Midterm Review Session

Brahm Capoor

With contributions by Juliette Woodrow

Logistics

February 14th, 3:30 - 4:20 p.m.

Rooms: NVIDIA, CEMEX and Bishop (which isn’t capitalized, unfortunately)

More details will be announced tomorrow!

Come a little early! Bring your phone/two-step authentication device

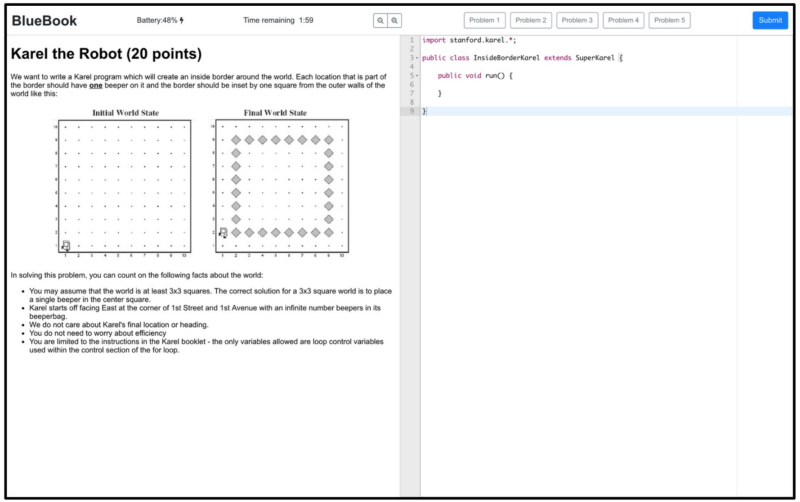
BlueBook

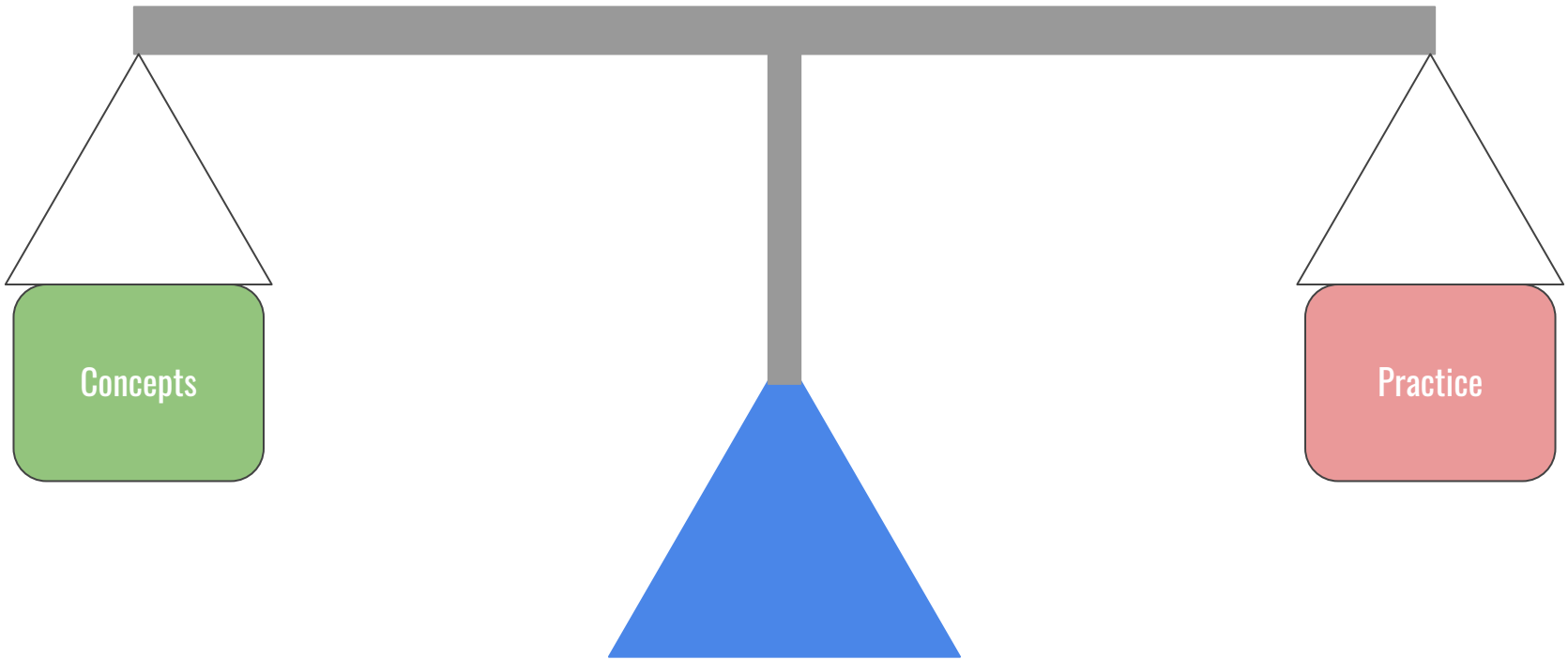
Download for Mac here

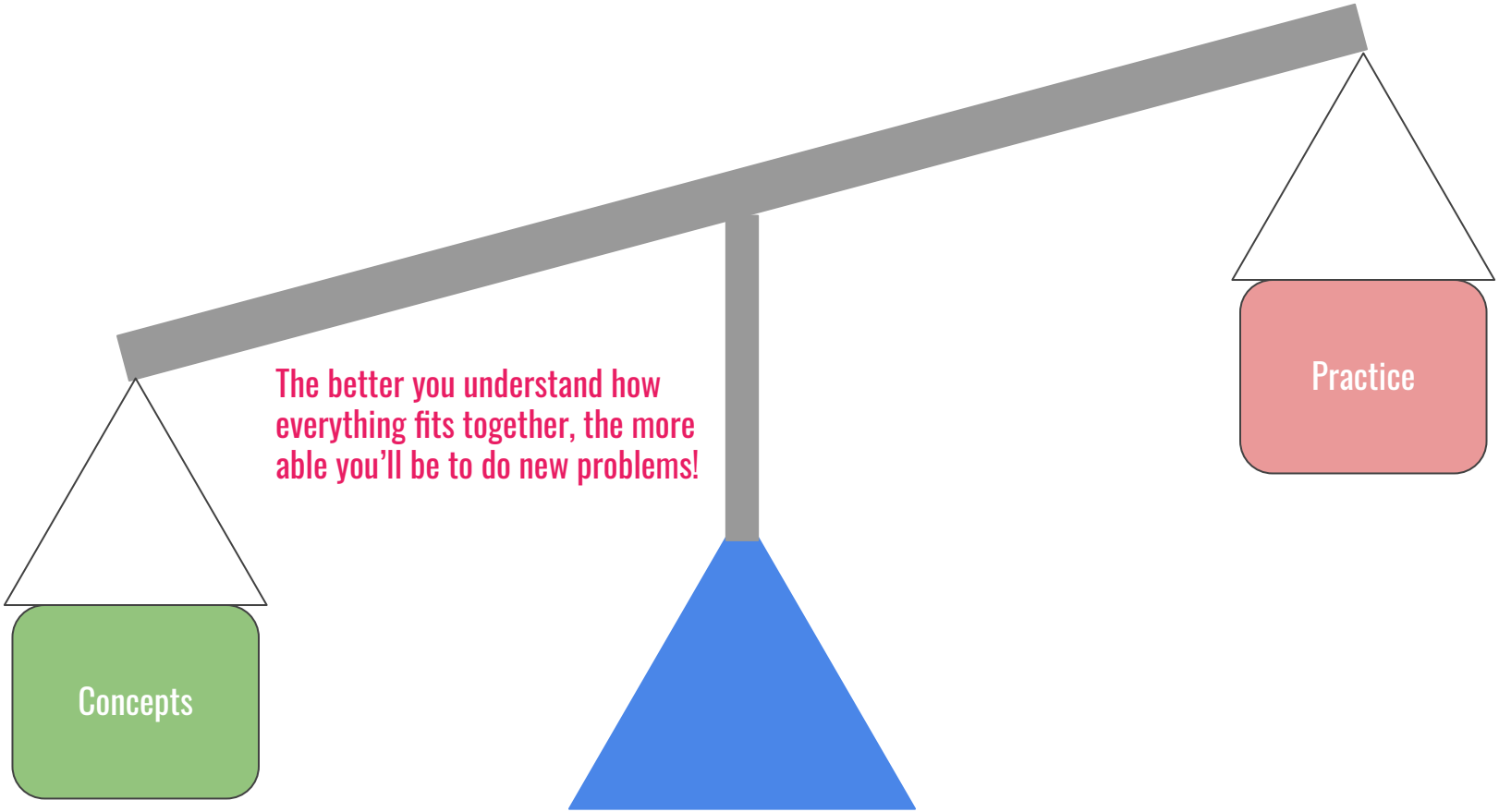
Download for Windows here

Handout here

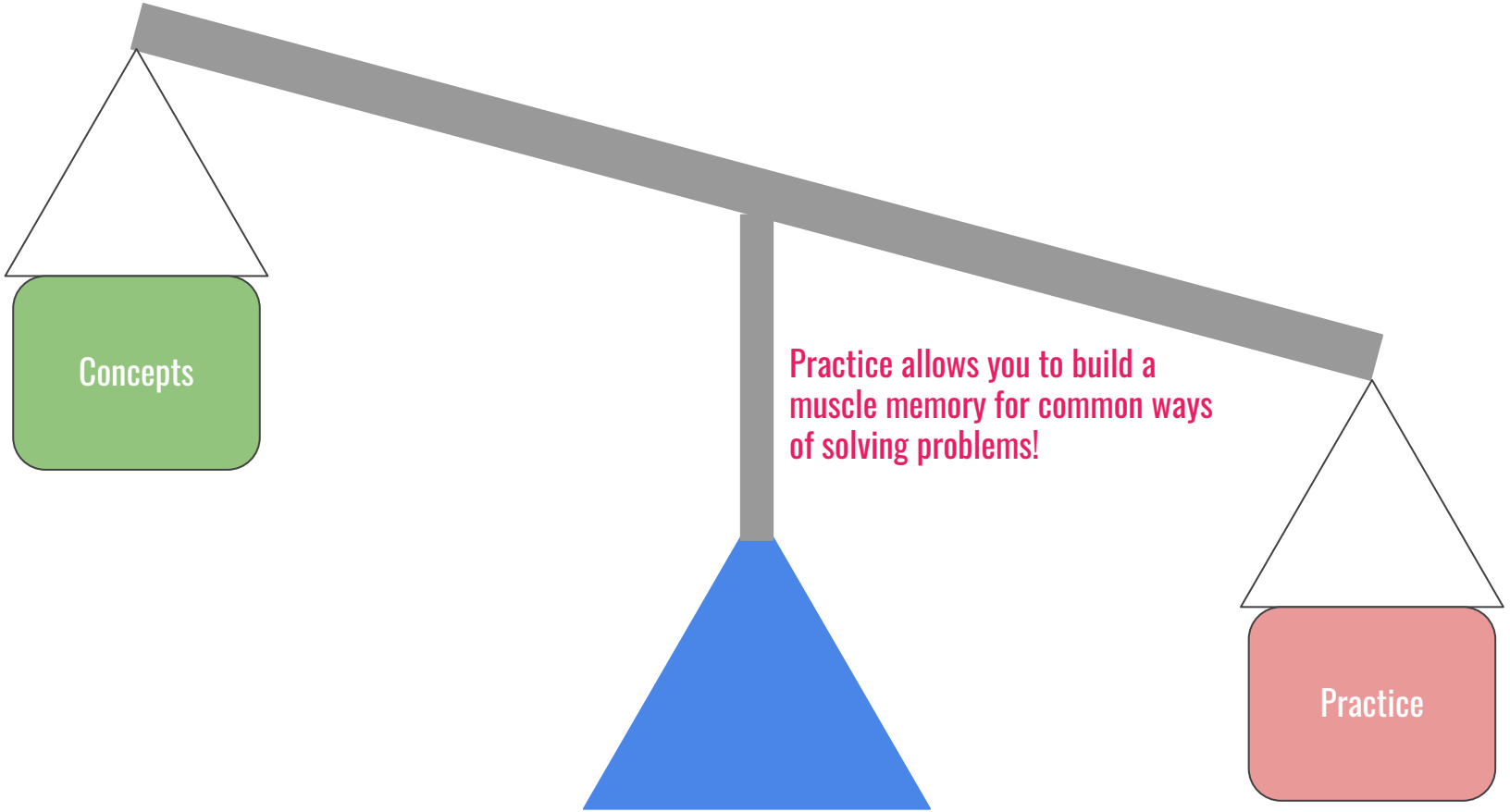
Make sure to have it installed and set up before the exam




Concepts
Practice



The better you understand how everything fits together, the more able you'll be to do new problems!


