Introduction to Computing and Programming in Python:
A Multimedia Approach

Chapter 3: Modifying Pictures using Loops

Chapter Learning Objectives

The media learning goals for this chapter are:

- To understand how images are digitized by taking advantage of limits in human vision.
- To identify different models for color, including RGB, the most common one for computers.
- To manipulate color values in pictures, like increasing or decreasing red values.
- To convert a color picture to grayscale, using more than one method.
- To negate a picture.

The computer science goals for this chapter are:

- To use a matrix representation in finding pixels in a picture.
- To use the objects *pictures* and *pixels*.
- To use iteration (with a `for` loop) for changing the color values of pixels in a picture.
- To nest blocks of code within one another.
- To choose between having a function *return* a value and just providing a *side effect*.
- To determine the *scope* of a variable name.

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.

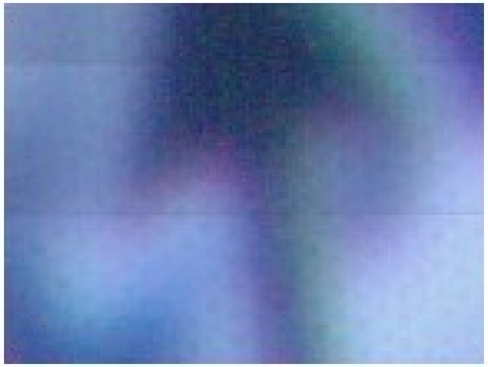
Luminance vs. Color

- 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 does luminance vs. color.

Digitizing pictures as bunches of little dots

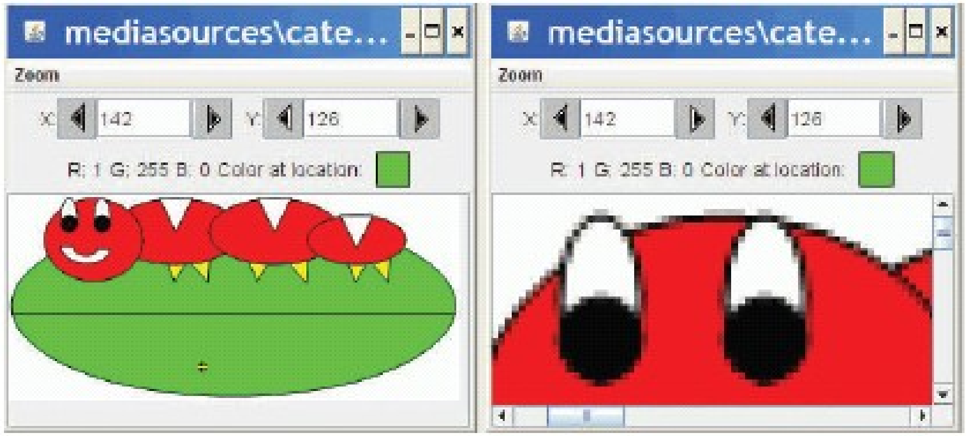
- We digitize pictures into lots of little dots
- Enough dots and it looks like a continuous whole to our eye
  - Our eye has limited resolution
  - Our background/depth *acuity* is particularly low
- Each picture element is referred to as a *pixel*



Pixels

- Pixels are *picture elements*
  - Each pixel object knows its *color*
  - It also knows where it is in its *picture*

```
>>> file = "c:/ip-book/mediasources/caterpillar.jpg"
>>> pict = makePicture(file)
>>> explore(pict)
```



When we zoom the picture to 500%, we can see individual pixels.

