CS106A Midterm Preparation

Please see the course page for information on the timing and logistics for the midterm. This page is about what sort of problems appear on our CS exams and how to study for them.

The midterm will be made of little code-writing questions which look similar to the problems seen in lecture and on the homeworks. It will look like a regular college exam, where you write short phrases of code on paper. This may sound a little strange, but in practice it works fine. We do not grade the questions all-or-nothing like a computer — instead, we look at the code carefully, identifying each part that is right to award partial credit. We also don't mark off for syntax. So if a student wrote this for a list

```
lst = []
lst.add('i'}
```

We would still give that full credit, even though the name of the function should be `.append()` and one parenthesis is a curly bracket. If we can see the right idea in there, that's good enough for full credit. We are interested in the algorithm in the solution code, not the syntax.

Topics on the exam: simple Bit (hw1), images/pixels/nested-loops (hw2), 2-d grids (hw3), strings, loops, simple lists (hw4). This is the material through the first four weeks.

Topics not on the exam: bit decomposition problems, bluescreen algorithm, file reading, writing main(). Int division `//` is not on our midterm, although it appears on some of the old questions here. We've added explanations for those examples.

We will provide a basic reference for the Python functions we've used (shown below), but other than that the exam is closed-note. Closed-note exams are actually better for the students, although you may be skeptical about that. With an open note exam, the exam questions have to morph to into weird, upside-down versions of the lecture examples. For an open-note exam, perversely, the questions all need to look different from the example code in the notes.

With a closed-note exam, we can take nice, normal example functions from lecture, e.g. brackets(s), and just make the exam out of those. The lecture and section and homework problems tour through important Python coding patterns, so they set the pattern for good exam problems too. You will notice that the practice problems below resemble the lecture examples and homework problems. Ideally, as you practice for the exam, you should feel that you can work out a pretty good solution to any problem we did in lecture, or section, or on a homework.

The exam grading will focus on getting the correct output, so if you use a for/i/range to get the right output, and our solution uses a for/ch/s .. either solution can get full credit so long as it produces the correct output.

The exam will provide enough time to write solutions, but not lots of extra time. Part of practicing your code writing is building skill and quickness. You want to be able to write the easy parts confidently, so you can take your time on the hard parts.

How To Study for an Exam

What is the goal of CS106A? We want everyone to pick up core coding knowledge and skills. These core skills are what we do in lecture, and section, and then on the homeworks, so the exam problems will look similar. The topics on the exam are **predictable**. Predictable = studying will pay off.

Three levels of working a code concept...

1. See a code in lecture with some understanding

2. Write code in section or on homework while looking at lecture examples

3. Write code on blank sheet of paper without any references

The bad news: It's a trap to get stuck at step 2. Your brain will look at one of our examples and think "oh yeah, I could have written that." On the day of the exam, you may look at that first empty area waiting for you to write something, and find your mind goes blank. The key is, you need to practice with the blank sheet, writing the code out, not just looking at the solution code. The good news..

(1) We publish many examples of the code patterns we care about. (2) We are providing many practice problems for you to use. (3) We will be forgiving of wrong

syntax when grading. We want the right algorithmic ideas, and we give partial credit for partially correct code (unlike when the computer grades something!).

Studying 1.0

How to study:

- Select a problem, could be one from lecture/section/homework or one below, or from this week's section which has practice problems. Study the solutions.
- Get a blank piece of paper (or a blank document where you can type). With no references at hand, try to write the solution
- Check your answer. If it's not perfect, that's fine. See where you went wrong, try again on a new paper.
- Make a quick sketch to try to get the index numbers right the first time. We will bring scratch paper to the exam.

Studying 2.0 - Experimental Practice Mode

We have a new feature on the experimental server. Go to a section, like Bit Puzzles:

Bit Puzzles

Go to the first problem. At the bottom of the screen, turn on Practice Mode. In this mode, your previously entered code is temporarily blocked, so you can try practicing things fresh again and again. It keeps a timer of how long you've been coding and how many problems you have solved. Many of the sections on the experimental server are perfect for studying for the midterm: bit puzzles, image nested, image section, string2, string3, string4, list1, strlisthw. Practice to build speed, skill, and confidence.

