Color replacements and targeted color replacement (`if`)

- Conditions: if ... then ...
- Expressions
- The use of: return
- Changing only some colors
- Posterising
- Commenting programs
- Sepia toned pictures

- 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

```
def turnRed():
  brown = makeColor(57,16,8)
  file = r"F:\comp2720\pics\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)
```

Original:

Talking through the program slowly

- Why aren't we taking any input? We don't want any: The 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"F:\comp2720\pics\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 latter half of the program

- 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%
- At the end, show and return the picture

```
def turnRed():
    brown = makeColor(57,16,8)
    file = r"F:\comp2720\pics\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

- We can test equality with ==
- We 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.

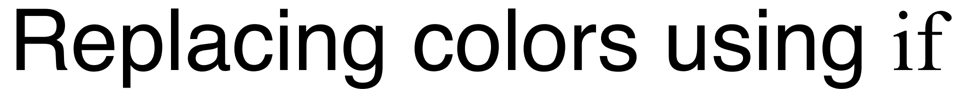
Note: Using return here!

- 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.

```
>>> redBarbara = turnRed()
```

Things to change

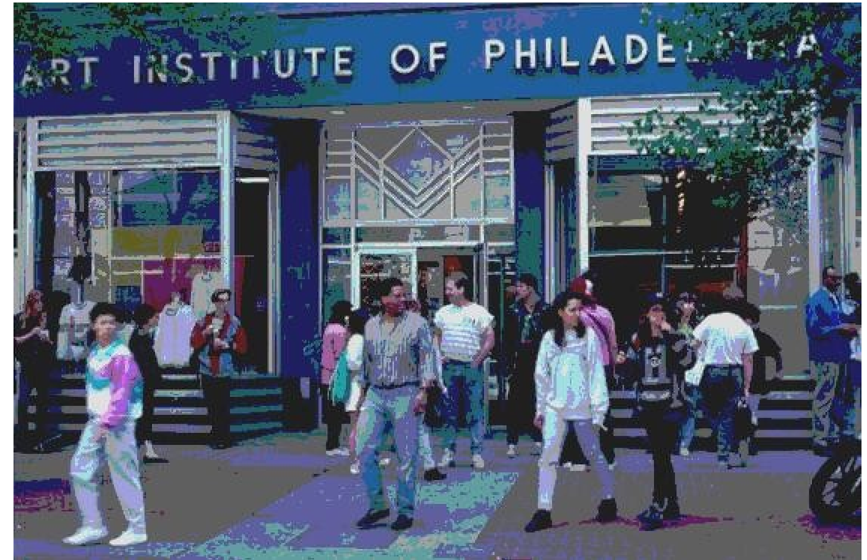
- Lower the threshold to get more pixels
    - But if it’s too low, you start messing with the wood behind her
- Increase the amount of redness
    - But if you go too high, you can get beyond the range



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.

Posterising: Reducing the range of colors



Posterising: 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
    - ...
- It requires a lot of if statements, but it’s really pretty simple.
- The end result is that a bunch of different colors gets set to a few colors.

Posterising function

```
def posterise(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)
```

- Any line that starts with a “#” is ignored by Python.
- This allows you to insert comments: Notes to yourself (or another programmer) that explains what’s going on here.
    - When programs get longer, there are lots of pieces to them, and it’s hard to figure out what each piece does.
    - Comments can help to understand a program.

Generating sepia-toned prints

- Pictures that are *sepia-toned* have a yellowish tint to them that we associate with older pictures.
- It’s not directly a matter of simply increasing the yellow in the picture, because it’s not a one-to-one correspondence.
  - Instead, colors in different ranges get mapped to other colors.
  - We can create such a mapping using `if`.

Example of sepia-toned prints

Here's how we do it

```
def sepiaTint(picture):
  # Convert image to greyscale
  greyScaleNew(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   # Why this here.. ?
      blue = blue*0.93

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

What's going on here?

