UWCSE BRIDGE Workshop August 4-5, 2008

Hal Perkins
Computer Science & Engineering
University of Washington
perkins@cs.washington.edu

What's Up?

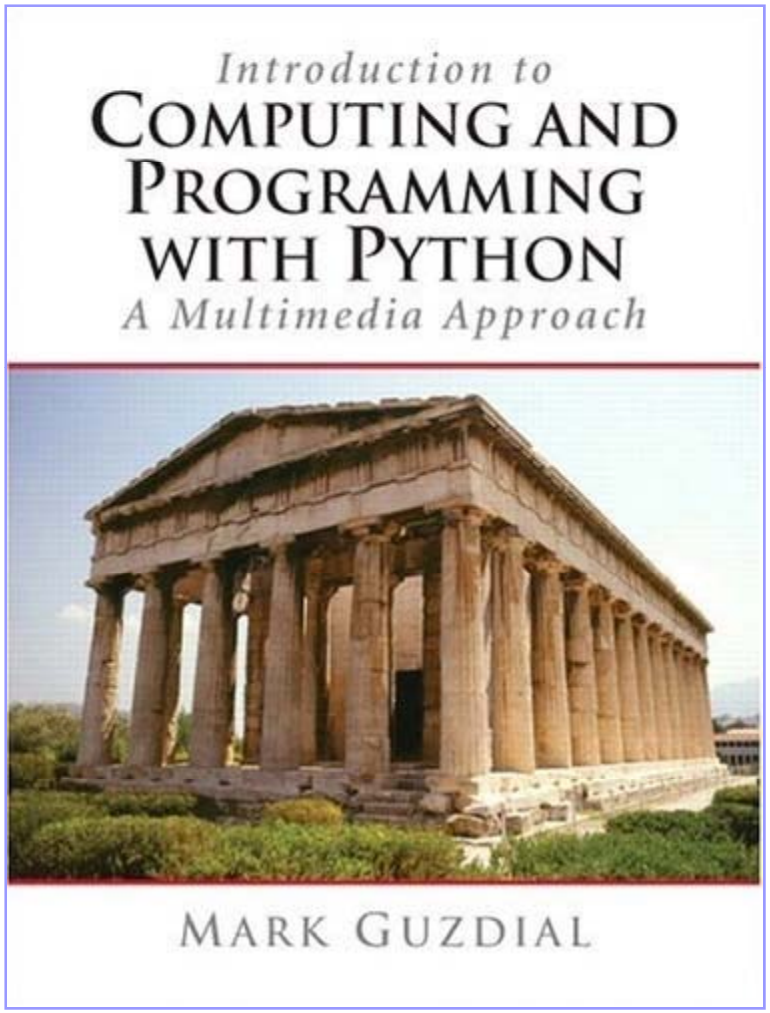
- In two afternoons:
  - Learn how to write programs in Python
  - Learn how digital images are stored in the computer
  - Use Python programs to change images and create new ones!
- Of course we won't learn *everything* there is to know in two days, but we'll make quite a start!!

Credits and Links

- These slides and ideas are largely taken from the media computation project at Georgia Tech
  - For software, links, etc. (for home – we already have what we need on the lab computers for today)
    - JES software: http://coweb.cc.gatech.edu/mediaComp-teach/26
    - More sample media, etc. http://coweb.cc.gatech.edu/cs1315/814
- Thanks to Mark Guzdial and to Barbara Ericson for suggestions and advice

If you like this…

- Get the book!
  - More Python
  - More about images
  - Movies, sound
  - More CSE!



Workshop Plan

- Today
  - Python basics
    - Python as a calculator; variables, expressions and assignment
    - Defining simple functions
  - Digital images
    - Representing pictures: pixels, rgb values
    - Simple image transformations: loops
    - Conditional statements (if time, otherwise tomorrow)
- Tomorrow
  - More complex image manipulation; image coordinates, nested loops
- Both days: Some talking, plenty of hands-on tinkering

Introductions

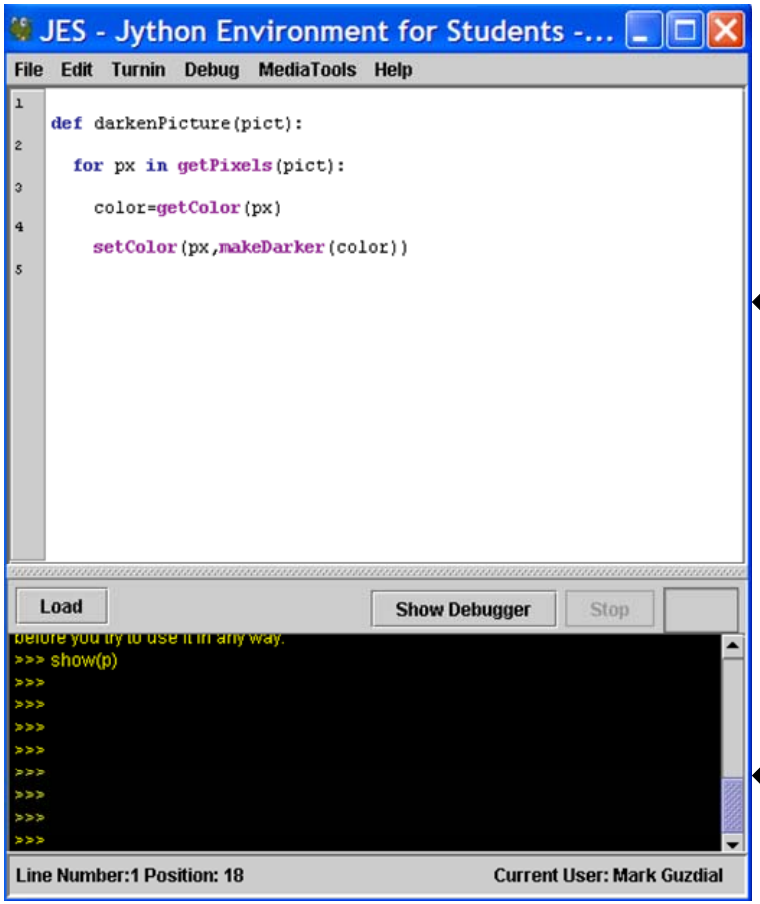
- Who are you?
- Where are you from?
- What’s your plan at UW?
- What do you want to get out of this workshop?

Python

- The programming language we will be using is called *Python*
  - *We* didn't invent Python—it was invented (and named) by researchers across the Internet
  - http://www.python.org
  - It's used by companies like Google, Industrial Light & Magic, Nextel, and others
  - Named after Monty Python – not after some sort of snake
- The *kind* of Python we're using is called Jython
  - It's Java-based Python
    - (We didn't invent that, either.)
  - http://www.jython.org
- We'll be using a specific tool to make Python programming easier, called JES.
  - We didn't invent that either (the folks at GATech did)

We will program in JES

- JES: Jython Environment for Students
- A simple *editor* (for entering in our *programs* or *recipes*): the *program area*
- A *command* area for entering in commands for Python to execute.



Program Area

Command Area

Python understands *commands*

- We can name data with **=**
- We can print values, expressions, anything with **print**

Using JES

```
>>> print 34 + 56
90
>>> print 34.1/46.5
0.7333333333333334
>>> print 22 * 33
726
>>> print 14 - 15
-1
>>> print "Hello"
Hello
>>> print "Hello" + "Y′all"
HelloY′all
```

Command Area Editing

- Up/down arrows walk through *command history*
- You can edit the line at the bottom
  - and then hit Return/Enter
  - that makes that last line execute

Expressions

- A formula to compute a value

  Example: 17 + 21 * 2

- Python has the usual arithmetic operations

+ - * /	plus, minus, times, divide
%	modulus (or remainder)
**	exponentiation

- The usual precedence (ordering) rules apply

  17 + 3 * 42 means 17 + (3 * 42)

  - You can write parentheses to change the grouping or make your meaning clear: (17 + 3) * 42

Division and Integers vs Floats

- If we use integers (whole numbers), / and % give us integer quotient and remainder: 7/3 7%3
  - Try it!
- We also have floating-point numbers with fractions and/or exponents: 1.0, 0.0, 3.14, 10e6
  - The computer approximation to real numbers
  - Arithmetic with floats or a mix of floats and integers gives a floating-point result
    - Compare: 1/3 vs 1.0/3.0, 7%3 vs 7.0%3.0
    - What happens if you mix them? 7.0/3
      - Try it!

Variables – Naming Things

- It often helps to give names to things

  ```
  farenheit = 72.0
  celsius = (farenheit – 32.0) * 5.0 / 9.0
  ```

- Pick whatever names you want! (almost)
  - Anything that starts with a letter followed by zero or more letters, digits, underscores (_), *except…*
  - There are a handful of *reserved words* (keywords) that mean something special to Python (if, for, def, return, etc.). You can't use these for your names.
    - A python-savvy editor will display them in a different color

Assignment

- What does *variable* = *expression* mean?
  1. First calculate the value of *expression*
  2. Then store that value in *variable*
- Things happen in that order.
- So, what does this mean? x = x + 1

  (Hint: never pronounce “=” as “equals”. It means “gets” or “becomes” in an assignment – say it that way!!)
- If the variable had a previous value it is replaced

Functions

- Python includes a lot of functions for math and other things
  - For instance: sqrt, sin, cos, max, min, …
- Use them in formulas

```
largest = max(a,b,c)
distance = sqrt(x**2 + y**2)
```

- Technicality: in standard Python you need to write “ from math import * ” (without the quotes) before you can use these functions. In JES this isn’t needed for the common ones.

Writing Functions

- Suppose we want to convert a bunch of temperatures from Fahrenheit to Celsius
  - Could type the formula over and over
    - A little easier if we use the up-arrow and edit, but still a pain
- Better: *define* our own function
  - (We'll call it f2c for now)
  - Then we can write

```
hot = f2c(110)
cold = f2c(-10)
nice = f2c(75)
```

Writing a recipe: Making our own functions

- To make a function, use the command **def**
- Then, the name of the function, and the names of the input values between parentheses (“(temp)”)
- End the line with a colon (“:”)
- The *body* of the recipe is indented (Hint: Use two or three spaces – a tab)
  - That’s called a *block*

```
def f2c(temp):
   return (temp-32.0) * 5.0/9.0
```

Making functions the easy way

- Get something working by typing commands in the command window (bottom half of JES)
- Enter the **def** command in the editing window (top part of JES)
- Copy-paste the right commands up into the recipe

Blocking is indicated for you in JES

- Statements that are indented the same, are in the same block.
- Statements in the same block as the cursor are enclosed in a blue box.

Different Ways to do Things

- There are many ways to name things and do things
- Try to write your code so it's easy for others (including yourself!) to understand

- Examples:

```
def vol1(l,w,h):
    return l*w*h

def vol2(length, width, height):
    return length*width*height

def vol3(length, width, height):
    area = length * width
    vol = area * height
    return vol
```

Saving Functions

- Once you've typed in your functions you need to save them in a file, the tell JES to "load" them
  - Use the regular File > Save command
  - A file containing Python code should normally have a name ending in ".py"
  - After saving the file, click the "Load Program" button
    - JES will tell you if it detects any punctuation (syntax) errors
    - If it does, fix, save, and reload
- You can reuse the functions next time by opening and reloading the file

Your Turn

- Log in, copy JES to your desktop, and start it
  - See the “Getting started” sheet, watch the demo, and ask questions
- Then do the first set of exercises
  - Use JES as a calculator, then
  - Define and use some functions

Image Processing

- Goals:
  - Give you the basic understanding of image processing, including how pictures are represented in a computer
  - Experiment with some interesting image transformation
- We won't put Photoshop, GIMP, ImageMagik out of business…
  - But you will have a much better idea of what they're doing!