Midterm Reference

The midterm will include printed on it a few reminders of python functions we have used. Of course the key is knowing how to call them. We will not generally grade off for syntax. The emphasis is on passing the right numbers to range() or in a slice, and example problems below all feature that sort of complexity. If a

problem requires some boilerplate, the problem statement will likely just include the boilerplate for you. Functions in square brackets below may have been mentioned briefly, but we haven't really used them yet, so none of the problems need them for a solution.

```
# CS106A Exam Reference Reminder
# [square brackets] denote functions listed here that we have not
used yet
# Exam questions will not depend on functions we have not used
yet
Bit:
  bit = Bit(filename)
  bit.front_clear() bit.left_clear() bit.right_clear()
  bit.move() bit.left() bit.right()
  bit.paint('red') [bit.erase()]

General functions:
  len() int() str() range() [list() sorted()]
String functions:
  isalpha() isdigit() isspace() isupper() islower()
  find() upper() lower() strip()
List functions:
  append() index() [extend() pop()  insert()]

SimpleImage:
  # read filename
  image = SimpleImage(filename)
  # create blank image
  image = SimpleImage.blank(width, height)

  # foreach loop
  for pixel in image:

  # range/y/x loop
  for y in range(image.height):
      for x in range(image.width):
          pixel = image.get_pixel(x, y)

Grid 2D:
  grid = Grid.build([['row', '0'], ['row', '1']])
  grid.width, grid.height - properties
  grid.in_bounds(x, y) - True if in bounds
  grid.get(x, y) - returns contents at that x,y, or None if empty
  grid.set(x, y, value) - sets new value into grid at x,y


Read lines out of a text fie:
```

```
with open(filename) as f:
    for line in f:
        line = line.strip()
```

Where To Get Practice Problems

We have a selection of practice problems below. Homework problems are also an excellent, and the exam problems will often be similar to homework problems. Section problems are also a good source of practice problems.

Midterm Practice Problems

Warmup Trace Problem

a. For each ??, write what value comes from the Python expression on each line.

```
>>> 3 + 4 - 1 + 2
??
>>>
>>> 1 + 2 * 3
??
>>>
>>> s = 'Summer'
>>> s[1]
??
>>>
>>> s[:2]
??
>>>
>>> s[2:]
??
```

b. Function Call - what 3 numbers print when the caller() function runs. This code runs without error.

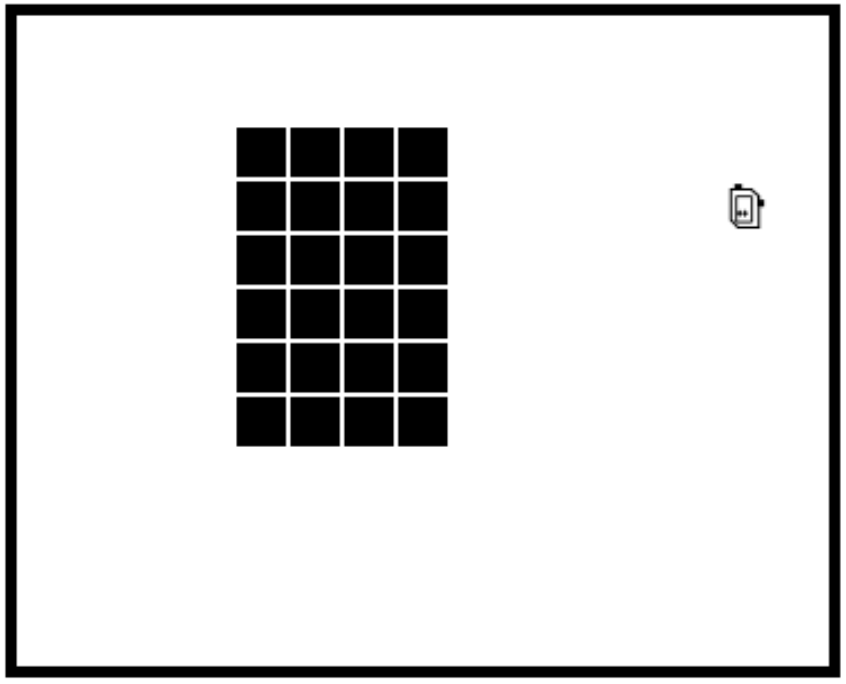
```
def one_more(a):
    b = a + 1
    return b


def caller():
    a = 1
    b = 2
```