Concepts
Practice allows you to build a muscle memory for common ways of solving problems!
Practice

Where to find practice problems

Practice Midterm + Additional Practice Problems

Section Handouts (especially this week's)

Scattered throughout these slides

Lecture slides and homework

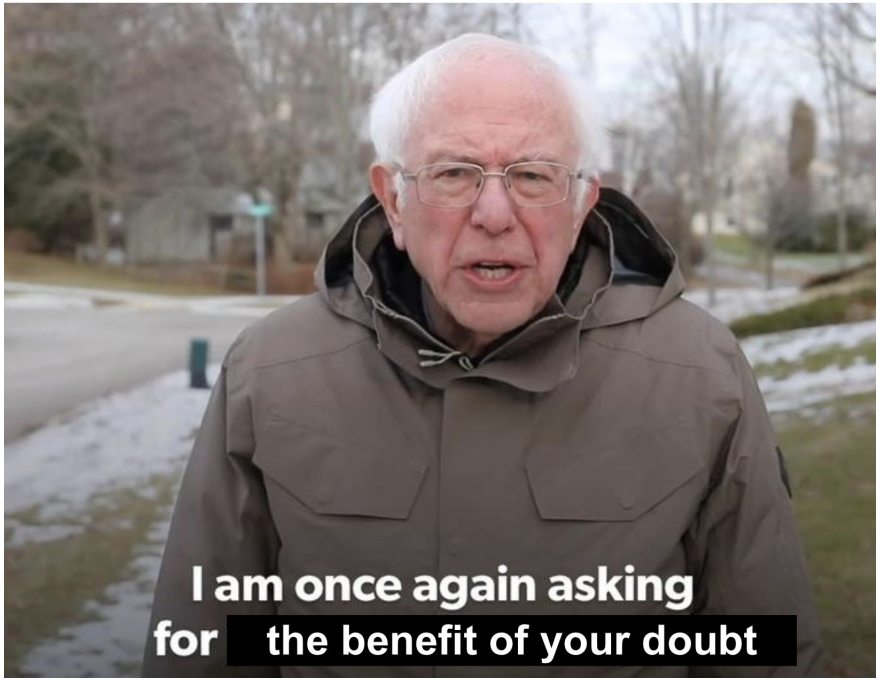
Don’t optimize for memorizing syntax...



...but it lets us give you partial credit

Your grader:

Your code:



The Game Plan

Variables & Control Flow

Functions *

Images *

Drawing

Strings *

Files

Grids *

Lists *

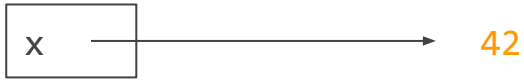
Parsing *

*sections with a * have practice problems*

Variables & Control Flow

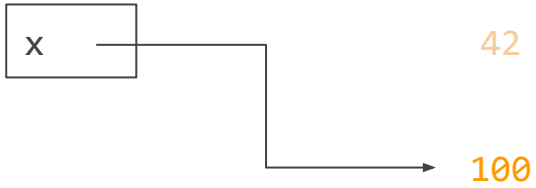
Variables: how we store information in a program

```
x = 42 # assigning a variable
```



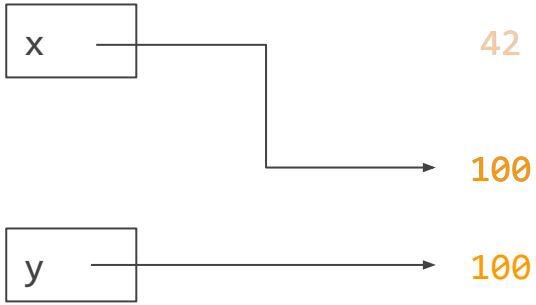
Variables: how we store information in a program

```
x = 42  # assigning a variable
x = 100 # reassigning the variable
```



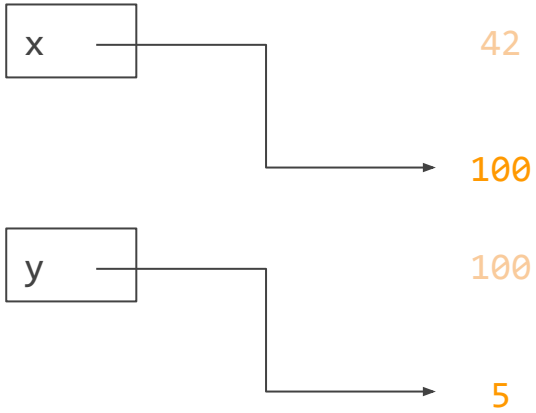
Variables: how we store information in a program

```
x = 42  # assigning a variable
x = 100 # reassigning the variable
y = x   # copying the value from
        # x into y
```



Variables: how we store information in a program

```
x = 42  # assigning a variable
x = 100 # reassigning the variable
y = x   # copying the value from
        # x into y
y = 5   # we only change y, not x
```



Other useful things to know about variables

You can add, subtract, divide, and multiply ints:

```
x = x + 10 | x = x - 42 | x = x // 8 | x = x * 6
```

Remember, dividing ints always rounds down, i.e. 5 // 2 = 2, *not* 2.5

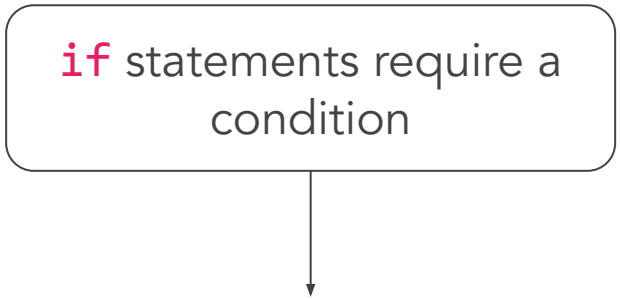
The operations above can be simplified as follows:

```
x += 10 | x -= 42 | x //= 8 | x *= 6
```

Control flow: the steps our program takes

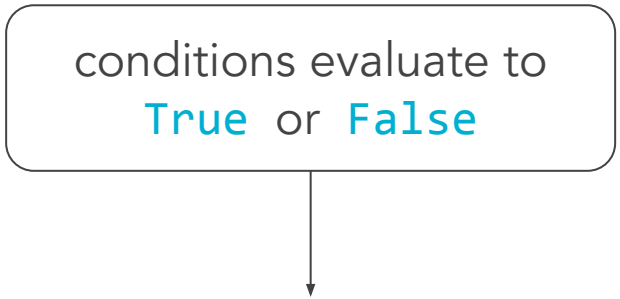
```
if bit.front_clear():
  # sick code here
```

Control flow: the steps our program takes



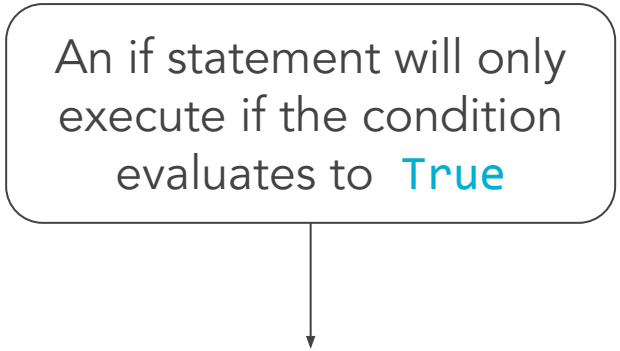

```
if bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes



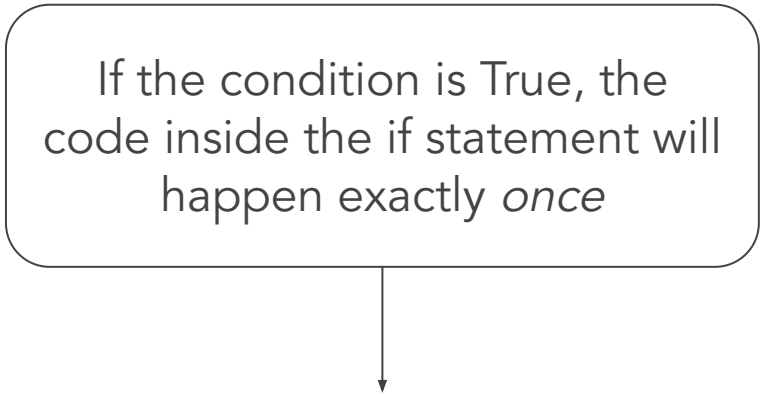

```
if bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes




```
if bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes




```
if bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes

Once the code inside the if statement has completed, the program moves on, *even if the condition is still* True



```
if bit.front_clear():
    # sick code here
# more sick code here
```

Control flow: the steps our program takes

```
while bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes

`while` loops also require a condition, which behaves in exactly the same way

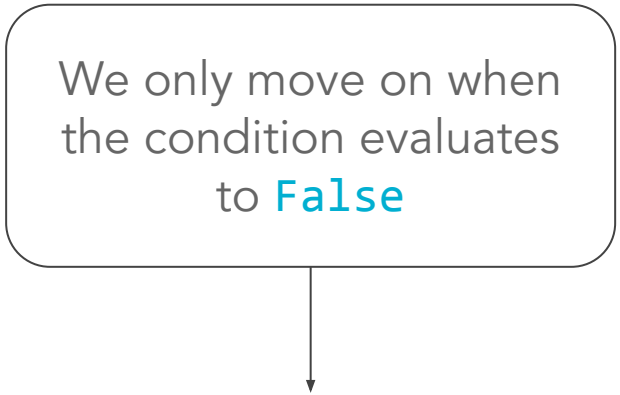
```
while bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes

The difference is that the `while` loop repeats so long as the condition is `True`

```
while bit.front_clear():
    # sick code here
```

Control flow: the steps our program takes




```
while bit.front_clear():
    # sick code here
# more sick code here
```

Control flow: the steps our program takes

```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes

A `for` loop goes through each of the elements of some collection of things

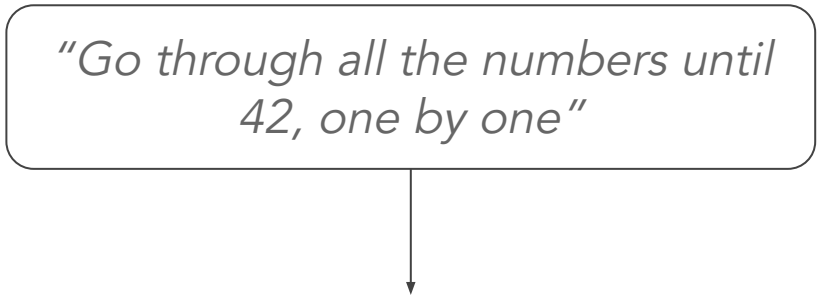
```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes

The range function gives us an ordered collection of all the non-negative integers below a particular number

```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes




```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes

"Go through all the pixels in `image`*, one by one"*

```
for pixel in image:
    # sick code here
```

Control flow: the steps our program takes

The `for` loop ends when we've gone through all the things in the collection

```
for pixel in image:
    # sick code here
# more sick code here
```

Other useful things to know about control flow

`range(42)` - all the numbers between 0 (inclusive) and 42 (exclusive)

`range(10, 42)` - all the numbers between 10 (inclusive) and 42 (exclusive)

`range(10, 42, 2)` - all the numbers between 10 (inclusive) and 42 (exclusive), going up by 2 each time

`range(42, 10, -2)` - all the numbers between 42 (inclusive) and 10 (exclusive), going down by 2 each time.

Functions

When we define a function, we make two promises:

```
def my_function(a, b):
    a += 2
    b = 7
    c = a * b
    return c + 1
```

When we define a function, we make two promises:

1. The inputs, or parameters, to the function

```
def my_function(a, b):
    a += 2
    b = 7
    c = a * b
    return c + 1
```

When we define a function, we make two promises:

1. The inputs, or parameters, to the function
2. What we're returning

```
def my_function(a, b):
    a += 2
    b = 7
    c = a * b
    return c + 1
```

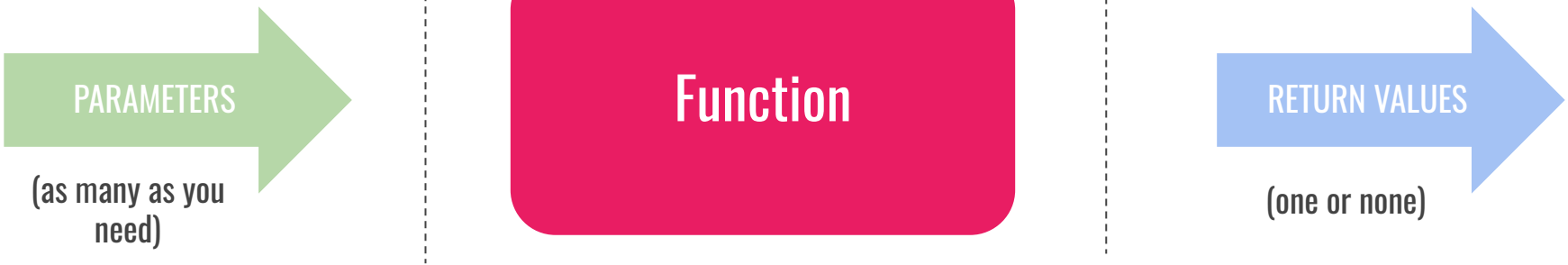
When we define a function, we make two promises:

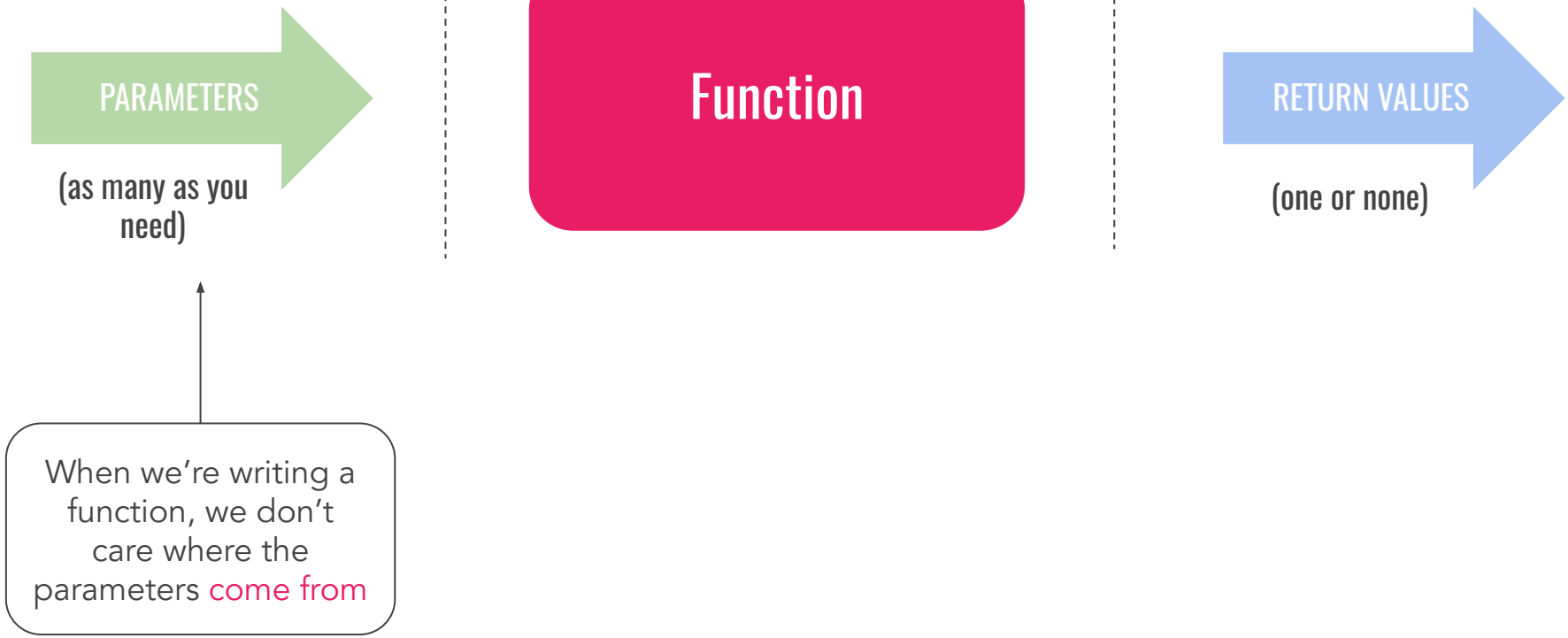
1. The inputs, or parameters, to the function
2. What we're returning

```
>>> my_function(2, 42)
29

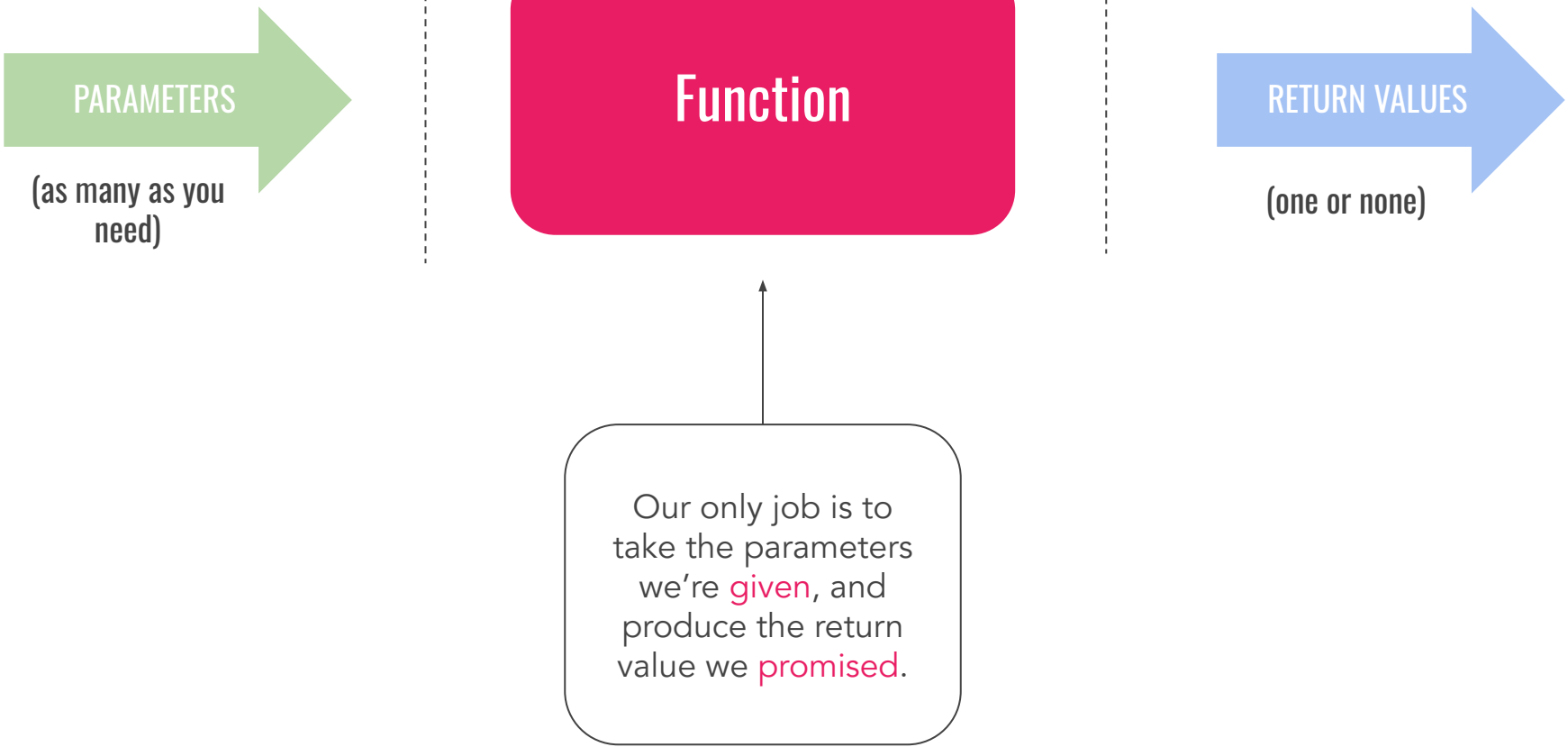
>>> my_function(10, 15)
85
```

```
def my_function(a, b):
    a += 2
    b = 7
    c = a * b
    return c + 1
```


PARAMETERS
(as many as you need)
Function
RETURN VALUES
(one or none)


PARAMETERS
(as many as you need)
Function
RETURN VALUES
(one or none)
When we're writing a function, we don't care where the parameters come from


PARAMETERS
(as many as you need)
Function
RETURN VALUES
(one or none)
...or where the return value is going


PARAMETERS
(as many as you need)
Function
RETURN VALUES
(one or none)
Our only job is to take the parameters we're given, and produce the return value we promised.

Other useful things to know about functions

Functions can't see each other's variables unless they're passed in as parameters or returned

Other useful things to know about functions

Functions can't see each other's variables unless they're passed in as parameters or returned

As a consequence, it's fine to have variables with the same name in different functions

Other useful things to know about functions

Functions can't see each other's variables unless they're passed in as parameters or returned

As a consequence, it's fine to have variables with the same name in different functions

A function can only change its own variables, and not those of its caller

```
def caller():
    x = 42
    callee(x)
    print(x) # this still prints 42

def callee(x):
    x = x + 10  # we're only changing callee's
                # copy of x
```

A great strategy for tracing programs: draw variables as boxes, and go line-by-line through the program

A good tracing practice problem: ‘Tracing’ from this week’s section handout

Printing vs Returning

Programs have a information flow, and a text output area, and those are separate.

- When a function returns something, that's information flowing out of the function to another function
- When a function prints something, that's information being displayed on the text output area (which is usually the terminal)

A useful metaphor is viewing a function as a painter inside a room

- Returning is like the painter leaving the room and telling you something
- Printing is like the painter hanging a painting inside the room
- The painter can do either of those things without affecting whether they do the other thing

Printing is sometimes described as a *side effect*, since it doesn't directly influence the flow of information in a program

Images

SimpleImageS

```
img = SimpleImage('buddy.png')
```

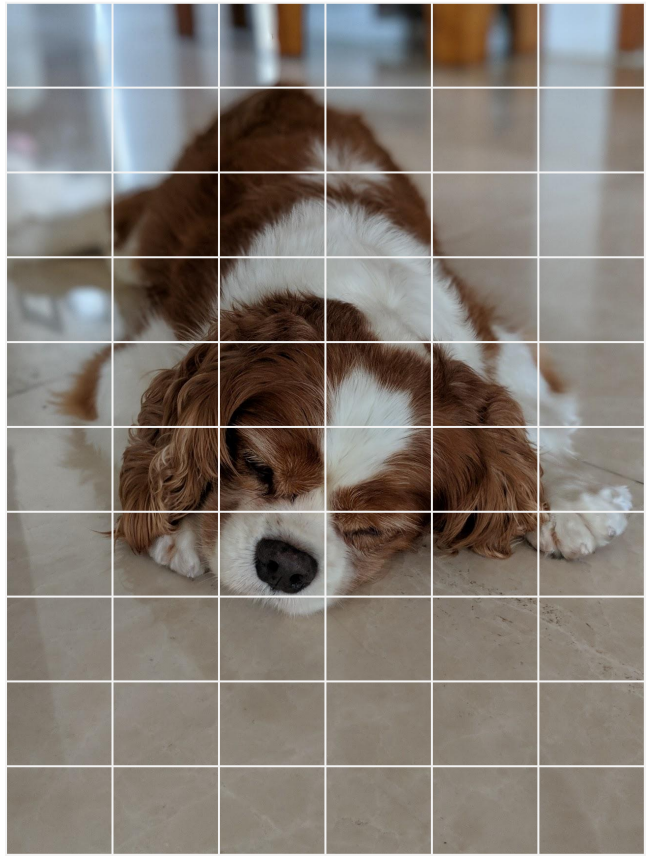


Images

```
img = SimpleImage('buddy.png')

height = img.height  # 10

width = image.width  # 6
```



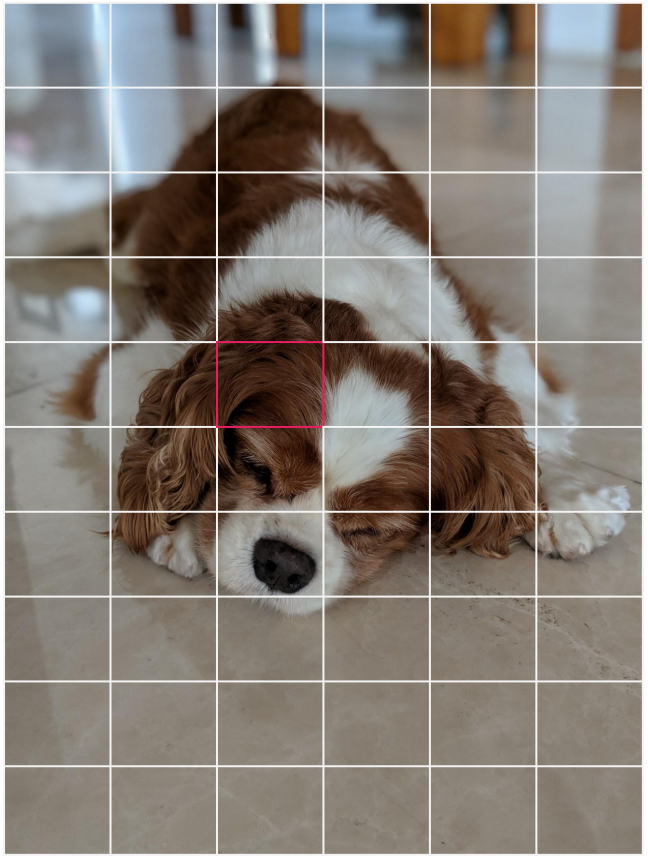
Images

```
img = SimpleImage('buddy.png')

height = img.height  # 10

width = image.width  # 6

pixel = img.get_pixel(2, 4)
```



Pixels

A pixel represents a single color, and is decomposed into three components, each of which is out of 255:

- How red the color is
- How green the color is
- How blue the color is

```
pixel = img.get_pixel(42, 100)

red_component = pixel.red # how red is this pixel?

pixel.green = 42 # changing how green this pixel is

pixel.blue = red_component // # update the blue component
```

Two common code patterns for images

```
for pixel in image:
    # we don’t have access to the coordinates of pixel

for y in range(image.height):
    for x in range(image.width):
        pixel = image.get_pixel(x, y)
        # now we do have access to the coordinates of pixel
```

Two common code patterns for images

```
for pixel in image:
    # we don't have access to the coordinates of pixel

for y in range(image.height):
    for x in range(image.width):
        pixel = image.get_pixel(x, y)
        # now we do have access to the coordinates of pixel
```

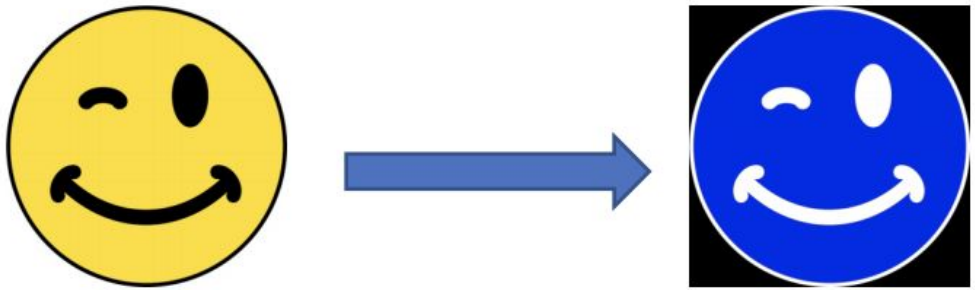
Both of these loops go over the pixels in the same order

A problem: `make_negative`

Implement the following function:

```
def make_negative(image)
```

that takes in a **`SimpleImage`** as a parameter and returns the *negative* of the image. The *negative* of an image is an image whose pixel's color components are set to their inverse. The inverse of a color component *c* is 255 - *c*.



