Midterm Prep Guide [READ ME!]

These practice midterms contain a variety of practice problems. For this review session, feel free to work on whatever you'd like to.

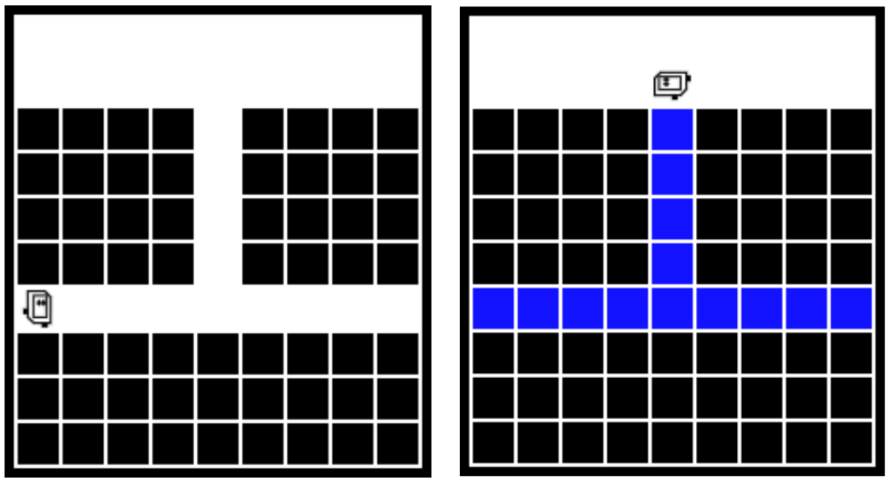
Note that I have not provided starter code for these problems — this is because I want you to practice writing answers without being able to run code (just like how you won't be able to run code during your exam).

Don't forget that there are also practice problems from each ACE section, the experimental server, and normal section problems, that you're welcome to work on.

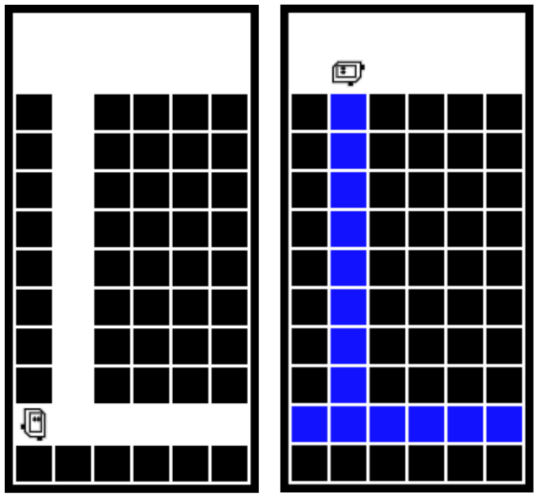
Bit Practice

Bit Program #1: Fill Well

Bit is at the bottom of a "well", and its job is to fill the "well" with "water" by painting each of the squares blue. Bit will start at the bottom-left corner of the "well" and should end above the well. Your code should work for different-sized worlds, where the vertical part of the well is always at least one square away from the edge of the world. (You should define your own helper functions to decompose the code!)



Pre/post-conditions

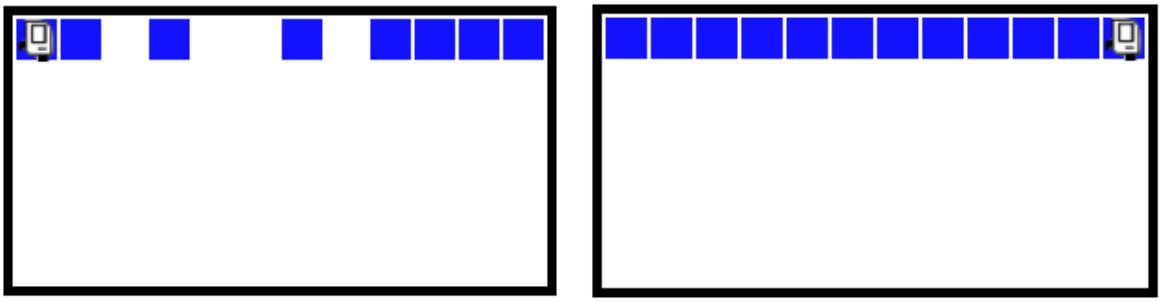


Pre/post-conditions in another world

```
def fill_well(filename):
    bit = Bit(filename)
    # your code here!
```

Bit Program #2: Repair Ceiling

Bit's job is to repair any "holes" in the ceiling. Bit begins in the top left corner of the world, facing right. Most of the "ceiling" (top row) is painted blue, but there are some holes. When Bit comes across these holes, it should paint them blue. Bit should end in the top right corner of the world, facing right. Write a function that accomplishes this task.