- First, we're calling greyScaleNew (lecture Pics-3, the function 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 greyscale 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 inputs uses (...)
    - Making a real function with outputs uses return ...
- Using functions to create programs (recipes) and executing them

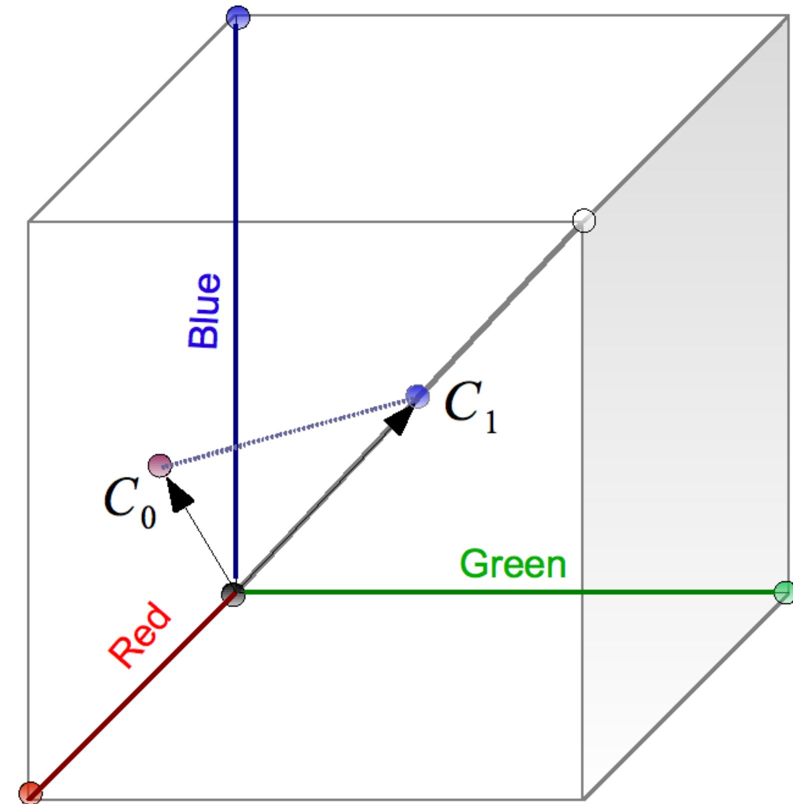
Colour interpolation

- What if we need to gradually convert one colour value into another colour value to create an effect of continuous change?
- Say, we want to make a picture fade its colours and become black-and-white one, or the other way around? (Like a “flashback from the past” in movies, when no colour film was available, and life “was” only coloured grey.)
- But we also want to preserve the luminance of every pixel.

Colour Cube

- The colour $\vec{C}_0$ is the original
- The color $\vec{C}_1$ is the same luminosity black-and-white
- We need to move gradually along the line $\overrightarrow{C_0 C_1}$ to convert the colour $\vec{C}_0$ into colour $\vec{C}_1$
- If you know a little of geometry, this is easy:

$$\vec{C}(t)=\vec{C}_0\cdot(1-t)+\vec{C}_1\cdot t\,,\quad t\in[0,\,1]$$



C_0 and C_1 have the same luminance. Any colour lying on the line C_0C_1 also have the same luminance. How to find C_0C_1?

Colour interpolation, the formula

- Initial colour $\vec{C}_0 = \vec{R} \cdot r_0 + \vec{G} \cdot g_0 + \vec{B} \cdot b_0$
- Final (grey) colour $\vec{C}_1 = \vec{R} \cdot \gamma + \vec{G} \cdot \gamma + \vec{B} \cdot \gamma \;,\; \gamma = \frac{(r_0 + g_0 + b_0)}{3}$
- interpolating colour $\vec{C}(t) = \vec{C}_0 \cdot (1 - t) + \vec{C}_1 \cdot t$
- interpolating colour in terms of primaries

$$\vec{C}(t) = \vec{R} \cdot r(t) + \vec{G} \cdot g(t) + \vec{B} \cdot b(t)$$

- interpolating primary components

$$r(t) = r_0 + \frac{g_0 + b_0 - 2 \cdot r_0}{3} \cdot t \,,\; (r, g, b) \rightarrow (g, b, r) \rightarrow (b, g, r)$$

Colour interpolation, the program

- That is what you will have to do in the lab 2 exercise 2.
- Create two functions:
  - `colourFade(picture, frac)`
    it will convert a colour picture into partially “black-and-whited”, with `frac=0` corresponding to the unchanged full coloured pictured, and `frac=1` – to the fully black-and-white
  - `fading(picture)`
    it will loop (say, ten times) over the value of `frac` from 0 to 1, and repaint the picture at every step, creating the effect of continuous colour fading.