Media Computation Workshop
Day 1

Mark Guzdial
College of Computing
Georgia Institute of Technology
guzdial@cc.gatech.edu
http://www.cc.gatech.edu/~mark.guzdial
http://www.georgiacomputes.org

Workshop Plan-Day 1

- 9 am: Introductions and overview of the workshop.
- 9:30-10:30: Introduction to Media Computation using Python
  - Pictures: Basic Filters
  - 10:30-10:45: Break
- 10:45-12:00: Compositing and scaling images.
- 12:00-1:00: Lunch
- 1:00-2:00: Tackling a homework assignment in Media Computation. *Making a collage*.
- 2:00-3:30: Introducing sound, sound manipulations, splicing sounds.
  - 3:30-3:45: Break
- 3:45-4:30: Tackling a homework assignment in Media Computation. *Making music*.

Workshop Plan-Day 2

- 9-10:00 am: Overview of results of Media Computation.
  - Why a contextualized computing education approach
  - Support available for teachers for adopting, adapting, and assessing.
  - 10:00-10:15: Break
- 10:15-12:00: Pictures and sounds in Java: Overview
- 12:00-1:00: Lunch
- 1:00-2:30: Movies in Media Computation
  - 2:30-2:45: Break
- 2:45-3:15: Discussion. *How might you use these kinds of assignments in your classes?*
- 3:15-4:30: Tackling a homework assignment in Media Computation. *Making a movie*.

Workshop Plan-Day 3

- 9-10:00 am: Introducing objects in a MediaComp way
  - Turtles and MIDI.
  - 10:00-10:15: Break
- 10:15-11:00: Linked lists of MIDI.
- 11:00-12:00: Linked lists and trees of pictures
- 12:00-1:00: Lunch
- 1:00-2:30: Tackling a homework assignment in Media Computation. *Creating linked list music* or *Making a movie with sound*.
  - 2:30-2:45: Break
- 2:45-3:30: Simulations, continuous and discrete
- 3:30-4:30: Creating the wildebeests and villagers: Making movies from simulations .

What’s on your CD

- MediaSources: Royalty-free JPEG and WAV files
- Material for this workshop
  - Workshop slides
- CS1-Python materials
  - MediaTools: Squeak-based media exploration tools
  - Chapters from the Media Computation book in Python
  - Course slides
  - Jython Environment for Students (JES) including new 3.0

What's on your CD - Continued

- CS1-Java materials
  - First 6 chapters of Media Computation book in Java
  - All slides
  - Exercises with solutions
  - DrJava: Great Java IDE for students
  - Classes
  - Slides for Alice + MediaComputation
- CS2-Java materials
  - Course notes
  - Course slides
  - Java Classes for data structures class
  - JMusic (MIDI support)

Introductions

- Who are you?
- Where are you from?
- What do you want to get out of this workshop/

General flow

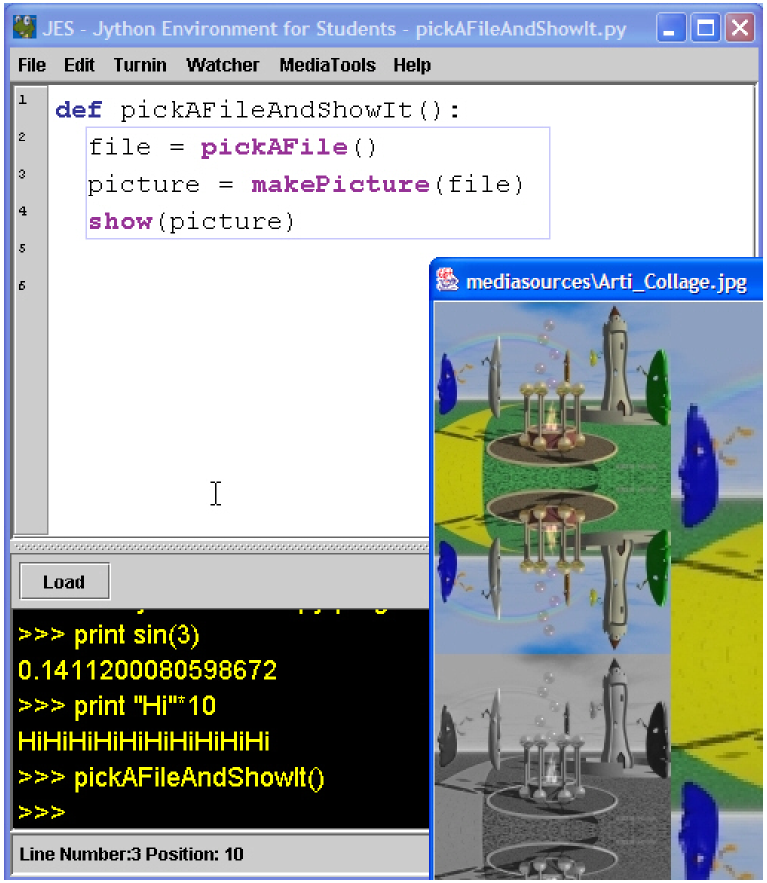
- For each approach:
  - Media Comp Python
  - Media Comp Java
  - Media Comp Data Structures in Java
- 1. An overview of the syllabus
- 2. Slides from class
  - Interspersed with comments for teachers (with blue background titles, like this one)

Multimedia CS1 in Python

- Focus: Learning programming and CS concepts within the context of media manipulation and creation
  - Converting images to grayscale and negatives, splicing and reversing sounds, writing programs to generate HTML, creating movies out of Web-accessed content.
  - ***Computing for communications, not calculation***

Python as the programming language

- *Huge* issue
- Use in commercial contexts authenticates the choice
  - IL&M, Google, Nextel, etc.
- Minimal syntax
- *Looks like* other programming languages
  - Potential for transfer

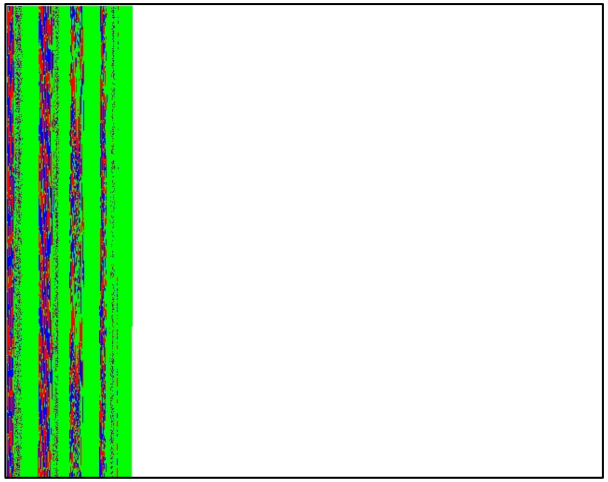


Rough overview of Syllabus

- Defining and executing functions
- Pictures
  - Psychophysics, data structures, defining functions, **for** loops, **if** conditionals
  - Bitmap vs. vector notations
- Sounds
  - Psychophysics, data structures, defining functions, **for** loops, **if** conditionals
  - Sampled sounds vs. synthesized, MP3 vs. MIDI
- Text
  - Converting between media, generating HTML, database, and networking
  - A little trees (directories) and hash tables (database)
- Movies
- ***Then***, Computer Science topics (last 1/3 class)

Some Computer Science Topics inter-mixed

- We talk about *algorithms* across media
  - Sampling a picture (to scale it) is the same algorithm as sampling a sound (to shift frequency)
  - Blending two pictures (fading one into the other) and two sounds is the same algorithm.
- We talk about *representations* and *mappings* (Goedel)
  - From samples to numbers (and into Excel), through a mapping to pixel colors
- We talk about design and debugging
  - But they mostly don't hear us



Computer Science Topics as solutions to *their* problems

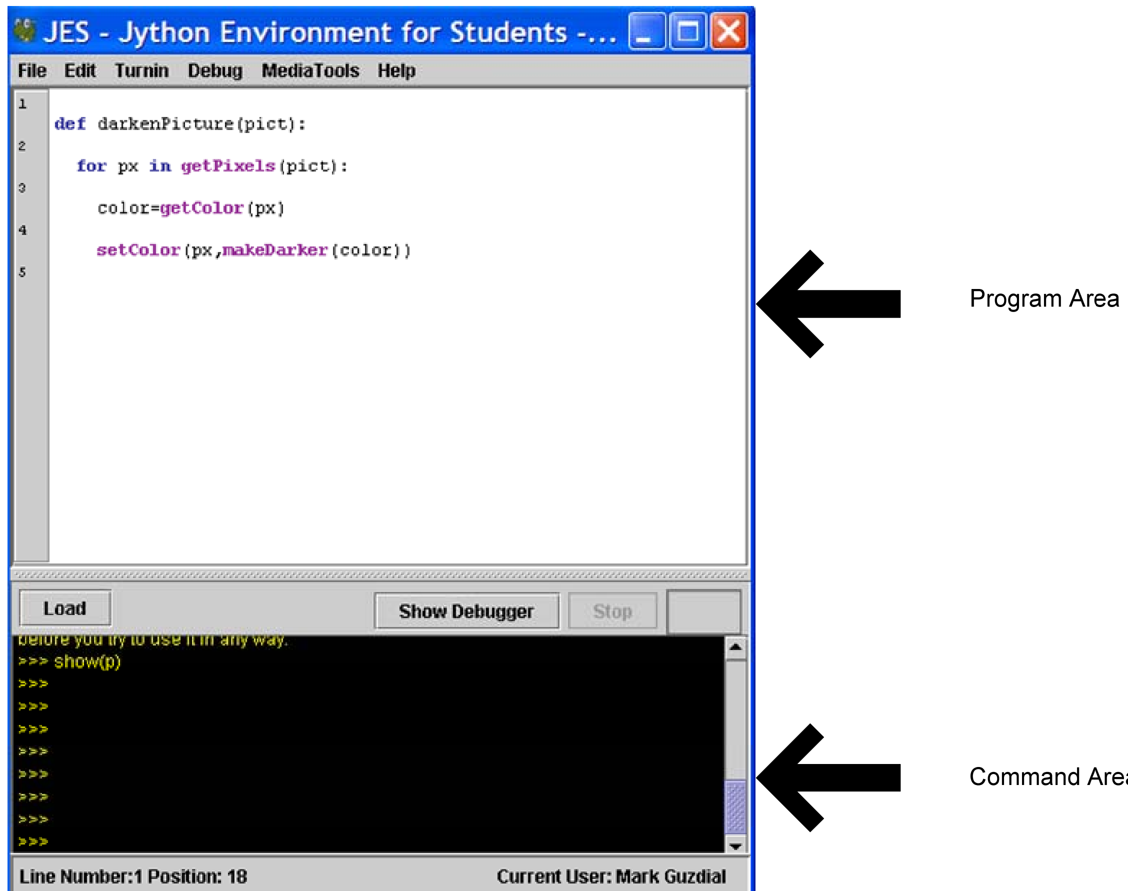
- "Why is PhotoShop so much faster?"
  - Compiling vs. interpreting
  - Machine language and how the computer works
- "Writing programs is *hard!* Are there ways to make it easier? Or at least shorter?"
  - Object-oriented programming
  - Functional programming and recursion
- "Movie-manipulating programs take a *long* time to execute. Why? How fast/slow can programs be?"
  - Algorithmic complexity

Installing JES (Jython Environment for Students)

- Installing JES and starting it up
  - Windows users:
    - Just copy the folder
    - Double-click JES application
  - Mac users:
    - Just copy the folder
    - Double-click the JES application
- There is always Help
  - Lots and lots of excellent help

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.