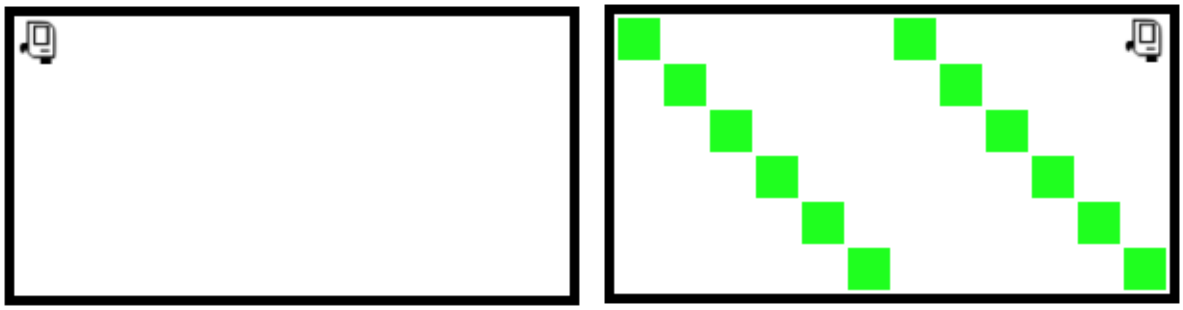


Pre/Post conditions for the world

```
def repair_ceiling(filename):
    bit = Bit(filename)
    # your code here!
```

Bit Program #3: Diagonal Stripes

Bit's job is to make green diagonal stripes. Beginning in the top left corner, Bit should make a series of diagonal green stripes down and to the right. Each stripe should be staggered; that is, a stripe should begin one column to the right of the last square in the previous stripe (no overlaps). You may assume that the width of the world will be a multiple of the height of the world; that is, every stripe will reach from the top to the bottom of the world. Bit should end in the top right corner.

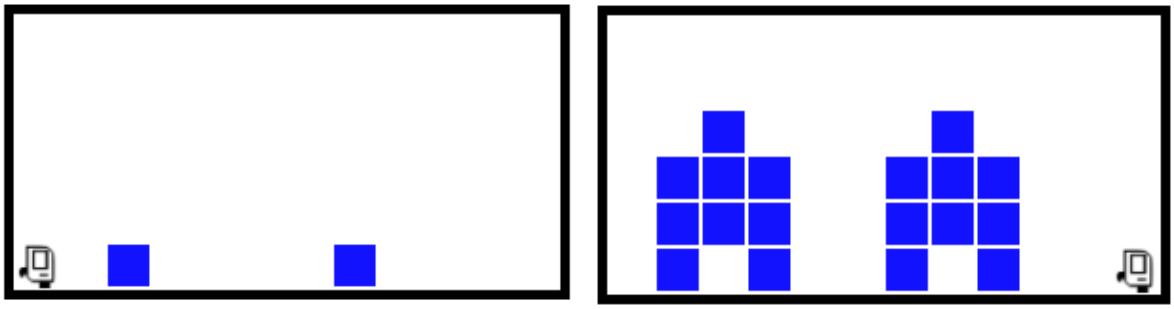


Pre/Post conditions for the world

```
def diagonal_stripes(filename):
    bit = Bit(filename)
    # your code here!
```

Bit Program #4: Build Hospitals

Bit's job is to "build hospitals" wherever it detects blue paint on the ground. Bit starts in the bottom left corner of the world and should traverse to the right across the world. When Bit encounters blue paint on the ground, it should erase that blue paint using **`bit.erase()`** and build a "hospital" composed of three staggered columns in blue paint (see diagram). Bit should end up in the bottom right corner of the world. The final hospital will be at least three columns away from the end of the world (don't worry about hitting a wall while building a hospital!)