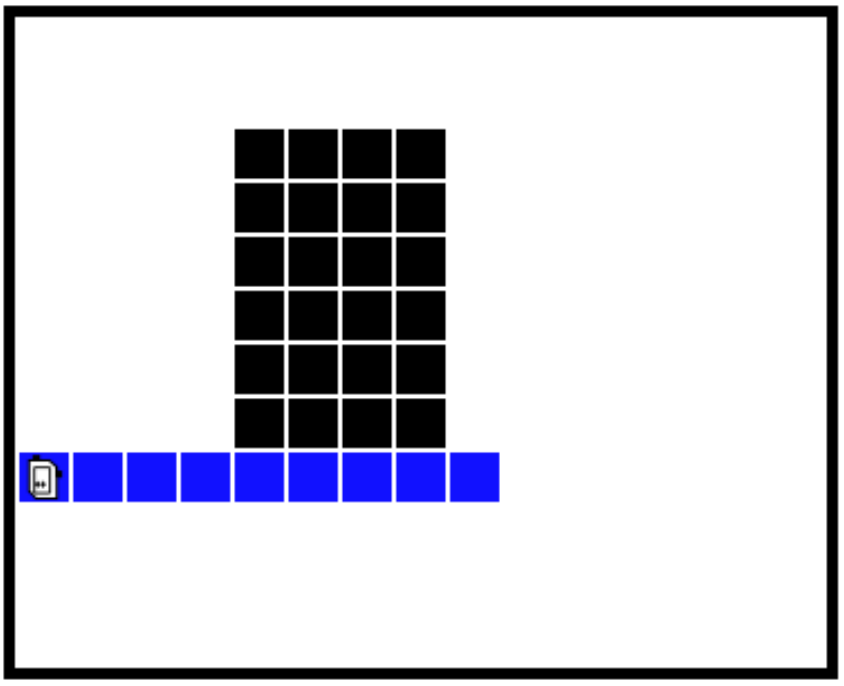
```
c = one_more(b)
print(a, b, c)
```

Bit Problem

Bit is in a world that contains a single rectangle of filled squares. Bit is to the right of the rectangle, facing a block on its right side as shown in this representative example:



Move bit forward to the edge of the rectangle. Then move bit down past the lower right corner of the rectangle. Finally, move bit to the left side of the world, painting every square occupied blue. Your code should work for any position and size rectangle, and bit will always start facing a block on the right side of the rectangle as in the example above.



```
def do_midterm(filename):
    bit = Bit(filename)
    pass
```

Image Problems

image1.

The provided code loads the image for the given filename. The int parameter **n** will be between 0 and the width of the image. Consider two vertical stripes, n pixels wide, running down the left and right sides of the image. Write nested loop code to set all the red values within the stripes to 0. Return the changed image.

```
def stripes(filename, n):
    image = SimpleImage(filename)
    # your code here


    return image
```

image2.

Given an image, create a new 'out' image which is three times as wide as the passed in image. Copy the original image to the rightmost third of the out image, but with

left and right reversed. Return the out image. The starter code includes the standard for y/x loops over the original image as a starting point.

```
def mirrorish(filename):
    image = SimpleImage(filename)

    # create out image:


    # loop x,y over original image
    for y in range(image.height):
        for x in range(image.width):
            # pass



    return out
```

image3.