Solving problems is about strategic procrastination: what's easy that we can do quickly?

```
def make_negative(image):
```

We definitely need to return the (modified) image, so let's do that first.

```
def make_negative(image):

    return image
```

Now, we definitely need to loop over the pixels of the image. Do we need their coordinates?

```
def make_negative(image):

    return image
```

...probably not, since we don't need to change the location of the pixels. Let's just use a single `for` loop.

```
def make_negative(image):
    for pixel in image:



    return image
```

Now, we've simplified the problem for ourselves: what do we do with a single pixel in order to solve the problem?

```
def make_negative(image):
    for pixel in image:



    return image
```

Invert it, just like the problem suggests!

```
def make_negative(image):
    for pixel in image:
        pixel.red = 255 - pixel.red
        pixel.green = 255 - pixel.green
        pixel.blue = 255 - pixel.blue
    return image
```

```
def make_negative(image):
    for pixel in image:
        pixel.red = 255 - pixel.red
        pixel.green = 255 - pixel.green
        pixel.blue = 255 - pixel.blue
    return image
```

Another problem:

Implement the following function:

```
def transform(image)
```

which takes in a `SimpleImage` as a parameter and returns an upside-down version of the image whose red pixels (those whose red components are more than 2 times the average color component of that pixel) have been replaced by their grayscale equivalents.

Some useful hints

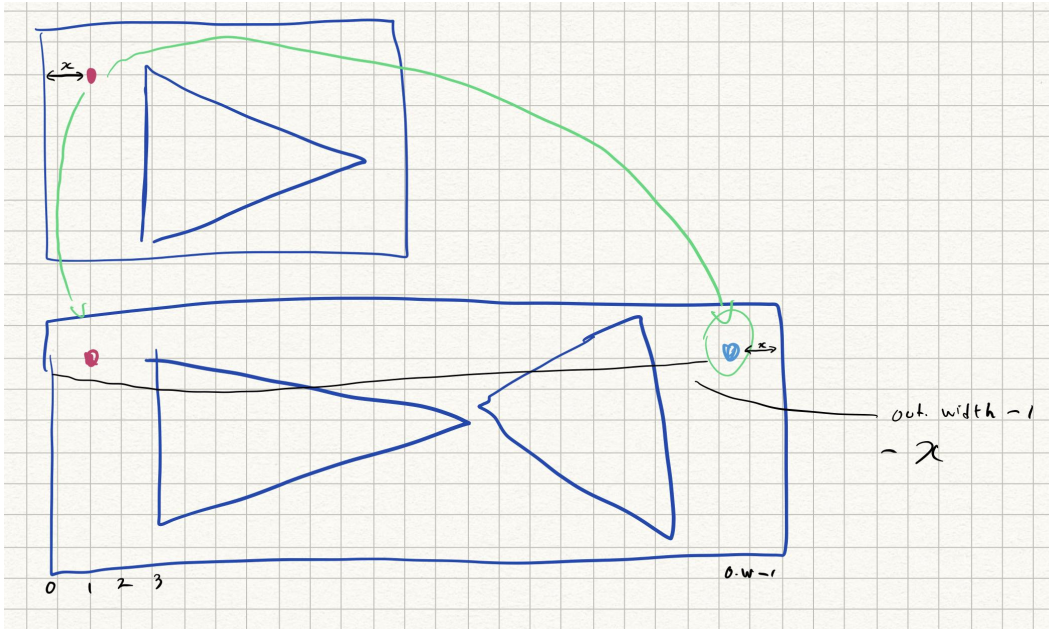
Whenever you're moving pixels around, it's usually easier to make a new image and copy pixels into it, like so:

```
out_image = SimpleImage.blank(image.width, image.height)
for out_y in range(out_image.height):
    for out_x in range(out_image.width):
        out_pixel = out_image.get_pixel(out_x, out_y)
        in_pixel = in_image.get_pixel(<calculate coordinates here>)
        out_pixel.red = # some value, possibly based on in_pixel
        out_pixel.green = # some value, possibly based on in_pixel
        out_pixel.blue = # some value, possibly based on in_pixel
```

Some useful hints

Whenever you're calculating coordinates, drawing diagrams is your best course of action, like so:

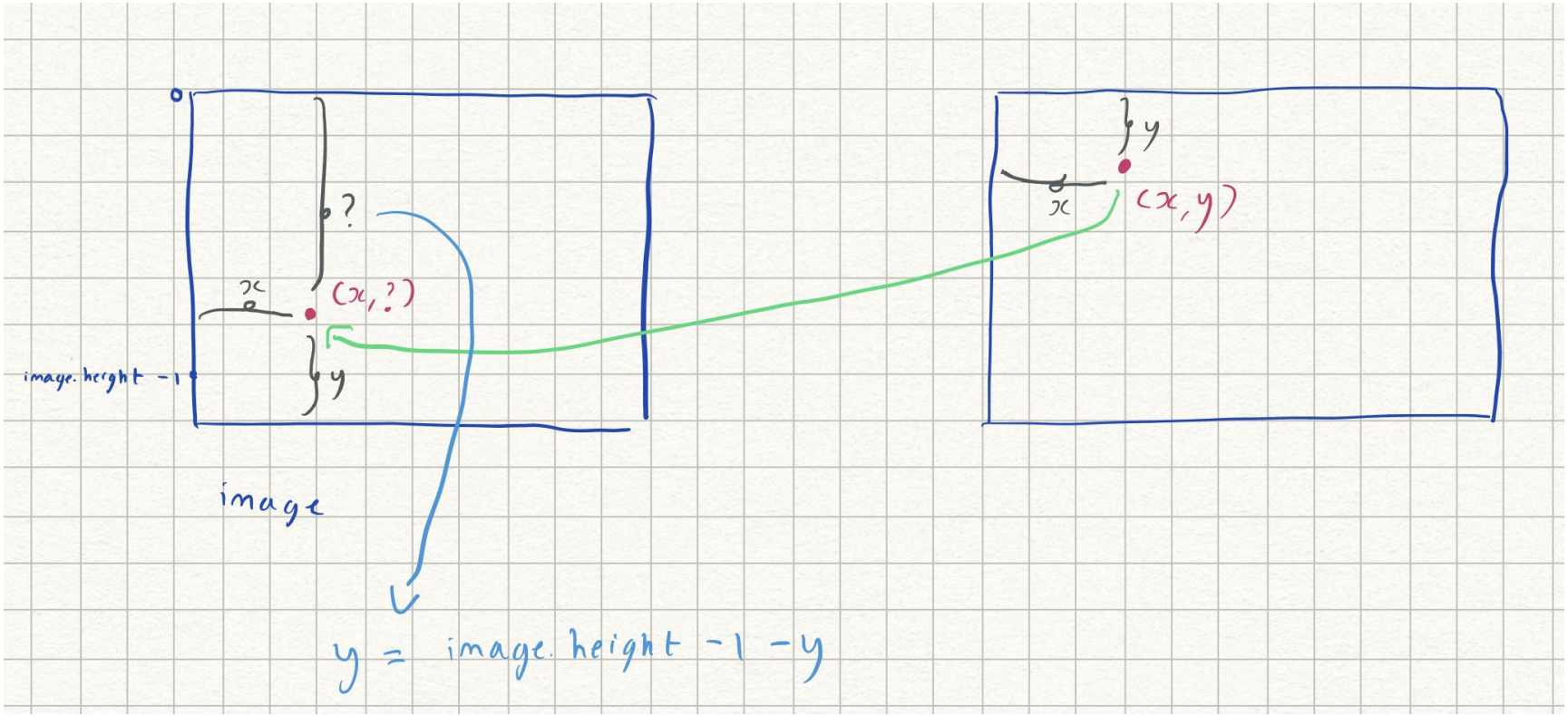
(This is an actual diagram I drew in office hours to explain the mirror problem! Your diagram doesn't need to be neat, it just needs help you calculate coordinates)



The solution

```
def transform(image):
    out_image = SimpleImage.blank(image.width, image.height)
    for out_y in range(out_image.height):
        for out_x in range(out_image.width):
            out_pixel = out_image.get_pixel(out_x, out_y)
            upside_down_y = image.height - 1 - out_y # see next slide
            in_pixel = image.get_pixel(out_x, upside_down_y)
            sum_color = in_pixel.red + in_pixel.green + in_pixel.blue
            average_color = sum_color // 3
            if in_pixel.red > 2 * average_color:
                out_pixel.red = average_color
                out_pixel.green = average_color
                out_pixel.blue = average_color
            else:
                out_pixel.red = in_pixel.red
                out_pixel.green = in_pixel.green
                out_pixel.blue = in_pixel.blue
    return out_image
```

The picture I drew



Looping over output pixels vs input pixels

Most image problems can be framed in one of two ways:

1. Iterating over the pixels of the input and figuring out where they should be copied to.
2. Iterating over the pixels of the output and figuring out where they should come from.

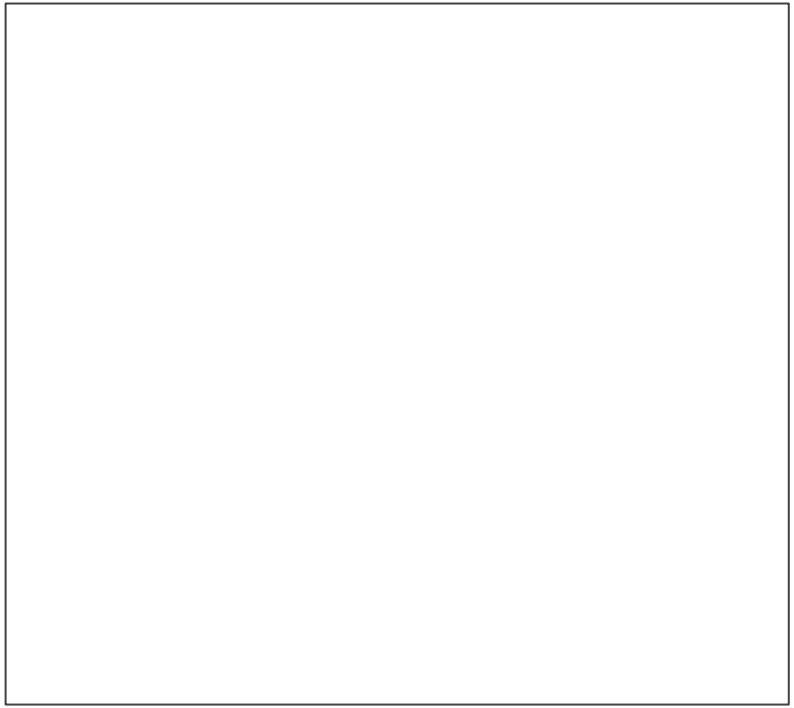
The problem is usually a lot easier if you think about it one way rather than the other. For example, in the `copies_n` function in homework 2, it's easier to think about taking a pixel from the input image and copying it to `n` different spots than it is to pick an arbitrary pixel in the output image and do some annoying math to figure out which input pixel you should be copying.

Conversely, consider the `squeeze_height` function from the same homework, in which you had to multiply output coordinates by a constant factor. In that case, it's more annoying to have to go over the input image and figure out which pixels you're keeping and which you're discarding. Thus, it probably makes sense to go over the pixels you're copying to, and figure out where they should come from.

Drawing

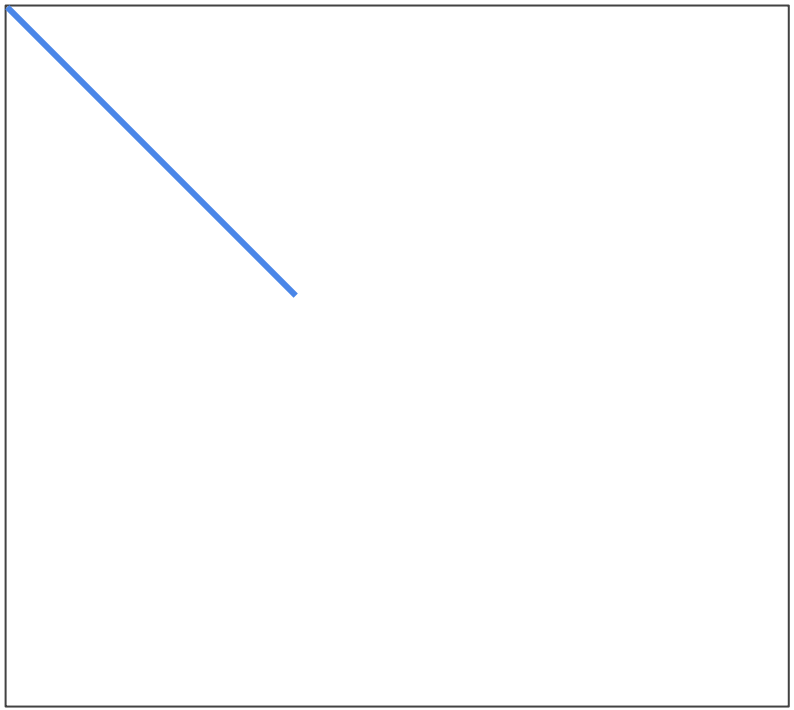
The canvas

```
canvas = DrawCanvas(<some params>)
```



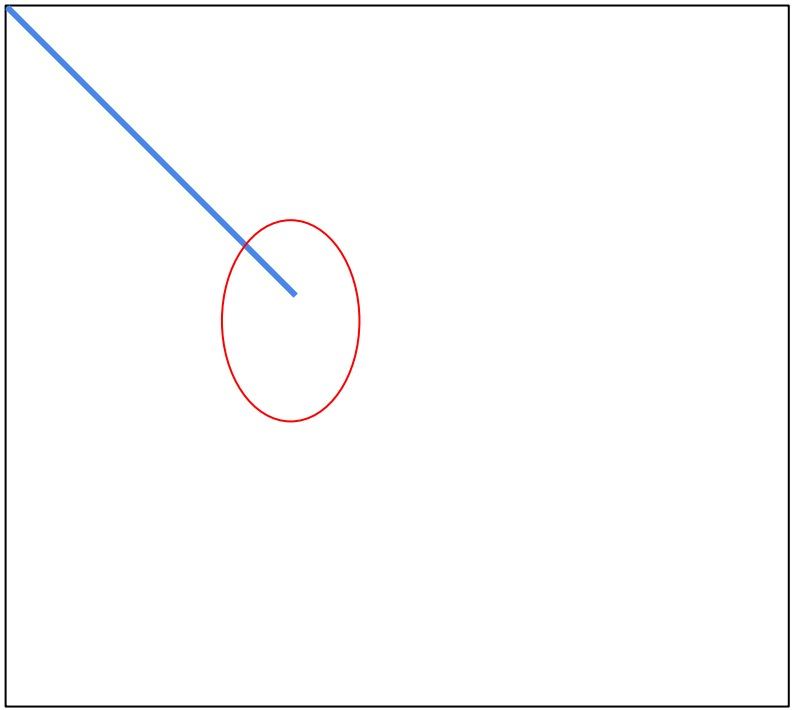
Drawing stuff

```
canvas = DrawCanvas(<some params>)
canvas.draw_line(0, 0, 50, 50, color='blue')
```