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" + "Mark"
HelloMark
```

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

Demonstrating JES for files and sounds

```
>>> print pickAFile()
/Users/guzdial/mediasources/barbara.jpg
>>> print makePicture(pickAFile())
Picture, filename /Users/guzdial/mediasources/barbara.jpg height 294 width 222
>>> print pickAFile()
/Users/guzdial/mediasources/hello.wav
>>> print makeSound(pickAFile())
Sound of length 54757
>>> print play(makeSound(pickAFile()))
None
>>> myfilename = pickAFile()
>>> print myfilename
/Users/guzdial/mediasources/barbara.jpg
>>> mypicture = makePicture(myfilename)
>>> print mypicture
Picture, filename /Users/guzdial/mediasources/barbara.jpg height 294 width 222
>>> show(mypicture)
```

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 ("(input1)")
- End the line with a colon (":")
- The *body* of the recipe is indented (Hint: Use three spaces)
  - That's called a *block*

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

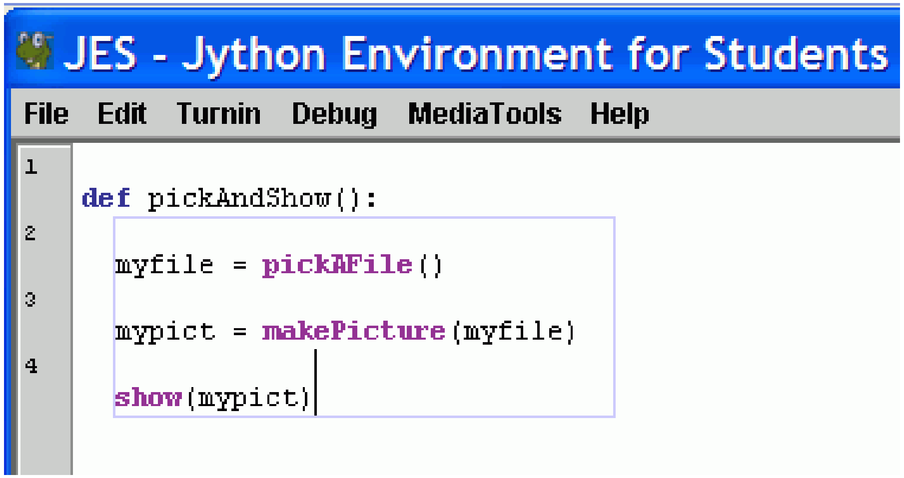
A recipe for playing picked sound files

```
def pickAndPlay():
  myfile = pickAFile()
  mysound = makeSound(myfile)
  play(mysound)
```

Note: myfile and **mysound**, inside **pickAndPlay()**, are *completely different* from the same names in the command area.

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.



A function for playing picked picture files

```
def pickAndShow():
  myfile = pickAFile()
  mypict = makePicture(myfile)
  show(mypict)
```

Explaining variables

- At this point, we'll do lots of variations of filenames and function composition.

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

def pas():
  show(makePicture(pickAFile()))
```

- For both pictures and sounds.
- This is our "Hello, World!"

Image Processing

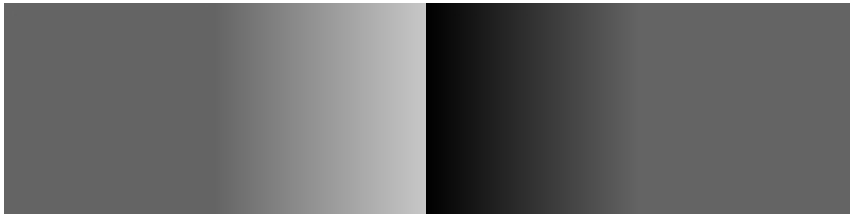
- Goals:
  - Give you the basic understanding of image processing, including psychophysics of sight,
  - Identify some interesting examples to use

We perceive light different from how it actually is

- Color is continuous
  - Visible light is in the wavelengths between 370 and 730 nanometers
    - That's 0.00000037 and 0.00000073 meters
- But we *perceive* light with color sensors that peak around 425 nm (blue), 550 nm (green), and 560 nm (red).
    - Our brain figures out which color is which by figuring out how much of each kind of sensor is responding
    - One implication: We perceive two kinds of "orange" — one that's *spectral* and one that's red+yellow (hits our color sensors just right)
    - Dogs and other simpler animals have only two kinds of sensors
      - They *do* see color. Just *less* color.

Luminance vs. Color

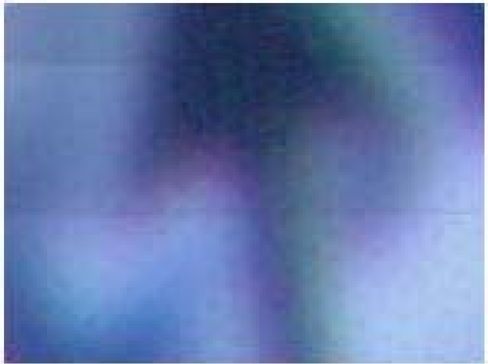
- We perceive borders of things, motion, depth via *luminance*
  - Luminance is *not* the amount of light, but our *perception* of the amount of light.
  - We see blue as "darker" than red, even if same amount of light.
- Much of our luminance perception is based on comparison to backgrounds, not raw values.



Luminance is actually *color blind*. Completely different part of the brain.

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 particularly 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*

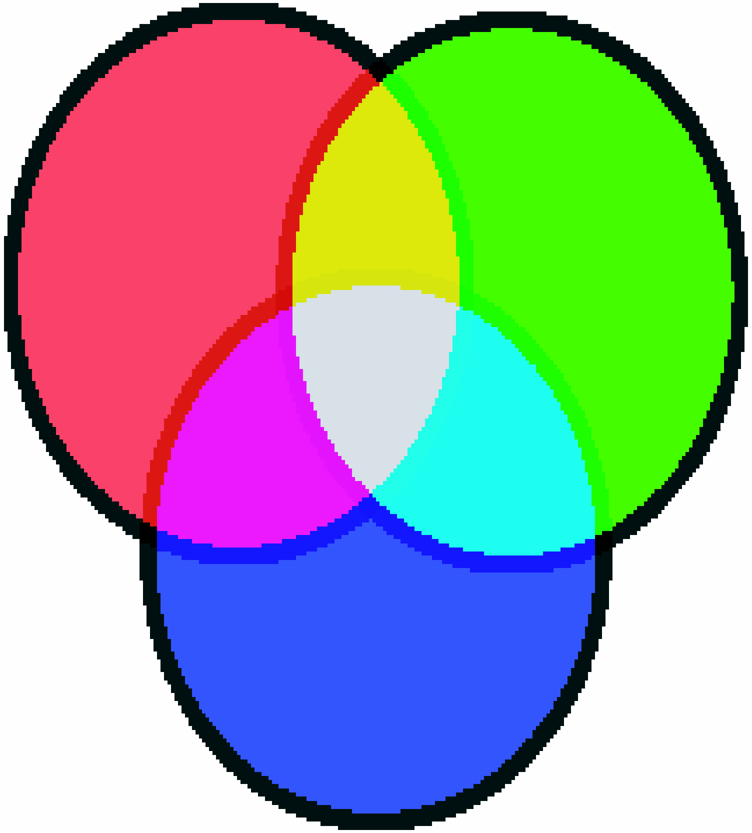


Encoding color

- 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

Basic Picture Functions

- makePicture(filename) creates and returns a picture object, from the JPEG file at the filename
- show(picture) displays a picture in a window
- We’ll learn functions for manipulating pictures later, like getColor, setColor, and repaint

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 (“(input1)”)
- End the line with a colon (“:”)
- The *body* of the recipe is indented (Hint: Use two spaces)
- Your function does **NOT** exist for JES until you *load* it

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="/Users/guzdial/mediasources/barbara.jpg"
>>> 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)
```

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="/Users/guzdial/mediasources/katie.jpg"
>>> picture=makePicture(file)
>>> show(picture)
>>> decreaseRed(picture)
>>> repaint(picture)
```

It’s not iteration—it’s a set operation

- Research on developing SQL in the 1970’s found that people are better at set operations than iteration.
  - For all records, get the last name, and if it starts with “G” then… => HARD!
  - For all records where the last name starts with “G”… => Reasonable!
- Because the Python **for** loop is a forEach, we can start out with treating it as a set operation:
  - “For all pixels in the picture…”

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 four eggs and crack them....
- Beat the eggs until...

