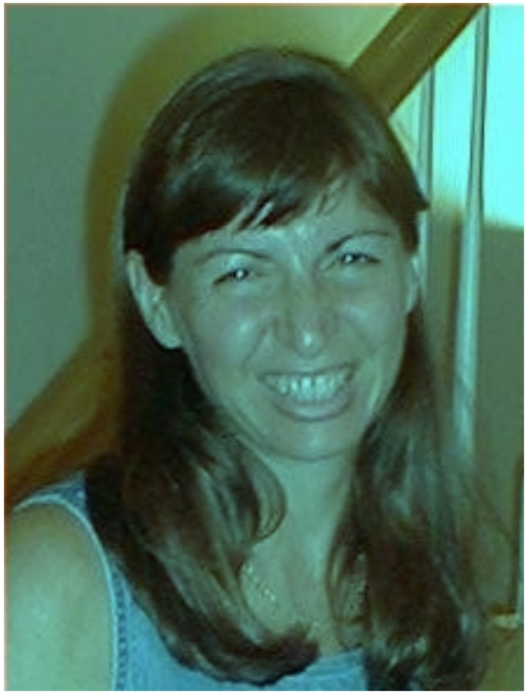
Chapter 4: Modifying Pictures using Loops (continued)

Looping through ALL the pixels

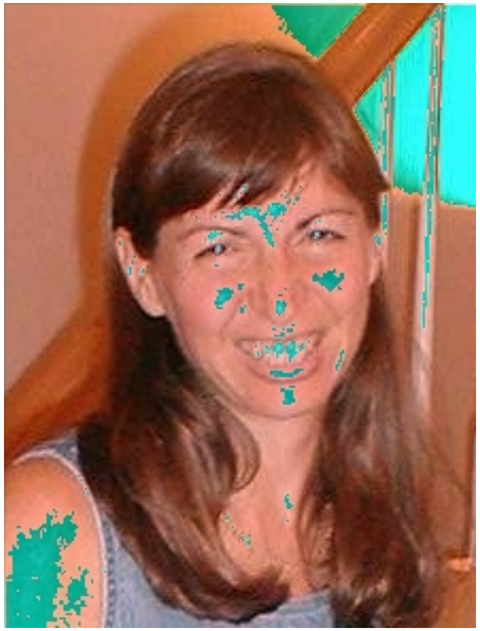
```
def decreaseRed(picture):
  pixels = getPixels(picture)
  for pixel in pixels:
    value=getRed(pixel)
    setRed(pixel,value*0.5)
```



```
Used like this:
>>> file=pickAFile()
>>> picture=makePicture(file)
>>> explore(picture)
>>> decreaseRed(picture)
>>> explore(picture)
```

Increasing Red

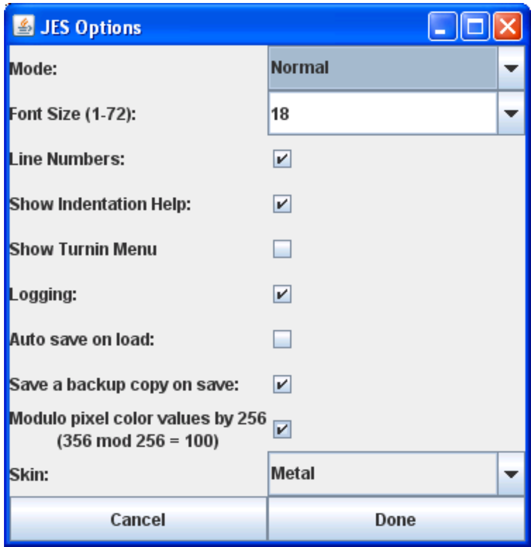
```
def increaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*1.2)
```



What happened here?!?

Remember that the limit for redness is 255.

If you go *beyond* 255, all kinds of weird things can happen if you have "Modulo" checked in Options.