Drawing stuff

```
canvas = DrawCanvas(<some params>)
canvas.draw_line(0, 0, 50, 50, color='blue')
canvas.draw_oval(40, 40, 20, 30, color='red')
```

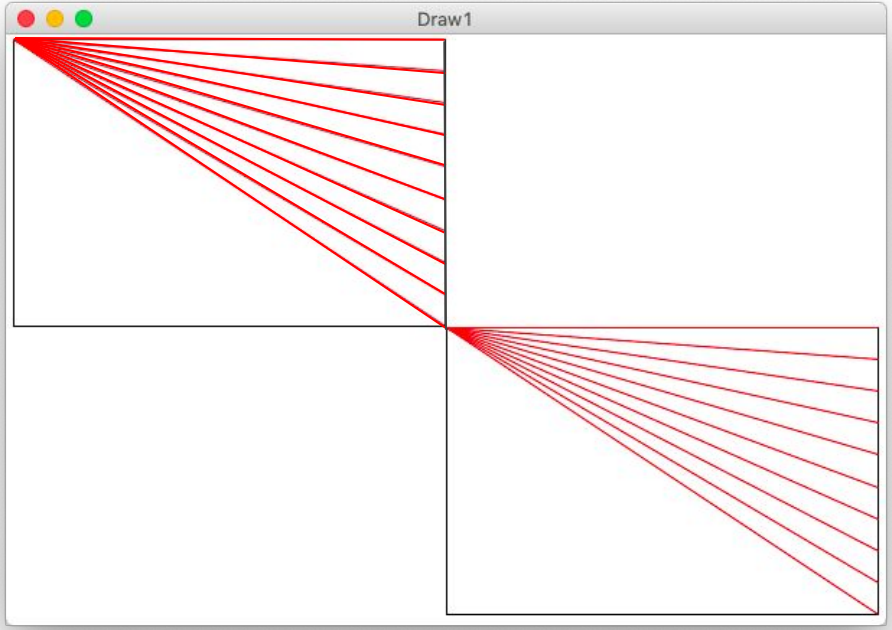


Drawing stuff

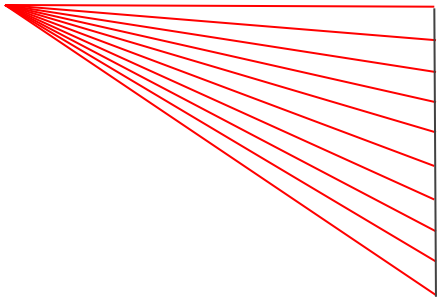
```
canvas = DrawCanvas(<some params>)
canvas.draw_line(0, 0, 50, 50, color='blue')
canvas.draw_oval(40, 40, 20, 30, color='red')
canvas.fill_rect(0, 0, canvas.width,
                 canvas.height, color='green')
```