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 <u>all</u> the pixels of **pict**. <u>For each</u> 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 <u>style</u>. The <u>meanings</u> 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*

Pixel, color r=135 g=131 b=105	Pixel, color r=133 g=114 b=46	Pixel, color r=134 g=114 b=45	…

p

getPixels()

picture

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**

picture

getPixels()

Pixel, color r=135 g=131 b=105	Pixel, color r=133 g=114 b=46	Pixel, color r=134 g=114 b=45	…

p

value = 135

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**

picture

getPixels()

Pixel, color **r=67** g=131 b=105	Pixel, color r=133 g=114 b=46	Pixel, color r=134 g=114 b=45	…

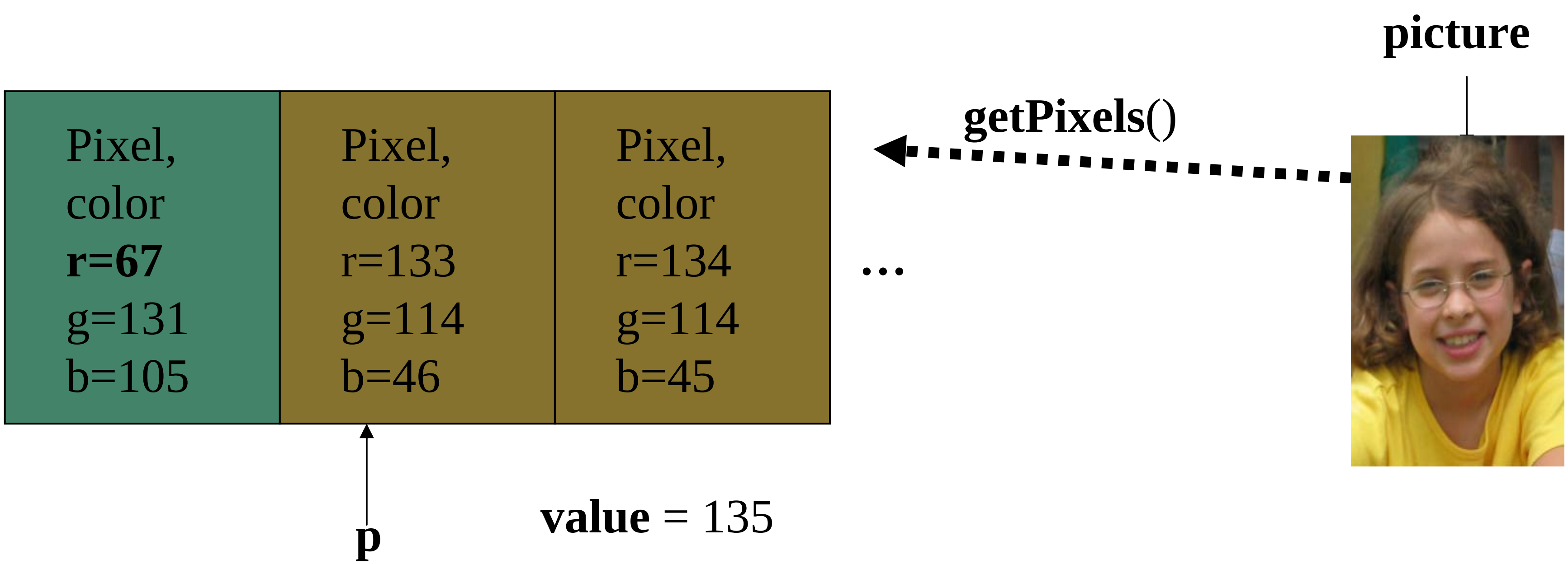
p

value = 135

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.

picture

getPixels()

Pixel, color **r=67** g=131 b=105	Pixel, color r=133 g=114 b=46	Pixel, color r=134 g=114 b=45	…

p

value = 133

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

Pixel, color **r=67** g=131 b=105	Pixel, color r=**66** g=114 b=46	Pixel, color r=134 g=114 b=45	…

p

value = 133

getPixels()

picture

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.

Increasing Red

Happens in JES 1.0, but not 2.0, optional in 3.0

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

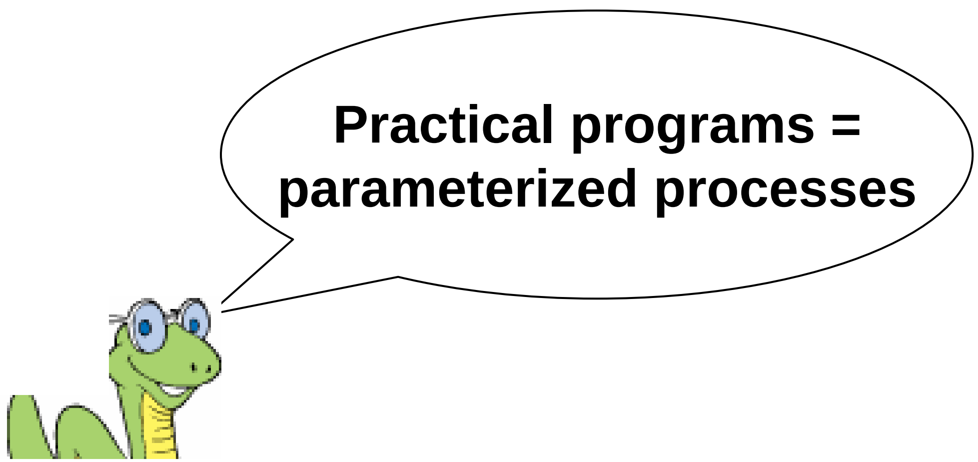
What happened here?!?

Remember that the limit for redness is 255.

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

How does increaseRed differ from decreaseRed?

- Well, it does increase rather than decrease red, but other than that…
  - It takes the same parameter input
  - It can also work for *any* picture
    - It's a specification of a *process* that'll work for *any* picture
    - There's nothing specific to any particular picture here.



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.

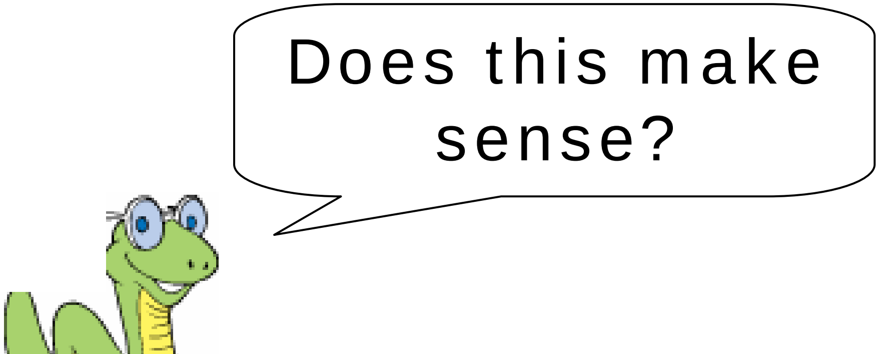
Converting to grayscale

- 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:

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

Converting to grayscale

```
def grayscale(picture):
  for p in getPixels(picture):
    sum = getRed(p) + getGreen(p) + getBlue(p)
    intensity = sum / 3
    setColor(p, makeColor(intensity, intensity, intensity))
```



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

Building a better grayscale

- We'll *weigh* red, green, and blue based on how light we perceive them to be, based on laboratory experiments.

```
def grayScaleNew(picture):
  for px in getPixels(picture):
    newRed = getRed(px) * 0.299
    newGreen = getGreen(px) * 0.587
    newBlue = getBlue(px) * 0.114
    luminance = newRed + newGreen + newBlue
    setColor(px, makeColor(luminance, luminance, luminance))
```

Lots and lots of filters

- There are many wonderful examples that we can do at this point.
- Students see them as all different.
- We know that they are all practice with simple loops.
- Here are a few more before we get to a more traditional **for** loop.
  - More in book (like chromakey, background subtraction, edge detection)

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
  - I used the MediaTools to find the RGB values for the brown of Barbara's hair
  - I 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
- The file is where the picture of Barbara is on my computer
- 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

```
if distance(color, brown) < 50.0:
    redness=getRed(px)*1.5
    blueness=getBlue(px)
    greenness=getGreen(px)
```

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)
```

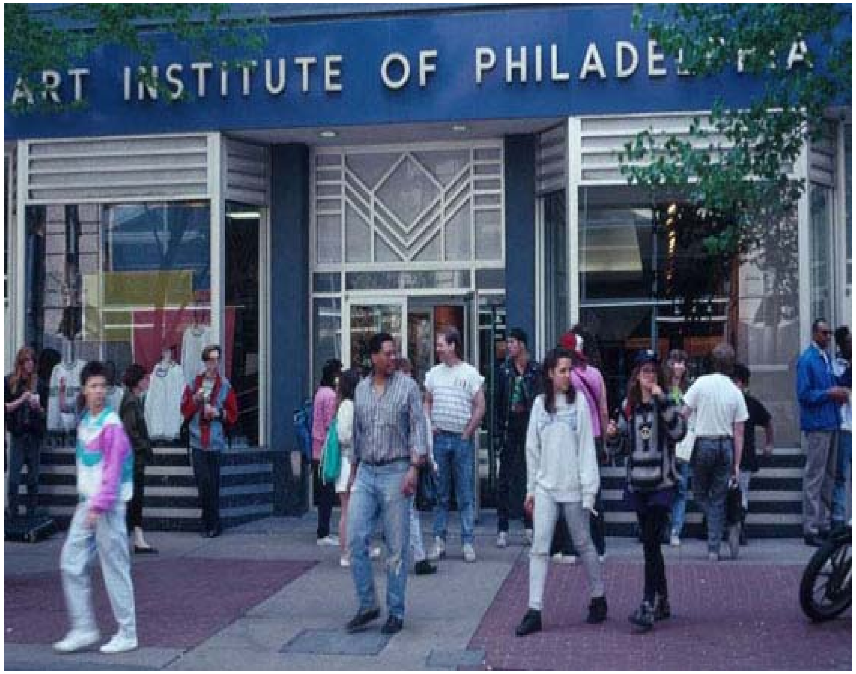
Things to change

- 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

Introducing 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.

Working the pixels by number

- To use **range**, 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 increaseRed2(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*1.1)
```

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)
```

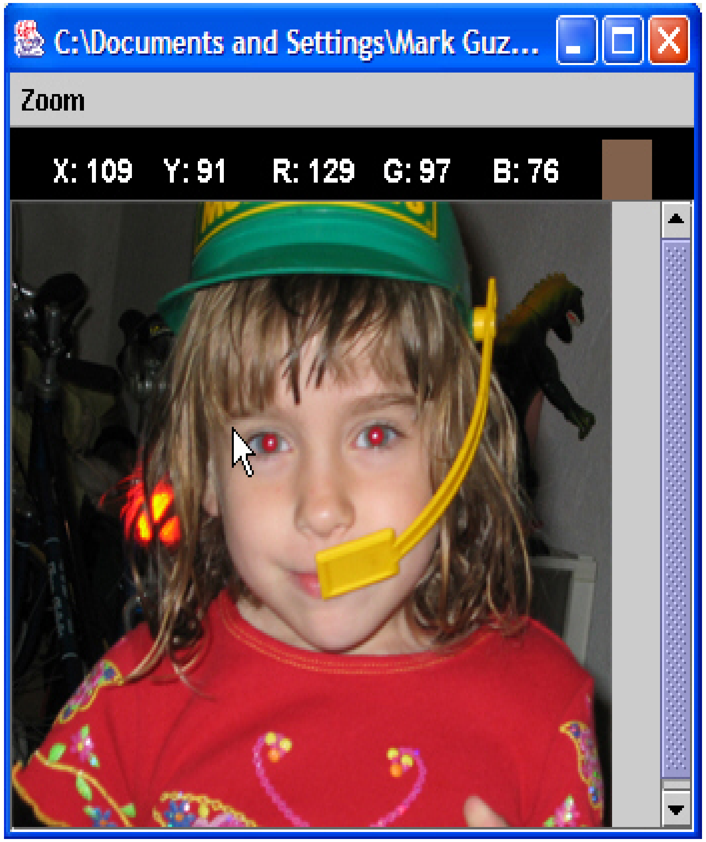
Could we do this without nested loops?

- Yes, but complicated IF

```
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!