Showing a Picture in JES

```
file = pickAFile()
picture = makePicture(file)
show(picture)
```

What does this do?

1. Variable file accesses the picture jpeg file on the disk
2. Variable picture is the picture bits copied to memory
3. Show draws the picture bits on the screen

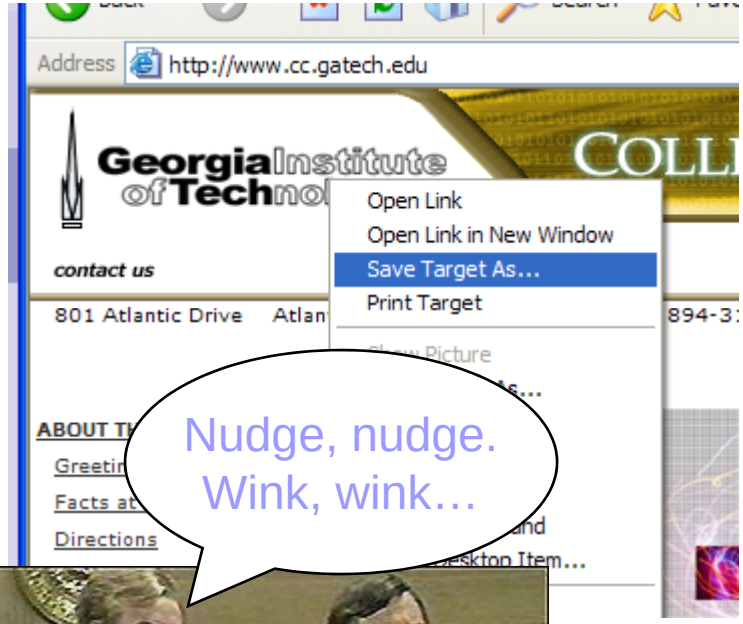
Another Function

- Since we do this a lot, let's make a function so we don't have to type it over and over again
  - We'll return a reference to the picture in memory so we can work with it

```
def pickAndShow():
  filename = pickAFile()
  picture = makePicture(filename)
  show(picture)
  return picture
```

Grabbing media from the Web

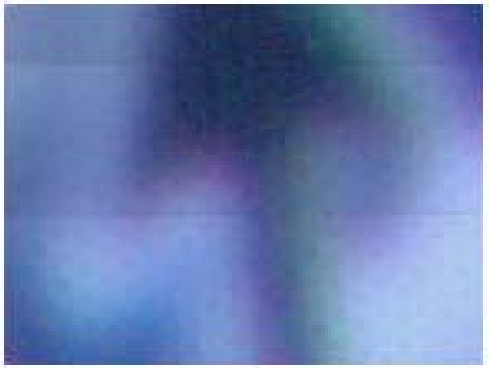
- Right-click (Windows) or Control-Click (Mac)
- Save Target As…
- Can *only* do JPEG images (.jpe, .jpg, .jpeg)



Most images on the Internet are copyright. You can download and use them *only* for your own use unless you have permission.

Digitizing pictures as bunches of little dots

- We digitize pictures into lots of little dots
- Enough dots and it looks like a continuous whole to our eye
  - Our eye has limited resolution
  - Our background/depth *acuity* is particulary low
- Each picture element is referred to as a *pixel*
- Pixels are *picture elements*
  - Each pixel object knows its *color*
  - It also knows where it is in its *picture*

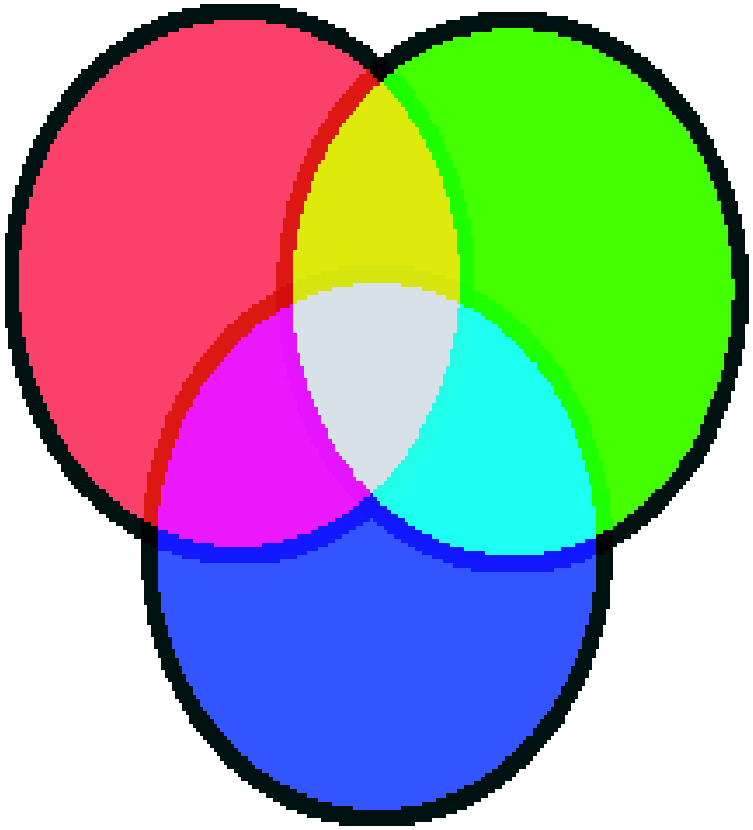


Encoding color

- Each pixel encodes color at that position in the picture
- Lots of encodings for color
  - Printers use CMYK: Cyan, Magenta, Yellow, and blacK.
  - Others use HSB for Hue, Saturation, and Brightness (also called HSV for Hue, Saturation, and Brightness
- We'll use the most common for computers
  - RGB: Red, Green, Blue

Encoding Color: RGB

- In RGB, each color has three component colors:
  - Amount of redness
  - Amount of greenness
  - Amount of blueness
- Each does appear as a separate dot on most devices, but our eye blends them.
- In most computer-based models of RGB, a single *byte* (8 bits) is used for each
  - So a complete RGB color is 24 bits, 8 bits of each



Encoding RGB

- Each component color (red, green, and blue) is encoded as a single byte
- Colors go from (0,0,0) to (255,255,255)
  - If all three components are the same, the color is in greyscale
    - (50,50,50) at (2,2)
  - (0,0,0) (at position (1,2) in example) is black
  - (255,255,255) is white

Use a loop!
Our first picture recipe

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*0.5)
```

Used like this:

```
>>> file=pickAFile()
>>> picture=makePicture(file)
>>> show(picture)
>>> decreaseRed(picture)
>>> repaint(picture)
```

Examples:

```
def clearRed(picture):
  for pixel in getPixels(picture):
    setRed(pixel,0)
```

```
def greyscale(picture):
  for p in getPixels(picture):
    redness=getRed(p)
    greenness=getGreen(p)
    blueness=getBlue(p)
    luminance=(redness+blueness+greenness)/3
    setColor(p,
        makeColor(luminance,luminance,luminance))
```

```
def negative(picture):
  for px in getPixels(picture):
    red=getRed(px)
    green=getGreen(px)
    blue=getBlue(px)
    negColor=makeColor(255-red,255-green,255-blue)
    setColor(px,negColor)
```

How do you make an omelet?

- Something to do with eggs…
- What do you do with each of the eggs?
- And then what do you do?

All useful recipes involve repetition

- Take <u>four</u> eggs and crack <u>them</u>....

- Beat the eggs <u>until</u>...

We need these repetition ("iteration") constructs in computer algorithms too

- Today we will introduce one of them

Decreasing the red in a picture

- Recipe: To decrease the red
- Ingredients: One picture, name it **pict**
- Step 1: Get all the pixels of **pict**. For each pixel **p** in the set of pixels…
- Step 2: Get the value of the red of pixel **p**, and set it to 50% of its original value

Use a for loop!
Our first picture recipe

```
def decreaseRed(pict):
  allPixels = getPixels(pict)
  for p in allPixels:
    value = getRed(p)
    setRed(p, value * 0.5)
```

The loop

- Note the indentation!

How for loops are written

```
def decreaseRed(pict):
  allPixels = getPixels(pict)
  for p in allPixels:
    value = getRed(p)
    setRed(p, value * 0.5)
```

- **for** is the name of the command
- An *index variable* is used to hold each of the different values of a sequence
- The word **in**
- A function that generates a *sequence*
  - **The index variable will be the name for one value in the sequence, each time through the loop**
- A colon (“:”)
- And a *block* (the indented lines of code)

What happens when a for loop is executed

- The *index variable* is set to an item in the *sequence*
- The block is executed
  - The variable is often used inside the block
- Then execution *loops* to the **for** statement, where the index variable gets set to the next item in the sequence
- Repeat until every value in the sequence was used.

getPixels returns a sequence of pixels

- Each pixel knows its color and place in the original picture
- Change the pixel, you change the picture
- So the loops here assign the index variable *p* to each pixel in the picture *picture*, one at a time.

```
def decreaseRed(picture):
  allPixels = getPixels(picture)
  for p in allPixels
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

or equivalently...

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

Do we need the variable *originalRed*?

- No: Having removed *allPixels*, we can also do without *originalRed* in the same way:
  - We can calculate the original red amount right when we are ready to change it.
  - It's a matter of programming style. The meanings are the same.

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

```
def decreaseRed(picture):
  for p in getPixels(picture):
    setRed(p, getRed(p) * 0.5)
```

Let's walk that through slowly…

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

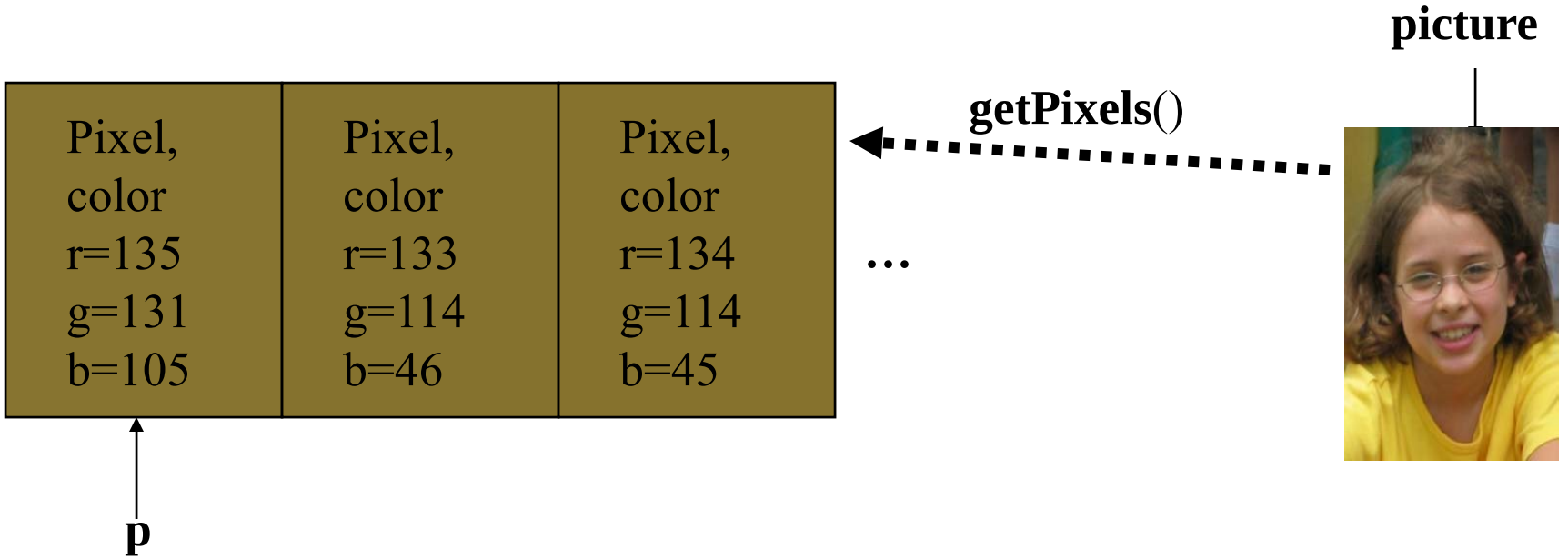
Here we take a picture object in as a parameter to the function and call it **picture**