Given an image and int n which is zero or more. Create a new 'out' image which is n pixels wider than the original. In a vertical stripe, n pixels wide, running down the left side of the output, set all the green values to 0, making it purple. Copy the original image to the area to the right of the stripe. Return the output image. The standard for y/x loop is provided as a start for copying over the original image.

```
def wider(filename, n):
    image = SimpleImage(filename)
    # create out image:


    # purple stripe at left:


    # copy image to space n pixels over
    # (loops provided)
    for y in range(image.height):
        for x in range(image.width):
            pass
```

Grid - Upright Problem

For the Grid problems you should understand the Peep and Movie lecture examples and your solution to the Sand homework: code that returns a Boolean test about the grid, code that potentially changes the grid, and the ability to write a Doctest about the grid.

Suppose we have a program similar to Sand, but where 's' is smoke. Every square in the grid is either 's' smoke, 'r' rock, or None empty.

The "upright" move, is a diagonal move, moving an 's' up one square and towards the right one square.

Before upright move at 0,1:

```
---
r r
s
---
```

After upright move:

```
---
rsr

---
```

a. Write the code for do_upright():

```
def do_upright(grid, x, y):
    """
    Given an x,y which will be in bounds.
    If there is a smoke at that x,y,
    and the destination "up-right" square is empty and in-bounds.
    Move the smoke from x,y to the upright square.
    Otherwise do nothing.
    """
    pass
```

b. Write 1 Doctest for the 3 by 2 case shown above. Some code is included, fill in the ?? of each line.

```
>>> grid = Grid.build(??
>>> do_upright(grid, ??
```

```
>>> ??
```

Grid - Is-Scared Problem

There are two functions for this problem, and you can get full credit for each function independent of what you do on the other function.

We have a Grid representing Yellowstone park, and every square is either a person `'p'`, a yelling person `'y'`, a bear `'b'`, or empty `None`.

a. is_scared(grid, x, y): Given a grid and an in-bounds x,y. Return True if there is a person or yelling person at that x,y, and there is a bear to the right of x,y or above x,y. Return False otherwise. (One could check more squares, but we are just checking two.)

b. run_away(grid): This function implements the idea that when a person is scared of a bear they they begin yelling and try to move one square left. Loop over the squares in the grid in the regular top to bottom, left to right order (this loop is provided in the starter code).

For each square, if is_scared() is True, (1) First set the person stored in the grid to 'y' (i.e. they are yelling). (2) Second, if there is an empty square to their left, then move the person to that square leaving their original square empty. You may call is_scared() even if you did not implement it.

```
def is_scared(grid, x, y):
    pass


def run_away(grid):
    for y in range(grid.height):
        for x in range(grid.width):
            pass
```

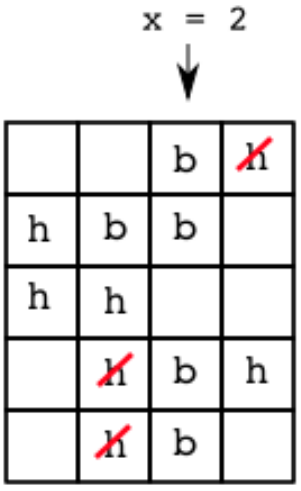
Grid - honey_dx()

There are two functions for this problem, and you can get full credit for each function independent of what you do on the other function.

Bears love honey. We have a Grid representing Yosemite national park, and every square is either a bear `'b'`, honey `'h'`, a tree `'t'`, or empty `None`.

a. honey_dx(grid, x, y): Given a grid and an in bounds x,y. If the square contains a bear, do the following: if the square immediately to the left contains honey, return -1. If the left does not contain honey, but the square to the right contains honey, return 1. In all other cases - no bear or no honey - return 0. So in effect, a 1 or -1 return value indicates that there's a bear, and the x direction of some honey.