A Picture is a *matrix* of pixels

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

0	1	2	3
15	12	13	10

	0	1	2	3
0	15	12	13	10
1	9	7	2	1
2	6	3	9	10

Referencing a matrix

	0	1	2	3
0	15	12	13	10
1	9	7	2	1
2	6	3	9	10

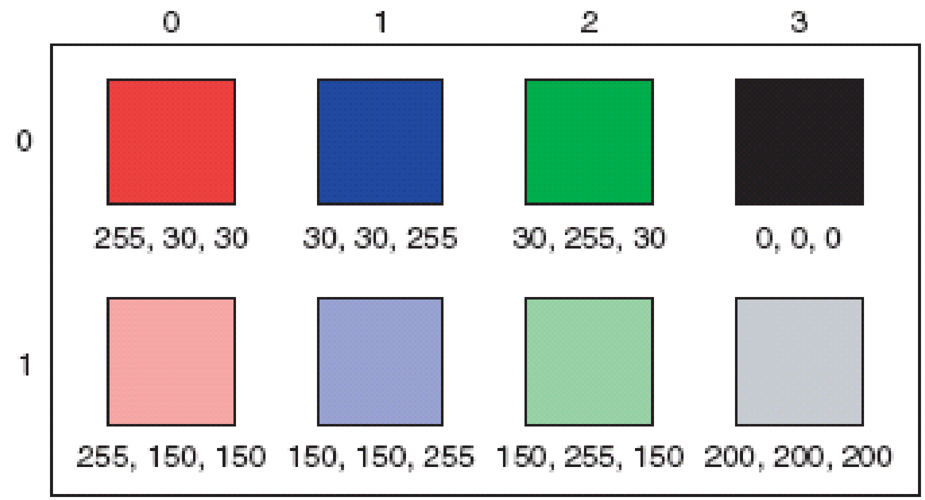
- We talk about positions in a matrix as (x,y), or (horizontal, vertical)
- Element (1,0) in the matrix at left is the value 12
- Element (0,2) is 6

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 Value)
- We'll use the most common for computers
  - RGB: Red, Green, Blue

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
    - (200,200,200) at (3,1)
  - (0,0,0) (at position (3,0) in example) is black
  - (255,255,255) is white



How much can we encode in 8 bits?

- Let's walk it through.
  - If we have one bit, we can represent two patterns: 0 and 1.
  - If we have two bits, we can represent four patterns: 00, 01, 10, and 11.
  - If we have three bits, we can represent eight patterns: 000, 001, 010, 011, 100, 101, 110, 111
- General rule: In *n* bits, we can have 2^n patterns
  - In 8 bits, we can have 2^8 patterns, or 256
  - If we make one pattern 0, then the highest value we can represent is 2^8-1, or 255

Is that enough?

- We're representing color in 24 (3 * 8) bits.
  - That's 16,777,216 (2^{24}) possible colors
  - Our eye can discern millions of colors, so it's probably pretty close
  - But the real limitation is the physical devices: We don't get 16 million colors out of a monitor
- Some graphics systems support 32 bits per pixel
  - May be more pixels for color, or an additional 8 bits to represent 256 levels of *translucence*

Size of images

	320 x 240 image	**640 x 480 image**	**1024 x 768 image**
24 bit color	230,400 bytes	921,600 bytes	2,359,296 bytes
32 bit color	307,200 bytes	1,228,800 bytes	3,145,728 bytes

Reminder: Manipulating Pictures

```
>>> file=pickAFile()
>>> print file
>>> picture=makePicture(file)
>>> print picture
```

This will show the height so you can figure out how big your picture object is (in terms for space).

What's a "picture"?

- An encoding that represents an image
  - Knows its height and width
  - Knows its filename
  - Knows its *window* if it's opened (via *show* and repainted with *repaint*)

Manipulating pixels

getPixel(picture,x,y) gets a single pixel.

getPixels(picture) gets *all* of them in an array. (Square brackets is a standard array reference notation—which we'll generally *not* use.)

```
>>> pixel=getPixel(picture,1,1)
>>> print pixel
Pixel, color=color r=168 g=131 b=105
>>> pixels=getPixels(picture)
>>> print pixels[0]
Pixel, color=color r=168 g=131 b=105
```

What can we do with a pixel?