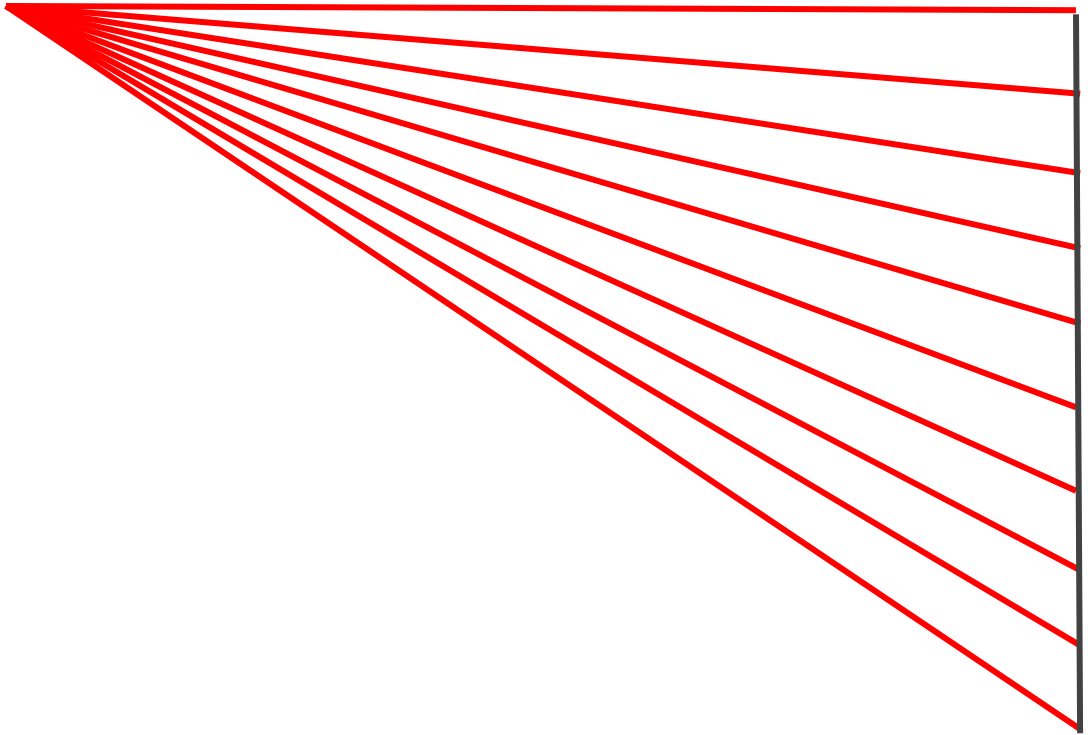
Fractional Drawing



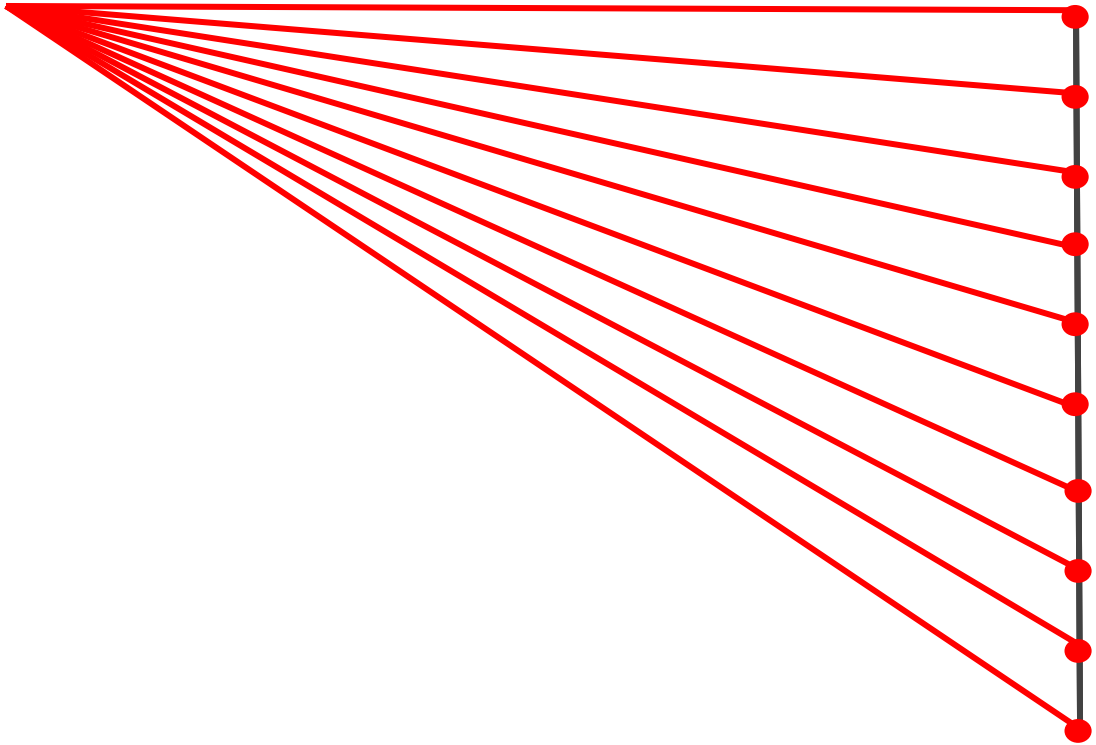
Fractional Drawing



Fractional Drawing



Fractional Drawing



Fractional Drawing



Fractional Drawing

All fractional drawing problems reduce to finding these points on a line



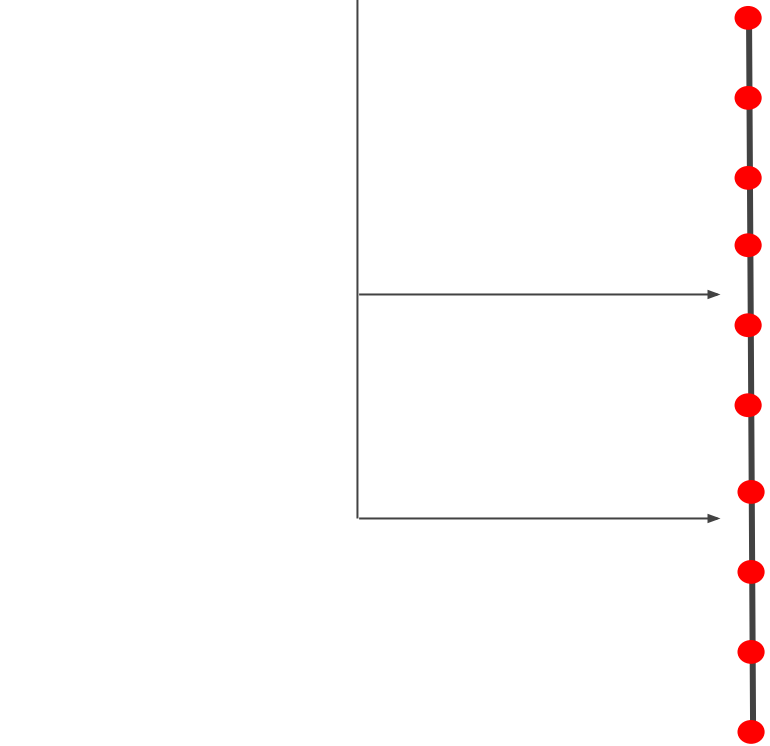
Fractional Drawing

If there are n lines drawn, there are n of these dots



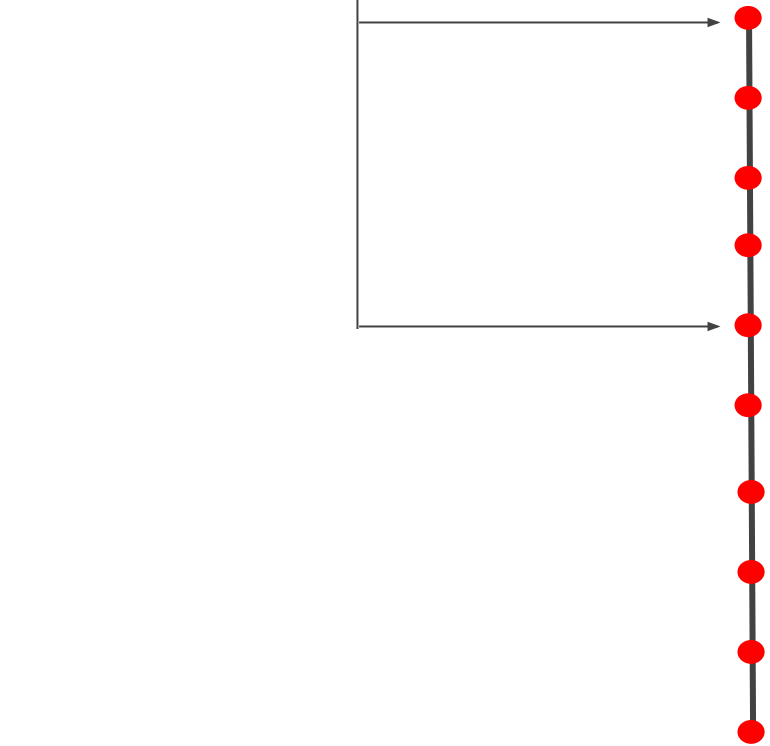
Fractional Drawing

...and at the k-th dot, we've drawn k - 1 of these line segments



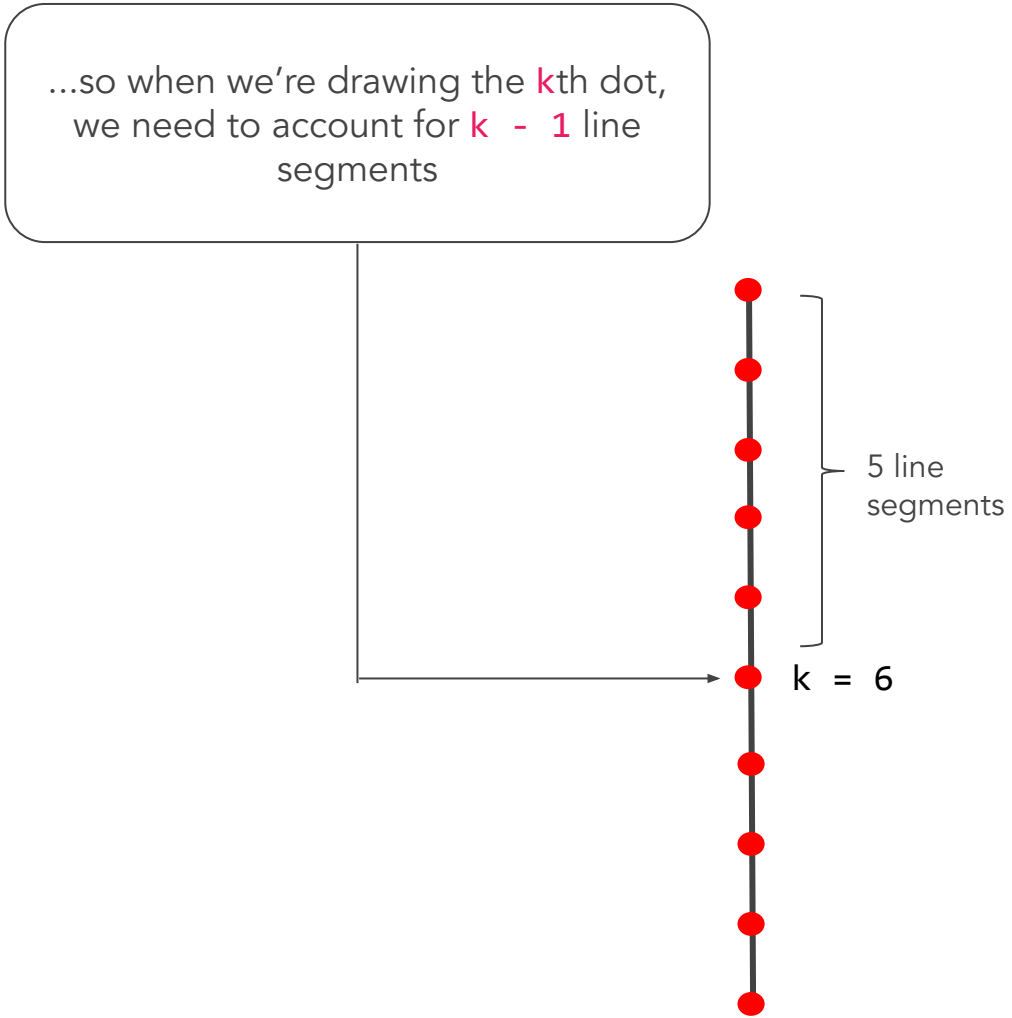
Fractional Drawing

Restated, the k = 1st dot has 0 line segments above it, and the k = 5th dot has 4 line segments above it.



Fractional Drawing

...so when we're drawing the kth dot, we need to account for k - 1 line segments



Fractional Drawing

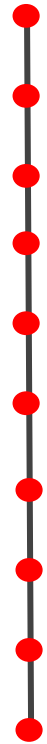
Restated, the number of line segments above each dot starts at 0 and goes up to n - 1



Fractional Drawing

...which is what a `for i in range(n)` loop gives you!

```
for i in range(n):
    # sick code here
```

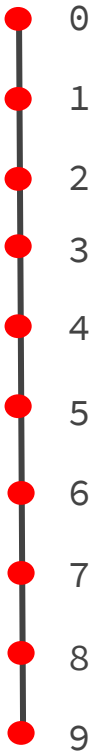


Fractional Drawing

```
for i in range(n):
    # sick code here
```

In this loop, `i` means two things:

1. The 0-indexed count of the points we're currently finding
2. The number of line segments *above* the point we're finding



Fractional Drawing

So for the `i`-th point, we need to find the distance from the top of the line to the point. Let's pick a random point.

```
for i in range(n):
    # sick code here
```



Fractional Drawing

So for the `i`-th point, we need to find the distance from the top of the line to the point. Let's pick a random point.

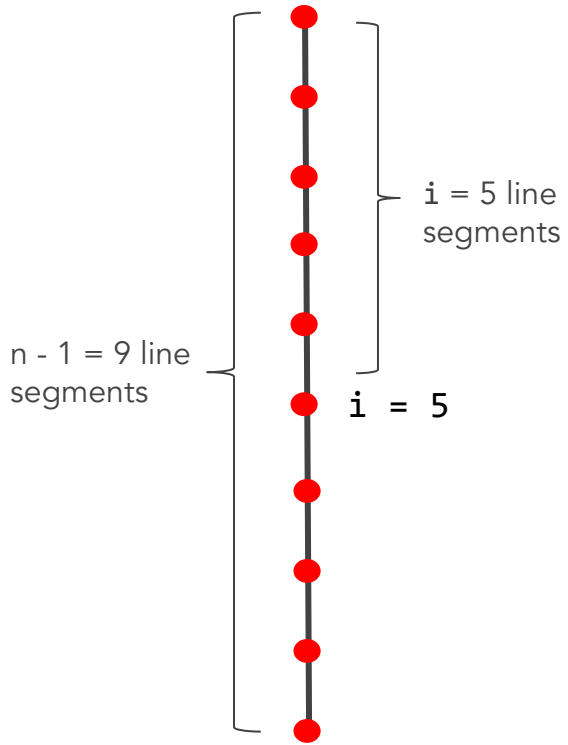
```
for i in range(n):
    # sick code here
```



Fractional Drawing

```
for i in range(n):
    # sick code here
```

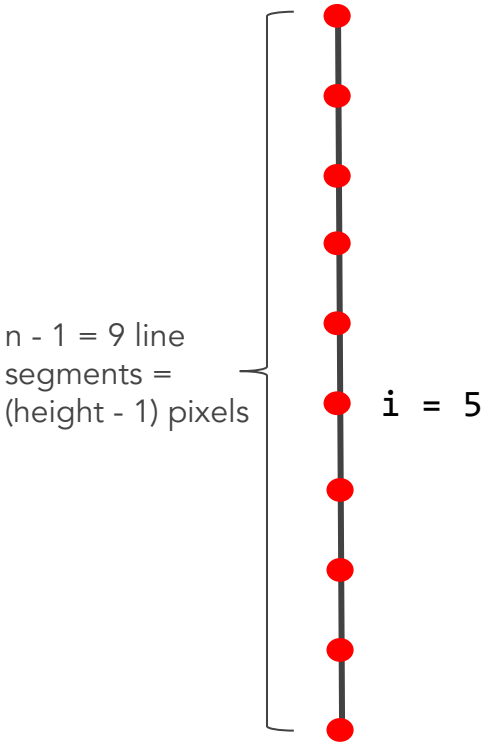
Of the `n - 1` = 9 line segments we could possibly draw, we're drawing `i` = 5 of them.



Fractional Drawing

```
for i in range(n):
    # sick code here
```

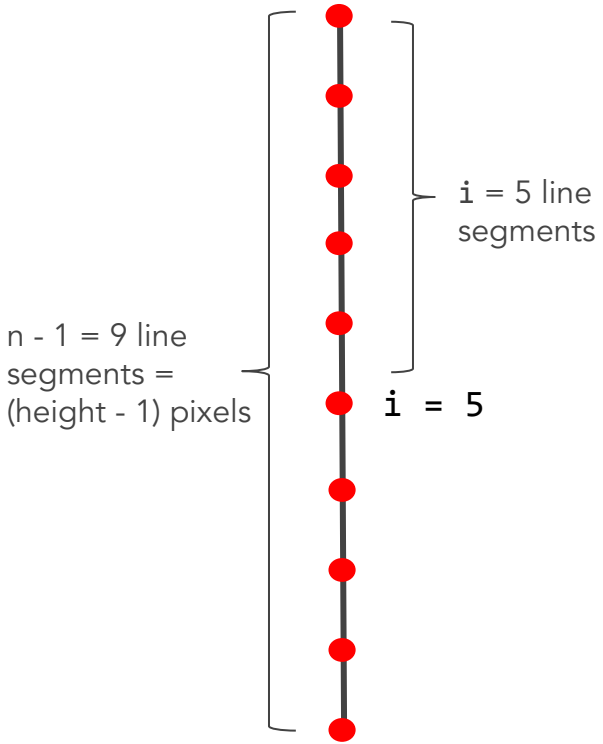
Together, the `n - 1` segments would constitute a distance of `(height - 1)` pixels



Fractional Drawing

So we can find the distance from the top of the line to the current point using fractional multiplication

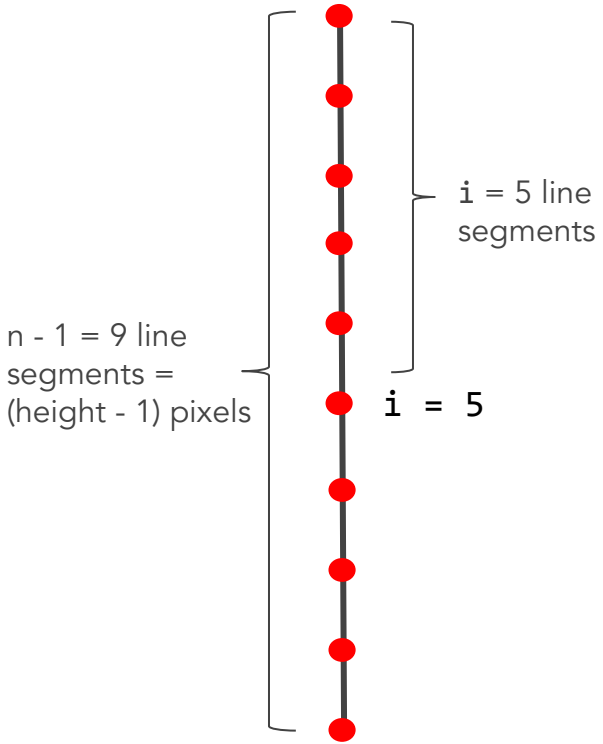
```
for i in range(n):
    y_add = i / (n - 1) * (height - 1)
```



Fractional Drawing

Note that we're using float division here. What would go wrong if we used integer division instead?

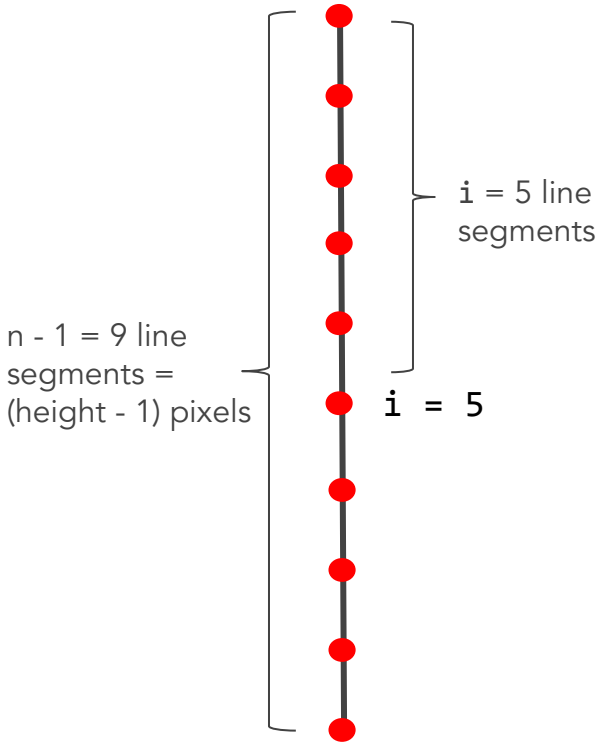
```
for i in range(n):
    y_add = i / (n - 1) * (height - 1)
```



Fractional Drawing

Now, we can just find the actual coordinate of the point and draw a line to it!

```
for i in range(n):
    y_add = i / (n - 1) * (height - 1)
    y_coord = line_start + y_add
```



Fractional Drawing

Take a look at the final solution to this problem and see if you can spot the same pattern here

```
def draw_lines1(canvas, x, y, width, height, n):
    canvas.draw_rect(x, y, width, height)
    for i in range(n):
        y_add = (i / (n - 1)) * (height - 1)
        canvas.draw_line(x, y, x + width - 1, y + y_add, color='red')
```

A quick note on grading

We get it: a lot of coordinate math is kind of annoying. One thing that makes it particularly tricky is that it's hard to tell just by looking at a program that a coordinate is slightly incorrect.

One way we'll account for that in our grading is by being particularly lenient on off-by-one errors in coordinate calculations. For example, if you forget the `- 1` in the `height - 1` calculation, you'll only be a *tiny* bit off so we wouldn't deduct points.

That's not a license to drop 1s, though - if you were to divide by `n` instead of `n - 1`, that results in very different output and we'd almost certainly deduct for that. Long story short: don't rely on this for points!

