lower()` to look at the lowercase form of each char in s.

only_ab(s) - v1

Here is our v1 attempted solution code. Use `s[i].lower()` to look at the lowercase version of each input char.

```
def only_ab(s):
    result = ''
    for i in range(len(s)):
        if s[i].lower() == 'a' or s[i].lower()
== 'b':
            result += s[i].lower()
    return result
```

Debug Technique-2 - Look at output/got + code

The got/output is like

```
only_ab('aABBccc') -> 'aabb'

expected output: 'aABB'
```

Look at the output.

Key question: Where does the output first go wrong? What line of the code writes that?

Look at that line. Think about how it could produce the observed output.

`reversed()` Function

The Python built-in reversed() function: return reversed form of a sequence such as from range().

Here is how reversed() alters the output of range():

```
range(5) -> 0, 1, 2, 3, 4
reversed(range(5)) -> 4, 3, 2, 1, 0
```

This fits into the regular for/i/range idiom to go through the same index numbers but in reverse order:

```
for i in reversed(range(5)):
    # i in here: 4, 3, 2, 1, 0
```

For more detail, see the guide Python range()

The reversed() function appears in part of homework-3.

(optional) Reverse String

Say we want to compute the reversed form of a string:

```
'Hello' -> 'olleH'
```

There are many ways to do this, and we might make a study of it later. For now, see if you can write the reverse2() version of the problem, using the reversed() function.

Plan: e.g. with `'Hello'`. Go through the index numbers in reverse order: 4, 3, ... So in the loop, `s[i]` will be in reverse order: `'o'`, then the `'l'` and so on. Use these to build the solution.

> reverse2()

There's actually a whole section of reverse string problems we may play with later - trying out various techniques.

Reverse Debug with print()

Now we'll show Debug-3 technique: add print() inside the code temporarily to get visibility into what it's doing. This works on the experimental server and in Doctests. The printing will make the Doctests fail, so it should only be in there temporarily.

reverse2() Solution With print()

Here's the reverse2() code with print() added

```
def reverse2(s):
    result = ''
    for i in reversed(range(len(s))):
        result += s[i]
        print(i, s[i], result)
    return result
```

Heres's what the output looks like in the experimental server - it shows the formal result first, and the print() output below that. This is kind of beautiful, revealing what's going on inside the loop:

```
'olleH'

4 o o
3 l ol
2 l oll
1 e olle
0 H olleH
```

Debug Print vs. Thought Experiment

So if you have a bug and the code is not computing what it's supposed to. Well first you just look at the output and try to just see what the bug is. That works sometimes. In your mind, you are thinking about what `s[i]` is going to be for each loop - a so called thought experiment.

But if you are stuck staring at the code, you could put some print() calls in there and it will just show you exactly what `s[i]` is for each run of the loop. This can be very clarifying technique if you are not spotting the bug at first. Like instead of using your brain to think what's going on with `s[i]`, just let the computer do it.