b. feed_column(grid, x): Given a grid and an in-bounds x value. The x value defines a column within the grid. This function feeds all the bears in the column. Do the following for every square in the column: Check each square by calling the the honey_dx() helper function. If the square contains a bear and there is honey to its left or right, set the honey square to `None` - in effect the bear eats it. If there is honey on both sides, eat the left side. Return the changed grid.




```
def honey_dx(grid, x, y):
    pass


def feed_column(grid, x):
    pass

    return grid
```

String Problems

str1. Given a string s. Return a string made of the upper case chars from s, so 'Hi TherE' returns 'HTE'.

```
def uppers(s):
```

str2. Given a string s. Find find the first '<' and the first '>' in the string. If these chars are not present or the '<' is not before the '>', return the empty string. Otherwise, return the chars that are between '<' and '>'. So 'xx<yyy>xxx' returns 'yyy'.

```
def foo(s):
    # your code here
```

str3. This nutty function returns a string made up of chars from s. For every char in s, if the char is a digit, compute the int value of that lone digit. If that int is a valid index number into s, append the char at that index to the result. So for example 'xyz029', returns 'xz' since 'x' is at index 0 and 'z' is at index 2. (Old exam problem that plays with char/int fluidity).

```
def nutty(s):
```

str4. Given a string s of even length, if the string length is 2 or less, return it unchanged. Otherwise take off the first and last chars. Consider the remaining middle piece. Split the middle into front and back halves. Swap the order of these two halves, and return the whole thing with the first and last chars restored. So 'x1234y' yields 'x3412y'.

2022 note: int division is not required for our midterm. For this problem, the back half of the string begins at index `len(s) // 2`, e.g. 3 for the example string.

```
def foo(s):
    # your code here
```

str5. Given a string s. Consider all the digit chars in s. Construct and return a **list** of all the digit chars in s. So for example 'A34and6' returns ['3', '4', '6']. Return the empty list if there are no digits.

```
def digits(s):
```

Crypto Logic

This problem is similar to the Crypto homework with the simplification that the encryption is done entirely with lowercase chars.

This "double" encryption uses two source lists, src1 and src2, and two slug lists, slug1 and slug2. All the lists contain lowercase chars and are the same length. Here is the double_encrypt() algorithm: use the lowercase version of the input char, which may be uppercase. If the char is in src1, return the char at the corresponding index in slug1.

If the char is in src2, return the char at the corresponding "reverse index" in slug2, e.g. like this for lists length 4:

```
index  reverse-index
0       3
1       2
2       1
3       0
```

If the char is not in src1 or src2, return the char unchanged. No char appears in both src1 and src2.

```
Double Encrypt Example (len 4)
src1  = ['a', 'b', 'c', 'd']
slug1 = ['s', 't', 'u', 'v']
src2  = ['e', 'f', 'g', 'h']
slug2 = ['w', 'x', 'y', 'z']
```

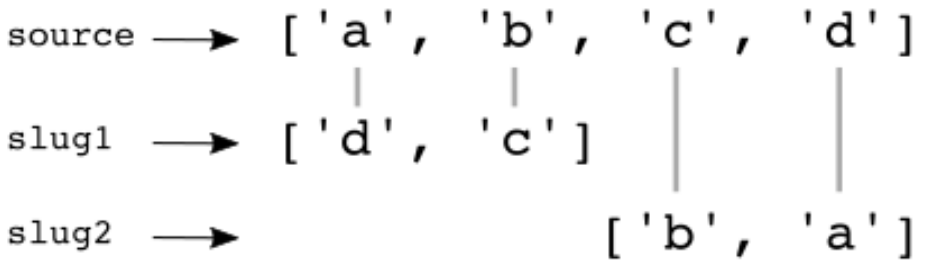
```
encrypt 'A' -> 's'
encrypt 'e' -> 'z'
encrypt 'f' -> 'y'
encrypt '$' -> '$'

def double_encrypt(src1, slug1, src2, slug2, char):
    pass
```

Crypto Double Slug Decryption

First we will describe the double-slug encryption system used in this problem, then you will write the **decrypt** function below. As a simplification, we will assume that uppercase chars do not exist for this problem.