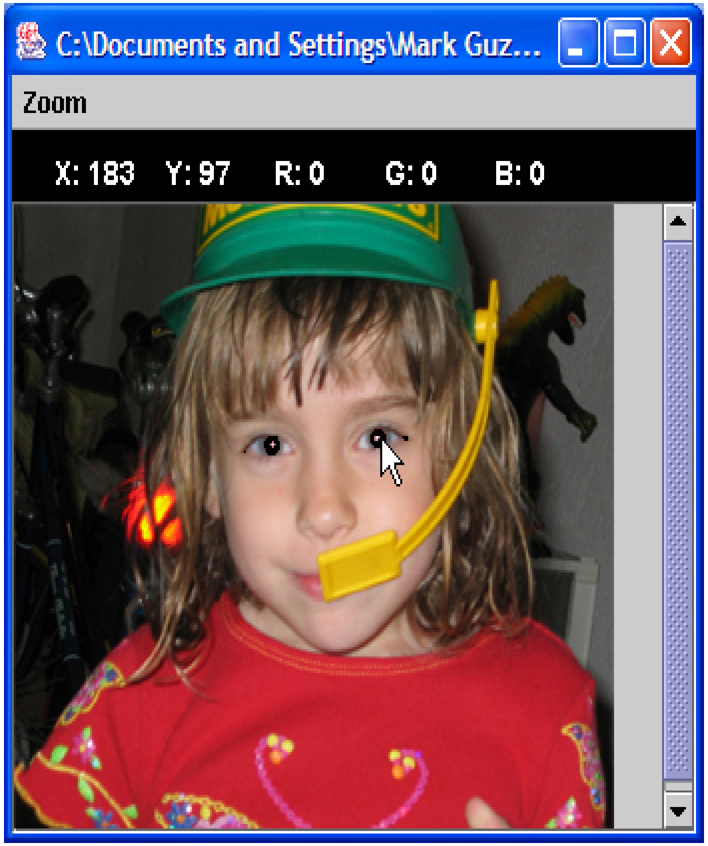
What we’re doing here:

- **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

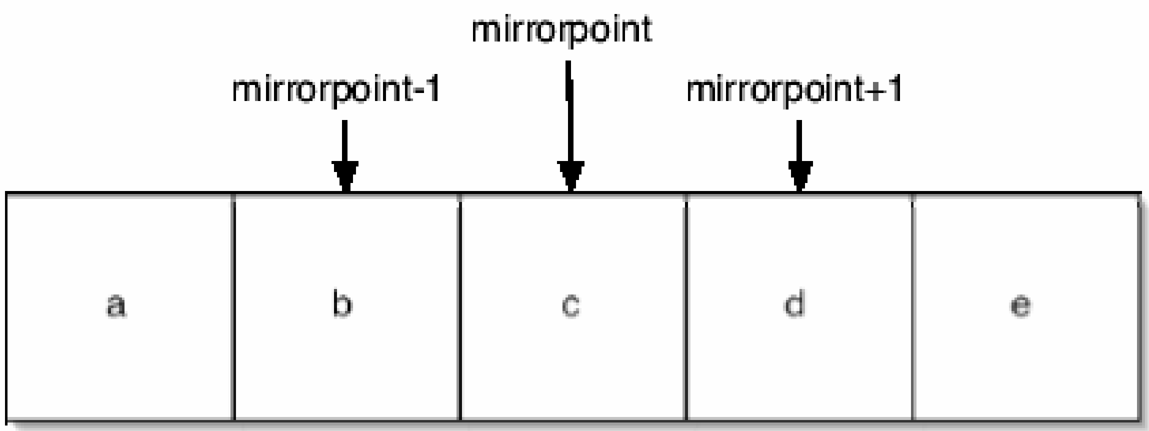


If you know where the pixels are: Mirroring

- 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)
```

Can we do it with a horizontal mirror?

```
def mirrorHorizontal(source):
  mirrorpoint = int(getHeight(source)/2)
  for yOffset in range(1,mirrorpoint):
    for x in range(1,getWidth(source)):
      pbottom = getPixel(source,x,yOffset+mirrorpoint)
      ptop = getPixel(source,x,mirrorpoint-yOffset)
      setColor(pbottom,getColor(ptop))
```

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 that I took on 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?

More Picture Methods

- Compositing and scaling
  - Necessary for making a collage

Copying pixels

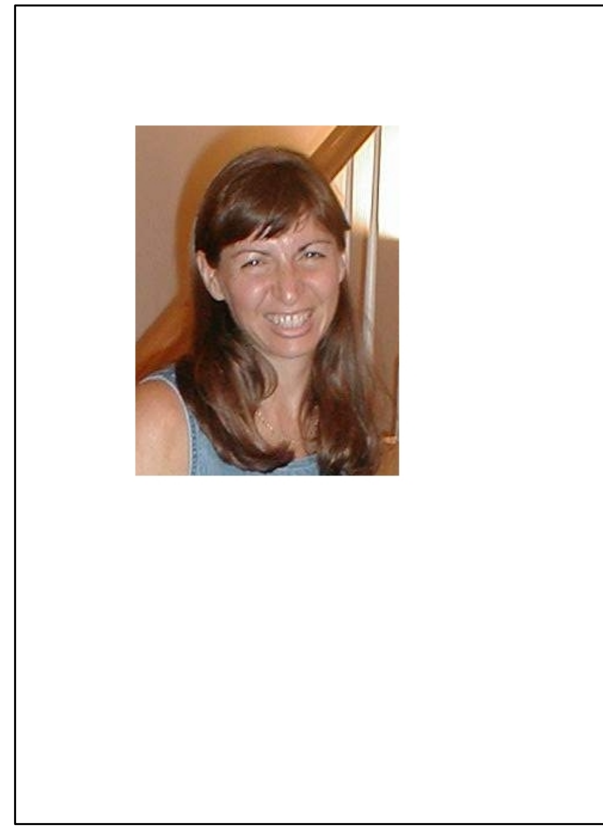
- In general, what we want to do is to keep track of a sourceX and sourceY, and a targetX and targetY.
  - We *increment* (add to them) in pairs
    - sourceX and targetX get incremented together
    - sourceY and targetY get incremented together
  - The tricky parts are:
    - Setting values *inside*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

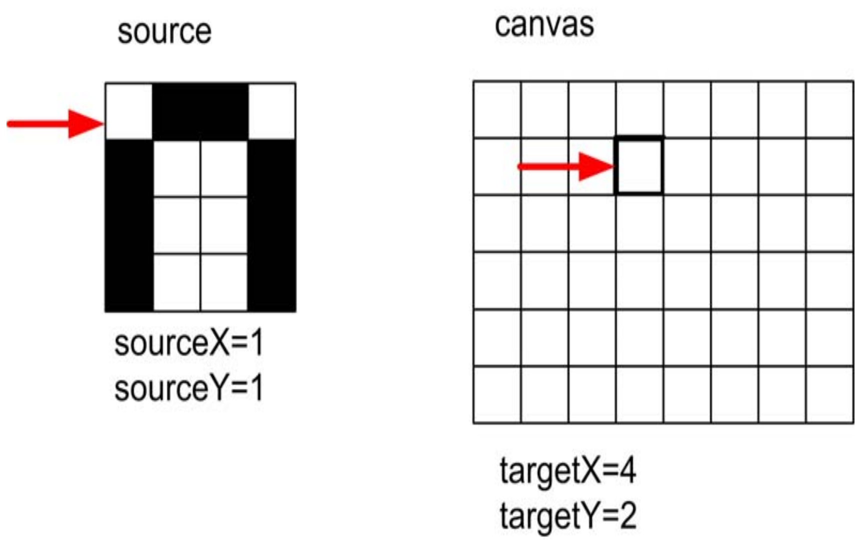
Copying into the middle of the canvas