picture

Now, get the pixels

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

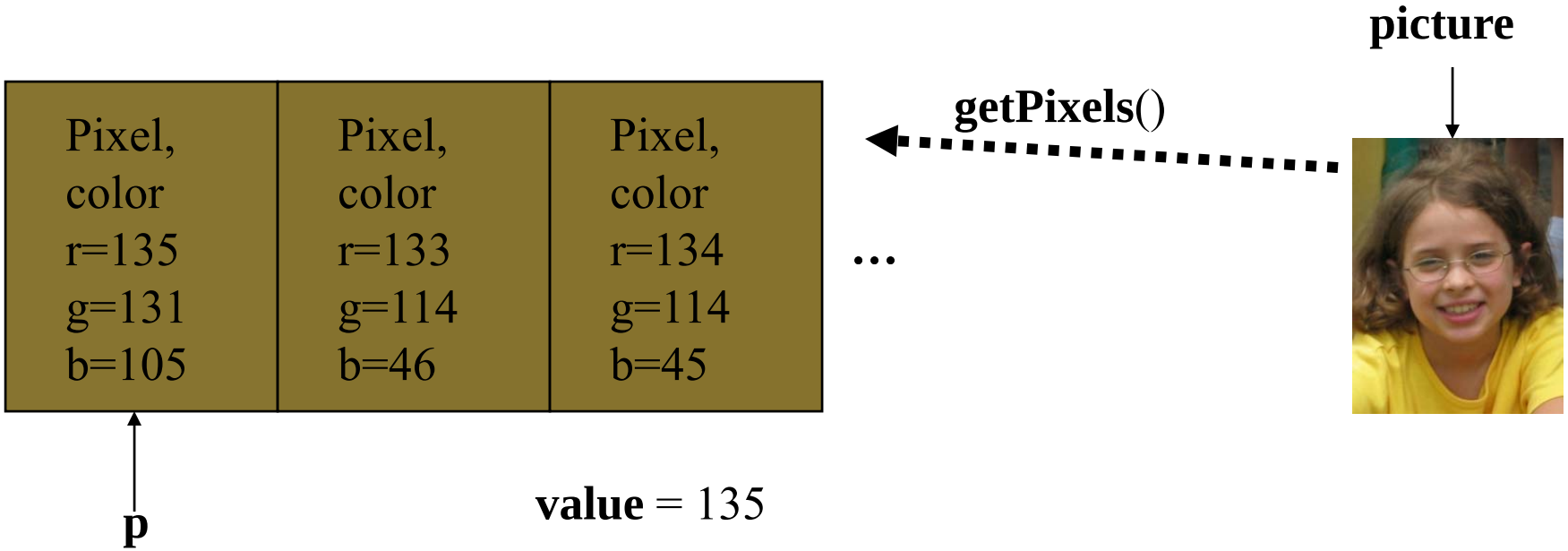
We get all the pixels from the **picture**, then make **p** be the name of each one *one at a time*



Get the red value from pixel

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

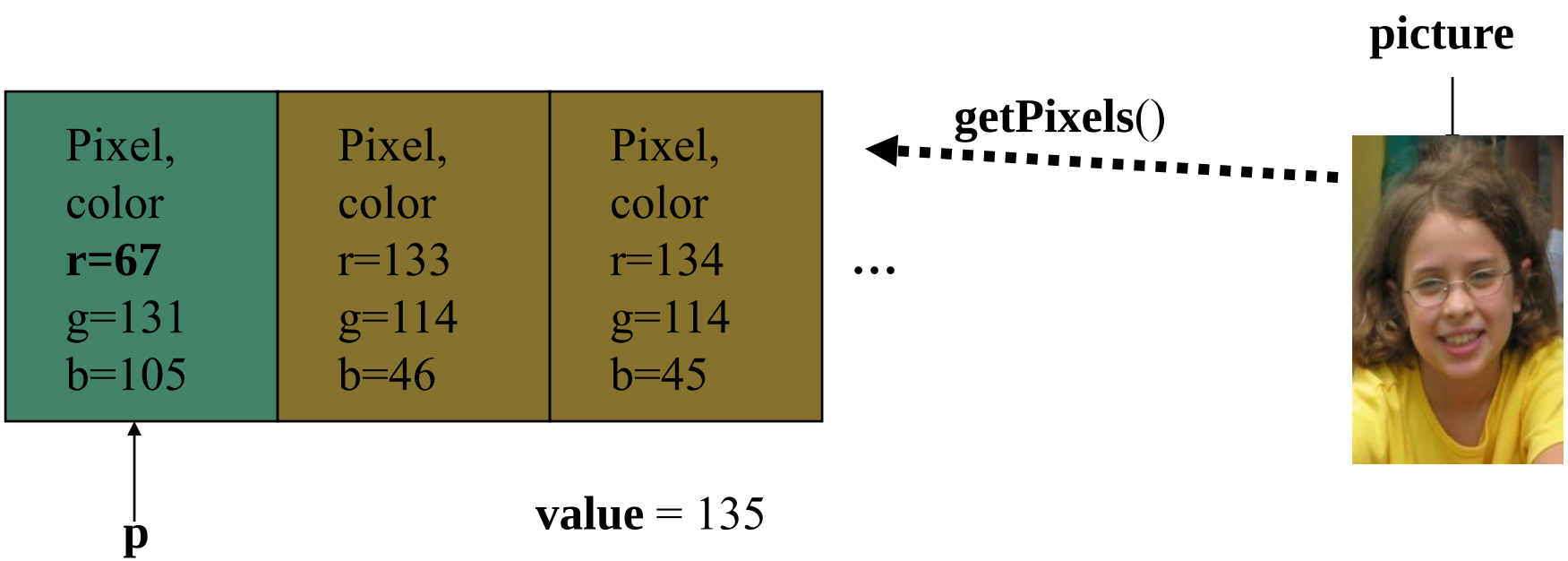
We get the red value of pixel **p** and name it **originalRed**



Now change the pixel

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

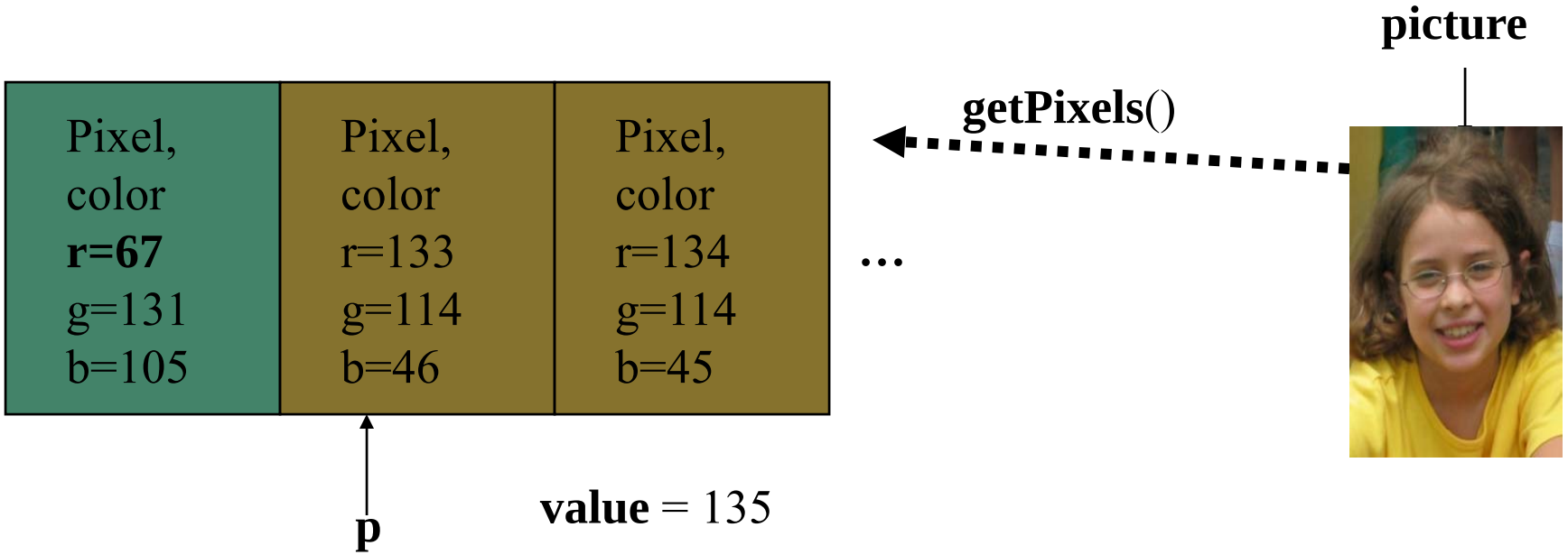
Set the red value of pixel **p** to 0.5 (50%) of **originalRed**



Then move on to the next pixel

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

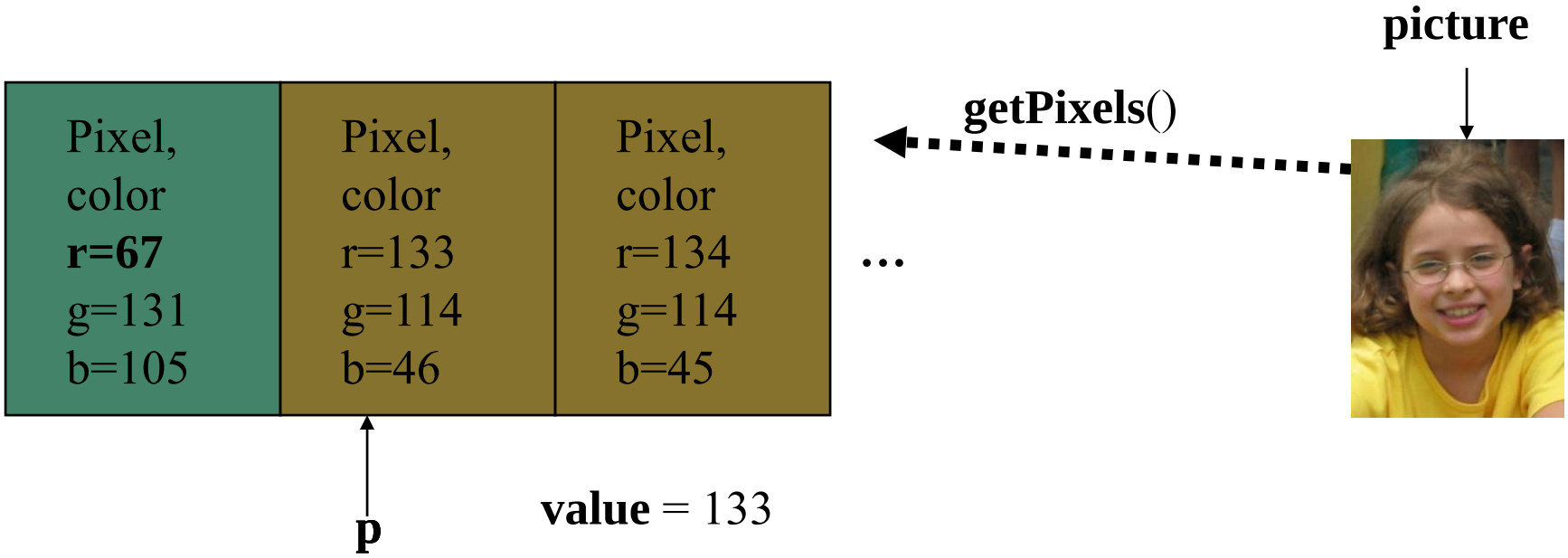
Move on to the next pixel and name *it* **p**