Double-slug encryption uses one source list and two slug lists, all containing lowercase chars. The source list is even length, and the two slugs are each half its length. The slug1 list holds the encrypted form of chars in the first half of the source list. The slug2 list holds the encrypted form of chars in the second half of the source list. No char is in both slug1 and slug2. Here is an example with a length-4 source list:



Encrypt examples with source/slugs above:

```
The encrypt of 'a' is 'd'
The encrypt of 'c' is 'b'
```

Write the code for the **decrypt** function: Given the lists of lowercase chars: source, slug1, slug2. And given lowercase ch which is encrypted if possible. Return the decrypted form of ch, or return ch unchanged if it is not encrypted (e.g. `'$'`). The index in the source list where the slug2-encryption begins is at `midway = len(source) // 2`, which is included in the starter code.

2022 note: int division operator `//` is not required for our midterm, so for this code it is provided. Int division is simply division that rounds down to the next round integer, so `57 // 10` is `5`.

Decrypt examples with source/slugs above:

```
The decrypt of 'd' is 'a'
The decrypt of 'b' is 'c'

def decrypt(source, slug1, slug2, ch):
    midway = len(source) // 2
    pass
```

Solutions

Solutions: Warmup

a. Python expressions

```
>>>>>> 3 + 4 - 1 + 2  # by default proceeds left-right
8
>>>
>>> 1 + 2 * 3       # */ go before +-
7
>>>
>>> s = 'Summer'
>>> s[1]
'u'
>>>
>>> s[:2]
'Su'
>>>
>>> s[2:]
'mmer'
>>>
```

b. Function-call

```
The output is
1 2 3

Key idea: the variable names in each function are independent
```

Solution: Bit

```
def do_midterm(filename):
    bit = Bit(filename)
    # up to block
    while bit.front_clear():
        bit.move()
    # down the side
    bit.left()
    while not bit.right_clear():
        bit.move()
    bit.right()
    bit.paint('blue')
```

```
    while bit.front_clear():
        bit.move()
        bit.paint('blue')
```

Solutions: Images

image1.

```
def stripes(filename):
    image = SimpleImage(filename)
    for y in range(image.height):
        for x in range(n):
            # setting both left and right sides
            # with one loop. Could be solved with
            # multiple loops.
            pixel_left = image.get_pixel(x, y)
            pixel_left.red = 0
            pixel_right = image.get_pixel(image.width - 1 - x, y)
            pixel_right.red = 0
    return image
```

image2.

```
def mirrorish(filename):
    image = SimpleImage(filename)
    # your code here
    out = SimpleImage.blank(image.width * 3, image.height)
    # loop x,y over original image
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            # key: locate pixel in out image
            pixel_out = out.get_pixel(out.width - 1 - x, y)
            pixel_out.red = pixel.red
            pixel_out.green = pixel.green
            pixel_out.blue = pixel.blue
    return out
```

image3.

```
def wider(filename, n):
    image = SimpleImage(filename)
    # your code here
    out = SimpleImage.blank(image.width + n, image.height)
    # purple stripe at left
    for y in range(image.height):
```

```
        for x in range(n):
            pixel_stripe = out.get_pixel(x, y)
            pixel_stripe.green = 0

    # copy image to space n pixels over
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            pixel_out = out.get_pixel(n + x, y)
            pixel_out.red = pixel.red
            pixel_out.green = pixel.green
            pixel_out.blue = pixel.blue

    return out
```

