Today: image co-ordinates, range, nested range, make drawing to figure out co-ords, using a parameter.

The live image problems today are linked into the notes below, also available in the experimental server sections image-nested (nested loops) and image-shift (x,y coords shift around)

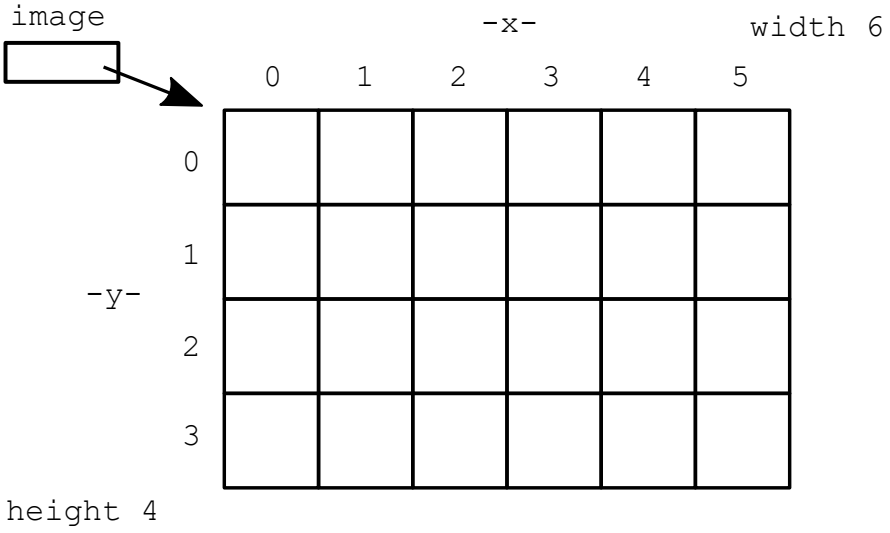
Strategy Note: Detail Oriented

- "Detail Oriented" - frequently seen on the resume!
- Book: Thinking Fast and Slow, by Kahneman
- Mostly your brain is not paying much attention and that's fine
- Get to Aqua-Stripe problem below .. very detail oriented
  Slow down and work the details
  Do not do it in your head
  Make a rough drawing to plot how the details fit together
  Then write the key line of code

Goal: Loop Over All x,y For an Image

We'll build up techniques below, and our goal is using the `range()` function and nested loops to go through all x,y of an image.

Here is a diagram of an image of 6 pixels width and 4 pixels height. First we'll look at the **x** and **y** numbers for all the coordinates.



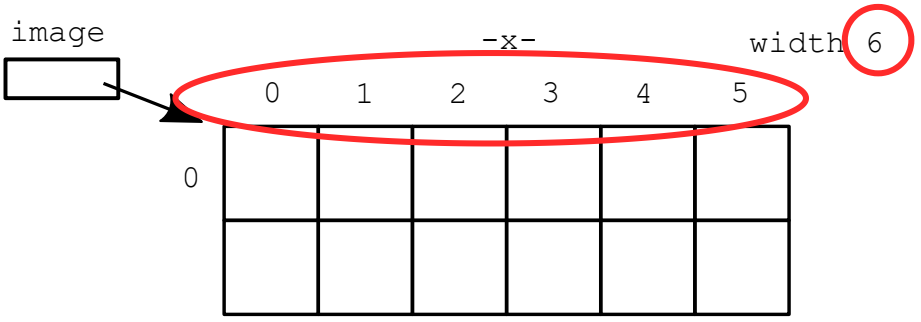
Zero Based Indexing

The pixels are numbered starting with 0 - "zero based indexing". This is an incredibly common scheme within computers, so you'll get used to it. Zero based indexing makes the math come out cleaner for some cases, which is why it is prevalent down here in the weeds.

Look at the top row of pixels.

The first pixel is at x=0, the next is x=1, and so on up to x=5 for the last pixel at the right side.

Key it's easy to think — well it's width 6 so the rightmost pixel is at x = 6. Nope! In zero based indexing, the last pixel is at x = (width - 1), or 5 as we see in this case.



More generally, if you have `n` things with zero-based indexing, the first is at `0` and the last is at `n - 1`.

This is not deeply difficult, but it's an easy "off by one error" (OBO) to make. We'll talk about OBO more later.

`range(n)` Function

See the Python Guide range

```
range(10) -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(6)  -> [0, 1, 2, 3, 4, 5]
range(3)  -> [0, 1, 2]

range(n)  -> [0, 1, 2 .... n-1]  # UBNI
```

- (last thing mon lecture)
- `range(n)` function
- We'll use this constantly, memorize how it works
- Given n, return the series of numbers 0..n-1

- e.g. `range(10)` -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- e.g. `range(5)` -> 0, 1, 2, 3, 4
- Start at 0, Up To But Not Including n - **UBNI**
- Alternate phrasing: `range(n)` returns n numbers starting with 0
- Design of range() fits perfectly with zero-based indexing
- With image **width** and **height**
  range(n) is perfect for generating x and y values into an image

for/range in Interpreter >>>

We can use the experimental server interpreter >>> for this as demo or exercise.