Get its red value
Get its red value

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

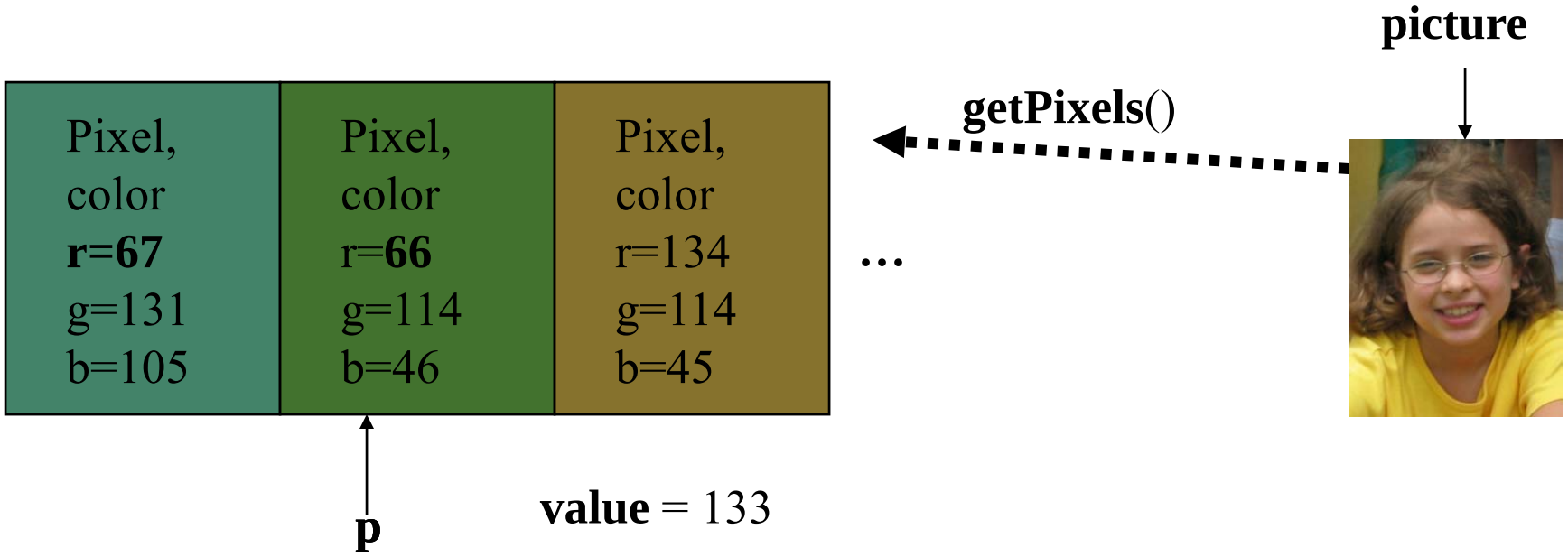
Set **originalRed** to the red value at the new **p**, then change the red at that new pixel.



And change *this* red value

```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

Change the red value at pixel **p** to 50% of value



And eventually, we do all pixels

- We go from this… to this!

“Tracing/Stepping/Walking through” the program

- What we just did is called “stepping” or “walking through” the program
  - You consider each step of the program, in the order that the computer would execute it
  - You consider what *exactly* would happen
  - You write down what values each variable (name) has at each point.
- It’s one of the most important *debugging* skills you can have.
  - And *everyone* has to do a *lot* of debugging, especially at first.

Clearing Blue

```
def clearBlue(picture):
  for p in getPixels(picture):
    setBlue(p, 0)
```

Again, this will work for any picture.

Try stepping through this one yourself!

Can we combine these? Why not!

- How do we turn this beach scene into a sunset?
- What happens at sunset?
  - At first, I tried increasing the red, but that made things like red specks in the sand REALLY prominent.
    - Wrap-around
  - New Theory: As the sun sets, less blue and green is visible, which makes things look more red.

A Sunset-generation Function

```
def makeSunset(picture):
  for p in getPixels(picture):
    value = getBlue(p)
    setBlue(p, value * 0.7)
    value = getGreen(p)
    setGreen(p, value * 0.7)
```

Creating a negative

- Let's think it through
  - R, G, B go from 0 to 255
  - Let's say Red is 10. That's very light red.
    - What's the opposite? LOTS of Red!
  - The negative of that would be 245: 255-10
- So, for each pixel, if we negate each color component in creating a new color, we negate the whole picture.

Creating a negative

```
def negative(picture):
  for px in getPixels(picture):
    red = getRed(px)
    green = getGreen(px)
    blue = getBlue(px)
    negColor = makeColor( 255-red, 255-green, 255-blue)
    setColor(px, negColor)
```

Original, negative, double negative

(This gives us a quick way to test our function: Call it twice and see if the result is equivalent to the original)

We call this a lossless transformation.

Converting to grayscale

- We know that if red=green=blue, we get gray
  - But what value do we set all three to?
- What we need is a value representing the darkness of the color, the *luminance*
- There are many ways, but one way that works reasonably well is dirt simple—simply take the average:

$$\frac{(red+green+blue)}{3}$$

Why can't we get back again?

- Converting to grayscale is different from computing a negative.
  - A negative transformation *retains* information.
- With grayscale, we've lost information
  - We no longer know what the ratios are between the reds, the greens, and the blues
  - We no longer know any particular value.

Media compressions are one kind of transformation.
Some are **lossless** (like negative);
Others are **lossy** (like grayscale)

But that's not really the *best* grayscale

- In reality, we don't perceive red, green, and blue as *equal* in their amount of luminance: How bright (or non-bright) something is.
  - We tend to see blue as "darker" and red as "brighter"
  - Even if, physically, the same amount of light is coming off of each
- Photoshop's grayscale is very nice: Very similar to the way that our eye sees it
  - B&W TV's are also pretty good
- A reasonable grayscale is to replace r, g, and b with luminance = r*0.299 + g*0.587 + b*0.114
  - Based on research into human vision

Saving Pictures

- Changing a picture only changes the bits in memory – it does not change the original file
- If you want to save a picture, you need to write the picture bits to a disk file

```
setMediaPath()          # pick directory for file - only need to
                        # do once, or whenever you change it

writePictureTo(picture, "filename.jpg")
                        # write file – pick the name you want
```

Lots and lots of filters

- There are many wonderful examples that we can do at this point.
- Your turn!
  - Try out some of the transformations we've seen
  - Create some new ones (see the exercise sheet)

Increasing Red

```
def increaseRed(picture):
  for p in getPixels(picture):
    value = getRed(p)
    setRed(p, value * 1.2)
```

What happened here?!?

Remember that the limit for redness is 255.

If you go beyond 255, all kinds of weird things might happen

Let’s try making Barbara a redhead!

- We could just try increasing the redness, but as we’ve seen, that has problems.
  - Overriding some red spots
  - And that’s more than just her hair
- If only we could increase the redness *only* of the brown areas of Barb’s head…

Treating pixels differently

- We can use the **if** statement to treat some pixels differently.
- For example, color replacement: Turning Barbara into a redhead
  - We used the MediaTools to find the RGB values for the brown of Barbara's hair
  - We then look for pixels that are close to that color (within a *threshold*), and increase by 50% the redness in those

Making Barb a redhead

Original:

```
def turnRed():
  brown = makeColor(57,16,8)
  file = r"C:\My Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for px in getPixels(picture):
    color = getColor(px)
    if distance(color, brown) < 50.0:
      redness=getRed(px)*1.5
      setRed(px,redness)
  show(picture)
  return(picture)
```

Digital makeover:

Talking through the program slowly

- Why aren't we taking any input? Don't want any: Recipe is specific to this one picture.
- The brown is the brownness that I figured out from MediaTools
- I need the picture to work with

```
def turnRed():
  brown = makeColor(57,16,8)
  file = r"C:\My Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
for px in getPixels(picture):
    color = getColor(px)
    if distance(color, brown) < 50.0:
      redness=getRed(px)*1.5
      setRed(px,redness)
show(picture)
```

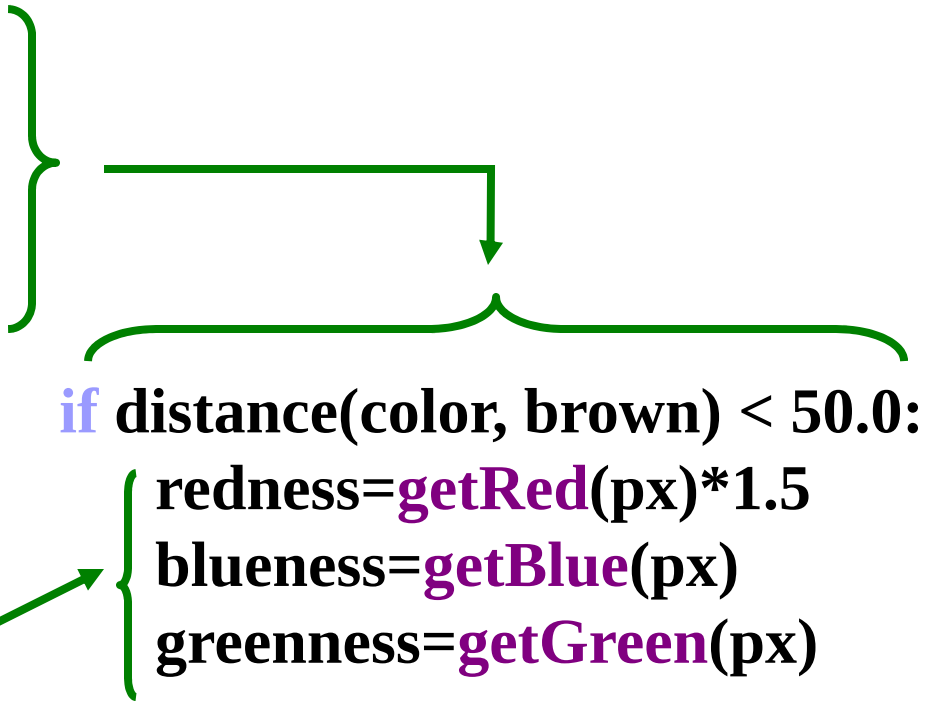
Walking through the for loop

- Now, for each pixel **px** in the picture, we
  - Get the color
  - See if it's within a distance of 50 from the brown we want to make more red
  - If so, increase the redness by 50%