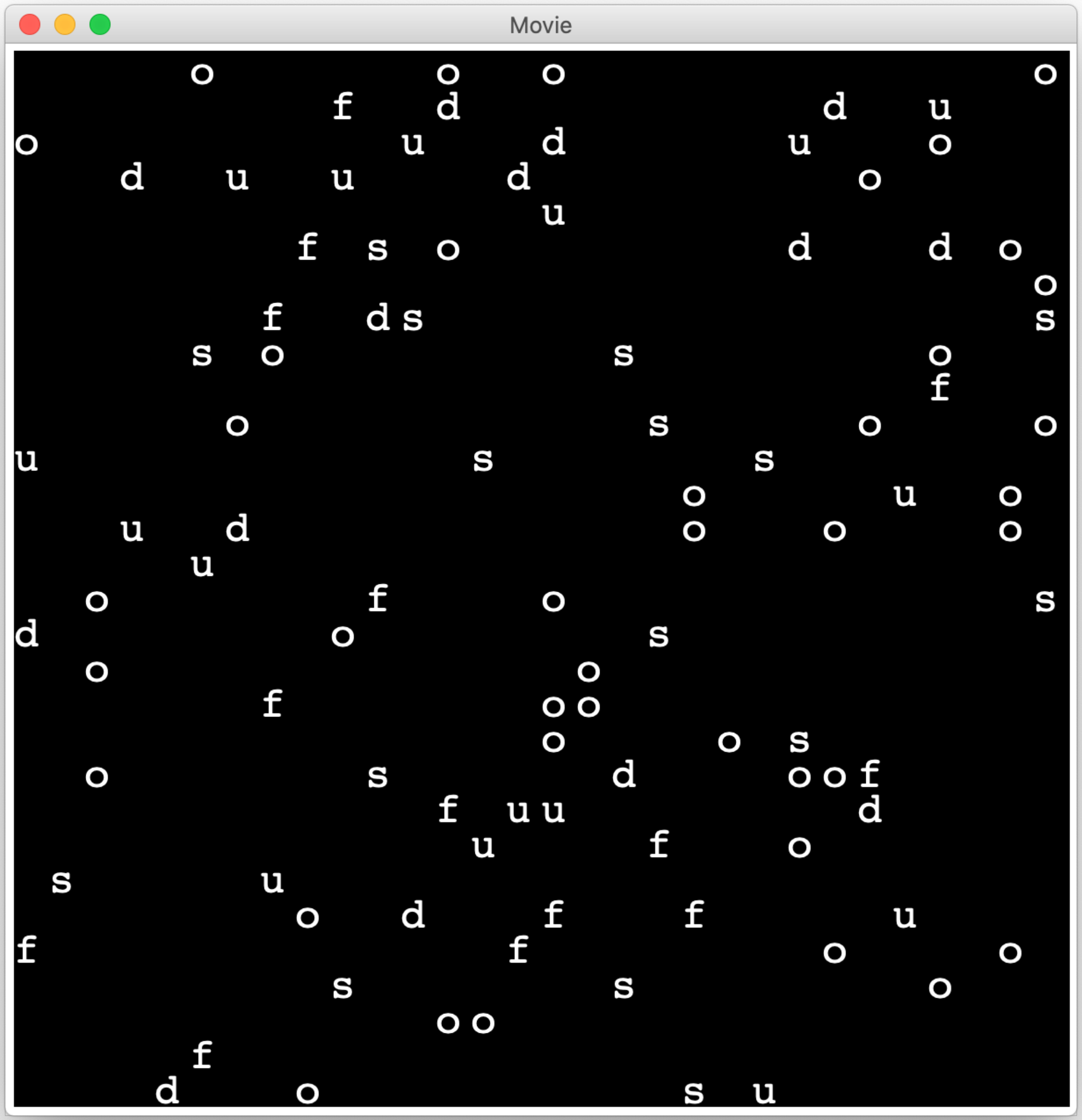
This works with Doctests too - the printed output appears in the Doctest window. Unfortunately, the printed output will make the Doctest fail, even if the code is correct.

Remove the print() code when you are done. It's a sort of temporary scaffolding and it will make all Doctests fail.

Today we will use this "movie" example and exercise: movie.zip

The movie-starter.py file is the code with bugs, and movie.py is a copy of that to work on, and movie-solution.py has the correct code.

Movie Project + Testing Themes



- Goal: we want an animation where letters fly leftwards on a black background
  The world is 90% blank
  10% random letters from word `'doofus'`
  Kind of like The Matrix
- Divide and Conquer - our mantra
- Test each function separately - our other mantra

- Don't debug the animation as it runs
- Debug a tiny, frozen Doctest case
- Huge time saver
- Today: Doctest for a 2d algorithm function

Recall: Grid

- Reference: Grid Reference
- `grid = Grid(4, 3)` - create, all None initially
- Zero based x,y coordinates for every square in the grid:
  origin at upper left
  x: 0..grid.width - 1
  y: 0..grid.height - 1
- `grid.width` - access width or height
- `grid.get(0, 0)` - returns contents at x,y (error if out of bounds)
- `grid.set(0, 0, 'a')` - set at x,y
- `grid.in_bounds(2, 2)` - returns True if x,y is in bounds

Example: set_edges() (optional)

Implement set_edges(), then write Doctests for it. We're not doing this in lecture, but it's an example.

```
def set_edges(grid):
    """
    Set all the squares along the left edge
 (x=0) to 'a'.
    Do the same for the right edge.
    Return the changed grid.
    """
    pass
```

Solution code:

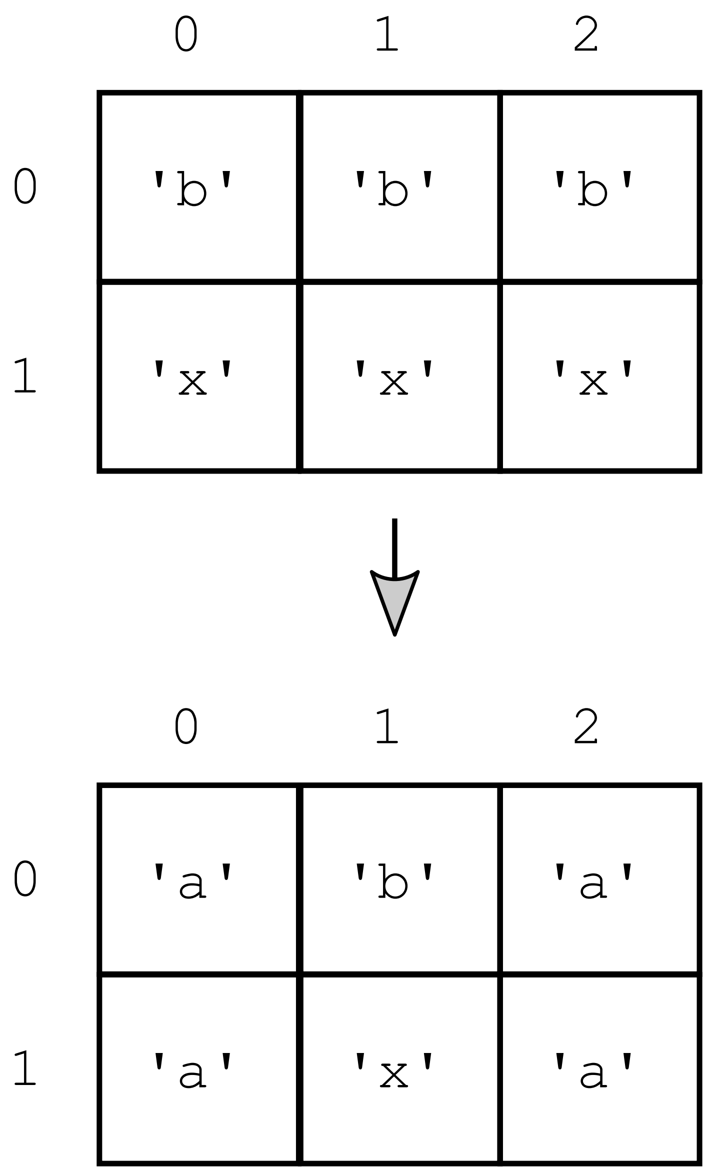
```
def set_edges(grid):
    """
    Set all the squares along the left edge
(x=0) to 'a'.
    Do the same for the right edge.
    Return the changed grid.
    """
    for y in range(grid.height):
        grid.set(0, y, 'a')
        grid.set(grid.width - 1, y, 'a')
    return grid
```

Q: How can we tell if that code works? With our image examples, at least you could look at the output, although that was not a perfect solution either. Really we want to be able to write test for a small case with visible data.

Write Test for set_edges()

- Write a test for set_edges()
- Literal format used to create and check grid values
- Can set a variable within the >>> Doctest, use on later line
  `>>> grid = Grid.build([['b', 'b', 'b'], ['x', 'x', 'x']])`

Here's a visualization - before and after - of grid and how set_edges() modifies it.



Here are the key 3 lines added to set_edges() that make the Doctest: (1) build a "before" grid, (2) call fn with it, (3) write out the expected result of the function call

```
        ...
        >>> grid = Grid.build([['b', 'b', 'b'],
  ['x', 'x', 'x']])
        >>> set_edges(grid)
        [['a', 'b', 'a'], ['a', 'x', 'a']]
        ...
```

Run Doctest in PyCharm

- Look at set_edges() in PyCharm
- Select the ">>>" Doctest
  Right click it .. Run Doctest
- Maybe have to click a 2nd time for PyCharm to recognize the test
- If the Run Doctest is not present in the menu
  You may need to close PyCharm and open the "movie" **folder** using the PyCharm open... menu
- With the set_edges() code correct, the test should pass
- (optional) Can try putting in a bug

Pseudo Random Numbers

"Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."
-John von Neumann (early CS giant)

Deterministic

A computer program is "deterministic" - each time you run the lines with the same input, they do exactly the same thing. Creating random numbers with deterministic code and inputs is impossible, so we settle for **pseudo random** numbers. These are numbers which are statistically random looking, but in fact are generated by a deterministic algorithm producing each "random" number in turn.