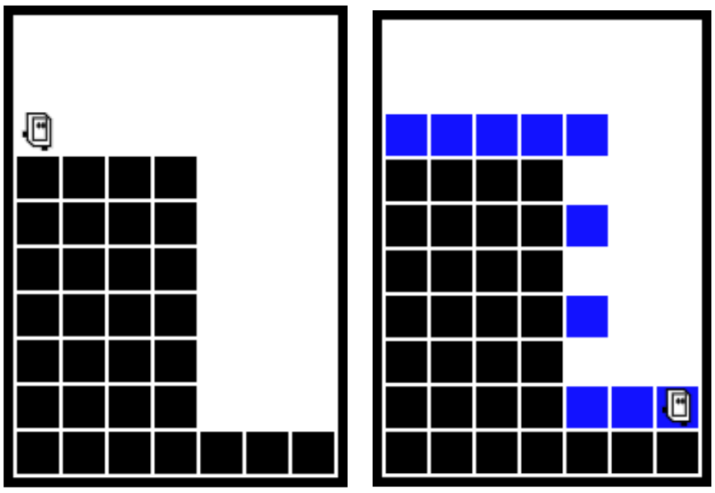


Pre/Post conditions for the world

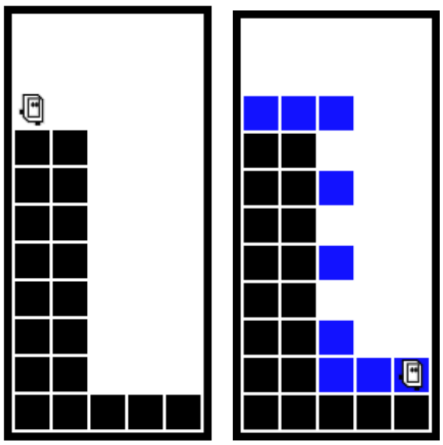
```
def build_hospitals(filename):
    bit = Bit(filename)
    # your code here!
```

Bit Program #5: Fast Waterfall

Bit is at the top of a cliff, and Bit's job is to fill in a "waterfall"! The horizontal parts of the waterfall should be completely filled-in, but the vertical part of the waterfall should be painted blue on every other square. See images below for the expected behavior for waterfalls of varying heights.



Pre/post-conditions (where waterfall is of odd height)



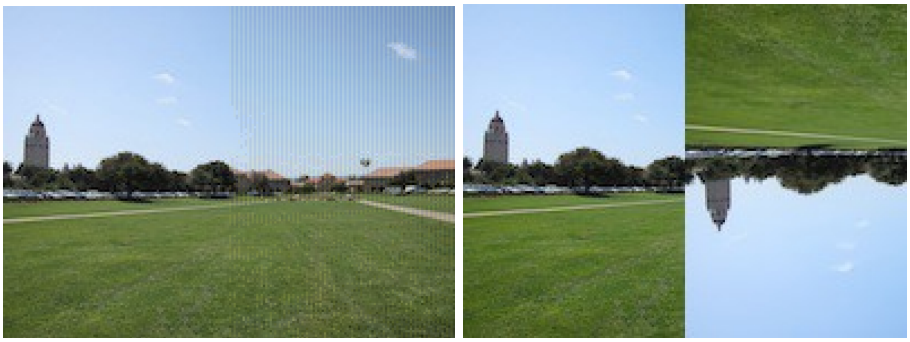
Pre/post-conditions (where waterfall is of even height)

```
def fast_waterfall(filename):
    bit = Bit(filename)
    # your code here!
```

Image Practice

Image Problem #1: Double Left Up

Write a program that takes in a filename, creates a SimpleImage, and then performs the following manipulation: Your program should take the left half of the image, and copy it on the right half, except it should be flipped upside-down as it is copied.

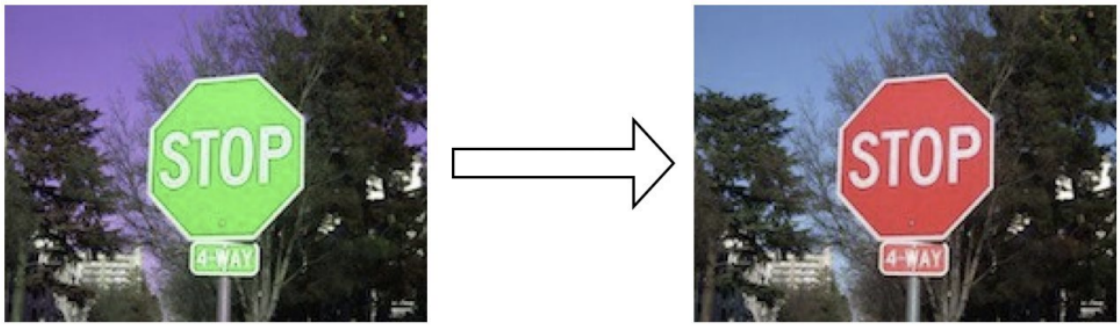


Left: original image
Right: expected output

```
def double_left_up(filename):
   # TODO: your code here!
```