```
file = r"C:\My Documents\mediasources\barbara.jpg"
picture=makePicture(file)
for px in getPixels(picture):
  color = getColor(px)
  if distance(color, brown) < 50.0:
     redness=getRed(px)*1.5
    setRed(px,redness)
show(picture)
return(picture)
```

How an if works

- **if** is the command name
- Next comes an expression: Some kind of true or false comparison
- Then a colon
- Then the body of the if — the things that will happen if the expression is true



Expressions

Bug alert!

= means “make them equal!”
== means “are they equal?”

- Can test equality with ==
- Can also test <, >, >=, <=, <> (not equals)
- In general, 0 is false, 1 is true
  - So you can have a function return a “true” or “false” value.

Returning from a function

- At the end, we **show** and **return** the picture
- Why are we using **return**?
  - Because the picture is created within the function
  - If we didn't return it, we couldn't get at it in the command area
- We could **print** the result, but we'd more likely assign it a name

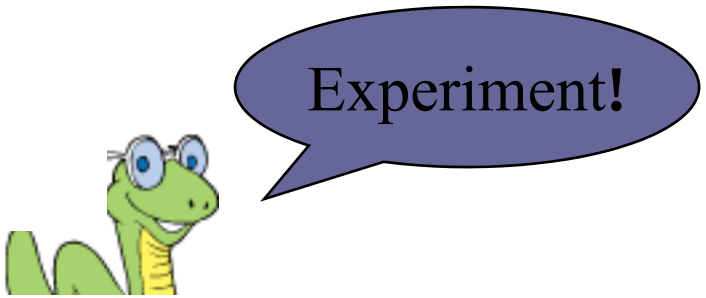
```
    redness=getRed(px) *1.5
    setRed(px,redness)
show(picture)
return(picture)
```

Things to change

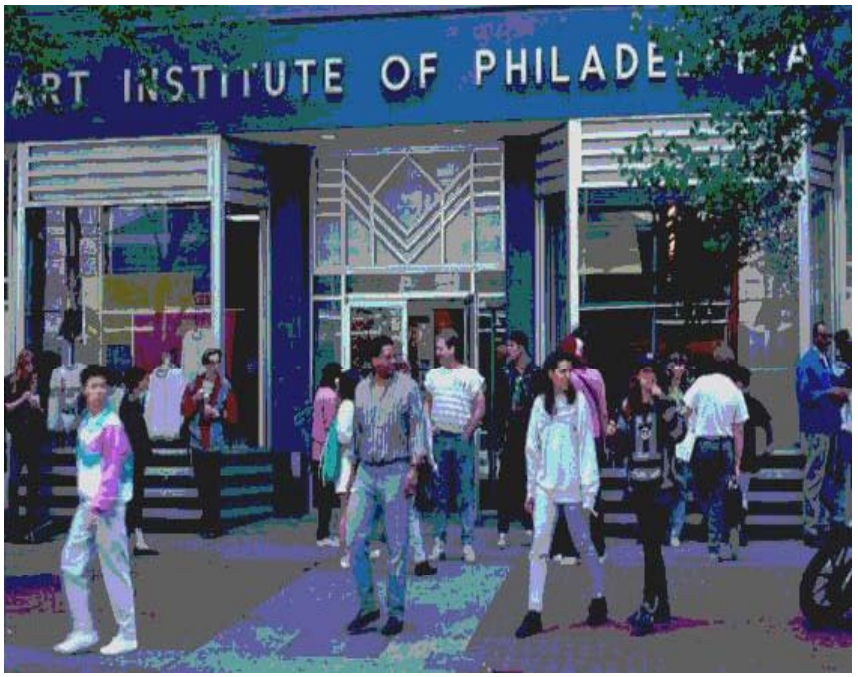
- Lower the threshold to get more pixels
  - But if it's too low, you start messing with the wood behind her
- Increase the amount of redness
  - But if you go too high, you can go beyond the range of valid color intensities (i.e. more than 255)

Replacing colors using if

- We don’t have to do one-to-one changes or replacements of color
- We can use **if** to decide if we want to make a change.
  - We could look for a range of colors, or one specific color.
  - We could use an operation (like multiplication) to set the new color, or we can set it to a specific value.
- It all depends on the effect that we want.



Posterizing:
Reducing the range of colors



Posterizing: How we do it

- We look for a *range* of colors, then map them to a *single* color.
  - If red is between 63 and 128, set it to 95
  - If green is less than 64, set it to 31
  - ...
- This requires many **if** statements, but the *idea* is pretty simple.
- The end result is that *many* colors, get reduced to a *few* colors

Posterizing function

```
def posterize(picture):
  #loop through the pixels
  for p in getPixels(picture):
    #get the RGB values
    red = getRed(p)
    green = getGreen(p)
    blue = getBlue(p)

    #check and set red values
    if(red < 64):
      setRed(p, 31)
    if(red > 63 and red < 128):
      setRed(p, 95)
    if(red > 127 and red < 192):
      setRed(p, 159)
    if(red > 191 and red < 256):
      setRed(p, 223)

    #check and set green values
    if(green < 64):
      setGreen(p, 31)
    if(green > 63 and green < 128):
      setGreen(p, 95)
    if(green > 127 and green < 192):
      setGreen(p, 159)
    if(green > 191 and green < 256):
      setGreen(p, 223)

    #check and set blue values
    if(blue < 64):
      setBlue(p, 31)
    if(blue > 63 and blue < 128):
      setBlue(p, 95)
    if(blue > 127 and blue < 192):
      setBlue(p, 159)
    if(blue > 191 and blue < 256):
      setBlue(p, 223)
```

What's with this "#" stuff?

- Any line that starts with # is *ignored* by Python.
- This allows you to insert *comments*: Notes to yourself (or another programmer) that explain what's going on here.
  - When programs get longer, and have lots of separate pieces, it's gets hard to figure out from the code alone what each piece does.
  - Comments can help explain the big picture.

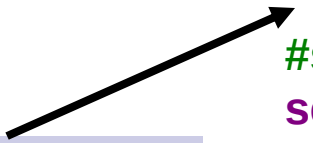
Generating sepia-toned prints

- Pictures that are *sepia-toned* have a yellowish tint to them that we associate with older photographs.
- It's not just a matter of increasing the amount of yellow in the picture, because it's not a one-to-one correspondence.
  - Instead, colors in different ranges get converted to other colors.
  - We can create such convertions using if

Example of sepia-toned prints

Here's how we do it

```
def sepiaTint(picture):
  #Convert image to greyscale
  greyScale(picture)

  #loop through picture to tint pixels
  for p in getPixels(picture):
     red = getRed(p)
     blue = getBlue(p)

    #tint shadows
    if (red < 63):
       red = red*1.1
       blue = blue*0.9

  #tint midtones
   if (red > 62 and red < 192):
      red = red*1.15
      blue = blue*0.85

   #tint highlights
   if (red > 191):
      red = red*1.08
      if (red > 255):
         red = 255

      blue = blue*0.93

   #set the new color values
   setBlue(p, blue)
   setRed(p, red)
```



Bug alert!
Make sure you indent the right amount

Reviewing: All the Programming We've Seen

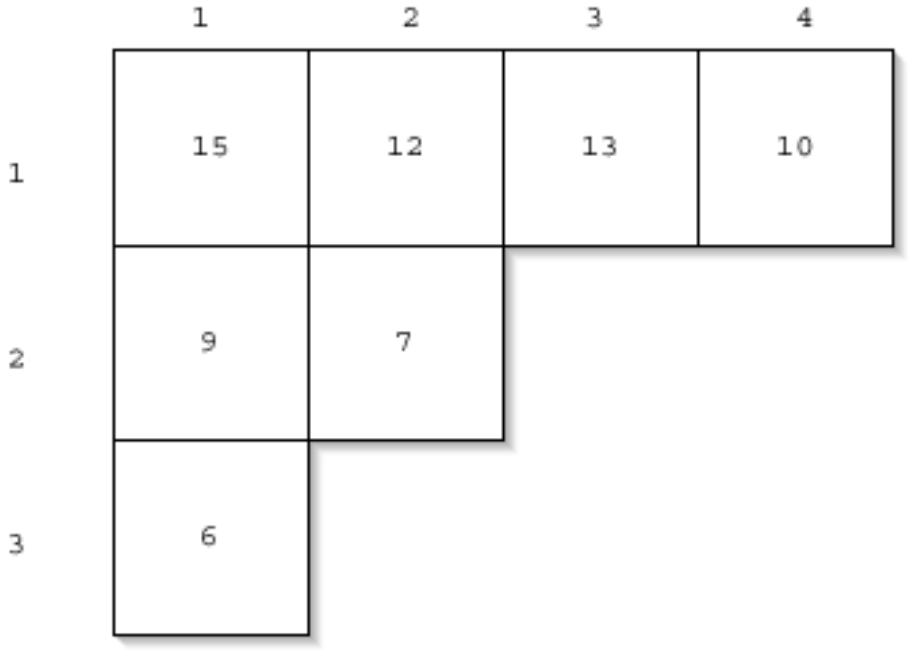
- Assigning names to values with **=**
- Printing with **print**
- Looping with **for**
- Testing with **if**
- Defining functions with **def**
  - **Making a real function with inputs uses ()**
  - **Making a real function with outputs uses return**
- Using functions to create programs (recipes) and executing them

What we can’t do (yet!)

- What if we want to copy of modify part of an image? Or combine images? Or flip an image upside down or sideways?
- So far all we can do is go through the pixels and change them regardless of their position
- To do more we need to know where the pixels are in the image

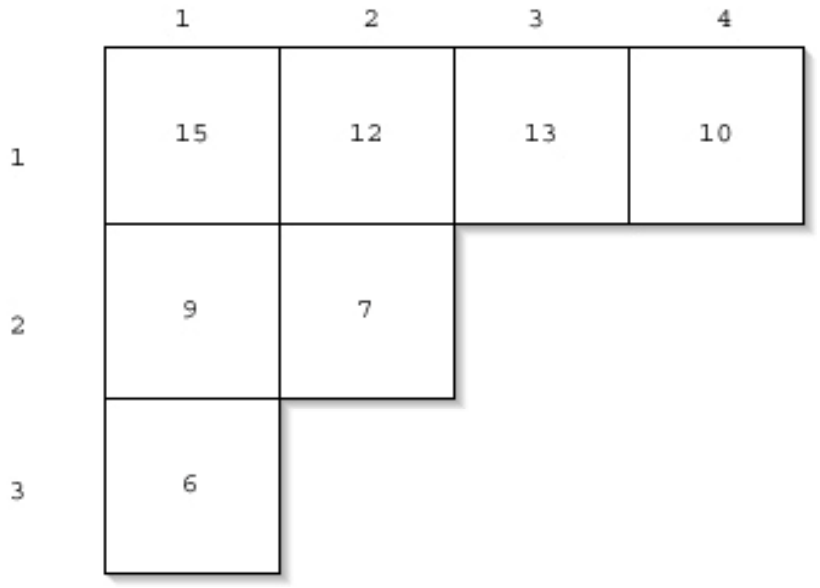
A Picture is a *matrix* of pixels

- It's not a continuous line of elements, that is, an *array*
- A picture has two dimensions: Width and Height
- We need a two-dimensional array: a *matrix*



Just the upper left hand corner of a matrix.

Referencing a matrix



- We talk about positions in a matrix as (x,y), or (horizontal, vertical)
- Element (2,1) in the matrix at left is the value 12
- Element (1,3) is 6

Pixel Functions

- Given a picture p,
- Retrieve the width and height

  w = getWidth(p)

  h = getHeight(p)
- Access a pixel at a location

  pixel = getPixel(p,xpos,ypos)

- Given a pixel,
- Get it's coordinates

  x = getX(pixel)

  y = getY(pixel)

- All the other functions to get/set colors, etc. work as usual

Working the pixels by number

- decreaseRed, but with explicit coordinates…
- We'll have to use *nested loops*
  - One to walk the width, the other to walk the height
    - Be sure to watch your blocks carefully!