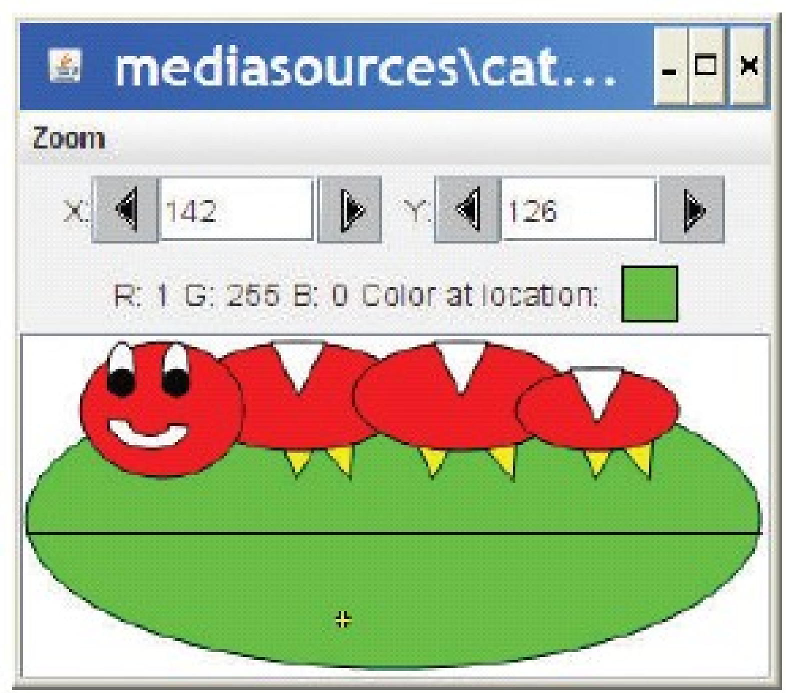
- getRed, getGreen, and getBlue are functions that take a pixel as input and return a value between 0 and 255
- setRed, setGreen, and setBlue are functions that take a pixel as input *and* a value between 0 and 255

We can also get, set, and make Colors

- getColor takes a pixel as input and returns a Color object with the color at that pixel
- setColor takes a pixel as input *and* a Color, then sets the pixel to that color
- makeColor takes red, green, and blue values (in that order) between 0 and 255, and returns a Color object
- pickAColor lets you use a color chooser and returns the chosen color
- We also have functions that can makeLighter and makeDarker an input color

How do you find out what RGB values you have? And where?

- Use the MediaTools!



The MediaTools menu knows what variables you have in the Command Area that contain pictures

Distance between colors?

- Sometimes you need to, e.g., when deciding if something is a "close enough" match
- How do we measure distance?
  - Pretend it's cartesian coordinate system
  - Distance between two points:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

$$\sqrt{(red_1 - red_2)^2 + (green_1 - green_2)^2 + (blue_1 - blue_2)^2}$$

Demonstrating: Manipulating Colors

```
>>> print getRed(pixel)
168
>>> setRed(pixel,255)
>>> print getRed(pixel)
255
>>> color=getColor(pixel)
>>> print color
color r=255 g=131 b=105
>>> setColor(pixel,color)
>>> newColor=makeColor(0,100,0)
>>> print newColor
color r=0 g=100 b=0
>>> setColor(pixel,newColor)
>>> print getColor(pixel)
color r=0 g=100 b=0
```

```
>>> print color
color r=81 g=63 b=51
>>> print newcolor
color r=255 g=51 b=51
>>> print distance(color,newcolor)
174.41330224498358
>>> print color
color r=168 g=131 b=105
>>> print makeDarker(color)
color r=117 g=91 b=73
>>> print color
color r=117 g=91 b=73
>>> newcolor=pickAColor()
>>> print newcolor
color r=255 g=51 b=51
```

Manipulating Pixels

- This is best seen in JES
- The point is we can manipulate individual pixels to change their colour.
- How? By selecting a pixel from an image and editing its color values!

Use a loop!
Our first picture recipe

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

Our first picture recipe works for *any* picture

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

Used like this:

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

How do you make an omelet?

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

All useful recipes involve repetition

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

- Beat the eggs <u>until</u>…

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

Decreasing the red in a picture

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

Use a for loop!
Our first picture recipe

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

The loop

- Note the indentation!