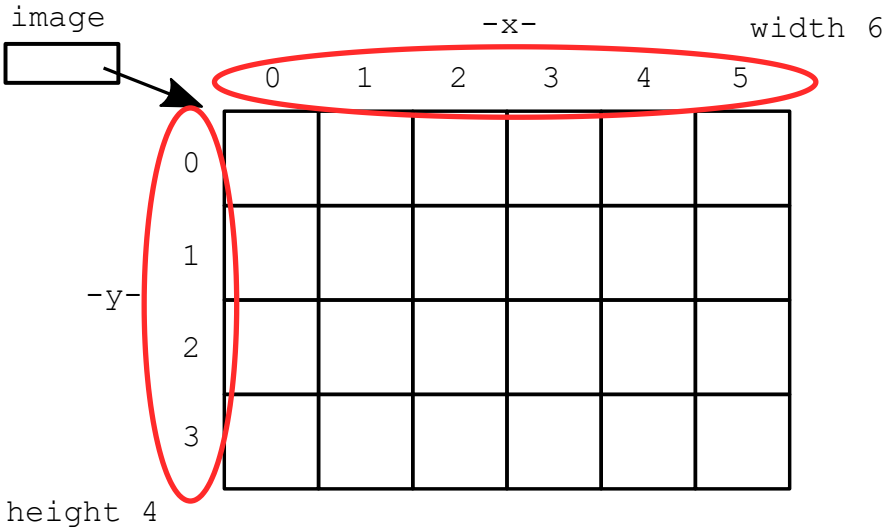
- The numbers generated by `range()` work in a for-loop
- e.g. `for x in range(10):`
- In loop:
  Each iteration of loop, x points to next value
  x points to: 0, 1, 2, 3 ..., 8, 9
- For now, the print() function displays what is passed to it in the parens
- See here how `range(10)` returns 10 numbers 0..9
- Try different numbers
- **Use up-arrow** to recall previous lines - super handy!

```
>>> for x in range(10):
        print('in loop:', x)
in loop: 0
in loop: 1
in loop: 2
in loop: 3
in loop: 4
in loop: 5
in loop: 6
in loop: 7
in loop: 8
in loop: 9
>>>
```

Observe: `range(10)` generates the numbers 0..9. Run the for loop over those numbers. The variable `x` points to each number from the collection, one number per loop body. The `print()` function prints out each one on a line, and we'll learn more about that later.

Demo: try a bigger number. Use the up-arrow (!!). "Sibling mode" - change the text to something like 'Nick is a big doofus' and then use the number 1000.

Want to Generate x,y Numbers For Image



1. Use range() - `range(image.width)`

Use range() to generate the numbers based on the image width/height.

```
# here image.width is 6
# image.height is 4

range(image.width)  -> [0, 1, 2, 3, 4, 5]
range(image.height) -> [0, 1, 2, 3]
```

2. Use for-x and for-y Loops

Use for loops to loop over the numbers from range():

```
for x in range(image.width):
    # x is 0, 1, 2 ...
    ...
```

```
for y in range(image.height):
    # y is 0, 1, 2
    ...
```

How To Combine the for-x and for-y Loops?

How to combine the two loops to generate all the x,y? Nested loops, a classic structure.

Nested Loops

- Need "nested" Loops to generate all x,y
- An "outer" loop first
- An "inner" loop nested inside the outer loop

Say we have an image width 6, height 4. Here are the nested loops to cover all its x,y. To go over an image, the "y" loop is first, and the "x" loop is nested inside it. With this structure, the loops go over the image from top to bottom.

```
for y in range(4):              # outer
    for x in range(6):          # inner
        # use x,y in here
```

How Does Nested Work?

Each run of a loop body is called an **iteration**. Here is the key rule: For each single iteration of outer, get all the iterations of inner.

The outer loop does one iteration (e,g, y = 0). Then inner goes through all the x values, x = 0, 1, 2, 3, 4, 5. Then outer does one iteration (y = 1), and inner goes through all the x values again.

```
y = 0                   # outer, y = 0
x = 0, 1, 2, .. 5       # inner, x goes through all

y = 1                   # outer, y = next value
x = 0, 1, 2, .. 5       # inner, x through all again

y = 2                   # outer, y = next value
```

```
  x = 0, 1, 2, .. 5    # inner, x through all again

  ...
```

e.g. y = 0, go through all the x's 0, 1, 2 .. 4, 5. Then for y = 1, go through all the x's again.

Nested Loop In Interpreter >>>

The print() function is standard Python and we'll use it more later. It takes one or more values separated by commas value in the parenthesis and prints them out as a line of text.

Run the nested loops in the experimental server (interpreter). You can see the key rule in action - one iteration of the outer loop selects one y number, and for that one y, the inner loop go through all the x numbers:

```
>>> for y in range(4):
        for x in range(6):
            print('x:', x, 'y:', y)
x: 0 y: 0
x: 1 y: 0
x: 2 y: 0
x: 3 y: 0
x: 4 y: 0
x: 5 y: 0
x: 0 y: 1
x: 1 y: 1
x: 2 y: 1
x: 3 y: 1
x: 4 y: 1
x: 5 y: 1
x: 0 y: 2
x: 1 y: 2
x: 2 y: 2
x: 3 y: 2
x: 4 y: 2
x: 5 y: 2
x: 0 y: 3
x: 1 y: 3
x: 2 y: 3
```

```
x: 3 y: 3
x: 4 y: 3
x: 5 y: 3
```

Why is y loop first? This way we go top to bottom — y=0, then y=1 and so on. This is the standard, traditional order for code to loop over an image, so we'll always do it this way (and if you encounter image code out in the wiworldld someday, it will tend to do it in this order too).

Nested Loop Visualization

Here is a picture, showing the order the nested y/x loops go through all the pixels - all of the top y=0 row, then the next y=1 row, and so on. This the same order as reading English text from top to bottom.