```
def decreaseRed2(picture):
  for x in range(1,getWidth(picture)):
    for y in range(1,getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px,value/2)
```

The function range

- Range returns a sequence between its first two inputs, possibly using a third input as the increment

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

That thing in [] is a sequence

```
>>> a=[1,2,3]
>>> print a
[1, 2, 3]
>>> a = a + 4
An attempt was made to call a function with a parameter of an invalid type
>>> a = a + [4]
>>> print a
[1, 2, 3, 4]
>>> a[0]
1
```

We can assign names to sequences, print them, add sequences, and access individual pieces of them.

We can also use **for** loops to process each element of a sequence.

Replacing colors in a range

Get the range using MediaTools

```
def turnRedInRange():
  brown = makeColor(57,16,8)
  file=r"C:\Documents and Settings\Mark Guzdial\My
Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for x in range(70,168):
    for y in range(56,190):
      px=getPixel(picture,x,y)
      color = getColor(px)
      if distance(color,brown)<50.0:
        redness=getRed(px)*1.5
        setRed(px,redness)
  show(picture)
  return(picture)
```

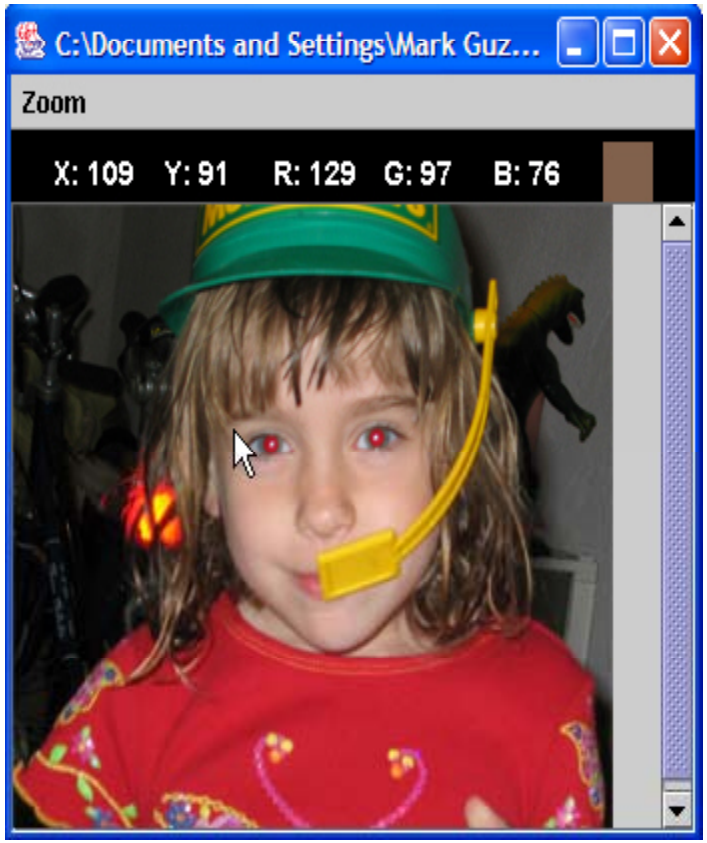
Could we do this without nested loops?

- Yes, but complicated IF
- AND we process many unneeded pixels

```
def turnRedInRange2():
  brown = makeColor(57,16,8)
  file=r"C:\Documents and Settings\Mark Guzdial\My
Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for p in getPixels(picture):
    x = getX(p)
    y = getY(p)
    if x >= 70 and x < 168:
      if y >=56 and y < 190:
        color = getColor(p)
        if distance(color,brown)<100.0:
          redness=getRed(p)*2.0
          setRed(p,redness)
  show(picture)
  return picture
```

Removing “Red Eye”

- When the flash of the camera catches the eye just right (especially with light colored eyes), we get bounce back from the back of the retina.
- This results in “red eye”
- We can replace the “red” with a color of our choosing.
- First, we figure out *where* the eyes are (x,y) using MediaTools



Removing Red Eye

```
def removeRedEye(pic,startX,startY,endX,endY,replacementcolor):
  red = makeColor(255,0,0)
  for x in range(startX,endX):
    for y in range(startY,endY):
      currentPixel = getPixel(pic,x,y)
      if (distance(red,getColor(currentPixel)) < 165):
        setColor(currentPixel,replacementcolor)
```

Why use a range? Because we don't want to replace her red dress!

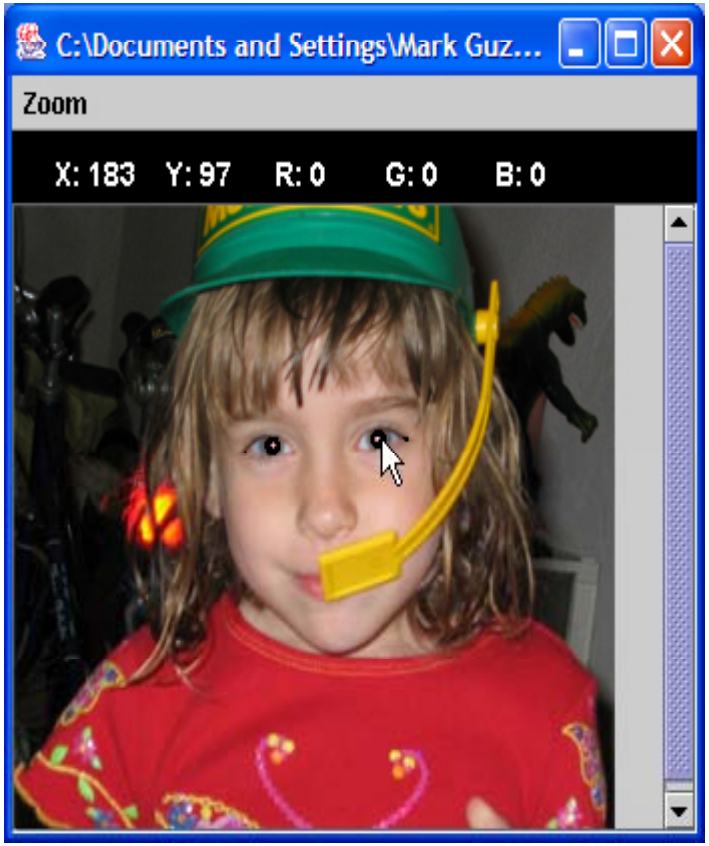
What we're doing here:

- **Within the rectangle of pixels (startX,startY) to (endX, endY)**
- **Find pixels close to red, then replace them with a new color**

“Fixing” it: Changing red to black

removeRedEye(jenny, 109, 91, 202, 107, makeColor(0,0,0))

- Jenny’s eyes are actually not black—could fix that
- Eye are also not mono-color
  - A better function would handle *gradations* of red and replace with *gradations* of the right eye color

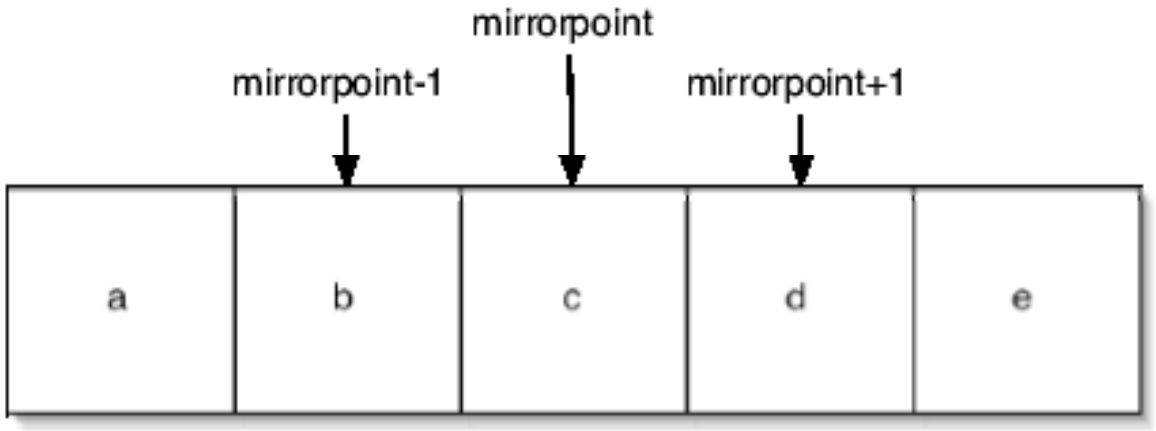


If you know where the pixels are: Mirroring

- Imagine a mirror horizontally across the picture, or vertically
- What would we see?
- How do generate that digitally?
  - We simply *copy* the colors of pixels from one place to another

Mirroring a picture

- Slicing a picture down the middle and sticking a mirror on the slice
- Do it by using a loop to measure a *difference*
  - The index variable is actually measuring distance from the mirrorpoint
- Then reference to either side of the mirror point using the difference



Recipe for mirroring

```
def mirrorVertical(source):
  mirrorpoint = int(getWidth(source)/2)
  for y in range(1,getHeight(source)):
    for xOffset in range(1,mirrorpoint):
      pright = getPixel(source, xOffset+mirrorpoint,y)
      pleft = getPixel(source, mirrorpoint-xOffset,y)
      c = getColor(pleft)
      setColor(pright,c)
```

Doing something useful with mirroring

- Mirroring can be used to create interesting effects, but it can also be used to create realistic effects.
- Consider this image from a trip to Athens, Greece.
  - Can we “repair” the temple by mirroring the complete part onto the broken part?

Figuring out where to mirror

- Use MediaTools to find the mirror point and the range that we want to copy

Program to mirror the temple

```
def mirrorTemple():
  source = makePicture(getMediaPath("temple.jpg"))
  mirrorpoint = 277
  lengthToCopy = mirrorpoint - 14
  for x in range(1,lengthToCopy):
    for y in range(28,98):
      p = getPixel(source,mirrorpoint-x,y)
      p2 = getPixel(source,mirrorpoint+x,y)
      setColor(p2,getColor(p))
  show(source)
  return source
```

Did it really work?

- It clearly did the mirroring, but that doesn't create a 100% realistic image.
- Check out the shadows: Which direction is the sun coming from?

Time for an exercise

- Write a function to take an image and flip it horizontally (left to right)

More Picture Methods

- Compositing and scaling
  - Necessary for making a collage

Copying pixels

- In general, what we want to do is to keep track of a sourceX and sourceY, and a targetX and targetY.
  - We *increment* (add to them) in pairs
    - sourceX and targetX get incremented together
    - sourceY and targetY get incremented together
  - The tricky parts are:
    - Setting values *inside* the body of loops
    - Incrementing at the *bottom* of loops

Copying Barb to a canvas

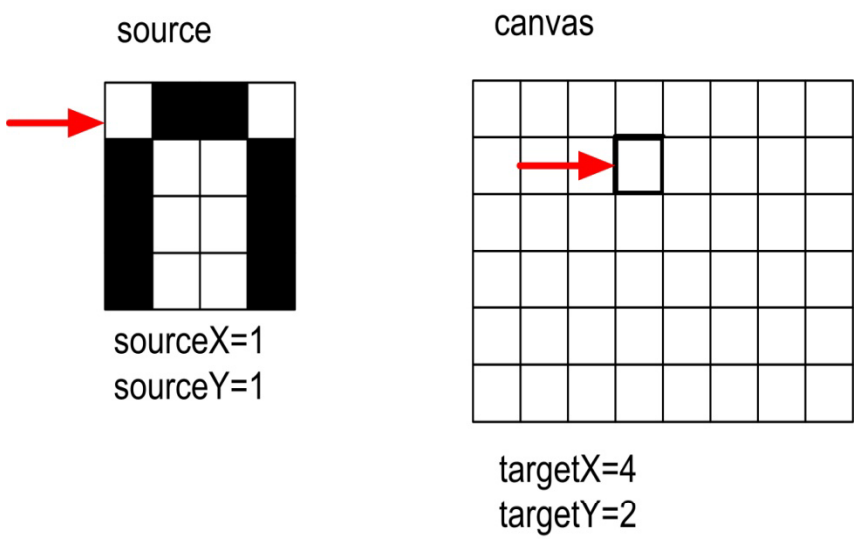
```
def copyBarb():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  targetX = 1
  for sourceX in range(1,getWidth(barb)):
    targetY = 1
    for sourceY in range(1,getHeight(barb)):
      color = getColor(getPixel(barb,sourceX,sourceY))
      setColor(getPixel(canvas,targetX,targetY), color)
      targetY = targetY + 1
    targetX = targetX + 1
  show(barb)
  show(canvas)
  return canvas
```

Copying into the middle of the canvas