Image Problem #2: Fix Mystery

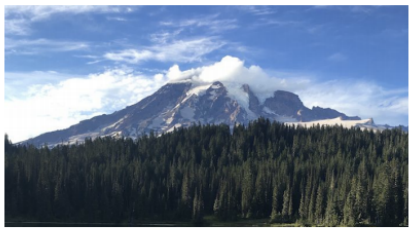
This function takes in the filename of an image where two of the RGB color components have been swapped consistently across all of its pixels. Figure out which two colors have been switched and write code to fix it, returning a SimpleImage object of the fixed image! (If you are colorblind or can't figure out which two colors have been switched, scroll down to see the answers about which channels are switched!)




```
def fix_mystery(filename):
    """
    Returns a SimpleImage where the switched colors have been
    corrected.
    """
    pass
```

Image Problem #3: Reflection

Write a function that returns an output SimpleImage that is twice the height of the original. The top half of the output image should be identical to the original image. The bottom half, however, should look like a reflection of the top half. The highest row in the top half should be "reflected" to be the lowest row in the bottom half. This results in a cool effect!



```
def reflect(filename):
    """
    Returns a SimpleImage twice the height of the original image,
    where a flipped version of the image is repeated in the bottom
    half.
    """
    pass
```

Image Problem #4: Triplicate

Given an image filename, create an output image that is 3 times as wide as the original.

a. Copy the original image, upside down, to the leftmost 1/3 of the output image.
b. Leave the middle copy of the image the same as the original.
c. The rightmost 1/3 of the output image should be a horizontal reflection of the original image (i.e. the top left corner pixel of the original image should be in the top right corner of the copied image).

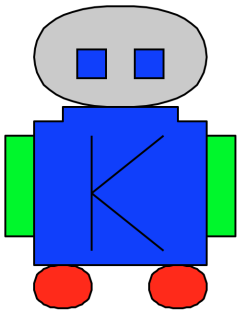


Figure 1: The original image that is triplicated below

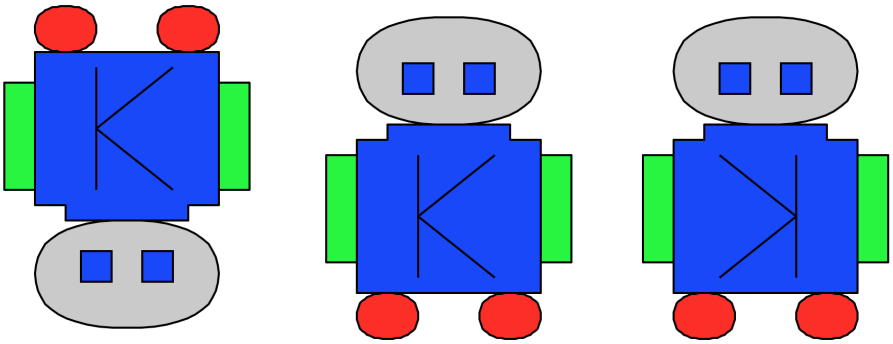
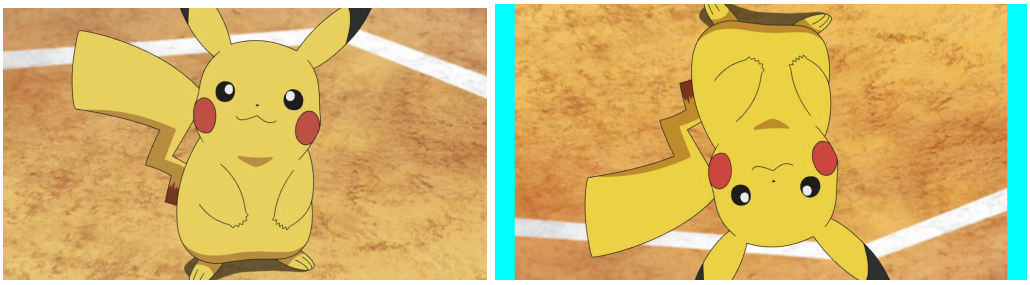


Figure 2: Triplicated version of an image. The original image is shown above.

```
def triplicate(filename):
    # YOUR CODE HERE
```

Image Problem #5: Stripes and Flip

Write a program that takes in a filename, creates a SimpleImage, and then creates a new SimpleImage that flips the original image upside-down and includes two cyan vertical stripes on the edges. The output image is the same size as the original.