Strings

A string is a variable type representing some arbitrary text:

```
s = 'We demand rigidly defined areas of doubt and uncertainty!'
```

and consists of a sequence of characters, which are indexed starting at 0.

```
eighth_char = s[7]  # 'n'
```

We can slice out arbitrary substrings by specifying the start and end indices:

```
string_part = s[10:20]  # 'rigidly de'
```

The start index is inclusive, and can be omitted if you want to start at the beginning of `s`

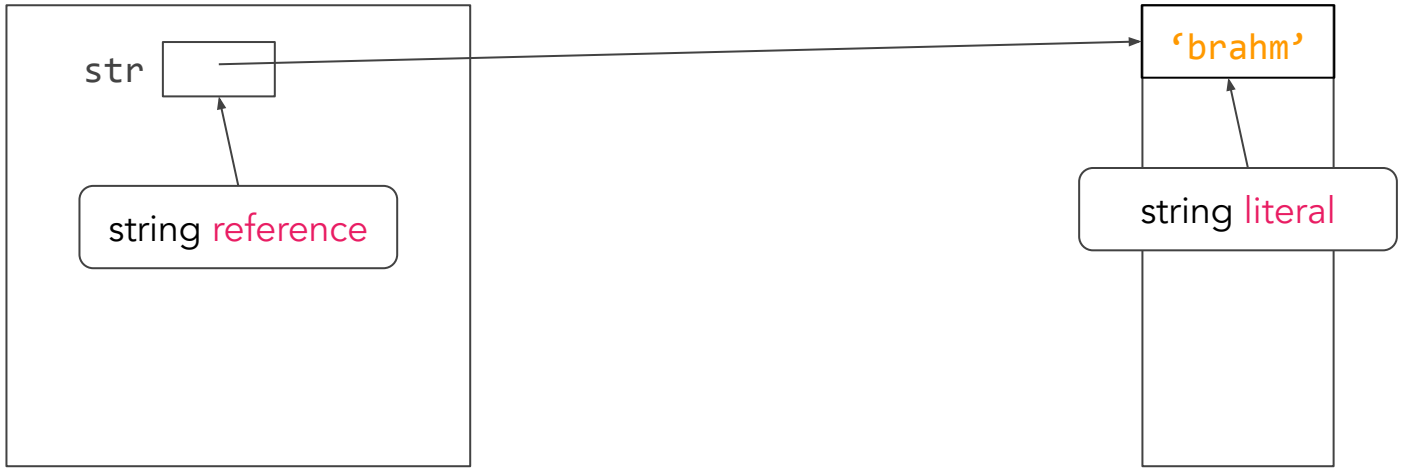
The end index is exclusive, and can be omitted if you want to go until the end of `s`

Useful String functions

```
>>> s = ‘So long, and thanks for all the fish’
>>> len(s)
36
>>> s.find(‘,’)     # returns the first index of the character
7
>>> s.find(‘z’)
-1                  # returns -1 if character isn’t present
>>> s.find(‘n’, 6)  # start searching from index 6
10
>>> s.lower()       # islower() also exists, returns true if all lowercase
‘so long, and thanks for all the fish’  # returns a string, doesn’t modify
>>> s.upper()       # isupper() also exists, returns true if all uppercase
‘SO LONG, AND THANKS FOR ALL THE FISH’
```

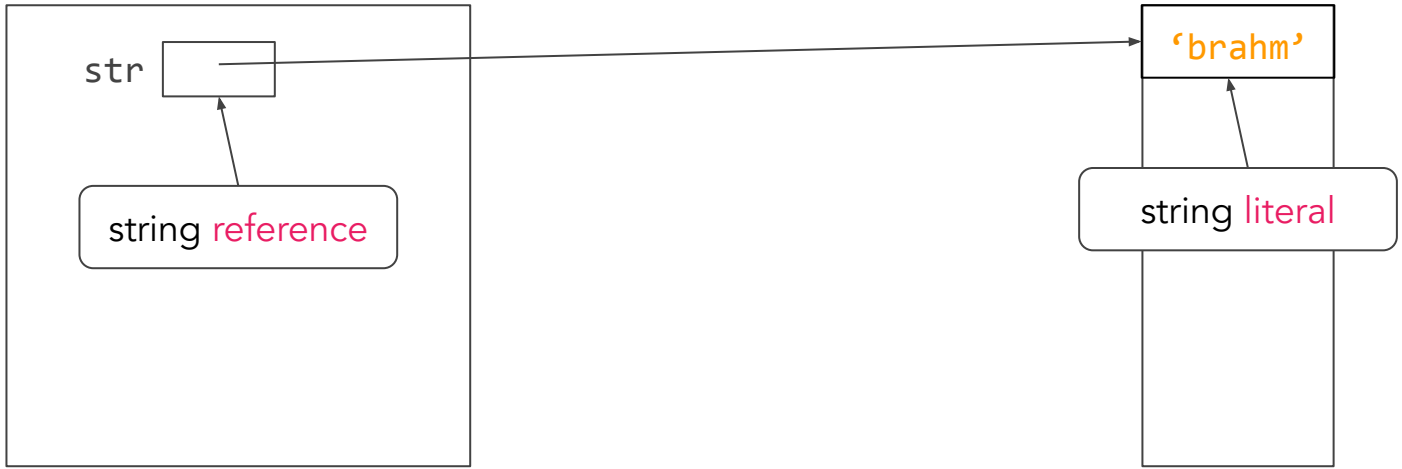
An important nuance: strings are immutable

```
str = 'brahm'
```



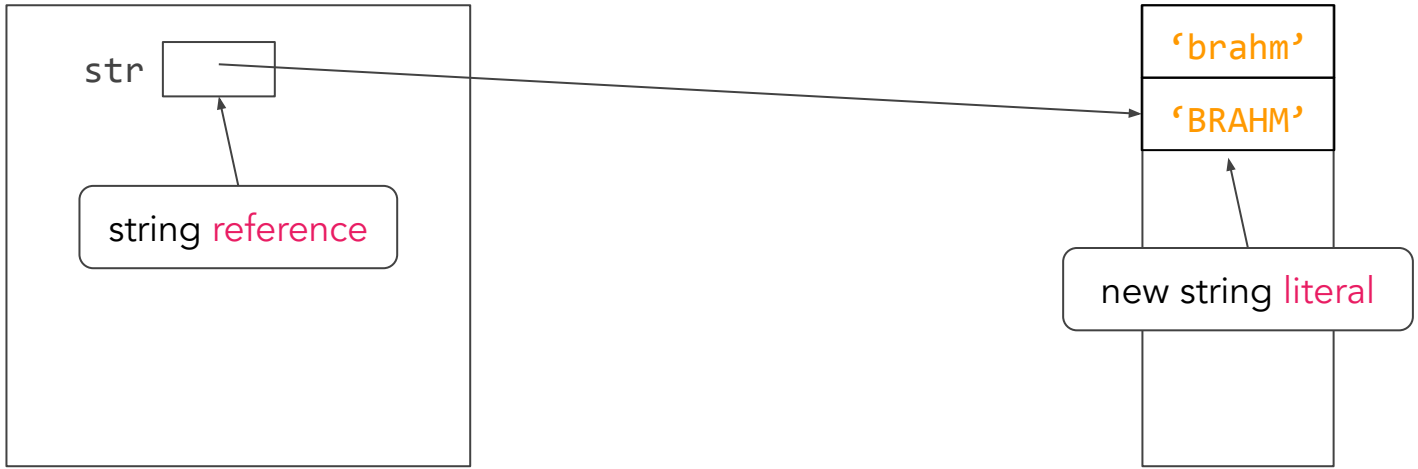
An important nuance: string literals are immutable

```
str = 'brahm'
```



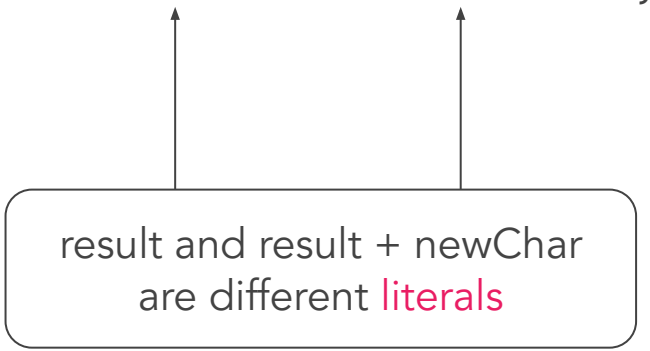
...but references aren't!

```
str = str.upper()
```



This leads to a common pattern for String problems

```
s = ‘banter’
result = ‘’
for i in range(len(s)):
     ch = s[i]
     newChar = # process ch
     result = result + newChar;
```



Why are Strings immutable?



There’s actually a cool reason! Come and chat about it afterwards or in office hours!

A problem: `remove_doubled_letters`

Implement the following function:

```
def remove_doubled_letters(s)
```

that takes in a String as a parameter and returns a new string with all doubled letters in the string replaced by a single letter. For example, calling

```
remove_doubled_letters('tresidder')
```

would return the string `'tresider'`. Similarly, if you call

```
remove_doubled_letters('bookkeeper')
```

the function would return the string `'bokeper'`.

Questions I’d ask myself

What do I do with each character?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one

Questions I'd ask myself

What do I do with each character?

If it isn't the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one

Is there anything else I'd need to think about?

Questions I'd ask myself

What do I do with each character?

If it isn't the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one

Is there anything else I'd need to think about?

The character at index 0 doesn't have a character before it but needs to go into the string

The solution

```
def removed_doubled_letters(s):
```

The solution

```
def removed_doubled_letters(s):
    result = ''
    for i in range(len(s)):
        ch = s[i]


    return result
```

The solution

```
def removed_doubled_letters(s):
    result = ''
    for i in range(len(s)):
        ch = s[i]
        if ch != s[i - 1]:
            result += ch
    return result
```

The solution

```
def removed_doubled_letters(s):
    result = ''
    for i in range(len(s)):
        ch = s[i]
        if i == 0 or ch != s[i - 1]:
            result += ch
    return result
```

Another problem: `extract_quote`

Implement the following function:

```
def extract_quote(s)
```

that takes in a String as a parameter and returns the text of the first quote in the string, or a blank string if there are no quotes. A quote is defined as a substring surrounded by two quotation marks (“).

You may assume that `s` only has quotation marks when it has quotes.

Another problem: `extract_quote`

Implement the following function:

```
def extract_quote(s)
```

that takes in a String as a parameter and returns the text of the first quote in the string, or a blank string if there are no quotes. A quote is defined as a substring surrounded by two quotation marks (“).

You may assume that `s` only has quotation marks when it has at least one quote.

My key insight

Do I need to loop over the characters here?

> *I probably could, but it sort of feels like the* `s.find()` *function is doing that for me*

The solution

```
def extract_quote(s):
    first_quote_index = s.find(‘“’)
    if first_quote_index == -1:
        return ‘’

    second_quote_index = s.find(‘“’, first_quote_index + 1)
    quote = s[first_quote_index + 1 : second_quote_index]

    return quote
```

Files

Files, in one slide

```
def print_out_file(filename):
    with open(filename) as f:
        for line in f:
            print(line, end=‘’)
```

Files, in ~~one~~ two slides

```
def print_out_file(filename):
    with open(filename) as f:
        for line in f:
            print(line, end='')
```

We open the file--extending a virtual cursor into it--and give that cursor a nickname of `f`

Files, in ~~one~~ ~~two~~ three slides

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

`f` is a cursor into collection of lines of text, so we can use a `for` loop to go through each of those lines

Files, in ~~one~~ ~~two~~ three slides

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

Each time the for loop happens, the cursor moves down to the next line

Files, in ~~one two three~~ four slides

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

Since the line in the file already has a newline character at the end, we shouldn't add another one when we print

```
Roses are red,\n
Violets are blue,\n
Poetry is hard,\n
So I give up.
```

Files, in ~~one two three four~~ don't worry about it slides

```
def process_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            # process the line
```

This is a general pattern you can use whenever you need to read through a file

Files, in ~~one two three four~~ don't worry about it slides

```
def process_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            # process the line
```

You can just pretty much stick this right in there without thinking! Just be careful to adjust the filename if needed.

Grids

Rather than go over grids from a theoretical perspective, let's go over the three kinds of things you'd be expected to write involving grids and the patterns associated:

1. A function that does some Boolean check of the grid
2. A function that potentially changes the grid
3. Writing small, targeted doctests

Checking portions of the grid

Usually, this will be checking that a coordinate is in bounds or has a particular value, which looks like this:

```
if grid.in_bounds(x, y):
    # legal coordinate
```

```
if grid.get(x, y) == <some value>:
    # process the value here
```

Changing the grid

`grid.get` and `grid.set` are your friends for problems that involving changing the grid. There are three questions involved in any changes to the grid:

1. Where do I get the value that needs to be put in a new spot?
   e.g. `val = grid.get(src_x, src_y)`

2. Where does this new value need to go?
   e.g `grid.set(src_y + 1, src_x, val)` moves `val` one spot down.

3. Do I need to do any cleanup/bookkeeping for the move?
   e.g `grid.set(src_x, src_y, None)` clears the source spot.

Writing doctests

Reasoning about (and making) large grids is annoying, so when you test your program's behaviour, come up with small examples that reflect the situations you're trying to test for. Usually, a 3x2 grid is enough to reason about a function.

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

A practice problem

Suppose we have a grid that represents Yosemite valley. Every square within the bounds of the grid is either a rock represented by `'r'`, tree `'t'`, or bear `'b'`. Write code for a `count_bears(grid, x, y)` function as follows: The given x,y is guaranteed be in bounds in the grid. Consider three squares: the square at x,y, the square to the left of x,y, and the square down from x,y. Return the count of the bears in those three squares, so the result is always one of 0, 1, 2, 3. So for example with this with=3 by height=2 grid

```
b b r
t b r
```

count at x,y = 0,0 -> 1
count at x,y = 1,0 -> 3
count at x,y = 2,0 -> 1

The solution

```
def count_bears(grid, x, y):
    count = 0


    return count
```

We know we need to return a `count`, so let's make one and return it at the end of the function (leaving ourselves strategic whitespace)

The solution

```
def count_bears(grid, x, y):
    count = 0

    if grid.get(x, y) == 'b':
        count += 1

    if grid.get(x - 1, y) == 'b':
        count += 1

    if grid.get(x, y + 1) == 'b':
        count += 1

     return count
```

Now, we need to check each of the coordinates that the question mentions: remember, (0, 0) is the top left corner.

The solution

```
def count_bears(grid, x, y):
    count = 0

    if grid.get(x, y) == 'b':
        count += 1

    if x - 1 >= 0 and grid.get(x - 1, y) == 'b':
        count += 1

    if y + 1 < grid.height and grid.get(x, y + 1) == 'b':
        count += 1

    return count
```

We know that (x, y) is in bounds, but we don't know about the coordinates of its neighbours, so we need to check those.