```
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
```



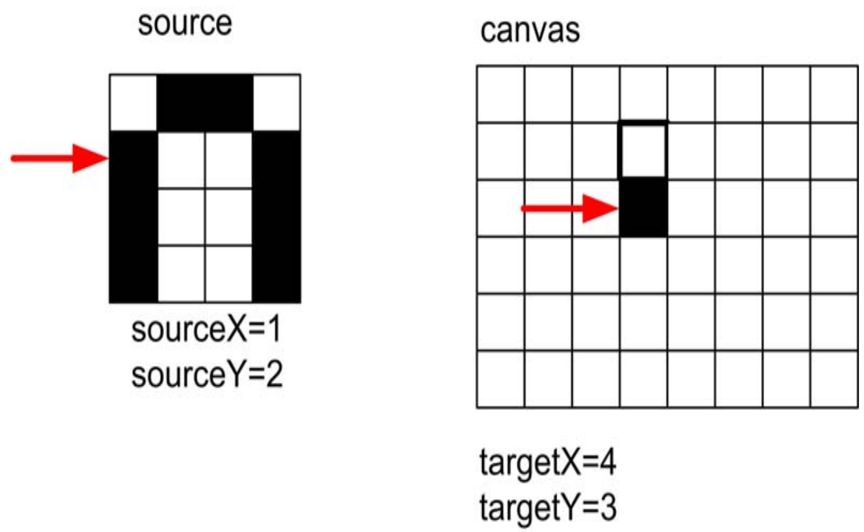
Copying: How it works

- Here's the initial setup:



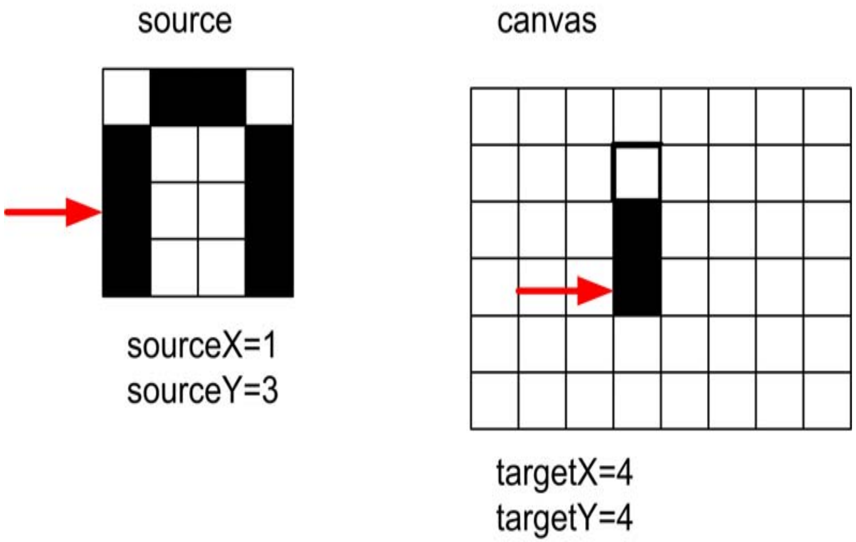
Copying: How it works 2

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



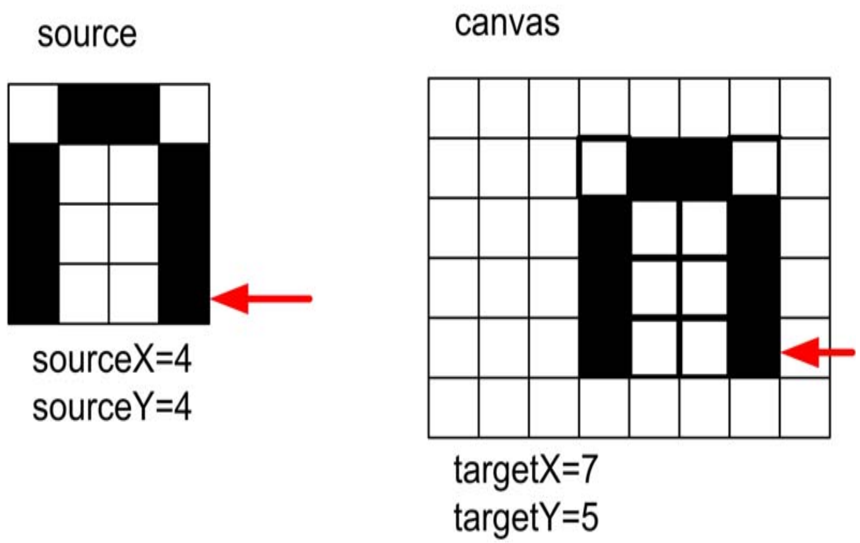
Copying: How it works 3

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



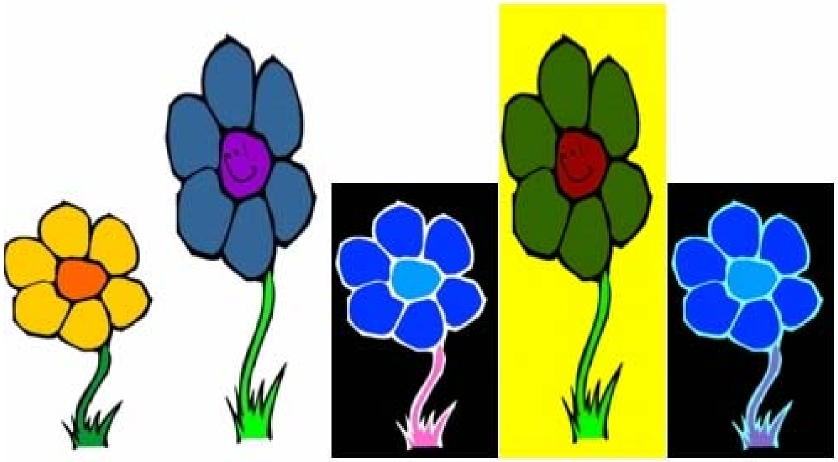
Copying: How it looks at the end

- Eventually, we copy every pixel



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
```

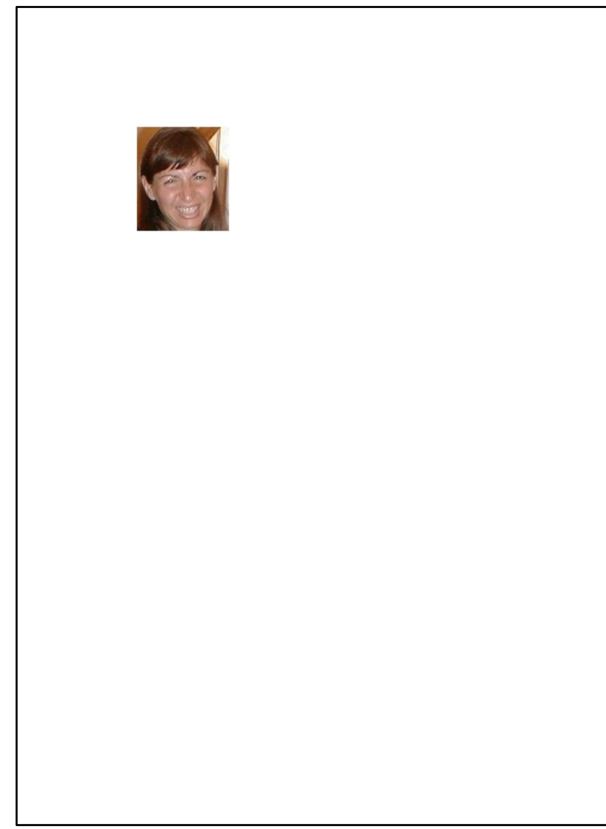
We can use targetX and targetY as the **for** loop index variables, and everything works the same.

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.

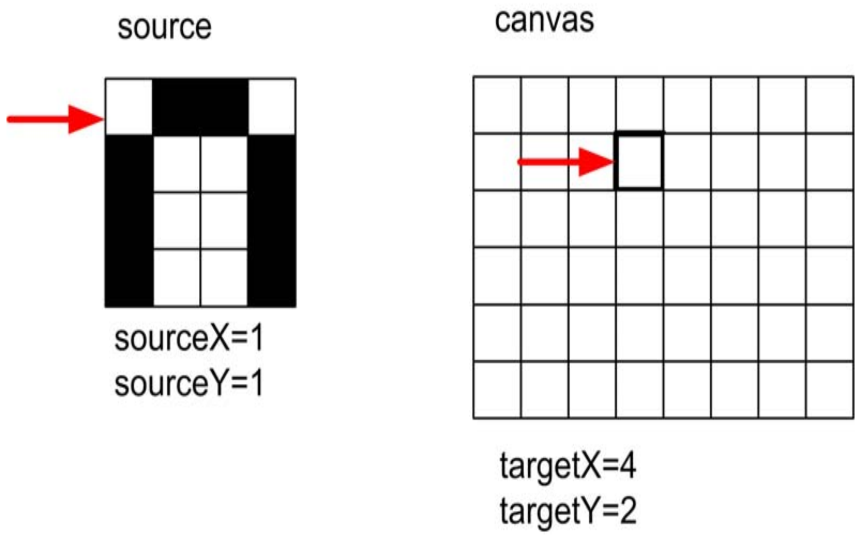
```
>>> print int(1)
1
>>> print int(1.5)
1
>>> print int(2)
2
>>> print int(2.5)
2
```