How for loops are written

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

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

What happens when a for loop is executed

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

getPixels returns a sequence of pixels

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

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

or equivalently…

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

Do we need the variable *originalRed*?

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

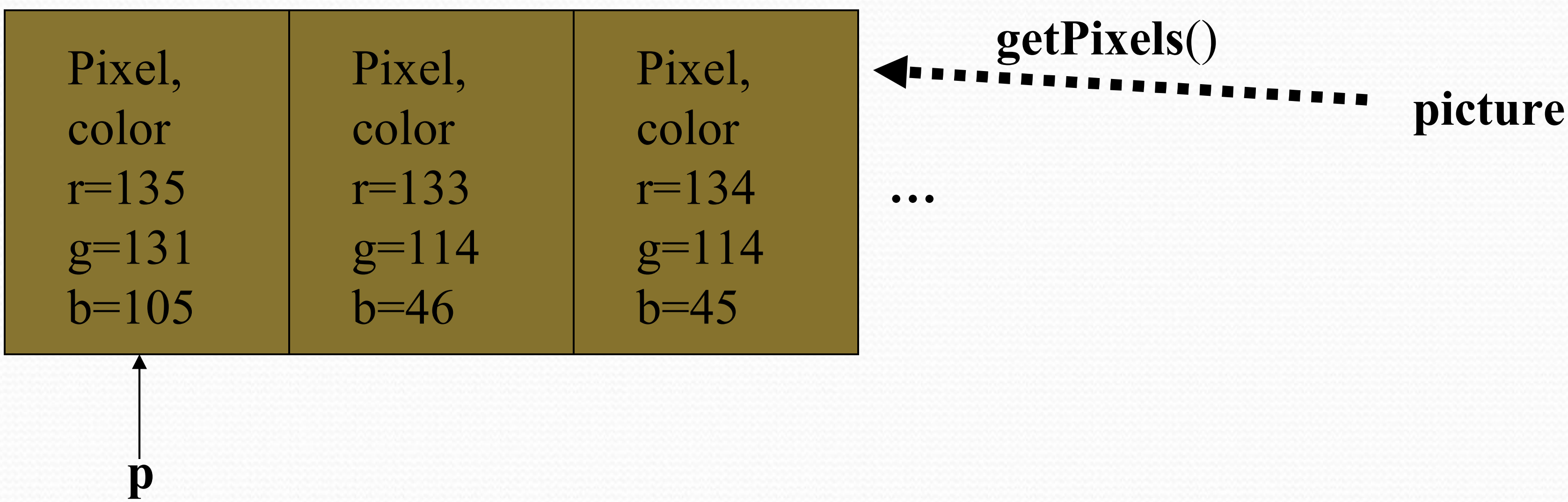
```
def decreaseRed(picture):
  for p in getPixels(picture):
    originalRed = getRed(p)
    setRed(p, originalRed * 0.5)
```