Lists

A list is a variable type representing a ~~list~~ linear collection of elements of any type:

```
num_list = [4, 2, 1, 3]
str_list = [‘ax’, ‘bc’]
```

They work pretty much exactly the same way as strings:

```
>>> len(num_list)
4
>>> str_list[1]
‘bc’
>>> num_list[1 : 3]
[2, 1]
```

You can make an empty list using square brackets

```
lst = []
```

And then stick stuff in it using the `append` method:

```
for i in range(5):
    lst.append(i)
# lst now is [0, 1, 2, 3, 4]
```

You can also stick another list at the end using the `extend` method:

```
second_lst = [8, 9, 10]
lst.extend(second_list)  # lst is now [0, 1, 2, 3, 4, 8, 9, 10]
```

Note that each of these functions modifies the list, rather than returning a new one.

You can also sort a list:

```
nums = [0, 9, 4, 5]
nums = sorted(nums)
```

and print it out:

```
print(nums)  # prints [0, 9, 4, 5]
```

and check whether it contains an element:

```
>>> 3 in nums
False
```

You can also put lists inside other lists!

```
>>> lst = []
>>> lst.append([1, 2, 3])
>>> lst.append([42, 100])
>>> print(lst)
[[1, 2, 3], [42, 100])
```



A problem: `multiples`

Implement the following function:

```
def multiples(lst)
```

that takes in as a parameter a list of numbers and returns a new list such that each of its elements a list containing the first 3 positive multiples of the corresponding element in the original list.

```
>>> multiples([5, 42, 100])
[[5, 10, 15], [42, 84, 126], [100, 200, 300]]
```

Your function should not modify the original `lst` parameter.

The solution

```
def multiples(lst):
    result = []
    for i in range(len(list):
        num = lst[i]



    return result
```

We'll start with a skeleton that looks mostly like our string problem structure: a `result`, a `for` loop, and a `return`.

The solution

```
def multiples(lst):
     result = []
     for i in range(len(list):
          num = lst[i]
          multiples_lst = []



     return result
```

Now, we're going to need to stick a list of multiples in `result`, so let's make that list

The solution

```
def multiples(lst):
    result = []
    for i in range(len(list):
        num = lst[i]
        multiples_lst = []


        result.append(multiples_lst)
    return result
```

Now, let's pretend that `multiples_lst` has the correct elements (it definitely doesn't, yet). We put it in `result`.

The solution

```
def multiples(lst):
     result = []
     for i in range(len(list):
          num = lst[i]
          multiples_lst = []
          for multiple in range(1, 4):
               multiples_lst.append(multiple * num)
          result.append(multiples_lst)
     return result
```

Now, we just need to fill `multiples_lst` with the first 3 multiples of `num`!

Parsing

When we're parsing a string, we're trying to extract *all* the substrings that match a particular pattern

- Extracting all the emails from a file
- Extracting words from noisy data
- Assignment 5 😱

There's a canonical structure we use to parse strings. If you understand that structure, you understand everything you need to for parsing!

An explanatory example: getting prerequisites

CS18SI: *William Janeway describes the relationship between technological development, capital markets, and the government as a three-player game. Scientists and entrepreneurs develop breakthrough innovations, aided and amplified by financial capital. Meanwhile, the government serves to either subsidize (as in wartime) or stymie (through regulations) technological development. Often, the advances in economic and military might due to technological advances lead to conflicts between competing countries, whether Japan and the U.S. in the 1970s, or China and the U.S. today. Within societies, technological innovation drives outcomes like increased life expectancy, wealth inequality, and in rare cases changes to paradigms of daily life. In this discussion-driven course, we will explore the ripple effects that technological developments have had and will continue to have on the geopolitical world stage, focusing on trends we as computer scientists are uniquely positioned to understand and predict the ramifications of. Prerequisites: The following are not required but will facilitate understanding of the topics covered: computer systems (CS110+), artificial intelligence (CS221, CS231N, CS229, or CS230), and theory (CS161, cryptography)*

An explanatory example: getting prerequisites

CS18SI: *William Janeway describes the relationship between technological development, capital markets, and the government as a three-player game. Scientists and entrepreneurs develop breakthrough innovations, aided and amplified by financial capital. Meanwhile, the government serves to either subsidize (as in wartime) or stymie (through regulations) technological development. Often, the advances in economic and military might due to technological advances lead to conflicts between competing countries, whether Japan and the U.S. in the 1970s, or China and the U.S. today. Within societies, technological innovation drives outcomes like increased life expectancy, wealth inequality, and in rare cases changes to paradigms of daily life. In this discussion-driven course, we will explore the ripple effects that technological developments have had and will continue to have on the geopolitical world stage, focusing on trends we as computer scientists are uniquely positioned to understand and predict the ramifications of. Prerequisites: The following are not required but will facilitate understanding of the topics covered: computer systems (CS110+), artificial intelligence (CS221, CS231N, CS229, or CS230), and theory (CS161, cryptography)*

An explanatory example: getting prerequisites

CS18SI: *William Janeway describes the relationship between technological development, capital markets, and the government as a three-player game. Scientists and entrepreneurs develop breakthrough innovations, aided and amplified by financial capital. Meanwhile, the government serves to either subsidize (as in wartime) or stymie (through regulations) technological development. Often, the advances in economic and military might due to technological advances lead to conflicts between competing countries, whether Japan and the U.S. in the 1970s, or China and the U.S. today. Within societies, technological innovation drives outcomes like increased life expectancy, wealth inequality, and in rare cases changes to paradigms of daily life. In this discussion-driven course, we will explore the ripple effects that technological developments have had and will continue to have on the geopolitical world stage, focusing on trends we as computer scientists are uniquely positioned to understand and predict the ramifications of. Prerequisites: The following are not required but will facilitate understanding of the topics covered: computer systems (CS110+), artificial intelligence (CS221, CS231N, CS229, or CS230), and theory (CS161, cryptography).*

Let's define a class name as *at least one letter*, followed by a sequence of letters and numbers. Given a string containing a class's description, we want to parse out all of the classes mentioned in the description.

How we do it

```
def get_prerequisites(s):
    prereqs = []


    begin = # ¯\_(ツ)_/¯




    end = # ¯\_(ツ)_/¯ ¯\_(ツ)_/¯


    prereq = s[begin : end]
    prereqs.append(prereq)

    return prereqs
```

Let's start simple, and pretend we only need to find one prereq, and magically know exactly where in the description it starts and ends

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = # ¯\_(ツ)_/¯

        end = # ¯\_(ツ)_/¯ ¯\_(ツ)_/¯

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

We need to do this for multiple prereqs, so let's use a loop. We'll keep pretending that **`begin`** and **`end`** are always correct, and we'll use a `while True` loop because we're lazy. We'll worry about ending the loop later.

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = 0

        end = # ¯\_(ツ)_/¯ ¯\_(ツ)_/¯

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

Now, let's try and get **`begin`** correct for the first prereq in the description. Because it needs some initial value, we'll just say `0` (the start of the string).

How we do it

```
def get_prerequisites(s):
    prereqs = []


    while True:
        begin = 0
        while not s[begin].isalpha():
            begin += 1




        end = # ¯\_(ツ)_/¯ ¯\_(ツ)_/¯



        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

Now, let's move **begin** along the string until we find an alphabetical character. So long as we're not at an alphabetical character yet, we move one to the right by adding 1 to **begin**.

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = 0
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        end = # ¯\_(ツ)_/¯ ¯\_(ツ)_/¯

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

We should probably make sure that **begin** doesn't fall off the end of the string. The maximum valid index in s is `len(s) - 1`, so we can just make sure that **begin** `< len(s)`

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = 0
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = # ¯\_(ツ)_/¯ ¯\_(ツ)_/¯



        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

Now that we're checking `begin < len(s)`, the inner `while` loop *might* end because there weren't any alphabetical characters and so we reached the end of the string. If that's the case, we should stop searching.

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = 0
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1



        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

Now that we have a correct value for **begin**, we know that **end** is *at least* one greater than **begin**, since the class name is at least one character long.

How we do it

```
def get_prerequisites(s):
    prereqs = []


    while True:
        begin = 0
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

Now, let's make sure **end** is correct. So long as it's at an alphanumeric character, we should shift it along one. When the loop ends, it should point at the *first* non-alphanumeric character after **begin**.

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = 0
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

Just like the last time, we should also make sure that **end** doesn't fall off the end of the string.

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = 0
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

This looks great! The only problem is that every time the outer while True loop restarts, we start searching from the beginning again. Ideally, we'd start searching from the end of the last prereq we found.

How we do it

Let's pretend we have a variable called **search**, which magically tells us where we should start searching from inthe while True loop. Now, at the beginning of the loop, we can just set **begin** equal to **search** instead of 0.

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = search
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)

    return prereqs
```

How we do it

```
def get_prerequisites(s):
    prereqs = []

    while True:
        begin = search
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)
        search = end + 1
    return prereqs
```

Then, at the end of the loop, we should indicate that we should search from the end of the prereq we just found. We know **end** is pointing at a non-alphanumeric character, so we can start searching from the one after that.

How we do it

```
def get_prerequisites(s):
    prereqs = []
    search = 0

    while True:
        begin = search
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)
        search = end + 1
    return prereqs
```

Now, all we need to do is make sure search has some initial value for when we begin. Since we want to start searching at the beginning of the string, we can just set it to `0`.

How we do it

```
def get_prerequisites(s):
    prereqs = []
    search = 0

    while True:
        begin = search
        while begin < len(s) and not s[begin].isalpha():
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum():
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)
        search = end + 1
    return prereqs
```

And we're done!

How we do it

```
def get_prerequisites(s):
    prereqs = []
    search = 0

    while True:
        begin = search
        while begin < len(s) and not s[begin]
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)
        search = end + 1
    return prereqs
```

A note: in practice, this doesn't work very well. Can you figure out why?

How we do it

```
def get_prerequisites(s):
    prereqs = []
    search = 0

    while True:
        begin = search
        while begin < len(s) and not s[begin]
            begin += 1

        if begin >= len(s):
            break

        end = begin + 1
        while end < len(s) and s[end].isalnum
            end += 1

        prereq = s[begin : end]
        prereqs.append(prereq)
        search = end + 1
    return prereqs
```

While it does correctly parse out all of the class names, it also parses out every single word in the course description.

One fix that seems to work is just checking that the prereq contains at least one number before appending it.

I omitted this because it's not the main focus here, but be sure to think about these sorts of things in your parsing solutions (they're rare, but not impossible).

This is a really hard piece of code to understand, and is a strict ceiling on how difficult the parsing problems on the exam will be.

Make sure you can understand it, and also answer these questions for yourself:

- Why don't we need to check `if end >= len(s)` and `break` the loop, like we do with `begin`?
- We made the decision to use a `while True` loop here because we were being lazy. Why is that actually the best decision? What's the difference between a `while True` loop and the other `while` loops you've seen so far?
- Would there have been any problems if we had set `search=end`, rather than `search=end + 1`? Also, is the `search` variable even necessary?

If you've internalized that code, and can answer those questions, you should be good to go for parsing problems. Understand also how you could adapt this structure to solve other parse problems you've seen in lecture or homework.

The answers to those questions are on the next few slides, but you should think about them and try and figure out your answers first.

Why don't we need to check `if end >= len(s)` and `break` the loop, like we do with `begin`?

Answer: the loop that increments end definitely could finish with `end` *pointing at the end of the string, but that's totally fine! We're not using* `end` *as an inclusive index (since slices are exclusive of the second index).*

The next time the `while True` *loop begins, search will be pointing off the end of the string, and so the* `while` *loop controlling* `begin` *won't even start and we'll break out of the* `while True` *loop because* `begin` *will be greater than the index at the end of the string.*

We made the decision to use a `while True` loop here because we were being lazy. Why is that actually the best decision? What's the difference between a `while True` loop and the other `while` loops you've seen so far?

A `while True` loop — combined with a `break` somewhere in the body of the loop — is sometimes called a loop and a half. The idea is that before you check the condition of the `while` loop, you want to do some work. For example, in this case, we want to check whether `begin` is pointing at the end of the string, but before we do that, we want to increment it past all the non-alphabetic characters we can. A loop and a half is a nice way of being able to do that work inside the while loop.

You could probably finesse some way of using a more conventional `while` loop here, but all the solutions I could think of are pretty messy and less generalizable to other parsing problems, so I'd caution against it.

Would there have been any problems if we had set `search=end`, rather than `search=end + 1`? Also, is the `search` variable even necessary?

Nope! End is pointing at a non-alphanumeric character, so we don't need to search through it, but we could, if we wanted to.

A nice intuition is that each of the inner `while` *loops acts sort of like a* `str.find()` *function, except instead of searching for a particular string, we're searching for a particular pattern of string. This helps explain why it's fine to search from a character that you know is not correct, but it won't break anything.*

And no, the `search` *variable isn't actually necessary. You could set begin to be equal to 0 at the beginning of the loop and equal to end at the end of the loop, and it would work the same way. This is a question of decomposition-by-variable, though.* `search` *represents where we'd like to start looking from, and* `begin` *represents the start of a particular pattern for us. Because those feel like different things, I'd store them in different variables.*

Parting words

“What should I do in these last 4 days?”

“What should I do in these last 4 days?”

Keep practicing! Come to office hours and the LaIR! Post on Piazza! Aim to appreciate the big ideas and the algorithmic details, not the syntax!

“What should I do in these last 4 days?”

Keep practicing! Come to office hours and the LaIR! Post on Piazza! Aim to appreciate the big ideas and the algorithmic details, not the syntax! Sleep!

We are not trying to trick you in this exam. You know how to do all these problems.

For many of you, this is your first programming exam. We know it's a weird thing to do, and we are doing our utmost to ensure that if you understand this material, you will do well.

Stay calm, focus on how to apply what you know to the problems, and keep making them easier for yourself, as we did in each of the problems tonight.

We are not trying to trick you in this exam. You know how to do all these problems.

For many of you, this is your first programming exam. We know it's a weird thing to do, and we are doing our utmost to ensure that if you understand this material, you will do well.

Stay calm, focus on how to apply what you know to the problems, and keep making them easier for yourself, as we did in each of the problems tonight.

Good luck!