Left: original image
Right: expected output

```
STRIPE_WIDTH = 40

def stripes_and_flip(filename):
   # TODO: your code here!
```

String Practice

String Problem #1: Upper Double Digits

Given a string that contains a combination of alphabetical, numerical, and other characters, write a program that returns a version of that string containing only alphabetical and numerical characters. The alphabetical characters in the resulting string should be made uppercase, and the numerical characters should be repeated.

Input: `'abc 123 !@#'`
Output: `'ABC112233'`

Input: `'I love ACE x100!'`
Output: `'ILOVEACEX110000'`

```
def upper_double_digits(s):
    # TODO: your code here!
```

String Problem #2: Parse Name

Given a string s, return a "parsed" version of the string containing only characters that appear in a legal name (alphabetical characters, '-', and spaces).

Input: `'123Son456nja J$#%@ohns***n-Y0u'`
Output: `'Sonja Johnson-Yu'`

```
def parse_name(s):
    # YOUR CODE HERE
```

String Problem #3: No Vowels But E

Given a string s, return a version of the string that omits all vowels in the original, except for ‘e’, and replaces every digit in the original string with the number 3. (Please use the VOWELS constant in your answer).

Input: `'ACE is great 123!'`
Output: `'CE s gret 333!'`

```
VOWELS = 'aeiou'

def no_vowels_but_e(s):
    # YOUR CODE HERE
```

String Problem #4: String Translation

You have arrived at a new land, where natives all speak a different language. It turns out that the language that the language they speak is very similar to English, with a few differences: This language does not have any vowels other than 'a', 'i' and 'u'. (i.e. 'e' and 'o' do not exist and instead are replaced by 'u' respectively). Additionally, each word ends with the suffix "mu". Last but not least, there does not exist uppercase letters.

Input: "**`Today`**"
Output: "**`tudaymu`**"

Input: "**`ILOVEACE`**"
Output: "**`iluvuacumu`**"

Input: "**`onthebed`**"
Output: "**`unthubudmu`**"

You have been hired to build a translator program that takes in simple English phrases and translate them into the native language of their land. (You can assume that string inputs will only contain alphabetical characters and spaces, so no additional error-checking.)

```
def string_translator(eng_string):
      # TODO: your code here
```

String Problem #5: Sarcasm Generator

Sometimes, sarcasm can be a good answer. But it can take some time to prepare a sarcastic response to someone's comment. BUT, since you're a programmer, you can write a program to automate them! In this exercise, you are given a string which only contains alphabetic characters and spaces, and your task is to change the string so that it sounds sarcastic.

Input: **`"you are so funny"`**
Output: **`"yOu aRe sO FuNnY"`**

Input: **`"interesting"`**
Output: **`"iNtErEsTiNg"`**

A sarcastic comment is defined as: For a given string, every even letter position (i.e. index) is lowercase and every odd letter position is uppercase. E.g.: "may" turns into "mAy". (Notice 0-index is even). (Hint: to check if a number is odd, use the condition **`number % 1 == 1`**)

```
def sarcasm_generator(str_comment):
      # TODO: your code here
```

String Problem #6: S and Other

Given a string that contains a combination of alphabetical, numerical, and other characters, write a program that returns a version of that string containing only "s"s and other non-alphanumerical characters.

Input: `'pqrsPQRS123#$%!'`
Output: `'sS#$%!'`

Input: `'Sonja is my ACE TA!'`
Output: `'S s   !'`

```
def s_and_other(s):
    # TODO: your code here!
```

More String Practice (slicing and .find)

More String Problem #1: Replace

Write a function that takes in a string **s** and another string **fill**. **s** contains a '[' character and then a ']' character. Return a new string with these characters and whatever is found between them with **fill**. Don't use s.replace().