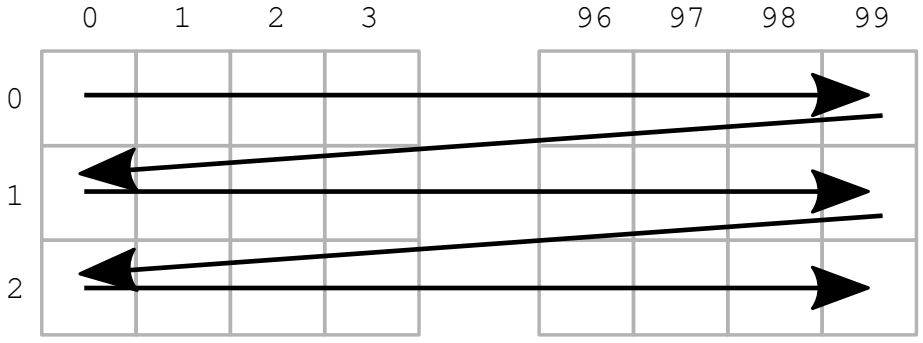
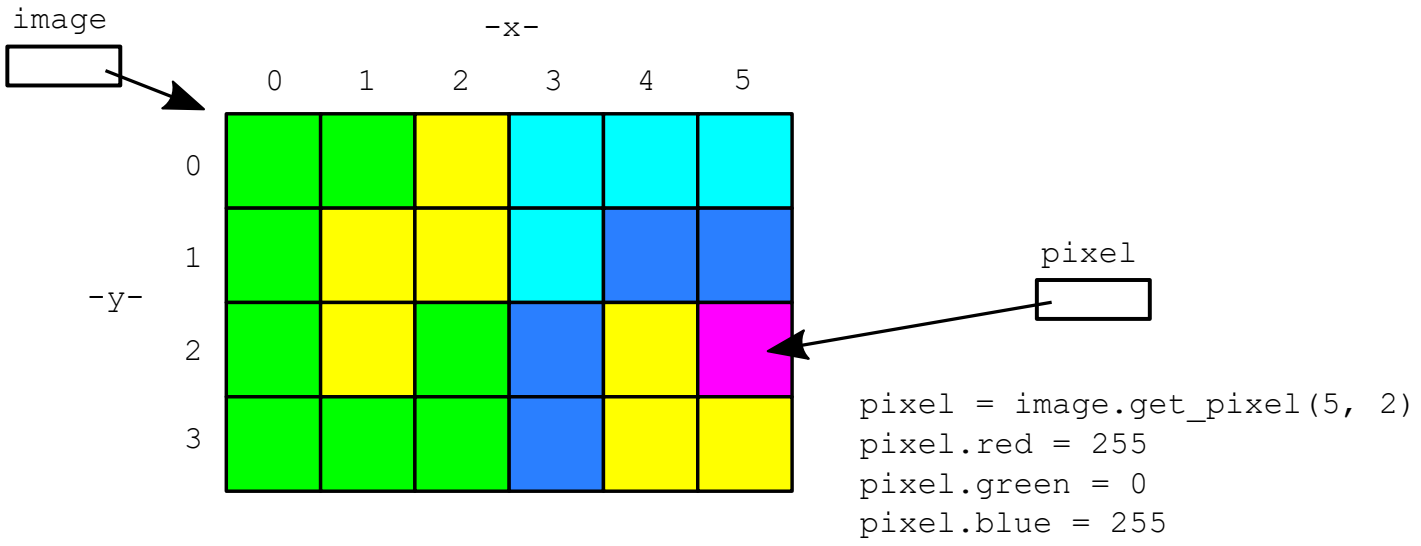


image.get_pixel(x, y)

- An image function that accesses one pixel
- `image.get_pixel(x, y)`
- If we have x,y numbers, gives us reference to that pixel
- Store reference in a variable, use .red .green on it
- Typically we use "pixel" as the variable name for this
- Error if the x,y are not valid, in-bounds numbers for the image

```
# get pixel at x=5 y=2 in "image",
# can use its .red etc.
pixel = image.get_pixel(5, 2)
pixel.red = 0
```