Aside: it is possible to create true random numbers by measuring a random physical process - getting the randomness from outside the determinism of the computer.

Aside: How To Get an Interpreter >>>

For more details, see the Python Guide Interpreter chapter

Ways to get an interpreter (apart from the experimental server)

1. With an open PyCharm project, click the "Python Console" button at the bottom .. that's an interpreter.

2. In the command line for your computer, type "python3" ("py" on Windows), and that runs the interpreter directly. Use ctrl-d (ctrl-z on windows) to exit.

Keep in mind that there are two different places where you type commands - your computer command line where you type commands like "date" or "pwd". Then there's the Python interpreter with the >>> prompt where you type python expressions.

The Random Module

- A "module" is a library of code we want to use
- Python has many built-in modules containing useful functions
- More module information later in the quarter
- Here the "random" module
- `import random` - this line once at the top of your file
- `random.randrange(n)` - returns 0..n-1 at random, uniformly distributed
- `random.choice('string')` - returns 1 char at random, uniformly distributed

Try random module in the "Python Console" tab at the lower-left of your PyCharm window to get an interpreter. This won't work right in the experimental server interpreter, so try Pycharm.

```
>>> import random   # hw3 starter code has
this already
>>>
>>> random.randrange(10)
1
>>> random.randrange(10)
3
>>> random.randrange(10)
9
>>> random.randrange(10)
1
>>> random.randrange(10)
```

```
8
>>> random.choice('doofus')
'o'
>>> random.choice('doofus')
'u'
>>> random.choice('doofus')
'o'
>>> random.choice('doofus')
'o'
>>> random.choice('doofus')
's'
>>> random.choice('doofus')
's'
>>> random.choice('doofus')
'o'
>>> random.choice('doofus')
's'
```

random_right() Function

The code for this one is provided to fill in letters at the right edge. Demonstrates some grid code for the movie problem. We're not testing this one - testing random behavior is a pain, although it is possible.

```
def random_right(grid):
    """
    Set the right edge of the grid to some
    random letters from 'doofus'.
    (provided)
    """
    for y in range(grid.height):
        if random.randrange(10) == 0:
            char = random.choice('doofus')
            grid.set(grid.width - 1, y, char)
    return grid
```

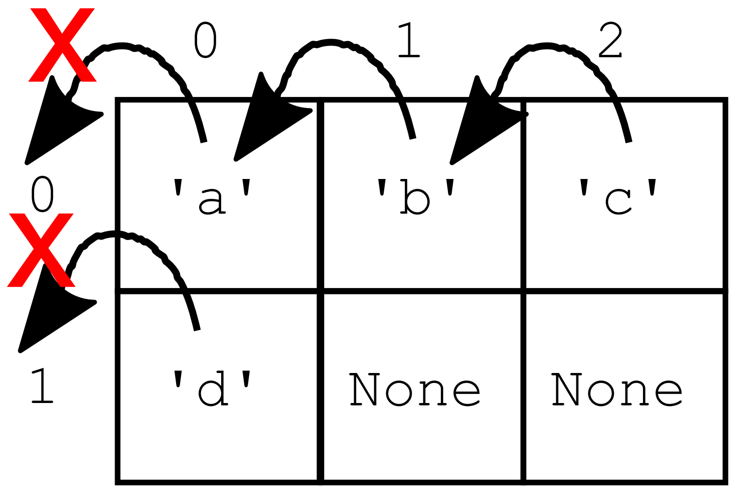
scroll_left(grid)

- The algorithmic core of this project
- Kind of tricky
- The Doctests are going to save us here
- First sketch out the idea without code
- Then we'll work out the code for it

Scroll Ideas

- For every x,y
- Want to "move" the `'b'` or `'c'` or whatever one to the left, e.g. its x-1
- Don't move `'a'` or `'d'` since its x-1 is out of bounds
- Don't move `None` which is 90% of squares
- Note: use drawing vs. "in your head" - a lot of detail here