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
```

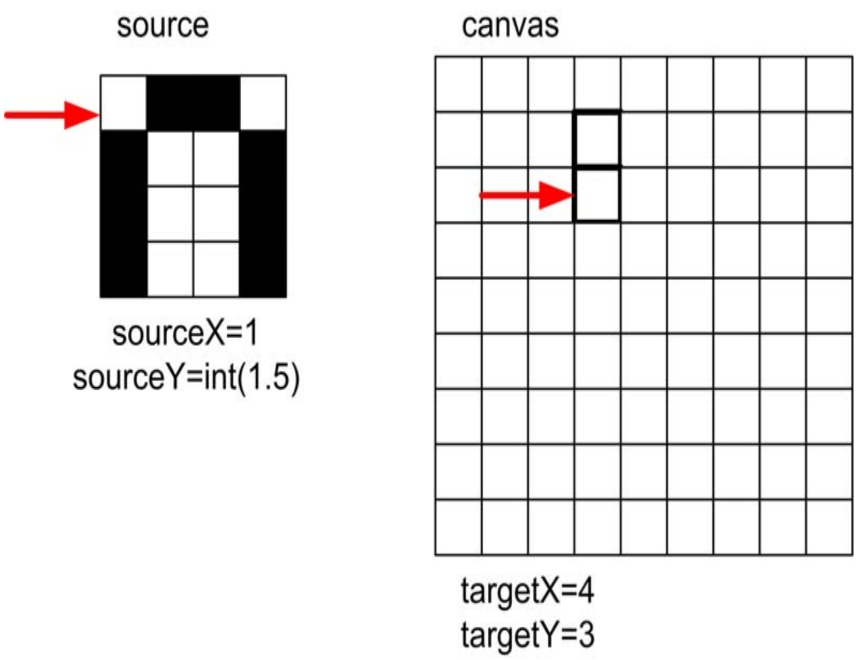
Scaling up: How it works

- Same basic setup as copying and rotating:



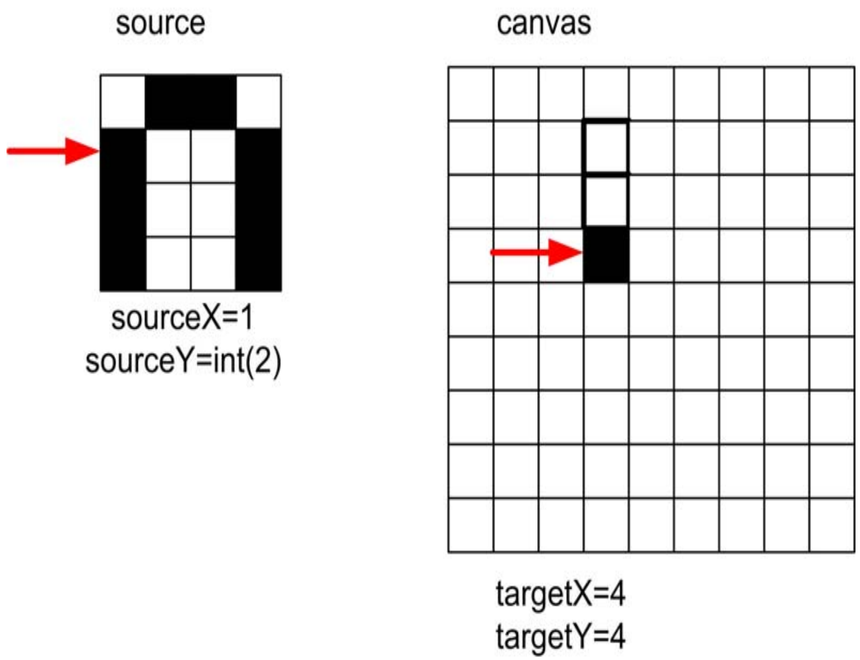
Scaling up: How it works 2

- 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.



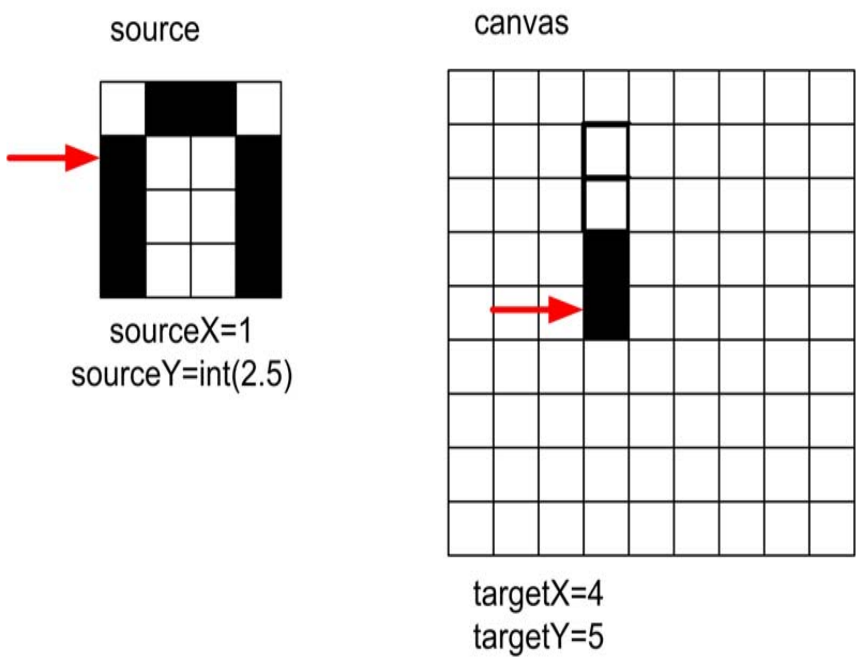
Scaling up: How it works 3

- Black pixels gets copied once…



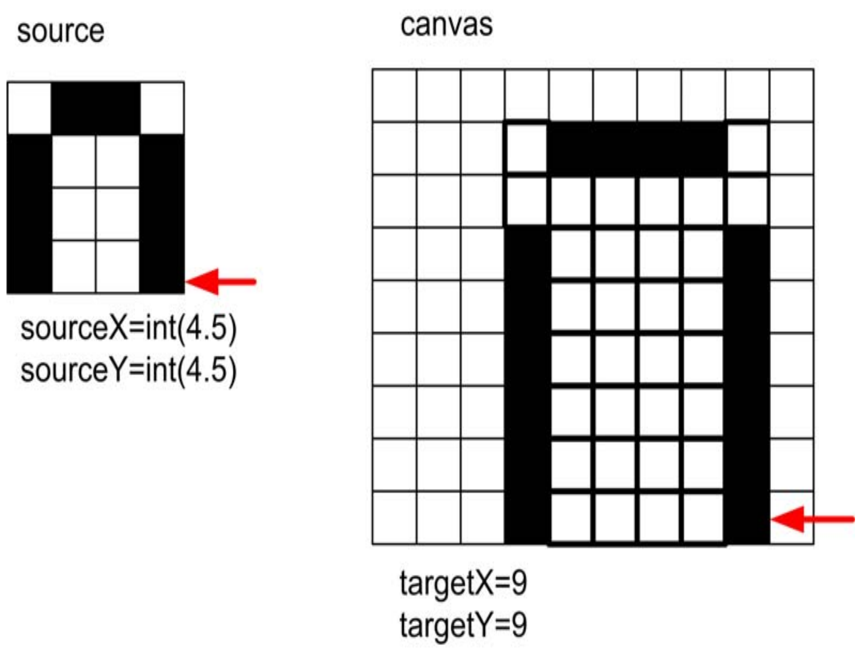
Scaling up: How it works 4

- 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



Homework Assignment!

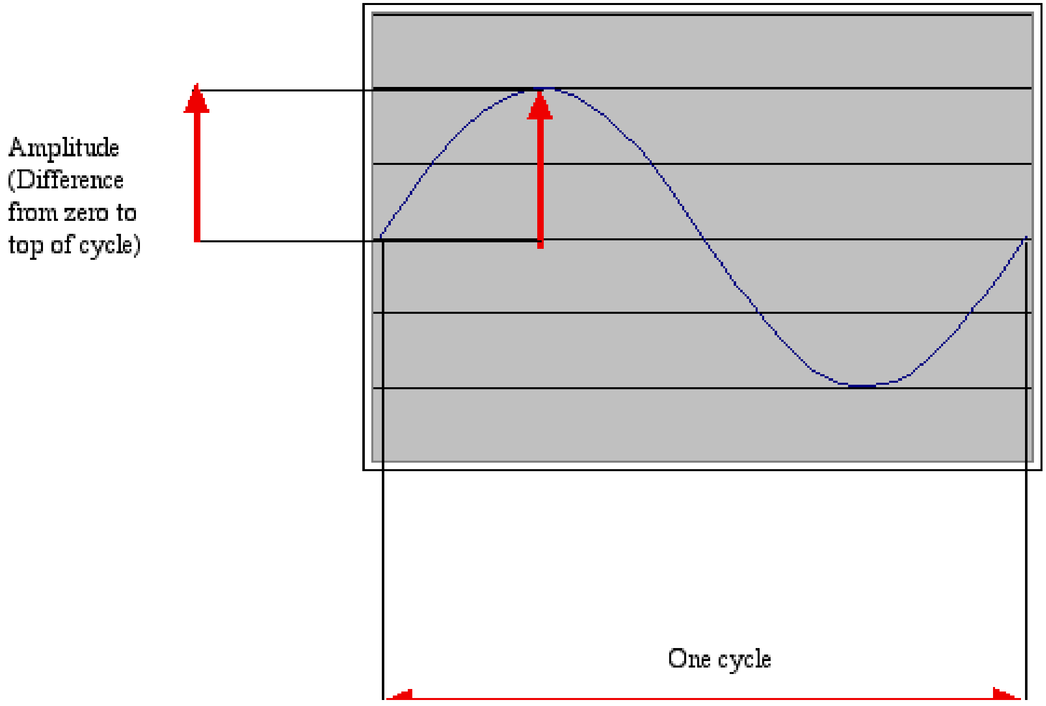
- Create a collage where the same picture appears at least three times:
  - Once in its original form
  - Then with *any* modification you want to make to it
    - Scale, crop, change colors, grayscale, edge detect, posterize, etc.
- Then mirror the whole canvas
  - Creates an attractive layout
  - Horizontal, vertical, or diagonal (if you want to work it out…)
- We'll spend an hour on this.
  - Save pictures with writePictureTo(picture,filename)
  - Share them at http://home.cc.gatech.edu/gacomputes
    - Key: "workshop"

Sound Processing

- Goals:
  - Give you the basic understanding of audio processing, including psycho-acoustics,
  - Identify some interesting examples to use.

How sound works:
Acoustics, the physics of sound

- Sounds are waves of air pressure
  - Sound comes in cycles
  - The *frequency* of a wave is the number of cycles per second (cps), or *Hertz*
    - (Complex sounds have more than one frequency in them.)
  - The amplitude is the maximum height of the wave



Live demos here!

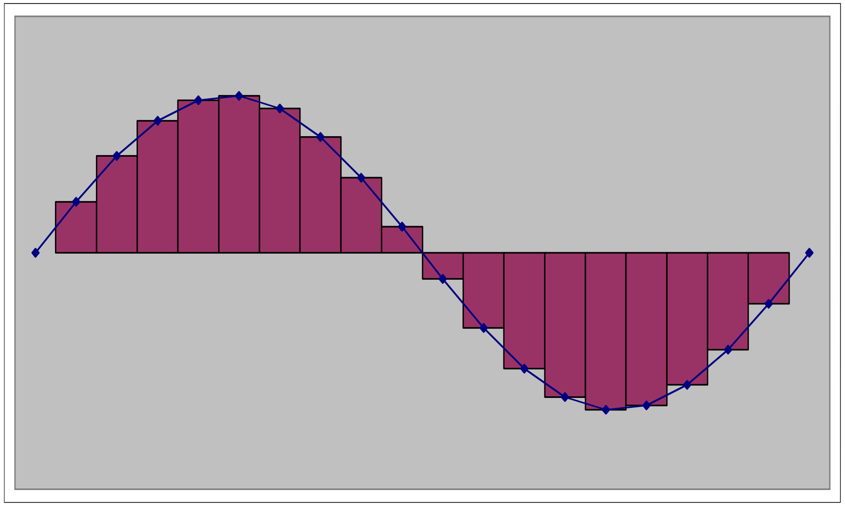
- Use the Squeak MediaTools to see real sound patterns.
- Try to bring in few musical instruments

Volume and pitch: Psychoacoustics, the psychology of sound

- Our perception of volume is related (logarithmically) to changes in amplitude
  - If the amplitude doubles, it's about a 3 decibel (dB) change.
  - A *decibel* is a ratio between two intensities: $10 * \log_{10}(I_1/I_2)$
  - As an absolute measure, it's in comparison to threshold of audibility
    - 0 dB can't be heard.
    - Normal speech is 60 dB.
    - A shout is about 80 dB
- Our perception of pitch is related (logarithmically) to changes in frequency
  - Higher frequencies are perceived as higher pitches
  - We can hear between 5 Hz and 20,000 Hz (20 kHz)
  - A above middle C is 440 Hz

Digitizing Sound: How do we get that into numbers?

- Remember in calculus, estimating the curve by creating rectangles?
- We can do the same to estimate the sound curve
  - Analog-to-digital conversion (ADC) will give us the amplitude at an instant as a number: a *sample*
  - How many samples do we need?



Nyquist Theorem

- We need twice as many samples as the maximum frequency in order to represent (and recreate, later) the original sound.
- The number of samples recorded per second is the *sampling rate*
  - If we capture 8000 samples per second, the highest frequency we can capture is 4000 Hz
    - That’s how phones work
  - If we capture more than 44,000 samples per second, we capture everything that we can hear (max 22,000 Hz)
    - CD quality is 44,100 samples per second

Digitizing sound in the computer

- Each sample is stored as a number (two bytes)
- What's the range of available combinations?
  - 16 bits, 2^{16} = 65,536
  - But we want both positive and negative values
    - To indicate compressions and rarefactions.
  - What if we use one bit to indicate positive (0) or negative (1)?
  - That leaves us with 15 bits
  - 15 bits, 2^{15} = 32,768
  - One of those combinations will stand for zero
    - We'll use a "positive" one, so that's one less pattern for positives
- Each sample can be between -32,768 and 32,767

Basic Sound Functions

- makeSound(filename) creates and returns a sound object, from the WAV file at the filename
- play(sound) makes the sound play (but doesn’t wait until it’s done)
- blockingPlay(sound) waits for the sound to finish
- We’ll learn more later like getSample and setSample

Working with sounds

- We'll use **pickAFile** and **makeSound** as we have before.
  - But now we want .wav files
- We'll use **getSamples** to get all the *sample objects* out of a sound
- We can also get the value at any index with **getSampleValueAt**
- Sounds also know their length (**getLength**) and their sampling rate (**getSamplingRate**)
- Can save sounds with **writeSoundTo(sound,"file.wav")**

Recipe to Increase the Volume