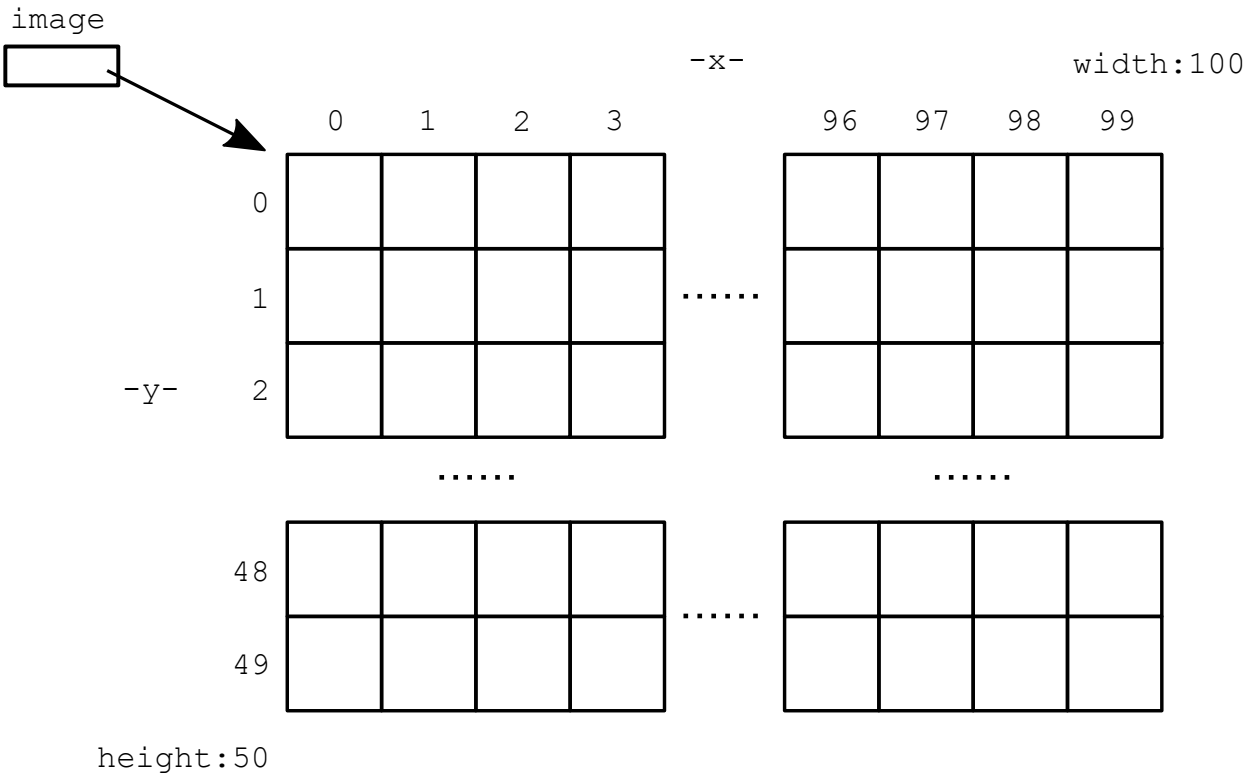
Example: Darker-Nested

This code works - pulls together all of the earlier topics in a running example.

> Darker Nested

Here is a version of our earlier "darker" algorithm, but written using nested range() loops. The nested loops load every pixel in the image and change the pixel to be darker. On the last line `return image` outputs the image at the end of the function (more on "return" next week). Run it to see what it outputs. Then we'll look at the code in detail.

```
def darker(filename):
    image = SimpleImage(filename)
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            pixel.red *= 0.5
            pixel.green *= 0.5
            pixel.blue *= 0.5
    return image
```



Observe Darker Nested Obsercations

- Standard for-y/for-x nested loops to go through all x,y (as above)
  Inside the loops..
  Call the `image.get_pixel()` function with current x,y
  Store reference to pixel in variable named "pixel"
- Operate on pixel
  `pixel.red *= 0.5`
- The result is
  Loop over whole image
  Access each pixel
  Operate on each pixel

Demos With Darker Nested

- Demo: add in inner loop: `print('x:', x, 'y:', y)`
  print() like this is a debugging trick we'll build on later
  See a line printed for each pixel, showing the whole x,y sequence
  Note the **return** is the formal function output, print output is secondary but handy for debugging sometimes
  Best with small images, otherwise print can produce astronomical amount of output

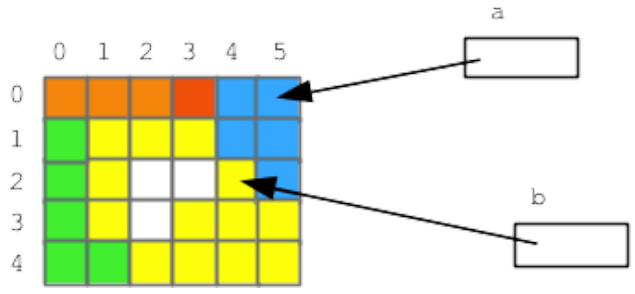
- Demo: try making x loop smaller - `range(image.width - 20)` - what do you see?
- Demo: what if we swap the roles of the y/x loops?
  It works fine, just a different order of the pixels
  Goes through the columns left-right. Then for each column, top to bottom
  That said, usually do it the standard for-y/for-x way

Nested y/x - Idiomatic

The nested y/x loop code has a lot of detail in it. The good news is, writing the loops this way to hit every pixel in an image is idiomatic. It's the same every time, so you can use it as a unit while you get used to its details.

How to Make 2 Pixels Look The Same?

- Suppose variables a and b refer to two pixels in an image
- Make pixel b look the same as a .. how?
- Pixels look the same if they have the same RGB numbers
- Make b look the same as a with three = for red/green/blue




```
# Make pixel b look the same as pixel a
b.red = a.red
b.green = a.green
b.blue = a.blue
```

How to Create a New, Blank Image?

Thus far the code has changed the original image. Now we'll create a new blank white "out" image and write changes to that. Here are a few examples of creating a new, blank image.

```
# Say filename is 'poppy.jpg' or whatever
# This loads that image into memory
image = SimpleImage(filename)

# 1. create a blank white 100 x 50 image, store in
variable
# named "out"
out = SimpleImage.blank(100, 50)

# 2. Create an out2 image the same size as the
original
out2 = SimpleImage.blank(image.width, image.height)

# 3. Create an image twice as wide as the original
image_wide = SimpleImage.blank(image.width * 2,
image.height)
```

Code With Two Images - getPixel()

In the code below, we have two images "image" and "out" - how to obtain a pixel in one image or the other? The key is which image is **before the dot** when the code calls get_pixel(). This is the essence of noun.verb function call form. Which image do we address the get_pixel() function call to?