Mode: Normal

Font Size (1-72): 18

Line Numbers: ☑

Show Indentation Help: ☑

Show Turnin Menu ☐

Logging: ☑

Auto save on load: ☐

Save a backup copy on save: ☑

Modulo pixel color values by 256 (356 mod 256 = 100) ☑

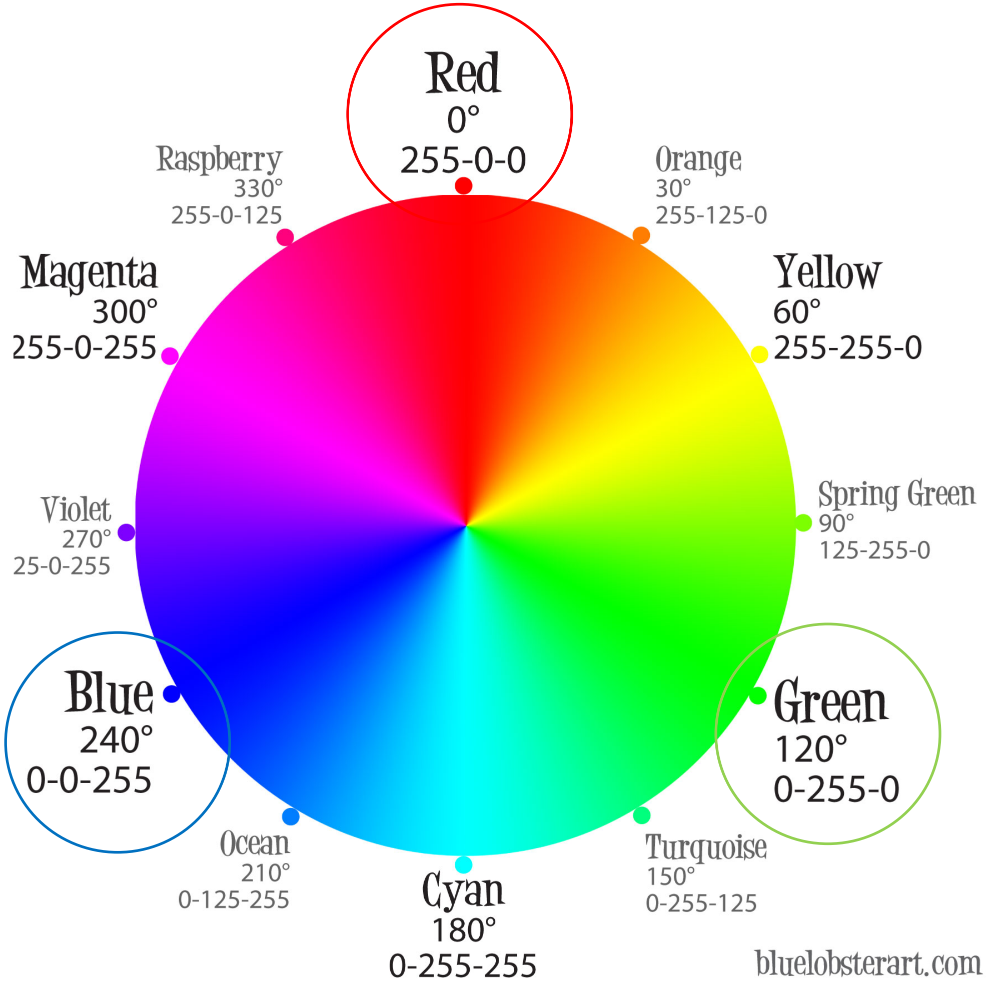
Skin: Metal

Colors and math

- RGB colors are a mix of red, green, blue VALUES (numbers!)
- Change the values, change the color