Input: 'I want [!@#$]' , 'waffle fries'
Output: 'I want waffle fries'

Input: 'I think [name] should do this task', 'Brian'
Output: 'I think Brian should do this task'

```
def replace(s, fill):
    # TODO: your code here
```

More String Problem #2: DNA Splicing

Given a DNA sequence containing characters of only A, T, C, G, we'd like to find the first instance of the subsequence 'ATG' and get rid of it. Write a function that takes in a DNA sequence as a string, and returns a new string with the first instance of the subsequence 'ATG' removed. If there is no 'ATG' inside the original DNA sequence, return the string unchanged.

Input: 'ATCGGGATGAC'
Output: 'ATCGGGAC'

Input: 'AATGACAATG'
Output: 'AACAATG'

```
def dna_splice(dna):
    # TODO: your code here!
```

More String Problem #3: Longer String

Write a function that, given a string with four '@' symbols, examines the substring contained within the first pair of '@' symbols and the substring contained within the second pair of '@' symbols, and returns the longer of the two. If both substrings are of the same length, return either one. You may find the **`s.find(target, start)`** function particularly useful here.

Input: 'qwerty@dog@qwerty@bird@qwerty'
Output: 'bird'

Input: '@happy@adfkjaslkdfja;s@sad@'
Output: 'happy'

```
def longer_string(s):
    # TODO: your code here!
```

More String Problem #4: Paired Parenthesis

Write a function that takes in a string with at most one ‘(‘ character and at most one ‘)’ character. Return True if the string has paired parenthesis or if it has no parenthesis at all. A string has paired parenthesis if it has a ‘(‘ and a ‘)’ character, where ‘)’ appears anywhere after the appearance of ‘(‘. Otherwise, return False.

Input: ‘1 + 2 + 3(33 * 5)’
Output: True

Input: ‘Hello my name is Brian’
Output: True

Input: ‘123 + 456)(33’
Output: False

Input: ‘123 / 123)’
Output: False

```
def paired_parenthesis(s):
    # TODO: your code here!
```

Trace Practice

Trace Problem #1: Treehouse Bill

The program below calculates a bill, specifically for Treehouse! But something weird is going on, and customers are getting overcharged...

We should be getting the following two output lines:

- Your total before tip is: $95.625.
- Your final price is: $119.53125.

Trace through the program and answer the following questions:

- What numbers are we getting instead?
- There are a couple of bugs in the code. What are they and how can we fix them?

```
"""
File: TreehouseBill.py
----------------------------
It's your job to figure out what this program does!
"""
# Constants
TAX_RATE = 0.0625
TIP_RATE = 0.25
SALAD_COST = 5
PIZZA_THRESHOLD = 4
LARGE_ORDER_PIZZA_COST = 70
SMALL_ORDER_PIZZA_COST = 20

def add_salad_costs(n):
    """Return the total cost of all n salads"""
    return n * SALAD_COST

def add_pizza_costs(n):
    """Return the total cost of all n pizzas."""
    if n < PIZZA_THRESHOLD:
        return SMALL_ORDER_PIZZA_COST
    else:
        return LARGE_ORDER_PIZZA_COST

def add_tax(total):
    """Return the total with tax"""
    total *= 1 + TAX_RATE

def add_tip(total):
    """Return the total with tip"""
    total *= 1 + TIP_RATE
    return total

def calculate_bill(num_pizzas, num_salads):
    """
    Given the total numbers of pizzas and salads, return
    the total cost of the meal.
```

```
    """
    total = 0
    total += add_salad_costs(num_salads)
    total += add_pizza_costs(num_pizzas)
    add_tax(total)
    print('Your total before tip is: $' + str(total) + '.')
    total = add_tip(total)
    return total

def main():
    num_salads = 4
    num_pizzas = 6
    final_price = calculate_bill(num_salads, num_pizzas)
    print('Your final price is: $' + str(final_price) + '.')
```

Trace Problem #2: The Mystery Bill

At the end of every month, Karel's Klinic and HosPytal (K&H) sends customers their bills for all services a given patient received, either from the K&H clinic or from the K&H hospital. The three steps for calculating a patient's medical bill are as follows:

- Summing together the costs for the clinical services the patient received
- Summing together the costs for the hospital services the patient received
- Subtracting the percentage cost covered by the patient's insurance company

However, some patients have been complaining that their totals aren't coming out correctly! The billing program was written by K&H's last programmer, who unfortunately didn't take CS106AP (so they didn't really know much about good style) and has now moved on to pursue their true passion of being a zookeeper. Luckily, you have some experience with both tracing code and building receipt programs, so K&H have asked for your help in debugging the existing billing program.