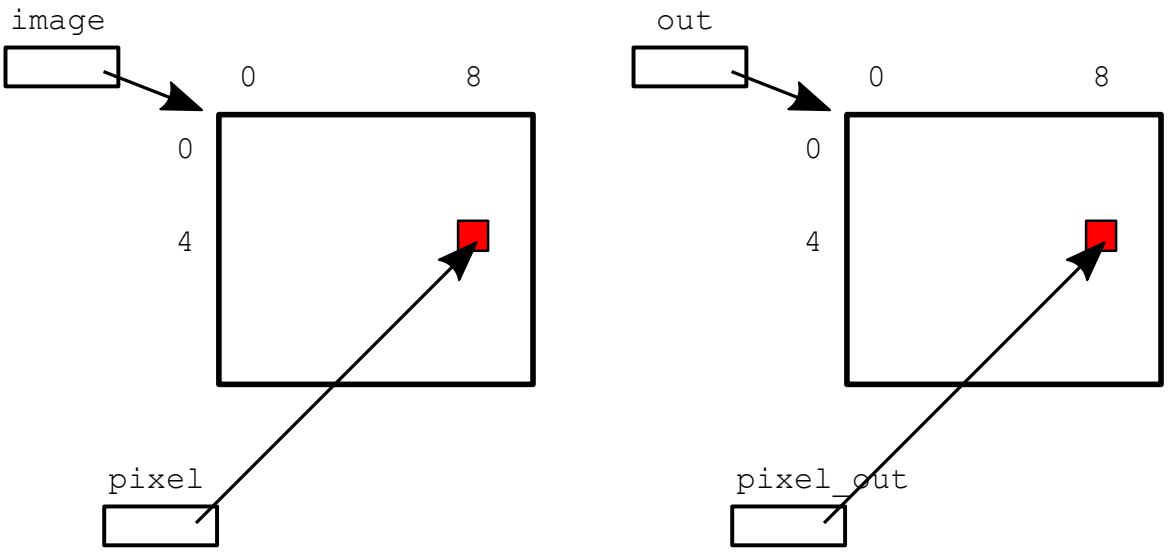
```
# Create original image
image = SimpleImage(filename)

# pixel points to pixel (8, 4) in original image
pixel = image.get_pixel(8, 4)


# Create out image, same size
out = SimpleImage.blank(image.width, image.height)

# pixel_out points to pixel (8, 4) in out image
pixel_out = out.get_pixel(8, 4)

# Could copy red from one to the other
pixel_out.red = pixel.red
```



Example: Darker Out

> Darker Out

The same "darker" algorithm, here writing the darker pixels to a separate "out" image, leaving the original image unchanged.

The "return xxx" line returns a completed value back to the caller code. Often the last line of a function. We'll use it in more detail later, but for these examples, it returns our "result" image.

Demo: try commenting out the return line. What does the function run do now?

Demo: try changing last line from `return out` to `return image` - what do you see and why?

```
def darker(filename):
    image = SimpleImage(filename)
    # Create out image, same size as original
    out =  SimpleImage.blank(image.width,
image.height)
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            pixel_out = out.get_pixel(x, y)
            pixel_out.red = pixel.red * 0.5
            pixel_out.green = pixel.green * 0.5
            pixel_out.blue = pixel.blue * 0.5
    return out
```

- Standard nested loops go through all x,y
- Variable `pixel` - points into original image
- Variable `pixel_out` - points into out image
- Copy over the data
- Return "out" when done
- Style: algorithm has two similar values, such as the two pixels here
  Use distinctive variable names to avoid confusing the two
  You can imagine a whole class of bugs where you meant to use one pixel but accidentally used the other
  Clear, distinct variable names can help

Aqua 10 - (skipping in lecture)

This shows writing pixels in the output at a different location than the input.

> Aqua 10

For the Aqua 10 problem, produce an image with a 10 pixel wide aqua stripe on the left, with a copy of the original image next to it, like this:

1. Make out image - 10 pixels wider than original

```
out = SimpleImage.blank(image.width + 10,
image.height)
```

2. How To Make Aqua Color?

- How to make aqua?
- White is 255 255 255
- Set red of white pixel 0
- That makes 0 255 255
- aka blue + green
- That's aqua!
- Someday when you are floating in warm, aqua waters .. we trust you will think back to your love of the RGB color system!

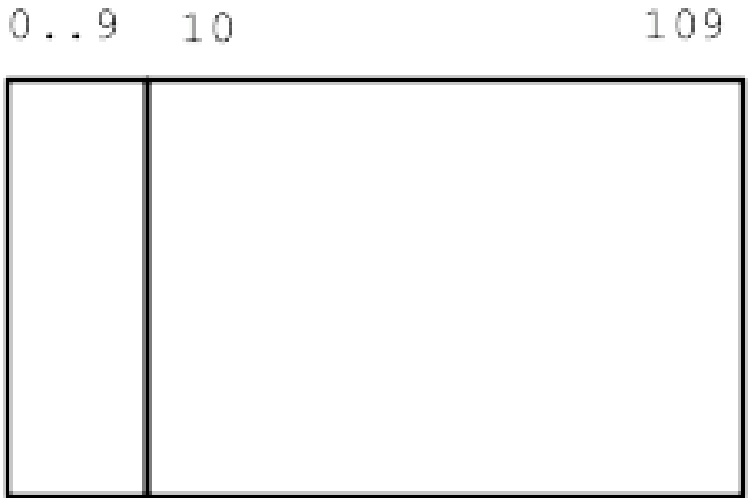
3. How To Loop Over the Stripe

- What are the x,y coordinates for 10 pixel wide stripe?
- What nested loops will hit them all?
- Run this first to see it work

```
# Create the 10-pixel aqua stripe
for y in range(image.height):
    for x in range(10):
        pixel_out = out.get_pixel(x, y)
        pixel_out.red = 0
```