```
def copyBarbMidway():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  targetX = 100
  for sourceX in range(1,getWidth(barb)):
    targetY = 100
    for sourceY in range(1,getHeight(barb)):
      color = getColor(getPixel(barb,sourceX,sourceY))
      setColor(getPixel(canvas,targetX,targetY), color)
      targetY = targetY + 1
    targetX = targetX + 1
  show(barb)
  show(canvas)
  return canvas
```

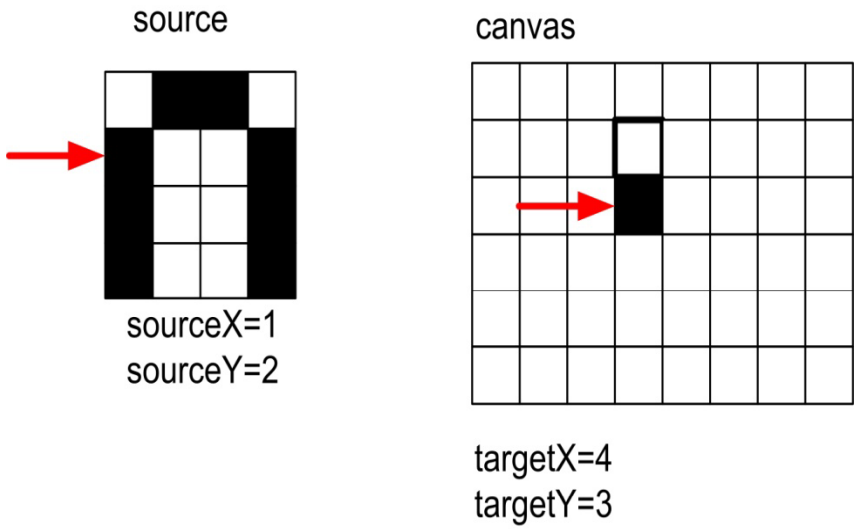
Copying: How it works

- Here's the initial setup:



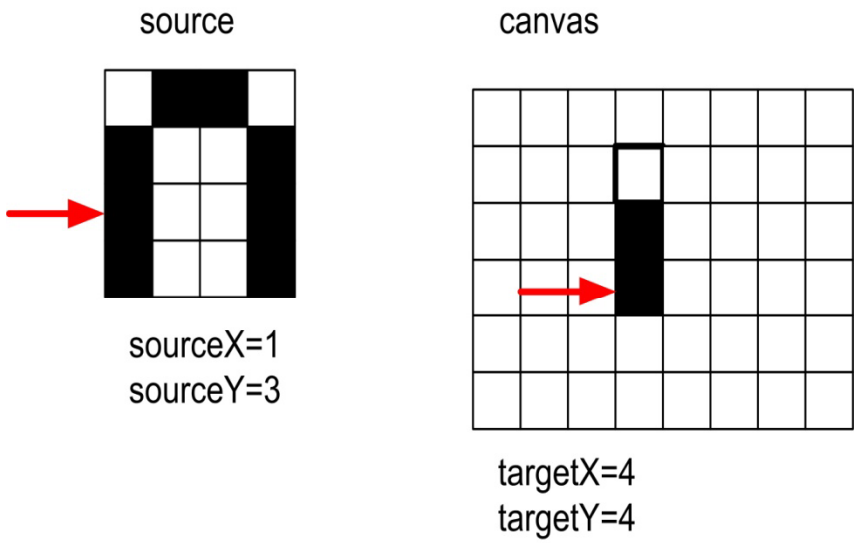
Copying: How it works 2

- After incrementing the sourceY and targetY once (whether in the **for** or via expression):



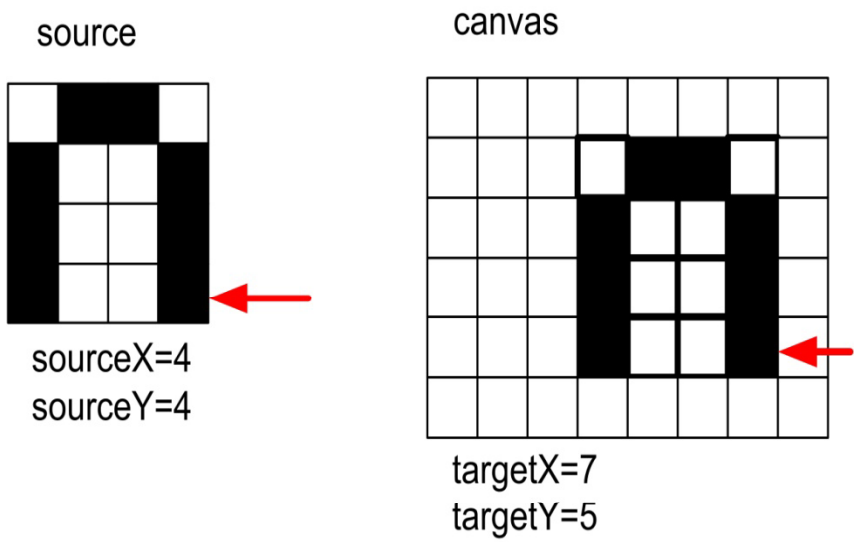
Copying: How it works 3

- After yet another increment of sourceY and targetY:
- When we finish that column, we increment sourceX and targetX, and start on the next column.



Copying: How it looks at the end

- Eventually, we copy every pixel

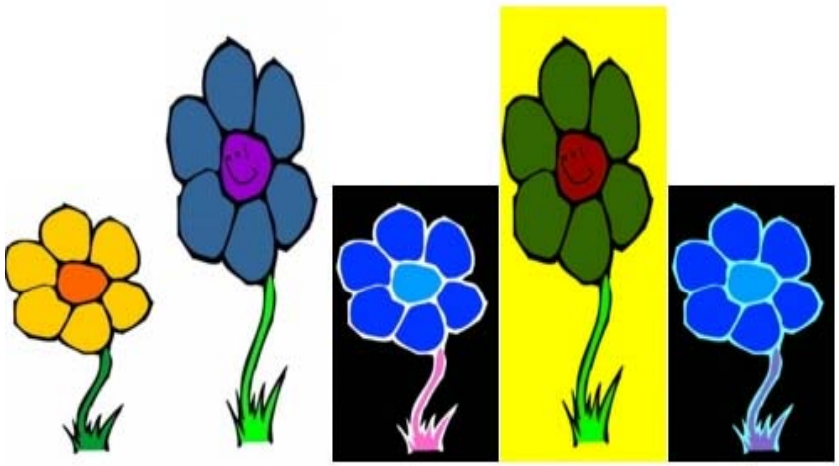


Blank Images

- A couple of ways to get a blank picture to use when creating images
  - Sample images contain empty images with names like 640x480.jpg
  - JES has a makeEmptyPicture(width,height) function that creates a picture without having to read a file

Making a collage

- Could we do something to the pictures we copy in?
  - Sure! Could either apply one of those functions *before* copying, or do something to the pixels *during* the copy.
- Could we copy more than one picture!
  - Of course! Make a collage!



```
def createCollage():
  flower1=makePicture(getMediaPath("flower1.jpg"))
  print flower1
  flower2=makePicture(getMediaPath("flower2.jpg"))
  print flower2
  canvas=makePicture(getMediaPath("640x480.jpg"))
  print canvas
  #First picture, at left edge
  targetX=1
  for sourceX in range(1,getWidth(flower1)):
    targetY=getHeight(canvas)-getHeight(flower1)-5
    for sourceY in range(1,getHeight(flower1)):
      px=getPixel(flower1,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Second picture, 100 pixels over
  targetX=100
  for sourceX in range(1,getWidth(flower2)):
    targetY=getHeight(canvas)-getHeight(flower2)-5
    for sourceY in range(1,getHeight(flower2)):
      px=getPixel(flower2,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
```

Page 76-77

```
  #Third picture, flower1 negated
  negative(flower1)
  targetX=200
  for sourceX in range(1,getWidth(flower1)):
    targetY=getHeight(canvas)-getHeight(flower1)-5
    for sourceY in range(1,getHeight(flower1)):
      px=getPixel(flower1,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Fourth picture, flower2 with no blue
  clearBlue(flower2)
  targetX=300
  for sourceX in range(1,getWidth(flower2)):
    targetY=getHeight(canvas)-getHeight(flower2)-5
    for sourceY in range(1,getHeight(flower2)):
      px=getPixel(flower2,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Fifth picture, flower1, negated with decreased red
  decreaseRed(flower1)
  targetX=400
  for sourceX in range(1,getWidth(flower1)):
    targetY=getHeight(canvas)-getHeight(flower1)-5
    for sourceY in range(1,getHeight(flower1)):
      px=getPixel(flower1,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  show(canvas)
  return(canvas)
```

Cropping: Just the face

```
def copyBarbsFace():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  targetX = 100
  for sourceX in range(45,200):
    targetY = 100
    for sourceY in range(25,200):
      color = getColor(getPixel(barb,sourceX,sourceY))
      setColor(getPixel(canvas,targetX,targetY), color)
      targetY = targetY + 1
    targetX = targetX + 1
  show(barb)
  show(canvas)
  return canvas
```

Again, swapping the loop works fine

```
def copyBarbsFace2():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+(200-45)):
    sourceY = 25
    for targetY in range(100,100+(200-25)):
      color = getColor(getPixel(barb,sourceX,sourceY))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 1
    sourceX = sourceX + 1
  show(barb)
  show(canvas)
  return canvas
```

We can use targetX and targetY as the **for** loop index variables, and everything works the same.

Scaling

- Scaling a picture (smaller or larger) has to do with *sampling* the source picture differently
  - When we just copy, we *sample* every pixel
  - If we want a smaller copy, we skip some pixels
    - We *sample* fewer pixels
  - If we want a larger copy, we duplicate some pixels
    - We *over-sample* some pixels

Scaling the picture down