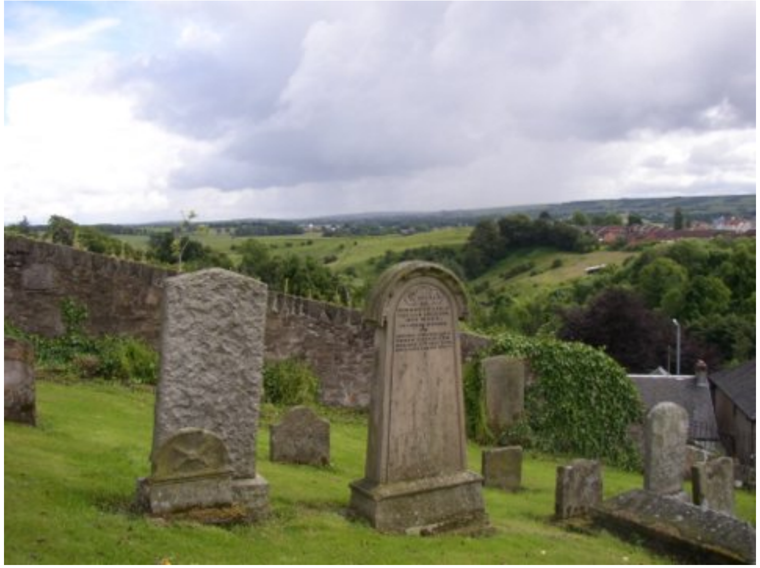
You can predict what something will look like

Remember: white = rgb(255,255,255)



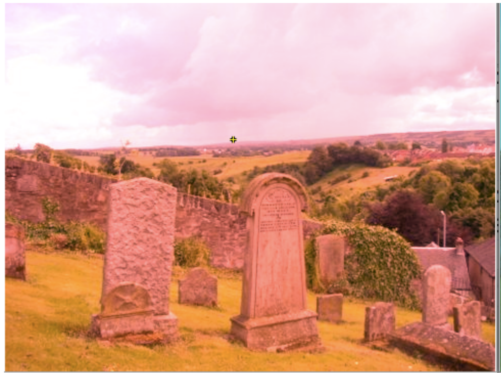
Which one of the below pictures was generated from this function:

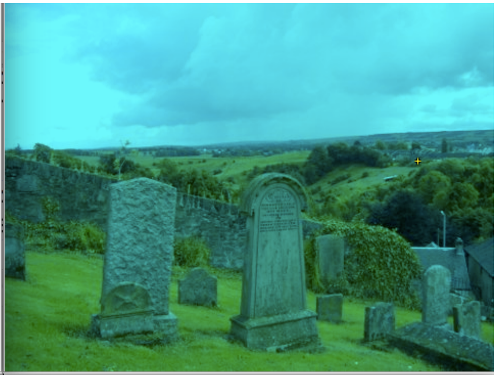
```
def change2(picture):
  for pixel in getPixels(picture):
    setBlue(pixel,0)
```



Refer to the color wheel!