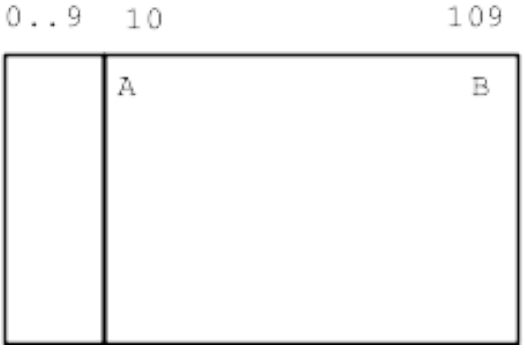
Drawing to think about coordinates...

```
original width is 100
out width is 110
```



4. Loop To Copy Over Original - Drawing!

How to copy the data from the original to the right side of the output? Need to think about the x,y values, which are not the same in the two images.



For x,y in original, what is the corresponding x,y in out?

Make a little chart (here "x" means in the original image)

```
pt    x      x_out
A     0      10
B     99     109

What's the pattern?
x_out = x + 10
```

Now we can write the key get_pixel() line below, to figure pixel_out for each original pixel.

Aqua 10 Solution

```
def aqua_stripe(filename):
    """
    Create an out image 10 pixels wider than the
original.
    (1) Set an aqua colored vertical stripe 10
pixels wide
    at the left by setting red to 0.
    (2) Copy the original image just to the right
    of the aqua stripe. Return the computed out
image.
    """
    image = SimpleImage(filename)
    # Create out image, 10 pixels wider than
original
    out = SimpleImage.blank(image.width + 10,
image.height)
    # Create the 10-pixel aqua stripe
    for y in range(image.height):
        for x in range(10):
            pixel_out = out.get_pixel(x, y)
            pixel_out.red = 0
    # Copy the original over - make drawing to guide
code here
    for y in range(image.height):
        for x in range(image.width):
```

```
            pixel = image.get_pixel(x, y)
            pixel_out = out.get_pixel(x + 10, y)   #
key line
            pixel_out.red = pixel.red
            pixel_out.green = pixel.green
            pixel_out.blue = pixel.blue
    return out
```

Experiments: try +11 instead of +10 - get bad-coord exception, Try +9, and image shifted slightly to left, can try the diff-stripes slider. (Can also try errors in mirror2 below.)

Strategy - Make a Drawing

It's hard to write the get_pixel() line with its coordinates just right doing it in your head. We make a drawing and take our time to get the details exactly right.

Concrete Numbers

Notice that our drawing was not general - just picking width = 100 as a concrete example. A single concrete example was good enough to get our thoughts organized, and then the formula worked out actually was general.

Off By One, OBO

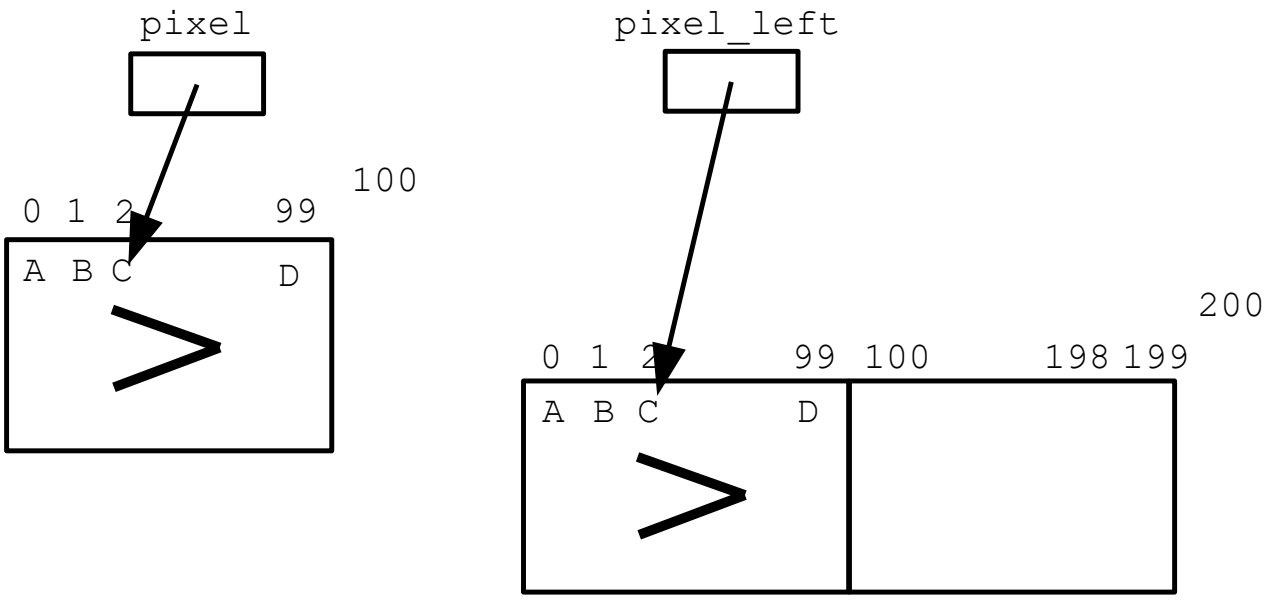
A common form of error in these complex indexing algorithms is being "off by one", like accessing the pixel at x = 100 when x = 99 is correct.

