Picture Color Manipulation

Using 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=r"c:\mediasources/katie.jpg"
>>> picture=makePicture(file)
>>> show(picture)
>>> decreaseRed(picture)
>>> repaint(picture)
```

Once we make it work for one picture, it will work for any picture

```
>>> file=pickAFile()
>>> pic=makePicture(file)
>>> decreaseRed(pic)
>>> show(pic)

Can repeat:

>>> decreaseRed(pic)
>>> repaint(pic)
```

Think about what we just did

- Did we change the program at all?
- Did it work for both of the examples?
- What was the input variable **picture** each time, then?
  - It was the value of whatever picture we provided as input!

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

Increasing Red

```
def increaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*1.3)
```

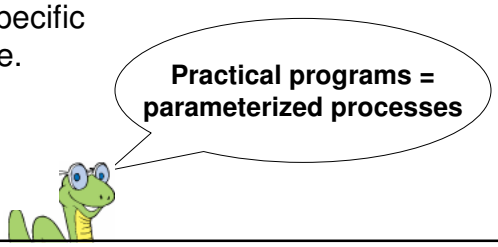
What happened here?!?

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 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 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.
    - That can't be how it really works
  - 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)

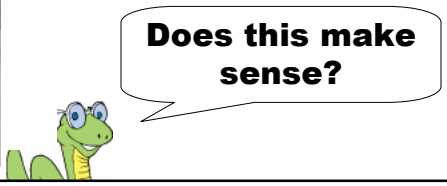
Converting to grayscale

- We know that if red=green=blue, we get grey
  - 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 lots of ways of getting it, 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):
    intensity = (getRed(p) + getGreen(p) + getBlue(p)) / 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 greyscale

- We'll *weight* 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))
```

Comparing the two grayscales: Average on left, weighted on right

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
  - Use the MediaTools to find the RGB values for the brown of Barbara's hair
  - 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 figured out from MediaTools
- The file is where the picture of Barbara is on the computer
- We 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)
  return(picture)
```

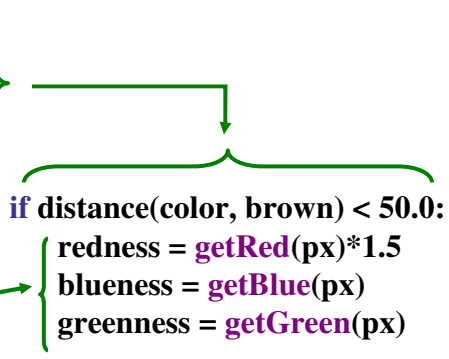
Walking through the for loop

- 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%

```
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)
```

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.

Expressions

- Can use and and or inside the expression we are testing in the if statement

```
red = getRed(px)
if (distance(color, brown) < 50.0) and (red > 200):
    redness = red * 1.5
    setRed(px, redness)
```

```
red = getRed(px)
if (distance(color, brown) < 50.0) or (red > 200):
    redness = red * 1.5
    setRed(px, redness)
```

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

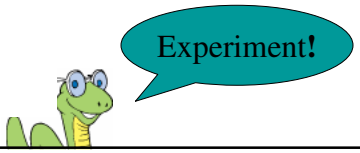
```
if distance(color, brown) < 50.0:
    redness = getRed(px)*1.5
    setRed(px, redness)
show(picture)
return(picture)
```

Things to change

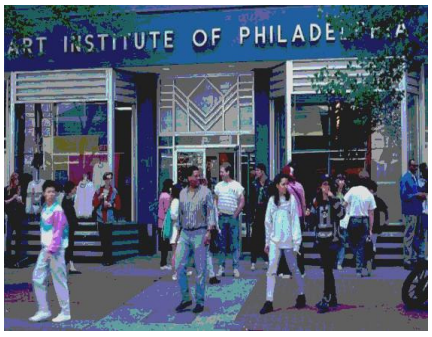
- Lower the threshold to get more pixels
  - But if it's too low, you start messing with the wood behind her
- Increase the amount of redness
  - But if you go too high, you can go beyond the range of valid color intensities (i.e. more than 255)

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 parts, it 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 brownish tint to them that we associate with older photographs.
- It's not just a matter of increasing the amount of brown 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 conversions using if

Example of sepia-toned prints

Here's how we do it

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

What's going on here?

- First, we're calling **grayScaleNew** (the one with weights).
  - It's perfectly okay to have one function calling another.
- We then manipulate the red (increasing) and the blue (decreasing) channels to bring out more yellows and oranges.
  - Why are we doing the comparisons on the red?
  - Why *not*? After grayscale conversion, all channels are the same!
  - Why these values?
  - Trial-and-error: Twiddling the values until it looks the way that you want

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 parameters uses ()**
  - **Making a real function with an output uses return**
- Using functions to create programs (recipes) and executing them