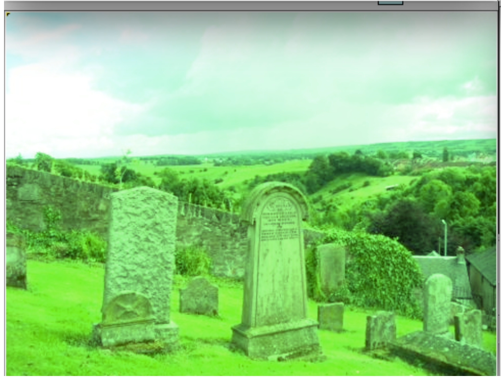
1

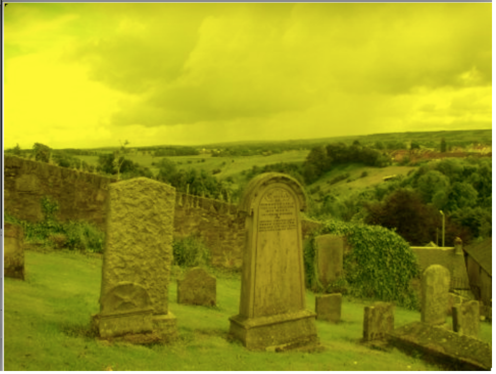




2

3





4

Recipe for 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)
```

Converting to greyscale

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

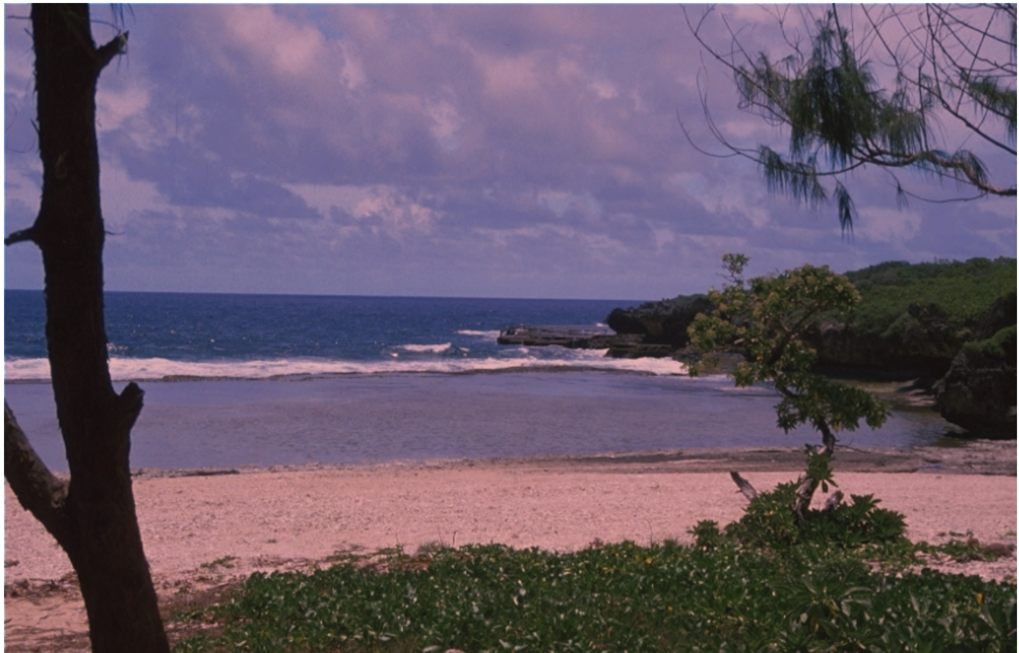
Building a better greyscale

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

```
def greyScaleNew(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))
```

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)
```



Building Better Functions

"Hierarchical decomposition"

Hierarchical decomposition

```
def makeSunset2(picture):
  reduceBlue(picture)
  reduceGreen(picture)

def reduceBlue(picture):
  for p in getPixels(picture):
    value=getBlue(p)
    setBlue(p,value *0.7)

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

- This one does the *same* thing as the earlier form.
- It's easier to read and understand: "To make a sunset is to reduceBlue and reduceGreen."
- We use *hierarchical decomposition* to break down the problem.
- This version is less inefficient, but that's okay. **It's easier for us (humans) to understand.**

Scope

- A function is its own context.
  - Input variables (placeholders) take on the value of the input values only for the life of the function
  - Only while it's executing
  - Variables defined within a function also only exist within the context of that function
  - The context of a function is also called its scope

Variables within functions *stay* within functions

- The variable **value** in **decreaseRed** is created *within* the scope of **decreaseRed**
  - That means that it only exists while decreseRed is executing

- If we tried to *print value* after running decreaseRed, it would work *ONLY* if we already had a variable defined in the Command Area
  - The name *value* within *decreaseRed* doesn't exist outside of that function
  - We call that a ***local*** variable

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

Consider these two functions

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

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

- First, it's perfectly okay to have *multiple* inputs to a function.
- The new decreaseRed now takes an input of the multiplier for the red value.
  - decreaseRed(picture,0.5) would do the same thing
  - decreaseRed(picture,1.25) would *increase* red 25%

Specifying Pixels by Index

Using square bracket notation ("[ ]") to jump to a set of pixels (instead of cycling through all the pixels all the time)

Another version of decreaseRed()

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

```
def decreaseRed2(picture):
  pixels = getPixels(picture)
  for index in range(0,len(pixels)):
    pixel = pixels[index]
    value=getRed(pixel)
    setRed(p,value*0.5)
```

Just ½ of picture decreaseRed

```
def decreaseRedHalf(picture):
  pixels = getPixels(picture)
  for index in range(0,len(pixels)/2):
    pixel = pixels[index]
    value = getRed(pixel)
    setRed(pixel, value * 0.5)
```