Example - Mirror1

> Mirror1

- This one is relatively simple, code is provided
- This problem will have two images in it
- Can call get_pixel() to obtain pixels in both
- Create a blank "out" image, twice as wide as original image:
  `out = SimpleImage.blank(image.width * 2, image.height)`
- This problem:

- 1. Create "out" twice as wide as original
- 2. Copy original image to left half
- Look at the get_pixel() calls
- "pixel" is in original image
- "pixel_left" is in out image (left half)
- In this case the x,y coords are the same in both images




```
def mirror1(filename):
    image = SimpleImage(filename)
    # Create out image with width * 2 of first image
    out = SimpleImage.blank(image.width * 2,
image.height)
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            # left copy
            pixel_left = out.get_pixel(x, y)
            pixel_left.red = pixel.red
            pixel_left.green = pixel.green
            pixel_left.blue = pixel.blue
            # right copy
            # nothing!
    return out
```

Exercise - Mirror2

> Mirror2

This a favorite example, bringing it all together. This algorithm pushes you to work out details carefully with a drawing, and the output is just neat.

Mirror2: Like mirror1, but also copy the original image to the right half of "out", but as a horizontally flipped mirror image. So the left half is a regular copy, and the right half is a mirror image. (Starter code does the left half).

Mirror2 Strategy

I think a reasonable reaction to reading that problem statement is: uh, what? How the heck is that going to work? But proceeding carefully we can get the details right.

Make a drawing of the image coordinates with concrete numbers, work out what the x,y coordinates are for input and output pixels. We'll go though the whole sequence right here.

- Like mirror1, place left-right swapped mirror image on right half
- A complex problem with coordinates, make a drawing
- Choose concrete numbers to work out an example
- e.g. say original width = 100
- Make diagram, say 4 ABCD "source" points
- Choose good variable names
  Use var names to keep the roles clear in the algo
  `pixel` - in source image, at x,y
  `pixel_left` - left side of out
  `pixel_right` - right side of out
- Key question:
  For each x in the original image, what is the x for pixel_right in the out image?

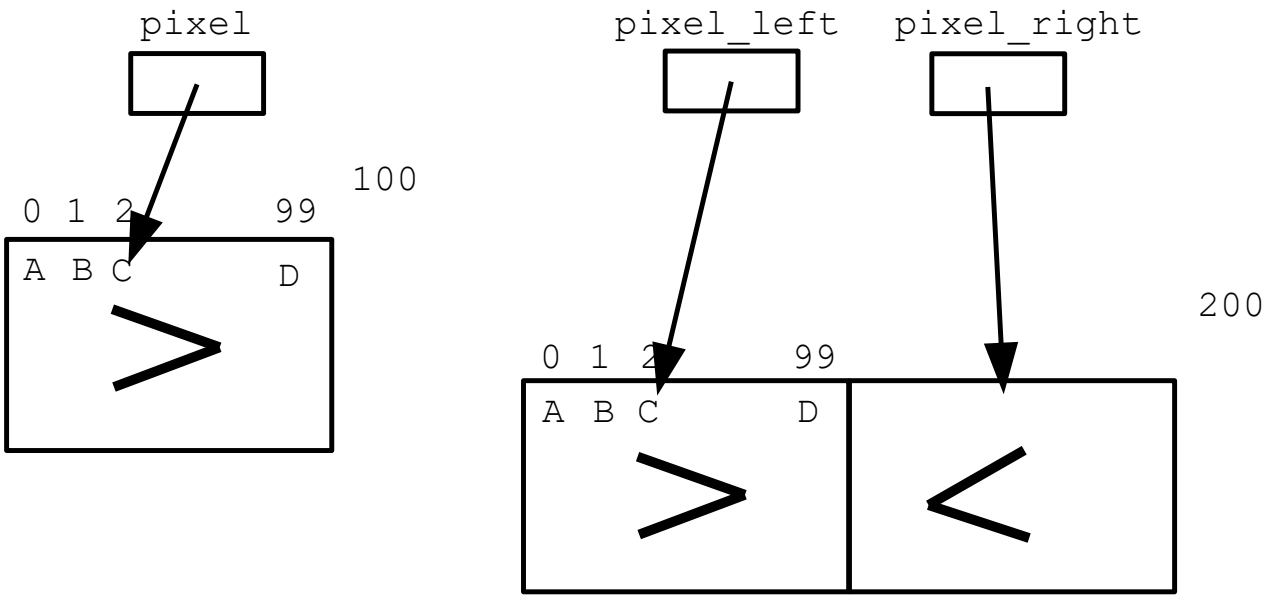
```
Here are 4 points in the original:
A: (0, 0)
B: (1, 0)
C: (2, 0)
D: (99, 0)

Sketch out where these points should land in the
```

output.
What is the x value for pixel_right for each of these?