In particular, K&H want you to answer the following questions:

1. We have a patient who received 6 clinical services and 2 hospital services and whose insurance will cover 50 percent of the total cost. Below is the function call relevant to this patient:

   ```
   zoo(6, 2, 0.50)
   ```

   What does the program output for this function call?

2. The correct output for the above function call (how much the patient should actually be paying) is 1300. What's going wrong in the program, and how can you fix it? (Please explain briefly in 1-2 sentences.)

HINT: The previous K&H programmer wrote one function for each of the steps described above, as well as a single function that calls the other three. To help you figure out which function is which, it's worth noting that hospital services cost $1000 each and that clinical services cost $100 each.

Using the input from question 1, start by tracing through the functions to figure out what each is doing. Once you've answered question 1 and have a better idea of what's going on, then try tackling question 2. **The full billing program is below:**

```
A = 100
B = 1000

def pangolin(h):
    return A* h

def sloth(c):
    return B *c

def kakapo(x, p): #
    x= x-x*p

def zoo(c,h,p): #(6, 2, 0.50) c = 6, h = 2, p = 0.5
    x=0
    x +=pangolin(c)
    x += sloth(h)
    kakapo(x,p)
    print(x)
```

Grid Practice

Grid Problem #1: Flip Grid

Remember how we were able to "flip" an image across an axis? We're going to do the same thing using a grid. Given a grid, change the location of each square in the grid, "flipping" across the y-axis so that elements in the first column become the elements in the last column, and vice versa. (One

way to approach this is building a new grid, but you can also do it “in-place” by changing the original grid).

before:

'a'	'b'	'c'	'd'
'e'	'f'	'g'	'h'
'i'	'j'	'k'	'l'

after:

'd'	'c'	'b'	'a'
'h'	'g'	'f'	'e'
'l'	'k'	'j'	'i'

```
def flip_grid(grid):
    """
    Example:
    >>> grid = Grid.build([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])
    >>> flip_grid(grid)
    [[3, 2, 1, 0], [7, 6, 5, 4], [11, 10, 9, 8]]
    """
    pass
```

Grid Problem #2: Make Vs

Given a grid filled with empty spaces (**`None`**) and **`'x'`**s, make upward-facing v-shapes at the x locations. This means that you should leave the x location intact but also add "x"s at the upper-left corner and the upper-right corner of that x square. None of the x's will result in overlapping v's, but you should make sure you are not adding an 'x's out of bounds.

This is illustrated in the following figure: at each x location, two additional x's are added at the top left and top right corners.

before:

None	None	None	None	None
None	X	None	None	None
None	None	None	X	None

after:

X	None	X	None	None
None	X	X	None	X
None	None	None	X	None

```
def make_vs(grid):
    """
    >>> grid = Grid.build([[None, None, None], [None, 'x', None]])
    >>> make_vs(grid)
    [['x', None, 'x'], [None, 'x', None]]
    """
    # YOUR CODE HERE
```

Grid Problem #3: Check Fire Move

We're writing a "fire" simulator (like Sand!), and you need to write the function that checks to see if a certain move is legal.

Info about the fire simulator:

- 'f' = fire
- 'r' = rock
- 'w' = wood
- None = air

Rules about allowed moves:

1. Only fire is allowed to move.
2. Fire can only move in-bounds.
3. Fire is allowed to move to wood ('w') or air (None) squares, but cannot move into fire or rock squares.
4. Fire cannot move downward.
5. Fire can only move to adjacent (or corner) squares. It cannot "skip".

```
def check_fire_move(grid, x1, y1, x2, y2):
    """
    >>> grid = Grid.build([[None, None, 'w'], ['r', 'f', None], ['r', None, 'r']])
    >>> check_fire_move(grid, 1, 1, 2, 0)
    True
    >>> check_fire_move(grid, 1, 0, 0, 0)
    False  # Rule 1
    >>> check_fire_move(grid, 1, 1, 0, 1)
    False  # Rule 3
    >>> check_fire_move(grid, 1, 1, 1, 2)
    False  # Rule 4
    """
    # TODO: YOUR CODE HERE
```