```
def increaseVolume(sound):
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value * 2)
```

Using it:

```
>>> f="/Users/guzdial/mediasources/gettysburg10.wav"
>>> s=makeSound(f)
>>> increaseVolume(s)
>>> play(s)
>>> writeSoundTo(s,"/Users/guzdial/mediasources/louder-g10.wav")
```

Decreasing the volume

```
def decreaseVolume(sound):
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value * 0.5)
```

This works *just* like **increaseVolume**, but we're *lowering* each sample by 50% instead of doubling it.

Maximizing volume

- How do we get maximal volume?
- It's a three-step process:
  - First, figure out the loudest sound (largest sample).
  - Next, figure out a multiplier needed to make that sound fill the available space.
    - We want to solve for *x* where *x * loudest = 32767*
    - So, *x = 32767/loudest*
  - Finally, multiply the multiplier times every sample

Maxing (*normalizing*) the sound

```
def normalize(sound):
    largest = 0
    for s in getSamples(sound):
        largest = max(largest,getSample(s) )
    multiplier = 32767.0 / largest

    print "Largest sample value in original sound was",  largest
    print "Multiplier is", multiplier

    for s in getSamples(sound):
        louder =  multiplier * getSample(s)
        setSample(s,louder)
```

Increasing volume by *sample index*

```
def increaseVolumeByRange(sound):
  for sampleIndex in range(1,getLength(sound)+1):
    value = getSampleValueAt(sound,sampleIndex)
    setSampleValueAt(sound,sampleIndex,value * 2)
```

This really is the same as:

```
def increaseVolume(sound):
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value * 2)
```

Recipe to play a sound backwards (Trace it!)

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```

Start at end of sound

Work backward

Return the processed sound for further use in the function that calls playBackward

How does this work?

- We make two copies of the sound
- The **srcSample** starts at the end, and the **destSample** goes from 1 to the end.
- Each time through the loop, we copy the sample value from the **srcSample** to the **destSample**

Note that the **destSample** is *increasing* by 1 each time through the loop, but **srcSample** is *decreasing* by 1 each time through the loop

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```

Starting out (3 samples here)

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```

You are here

12	25	13

source

12	25	13

dest

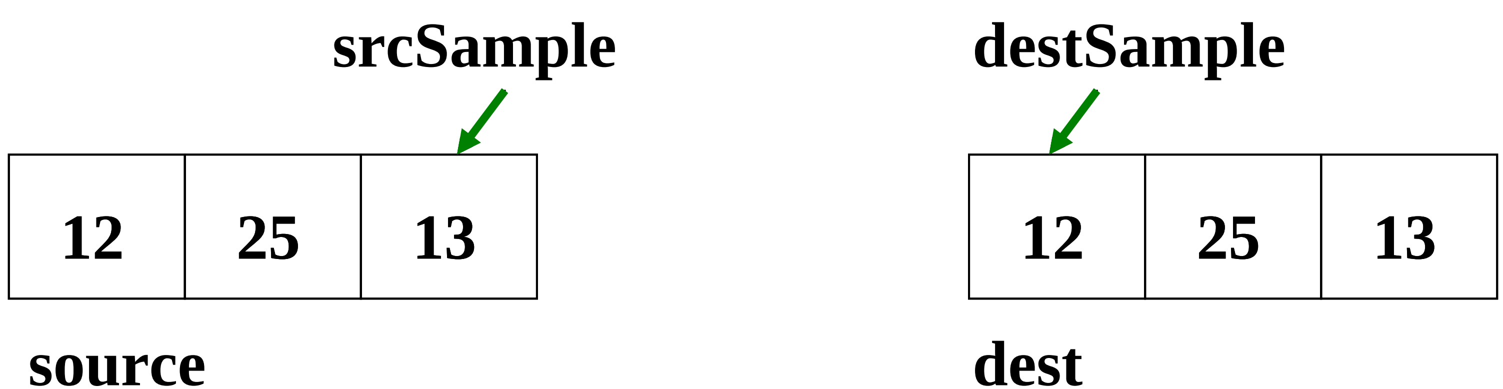
Ready for the copy

Ready for the copy

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```



Do the copy

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```

Ready for the next one?

Ready for the next one?

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```

Moving them together

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```

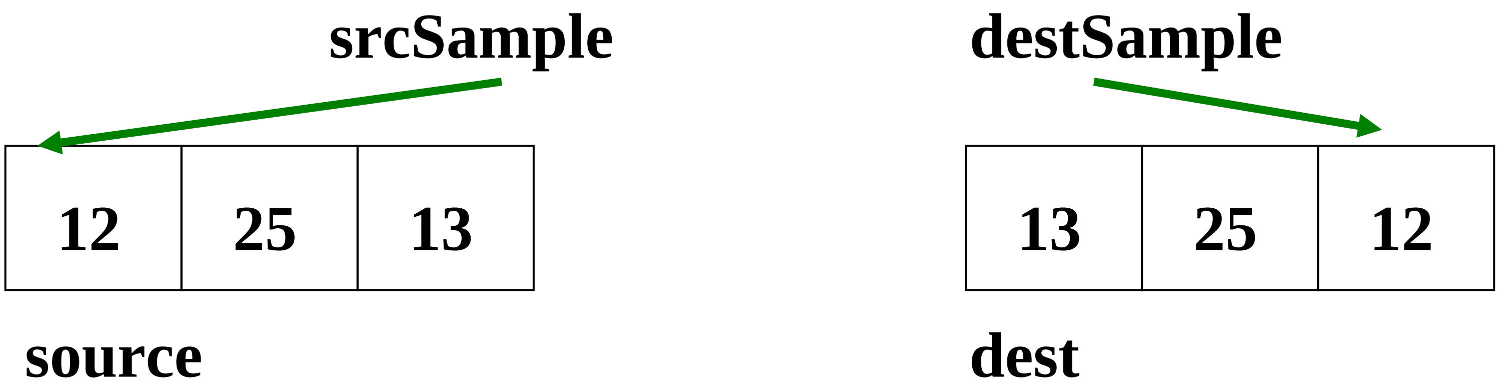
You are here

How we end up

```
def playBackward(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = getLength(source)
  for destSample in range(1, getLength(dest)+1):
    srcVolume = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, srcVolume)
    srcSample = srcSample - 1

  return dest
```



Recipe for halving the frequency of a sound

```
def half(filename):
  source = makeSound(filename)
  dest = makeSound(filename)

  srcSample = 1
  for destSample in range(1, getLength(dest)+1):
    volume = getSampleValueAt(source,            int(srcSample) )
    setSampleValueAt(dest, destSample, volume)
    srcSample = srcSample + 0.5

  play(dest)
  return dest
```

This is how a sampling synthesizer works!

Here are the piece that do it

Changing pitch of sound vs. changing picture size

```
def copyBarbsFaceLarger():
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  sourceX = 45
  for targetX in range(100,100+((200-45)*2)):
    sourceY = 25
    for targetY in range(100,100+((200-25)*2)):
      px = getPixel(barb,int(sourceX),int(sourceY))
      color = getColor(px)
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 0.5
    sourceX = sourceX + 0.5
  show(barb)
  show(canvas)
  return canvas
```

1

2

3

```
def half(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  srcSample = 1
  for destSample in range(1, getLength(dest)+1):
    vol = getSampleValueAt( source, int(srcSample))
    setSampleValueAt(dest, destSample, vol)
    srcSample = srcSample + 0.5

  play(dest)
  return dest
```

1

2

3

Both of them are *sampling*

- Both of them have three parts:
  - A start where objects are set up
  - A loop where samples or pixels are copied from one place to another

    1

    2
    - To decrease the frequency or the size, we take each sample/pixel twice
    - In both cases, we do that by incrementing the index by 0.5 and taking the integer of the index
  - Finishing up and returning the result

3

Recipe to double the frequency of a sound

Here's the critical piece: We skip every other sample in the source!

```
def double(filename):
  source = makeSound(filename)
  target = makeSound(filename)
  targetIndex = 1
  for sourceIndex in range(1, getLength(source)+1, 2):
    setSampleValueAt( target, targetIndex,
          getSampleValueAt( source, sourceIndex))
    targetIndex = targetIndex + 1
  #Clear out the rest of the target sound -- it's only half full!
  for secondHalf in range( getLength( target)/2, getLength( target)):
    setSampleValueAt(target,targetIndex,0)
    targetIndex = targetIndex + 1
  play(target)
  return target
```

What happens if we don’t “clear out” the end?

Try this out!