Solution: Grid Upright

a. Write the code for do_upright():

```
def do_upright(grid, x, y):
    """
    Given an x,y which will be in bounds.
    If there is a smoke at that x,y,
    and the destination "up-right" square is empty and in-bounds.
    Move the smoke from x,y to the upright square.
    Otherwise do nothing.
    """
    if grid.get(x, y) != 's':
        return grid
    to_x = x + 1  # using vars, though not required
    to_y = y - 1
    if grid.in_bounds(to_x, to_y):
        if grid.get(to_x, to_y) == None:
            grid.set(to_x, to_y, 's')
            grid.set(x, y, None)
    return grid
```

b. Grid doctest

```
    >>> grid = Grid.build([['r', None, 'r'], ['s', None, None]])
    >>> do_upright(grid, 0, 1)
    [['r', 's', 'r'], [None, None, None]]
```

Solution: Grid Is-Scared

```
def is_scared(grid, x, y):
    # pass
    # A person there
    if grid.get(x, y) == 'p' or grid.get(x, y) == 'y':
        # bear to the right!
        if grid.in_bounds(x + 1, y) and grid.get(x + 1, y) ==
'b':
            return True
        # bear above!
        if grid.in_bounds(x, y - 1) and grid.get(x, y - 1) ==
'b':
            return True
    return False


def run_away(grid):
    for y in range(grid.height):
        for x in range(grid.width):
            # pass
            if is_scared(grid, x, y):
                grid.set(x, y, 'y')   # begin yelling
                if grid.in_bounds(x - 1, y) and grid.get(x - 1,
y) == None:
                    # Move the 'y' one to the left
                    grid.set(x - 1, y, 'y')
                    grid.set(x, y, None)
```

Solution: Grid honey_dx()

```
def honey_dx(grid, x, y):
    if grid.get(x, y) != 'b':
        return 0

    if grid.in_bounds(x - 1, y) and grid.get(x - 1, y) == 'h':
        return -1

    if grid.in_bounds(x + 1, y) and grid.get(x + 1, y) == 'h':
        return 1

    return 0


def feed_column(grid, x):
    pass
    for y in range(grid.height):
        dx = honey_dx(grid, x, y)  # get the dx for this square
        # this -1/+1 way is fine
```

```
        if dx == -1:
            grid.set(x - 1, y, None)
        if dx == 1:
            grid.set(x + 1, y, None)
        # the cool way
        # if dx != 0:
        #     grid.set(x + dx, y, None)  # nom nom nom nom
    return grid
    # Subtle issue: our solution calls honey_dx() once
    # and stores the returned dx in a var. If a solution calls
the function
    # twice, it can eat honey twice. This is very subtle, so the
deduction
    # was very minimal.
```

Solutions: Strings

str1.

```
def uppers(s):
    result = ''
    for ch in s:
        if ch.isupper():
            result += ch
    return result
```

str2.

```
def foo(s):
    start = s.find('<')
    end = s.find('>')
    if start == -1 or end == -1 or end < start:
        return ''
    return s[start + 1:end]
```

str3.

```
def nutty(s):
    result = ''
    for i in range(len(s)):
        # using for/i/range, but
        # for/ch/s works fine too
        if s[i].isdigit():
            val = int(s[i])
            if val < len(s):  # valid index
```

```
                result += s[val]
    return result
```

str4.

```
def foo(s):
    if len(s) <= 2:
        return s
    first = s[0]
    last = s[len(s) - 1]
    mid = s[1:len(s) - 1]
    halfway = len(mid) // 2  # int div needed
    return first + mid[halfway:] + mid[:halfway] + last
```

str5.

```
def digits(s):
    result = []
    for ch in s:
        if ch.isdigit():
            result.append(ch)
    return result
```

Solution: Crypto

```
def double_encrypt(src1, slug1, src2, slug2, char):
    lower = char.lower()
    if lower in src1:
        idx = src1.index(lower)
        return slug1[idx]
    if lower in src2:
        idx = src2.index(lower)
        rev = len(src2) - idx - 1
        return slug2[rev]
    return char
```

Solution: Double Slug

```
def decrypt(source, slug1, slug2, ch):
    midway = len(source) // 2
    if ch in slug1:
        idx = slug1.index(ch)
        return source[idx]
    if ch in slug2:
        idx = slug2.index(ch)
```

```
    return source[midway + idx]
return ch
```