Grid Problem #4: Transpose [7 min]

This function is passed an n x n Grid. The function should return a new grid that transposes the ordering of the original list: the first row of the new grid is composed of the first column of the original grid. The second row of the new grid is composed of the second column in the original grid, and so on and so forth. (Hint: you should make a new grid!)

before:

'a'	'b'	'c'	'd'
'e'	'f'	'g'	'h'
'i'	'j'	'k'	'l'

after:

'a'	'e'	'i'
'b'	'f'	'j'
'c'	'g'	'k'
'd'	'h'	'l'

```
def transpose(grid):
    """
    >>> transpose(Grid.build([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
    [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
    >>> transpose(Grid.build([[11, 12, 13],[14, 15, 16]]))
    [[11, 14], [12, 15], [13, 16]]
    """
    pass
```

Grid Problem #5: Scroll Left and Up

Given a Grid, write a function that will "scroll" all the elements in the Grid up and to the left. For each square in the grid, move its contents up 1 and to the left 1, regardless of whether or not it contains None or some other value. Replace values of the original square with None.

```
def scroll_left_up(grid):
    """
    >>> grid = Grid.build([['a', 'b', 'c'], ['d', 'e', 'f']])
    >>> scroll_left_up(grid)
    [['e', 'f', None], [None, None, None]]
    """
    pass
```

List Practice

List Problem #1: Is Odd Index

Write a program that takes in a list of strings, a string, and return True if that string is present at an odd index in the list. For example, in the list ['pangolins', 'are', 'six', 'inches', 'at', 'birth'], 'birth' is at an odd index since it is at index 5 in the list. ('pangolins' is at index 0, 'are' is at index 1, and so on and so forth). If the string is not present in the list or it is but the index is even, the function should return False.

```
def is_odd_index(str_lst, s):
    """
    >>> is_odd_index(['all', 'pangolins', 'are', 'solitary'], 'solitary')
    True
    >>> is_odd_index(['pangolins', 'eat', 'ants'], 'berries')
    False
    """
    pass
```

List Problem #2: Find News Keywords

Given a newspaper headline written as a list of strings and a list of "search words", return a list of search words that appeared in the headline. The headlines will be all uppercase, but the search words might not be. You should return a list of uppercase strings. You can assume that there are no repeated words in the headline.

```
def find_news_keywords(headline, search_words):
    """
    >>> find_news_keywords(['SMALLEST', 'PANGOLIN'], ['pangolin', 'scales'])
    ['PANGOLIN']
    >>> find_news_keywords(['ACE', 'VOTED', 'BEST', 'CLASS'], ['Ace', 'best'])
    ['ACE', 'BEST']
    """
    pass
```

List Problem #3: Compute Reverse Slug

Now that someone has cracked the encryption code you used for Crypto, it's time to come up with a new code! In the homework, you put the alphabetic characters in the key at the front of the slug and then filled it in with the remaining alphabetic characters. This time, you should put all the alphabetic characters in the key *at the end of the slug* instead of at the front. Ignore non-alphabetic characters.

```
ALPHABET = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
'z']

def compute_reverse_slug(key):
    """
    >>> compute_reverse_slug('pangolin')
    ['b', 'c', 'd', 'e', 'f', 'h', 'j', 'k', 'm', 'q', 'r', 's', 't',
'u', 'v', 'w', 'x', 'y', 'z', 'p', 'a', 'n', 'g', 'o', 'l', 'i']
    >>> compute_reverse_slug('ace!!!')
    ['b', 'd', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'a', 'c', 'e']
    """
    pass
```

List Problem #4: Is Top Shelf

Given a list of foods, a corresponding list that stores which shelf the food belongs on, and a food, write a function that will return True if the food belongs on the top shelf and False otherwise. You can assume that the food is guaranteed to be in the food list.

Inputs:

- foods (list of strings)
- shelf_list (list of strings) - 1st index in shelf_list corresponds with 1st index in foods
    - can be 'top', 'mid', or 'bottom'
- food (string)

Returns:

- True if food belongs 'top' shelf and False otherwise

```
def is_top_shelf(foods, shelf_list, food):
    """
    >>> is_top_shelf(['pear', 'parsley'], ['top', 'bottom'], 'pear')
    True
    >>> is_top_shelf(['parsley', 'pork'], ['bottom', 'mid'], 'parsley')
    False
    """
    pass
```