```
def copyBarbsFaceSmaller():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+((200-45)/2)):
    sourceY = 25
    for targetY in range(100,100+((200-25)/2)):
      color = getColor(getPixel(barb,sourceX,sourceY))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 2
    sourceX = sourceX + 2
  show(barb)
  show(canvas)
  return canvas
```

Scaling Up: Growing the picture

- To grow a picture, we simply duplicate some pixels
- We do this by incrementing by 0.5, but only use the integer part.

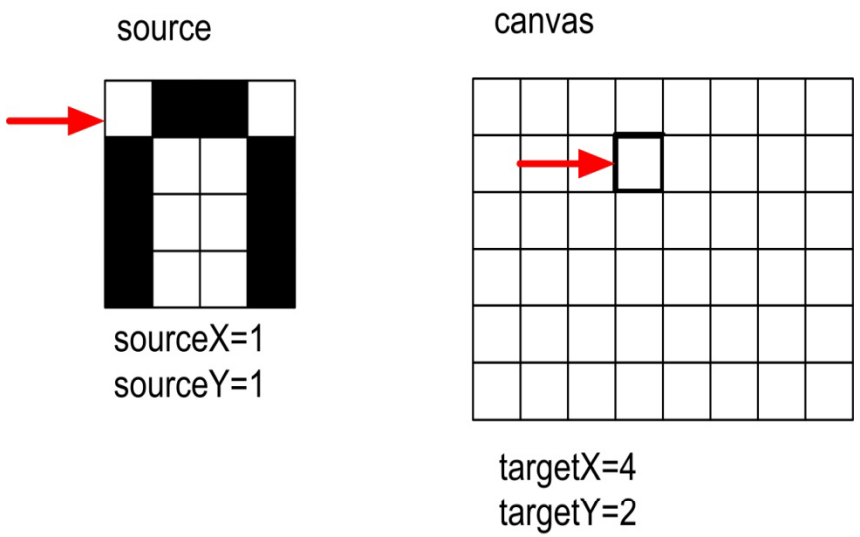
```
>>> print int(1)
1
>>> print int(1.5)
1
>>> print int(2)
2
>>> print int(2.5)
2
```

Scaling the picture up

```
def copyBarbsFaceLarger():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+((200-45)*2)):
    sourceY = 25
    for targetY in range(100,100+((200-25)*2)):
      color = getColor(getPixel(barb,int(sourceX),int(sourceY)))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 0.5
    sourceX = sourceX + 0.5
  show(barb)
  show(canvas)
  return canvas
```

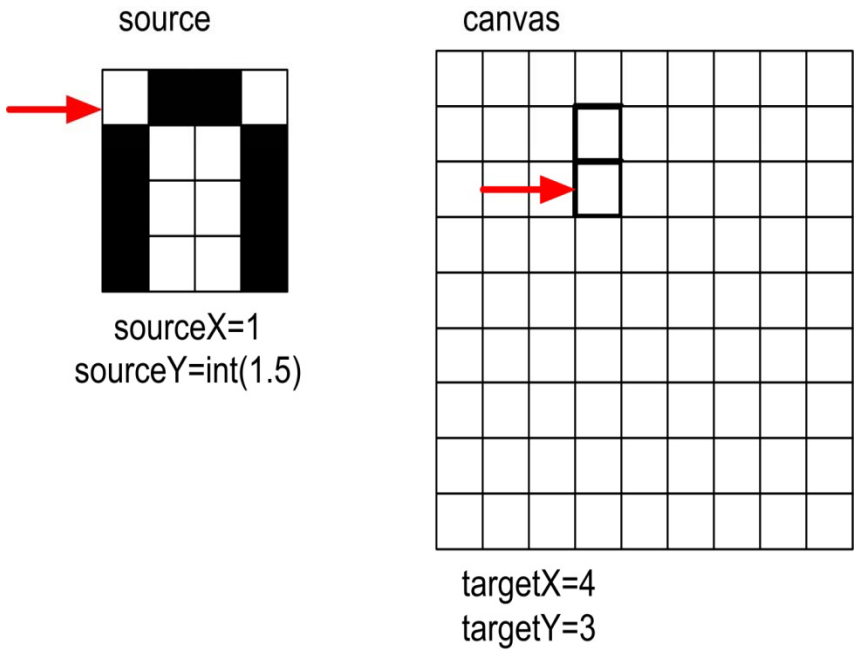
Scaling up: How it works

- Same basic setup as copying and rotating:



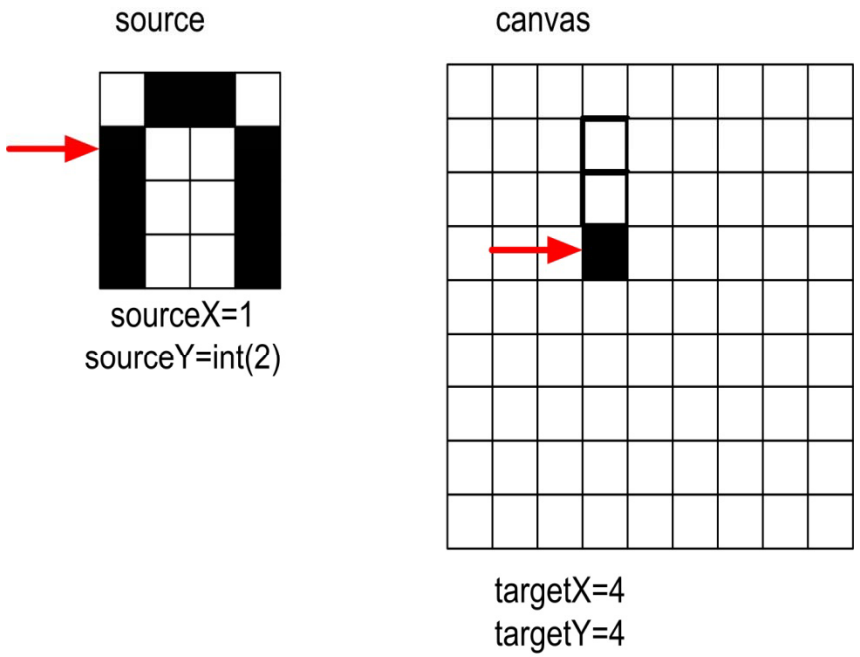
Scaling up: How it works 2

- But as we increment by *only 0.5*, and we use the **int()** function, we end up taking every pixel *twice.*
- Here, the blank pixel at (1,1) in the source gets copied twice onto the canvas.



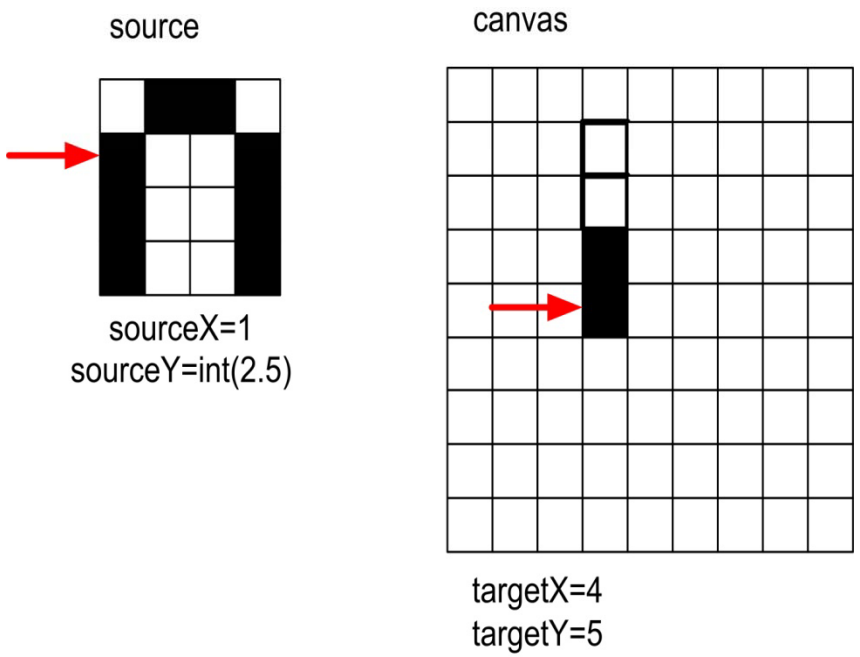
Scaling up: How it works 3

- Black pixels gets copied once…



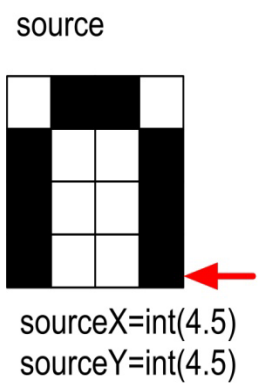
Scaling up: How it works 4

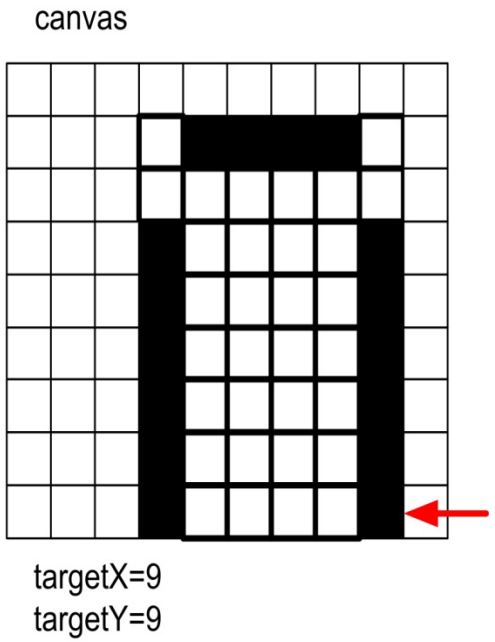
- And twice…



Scaling up: How it ends up

- We end up in the same place in the source, but twice as much in the target.
- Notice the degradation:
  - Gaps that weren't there previously
  - Curves would get "choppy": Pixelated



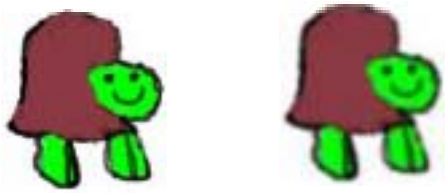


One Last Transformation - Bluring

- There are many ways to blur an image
- Here's a simple one – replace the r,g,b values in each pixel with the average of that pixel's rgb values and the ones above, below, to the left, and to the right
- BUT: we can't do this in a single pass over an image, we need to make a copy. Why?

The Code

```
def blur(source):
  """Return a new picture that is a blured copy of source """
  target = makeEmptyPicture(getWidth(source), getHeight(source))
  for x in range(2, getWidth(source)):
    for y in range(2, getHeight(source)):
      top = getPixel(source,x,y-1)
      left = getPixel(source,x-1,y)
      bottom = getPixel(source,x,y+1)
      right = getPixel(source,x+1,y)
      center = getPixel(source,x,y)
      newRed = (getRed(top) + getRed(left) + getRed(bottom) + getRed(right) + getRed(center)) / 5
      newGreen = (getGreen(top) + getGreen(left) + getGreen(bottom) + getGreen(right) + getGreen(center)) / 5
      newBlue = (getBlue(top) + getBlue(left) + getBlue(bottom) + getBlue(right) + getBlue(center)) / 5
      newPixel = getPixel(target,x,y)
      setColor(newPixel, makeColor(newRed, newGreen, newBlue))
  return target
```



- Unlike the other transformations, this creates a new image and returns it. The caller can show it, save it, or whatever
- Notice that we're careful not to reference x, y coordinates off the edge of the picture

Better Bluring

- Photoshop, GIMP and others have more elaborate blurring algorithms that take more neighbors into account and weigh the pixels more the closer they are.
- For instance, we could use the following weights to calculate each pixel from the 3x3 grid that surrounds it (multiply the colors by these weights then divide by the sum)

1	2	1
2	4	2
1	2	1

More Transformations
More Python

- We've barely gotten started
- There's a whole world of digital media and algorithms out there
- There's more to programming
  - But we've hit some real key points: expressions, variables, assignment, conditionals, loops, functions
- Not bad for two afternoons! Congratulations!!!

Homework Assignment!

- Create a collage where the same picture appears at least three times:
  - Once in its original form
  - Then with any modification you want to make to it
    - Scale, crop, change colors, grayscale, edge detect, posterize, etc.
- Then mirror the whole canvas
  - Creates an attractive layout
  - Horizontal, vertical, or diagonal (if you want to work it out…)
- Hint: write functions – particularly if you wind up copying and pasting the same code a lot
  - Can you simplify things by creating a function and calling it several times with different arguments?