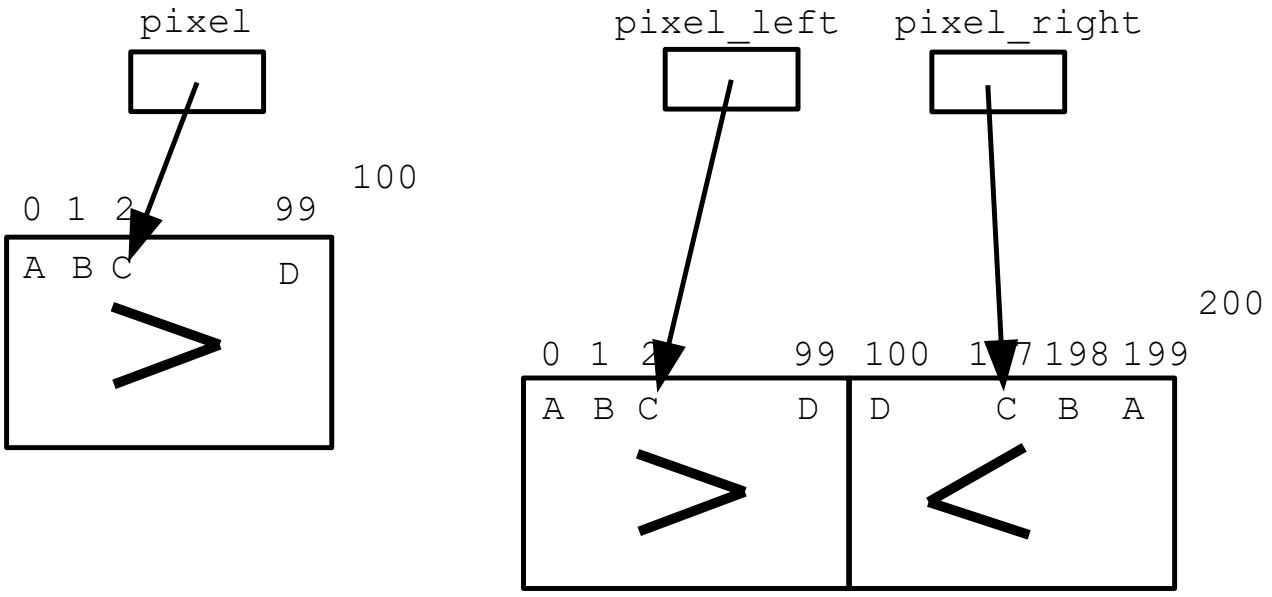
Sketch out ABCD Values

Try completing drawing with ABCD values. This is a great example of slowing down, working out the details. We start knowing what we want the output to look like, proceed down to the coordinate details.



- Figure out the formula to compute out pixel_right x,y from source x,y
- (Demo: go through points, figure out formula)
- (Demo: put in wrong formula, see bad x,y error message)
- Strategy: concrete numbers + drawing .. work out code details
- Diff slider draws yellow bars where the output seems wrong to the system

Here is the drawing with the numbers filled in



ABCD Table Solution Show

mirror2 Solution Code

```
def mirror2(filename):
    image = SimpleImage(filename)
    out = SimpleImage.blank(image.width * 2,
image.height)
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            # left copy
            pixel_left = out.get_pixel(x, y)
            pixel_left.red = pixel.red
            pixel_left.green = pixel.green
            pixel_left.blue = pixel.blue
            # right copy
            # this is the key spot
            # have: pixel at x,y in image
            # want: pixel_right at ??? to write to
            pixel_right = out.get_pixel(out.width -
1 - x, y)
            pixel_right.red = pixel.red
            pixel_right.green = pixel.green
```

```
            pixel_right.blue = pixel.blue
    return out
```

Debugging: Put in a bug

Remove the "- 1" from formula above, so out_x value is one too big. A very common form of Off By One error. What happens when we run it?

Off By One Error - OBO - Classic!

Off By One error - OBO - a very common error in computer code. Surely you will write some of these in CS106A. It has its own acronym and wikipedia page.

Side N - Parameter

> Side N

side_n: The "n" parameter is an int value, zero or more. The code in the function should use whatever value is in n. (Values of n appear in the Cases menu.) Create an out image with a copy of the original image with n-pixel-wide blank areas added on its left and right sides. Return the out image.

def / Parameter

We'll start down the path with parameters a little here. A "parameter" is listed within the parenthesis.

```
def side_n(filename, n):
```

Each parameter represents a value that comes **in** to the function when it runs. The function just uses each parameter. We'll worry about where the parameter value comes from later. For today: treat the parameter like a variable that has a value in it and the code simply use each parameter, knowing its value is already set.

Side N - Blank Image

For example, we have the "n" parameter to side_n(), specifying how wide the blank space is on each side. What is the line to make the new blank image? How wide should it be. The width of the out image is the width of the original, plus two n-wide areas. So the whole width is `image.width + 2 * n`

```
  out = SimpleImage.blank(image.width + 2 * n,
  image.height)
```

Notice how the `n` is just in the code. This works, because each parameter is set up with the proper value in it before the function runs.

Side N Solution

```
  def side_n(filename, n):
      image = SimpleImage(filename)
      # Create out image, 2 * n pixels wider than
  original
      out = SimpleImage.blank(image.width + 2 * n,
  image.height)

      # Copy the original over - shifting rightward by
  n
      for y in range(image.height):
          for x in range(image.width):
              pixel = image.get_pixel(x, y)
              pixel_out = out.get_pixel(x + n, y)  #
  shift by n
              pixel_out.red = pixel.red
              pixel_out.green = pixel.green
              pixel_out.blue = pixel.blue
      return out
```

(Optional) Extra Practice - Mirror3

Here's another variation on the 2-image side by side form, this one with the left image upside down:

> Mirror3

(optional) Keyboard Shortcuts

See Python Guide: Keyboard Shortcuts

These work on the experimental server and other places, including the PyCharm tool we'll show you on Fri. Ctrl-k works in GMail .. so satisfying!

- Run = cmd-enter
- Indent / Unindent (cmd-tab, shift-cmd-tab)
- Comment / Uncomment (cmd-/, shift-cmd-/)
- ctrl-k = delete line, super handy!