Think about scroll_left()



scroll_left() Plan

- Write some code in scroll_left() - version 1
- Move square like `'b'` to its `x - 1`
- Don't move the `'a'`, there is no square to its left
- Version 1 shown below
- Has some bugs

scroll_left() v1 with bugs

```
def scroll_left(grid):
    """
    Implement scroll_left as in lecture notes.
    """
    # v1 - has bugs
    for y in range(grid.height):
        for x in range(grid.width):
            # Move letter at x,y leftwards
            val = grid.get(x, y)
            if val != None and
grid.in_bounds(x - 1, y):
                grid.set(x - 1, y, val)
    return grid
```

- Run this in the full GUI. It's buggy, but at least it's funny.
- Observe: small bugs can create big output effects
  Your output may be totally haywire
  But the bug may just be a -1 somewhere
- Running the whole program .. not a good way to debug
- Want: small, frozen, visible test case to look at - Doctest

Make Test Case - Input and Expected

Need concrete cases to write Doctest. They can be small! An input grid, and the expected output grid. That's what makes one test case - an input and expected. We could also call these "before" and "after" pictures.

Doctest input grid (before)

	0	1	2
0	'a'	'b'	'c'
1	'd'	None	None

```
[['a', 'b', 'c'], ['d', None, None]]
```

Doctest expected grid (after)

	0	1	2
0	'b'	'c'	None
1	None	None	None

```
[['b', 'c', None], [None, None, None]]
```

Debug scroll_left() With Doctests

Run the Doctest to debug the code.

Key: look at the **got**, and the and **code** which produced it. Sometimes you can see the problem just looking at these. We'll keep working on the code with this test until it is perfect.

```
def scroll_left(grid):
    """
    Implement scroll_left as in lecture notes.
    >>> grid = Grid.build([['a', 'b', 'c'],
['d', None, None]])
    >>> scroll_left(grid)
    [['b', 'c', None], [None, None, None]]
    """
```

Work on scroll_left()

How do you debug a function? Run its small, frozen Doctests, look at the **got** and the **code** - all of which the Doctest prints out.

- The bug has to do with blanking out a square copying from
- Also dealing with x=0
- **Look at the got output** from the Doctest
- Then look at your code .. often that's enough
- Fix the code in lecture. Need to blank out moved-from squares.
- Fix-1. Add within if: `grid.set(x, y, None)`
- Fix-2. Un-indent above, so outside the if
- Instead of in_bounds(), checking `x > 0` would work too
  Since we are only worried about going off the left edge

scroll_left() Solution

Here is the code with bugs fixed and the Doctest now passes.

```
def scroll_left(grid):
    """
    Implement scroll_left as in lecture notes.
    >>> grid = Grid.build([['a', 'b', 'c'],
['d', None, None]])
    >>> scroll_left(grid)
    [['b', 'c', None], [None, None, None]]
    """
    for y in range(grid.height):
        for x in range(grid.width):
            # Move letter at x,y leftwards
            val = grid.get(x, y)
            if val != None and
grid.in_bounds(x - 1, y):
                grid.set(x - 1, y, val)
            grid.set(x, y, None)
    return grid
```

Run Movie

- Then run the movie program again
- Can specify width/height numbers, 30 30 is the default

```
$ python3 movie.py
$
$ python3 movie.py 80 40   # bigger window
```

Testing - Strategy Conclusions

- Run whole program, see a bug
- A small bug can still produce big, crazy output
- Hard to debug with all the code at once

- Write a Doctest per function
- Doctest gives you what you need, no extra code:
- 1. Small
- 2. Frozen
- 3. Visible
- Look at the test case **got** to debug the code
- Note debugged scroll_left() with a tiny 3x2 test case
- Once the code worked for the tiny case, the whole big program worked perfectly

Demo: HW3 Sand Program

- Watching a demo of the program
  looks like a lot of work
- This program is very decomposable
- It's 4 functions
- Each function with its own Doctests
- We provide many Doctests, you write some
- It's possible it will work the first time you run all your code
  Doctests FTW
- This thing is a little fun to play with once done
- Even your parents can understand what it does!
- Then try to explain Doctests and grid literals to them!