```
def double(filename):
  source = makeSound(filename)
  target = makeSound(filename)
  targetIndex = 1
  for sourceIndex in range(1, getLength(source)+1, 2):
    setSampleValueAt( target, targetIndex,
             getSampleValueAt( source, sourceIndex))
    targetIndex = targetIndex + 1
  #Clear out the rest of the target sound -- it's only half full!
  #for secondHalf in range( getLength( target)/2, getLength( target)):
  #  setSampleValueAt(target,targetIndex,0)
  #  targetIndex = targetIndex + 1
  play(target)
  return target
```

“Switch off” these lines of code by commenting them out.

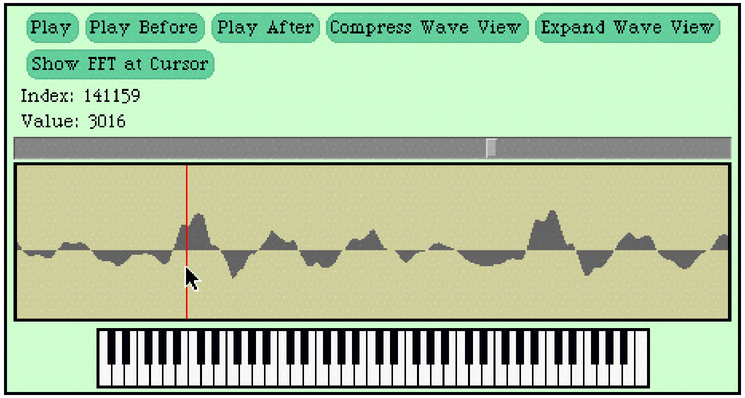
Splicing Sounds

- Splicing gets its name from literally cutting and pasting pieces of magnetic tape together
- Doing it digitally is easy (in principle), but painstaking
- Say we want to splice pieces of speech together:
  - We find where the end points of words are
  - We copy the samples into the right places to make the words come out as we want them
  - (We can also change the volume of the words as we move them, to increase or decrease emphasis and make it sound more natural.)

Finding the word end-points

- Using *MediaTools* and play before/after cursor, can figure out the index numbers where each word ends

Word	Ending index
We	15730
the	17407
People	26726
of	32131
the	33413
United	40052
States	55510



Now, it's all about copying

- We have to keep track of the source and target indices, **srcSample** and **destSample**

```
destSample = Where-the-incoming-sound-should-start
for srcSample in range(startingPoint, endingPoint):
    sampleValue = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, sampleValue)
    destSample = destSample + 1
```

The Whole Splice

```
def splicePreamble():
  file = "/Users/guzdial/mediasources/preamble10.wav"
  source = makeSound(file)
  dest = makeSound(file)   # This will be the newly spliced sound
  destSample=17408         # targetIndex starts at just after "We the" in the new sound
  for srcSample in range( 33414, 40052):  # Where the word "United" is in the sound
    setSampleValueAt(dest, destSample,  getSampleValueAt( source, srcSample))
    destSample = destSample + 1
  for srcSample in range(17408, 26726):   # Where the word "People" is in the sound
    setSampleValueAt(dest, destSample, getSampleValueAt( source,  srcSample))
    destSample = destSample + 1
  for index in range(1, 1000):                     #Stick some quiet space after that
    setSampleValueAt(dest,  destSample, 0)
    destSample = destSample + 1
  play(dest)                                       #Let's hear and return the result
  return dest
```

What's going on here?

- First, set up a source and target.
- Next, we copy "United" (samples 33414 to 40052) after "We the" (sample 17408)
  - That means that we end up at 17408+(40052-33414) = 17408+6638=24046
  - Where does "People" start?
- Next, we copy "People" (17408 to 26726) immediately afterward.
  - Do we have to copy "of" to?
  - Or is there a pause in there that we can make use of?
- Finally, we insert a little (1/441-th of a second) of space – 0's

Word	Ending index
We	15730
the	17407
People	26726
of	32131
the	33413
United	40052
States	55510

What if we didn't do that second copy? Or the pause?

```
def splicePreamble():
  file = "/Users/guzdial/mediasources/preamble10.wav"
  source = makeSound(file)
  dest = makeSound(file)   # This will be the newly spliced sound
  destSample=17408         # targetIndex starts at just after "We the" in the new sound
  for srcSample in range( 33414, 40052):  # Where the word "United" is in the sound
    setSampleValueAt(dest, destSample,  getSampleValueAt( source, srcSample))
    destSample = destSample + 1
   #for srcSample in range(17408, 26726):   # Where the word "People" is in the sound
   #setSampleValueAt(dest, destSample, getSampleValueAt( source,  srcSample))
    #destSample = destSample + 1
  #for index in range(1, 1000):                    #Stick some quiet space after that
    #setSampleValueAt(dest,  destSample, 0)
    #destSample = destSample + 1
  play(dest)                                       #Let's hear and return the result
  return dest
```

Changing the splice

- What if we wanted to increase or decrease the volume of an inserted word?
  - Simple! Multiply each sample by something as it's pulled from the source.
- Could we do something like slowly increase volume (emphasis) or normalize the sound?
  - Sure! Just like we've done in past programs, but instead of working across *all* samples, we work across only the samples in that sound!

Making more complex sounds

- We know that natural sounds are often the combination of multiple sounds.
- Adding waves in physics or math is hard.
- In computer science, it's easy! Simply add the samples at the same index in the two waves:

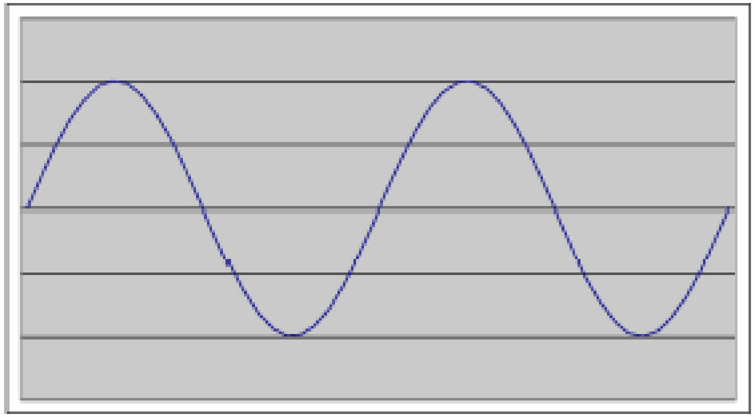
```
for srcSample in range(1, getLength(source)+1):
  destValue=getSampleValueAt(dest, srcSample)
  srcValue=getSampleValueAt(source,srcSample)
  setSampleValueAt(source, srcSample, srcValue+destValue)
```

Adding sounds

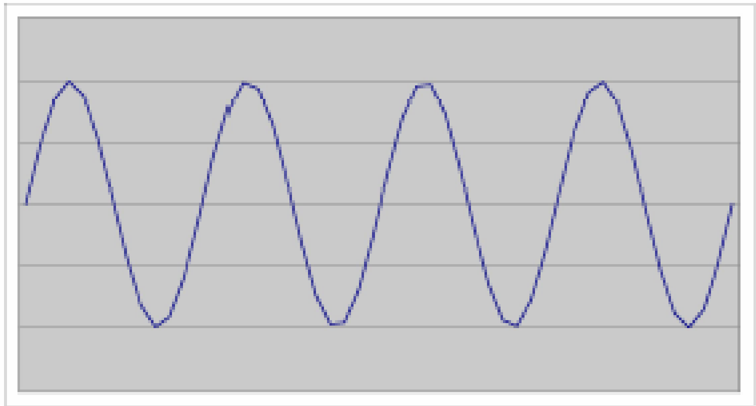
The first two are sine waves generated in Excel.

The third is just the sum of the first two columns.

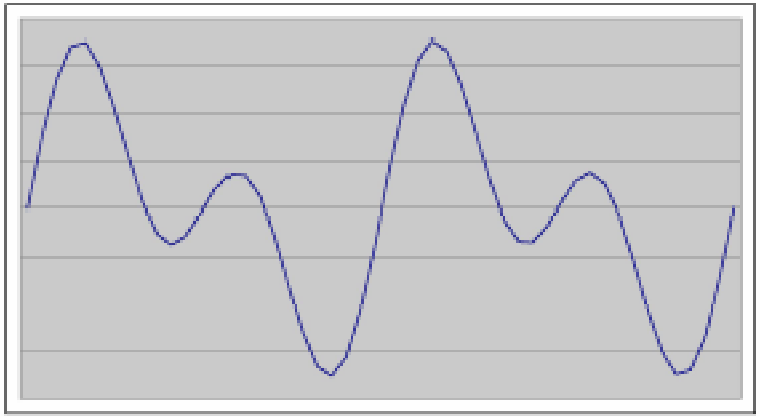
a



b



a + b = c



Uses for adding sounds

- We can mix sounds
  - We even know how to change the volumes of the two sounds, even over time (e.g., fading in or fading out)
- We can create echoes
- We can add sine (or other) waves together to create kinds of instruments/sounds that do not physically exist, but which sound interesting and complex

A function for adding two sounds

```
def addSoundInto(sound1, sound2):

  for sampleNmr in range(1, getLength(sound1)+1):
    sample1 = getSampleValueAt(sound1, sampleNmr)
    sample2 = getSampleValueAt(sound2, sampleNmr)
    setSampleValueAt(sound2, sampleNmr, sample1 + sample2)
```

Notice that this adds sound1 and sound by adding sound1 *into* sound2

Making a chord by mixing three notes

```
>>> setMediaFolder()
New media folder: C:\Documents and Settings\Mark Guzdial\My
Documents\mediasources\
>>> getMediaPath("bassoon-c4.wav")
'C:\\Documents and Settings\\Mark Guzdial\\My
Documents\\mediasources\\bassoon-c4.wav'
>>> c4=makeSound(getMediaPath("bassoon-c4.wav"))
>>> e4=makeSound(getMediaPath("bassoon-e4.wav"))
>>> g4=makeSound(getMediaPath("bassoon-g4.wav"))
>>> addSoundInto(e4,c4)
>>> play(c4)
>>> addSoundInto(g4,c4)
>>> play(c4)
```

Adding sounds with a delay

```
def makeChord(sound1, sound2, sound3):
  for index in range(1, getLength(sound1)):
    s1Sample = getSampleValueAt(sound1, index)
    if index > 1000:
      s2Sample = getSampleValueAt(sound2, index - 1000)
      setSampleValueAt(sound1, index, s1Sample + s2Sample)
    if index > 2000:
      s3Sample = getSampleValueAt(sound3, index - 2000)
      setSampleValueAt(sound1, index, s1Sample + s2Sample + s3Sample)
```

-Add in sound2 after 1000 samples

-Add in sound3 after 2000 samples

Note that in this version we're adding into sound1!

Homework Assignment!

- Option #1: Create an *audio* collage where the same sound is spliced in at least three times:
  - Once in its original form
  - Then with *any* modification you want to make to it
    - Reverse, scale up or down.
- Option #2: Make music (it's up to you what you do!)
  - Look in MusicSounds folder in MediaSources
    - Several instruments, different notes
  - Shift frequencies to get new tones
  - Crop to get shorter notes
- We'll spend an hour on this.
  - Save pictures with writeSoundTo(sound,filename)
  - Share them at http://home.cc.gatech.edu/gacomputes
    - Key: "workshop"