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

Now, get the pixels

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

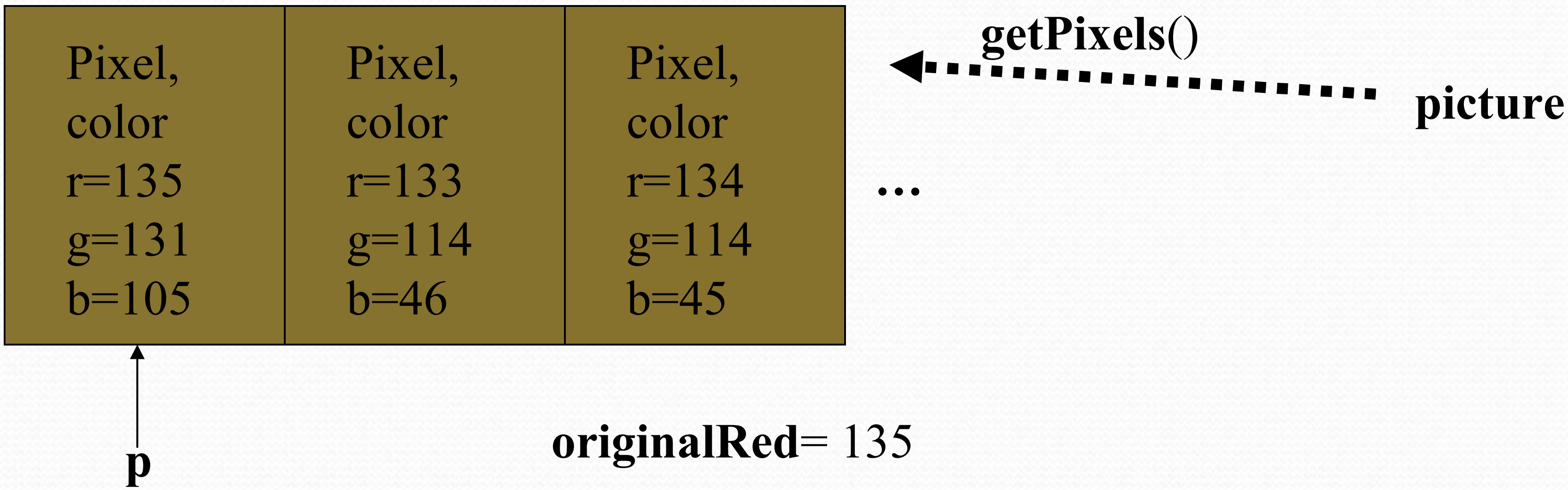
We get all the pixels from the **picture**, then make **p** be the name of each one *one at a time*



Get the red value from pixel

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

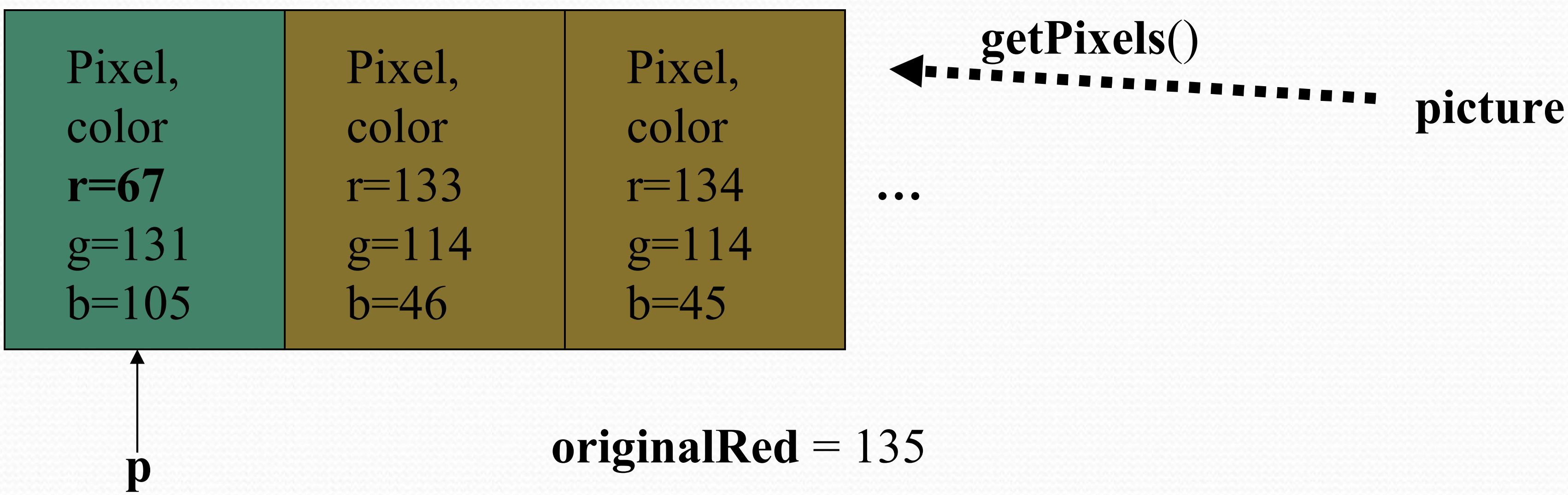
We get the red value of pixel **p** and name it **originalRed**



Now change the pixel

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

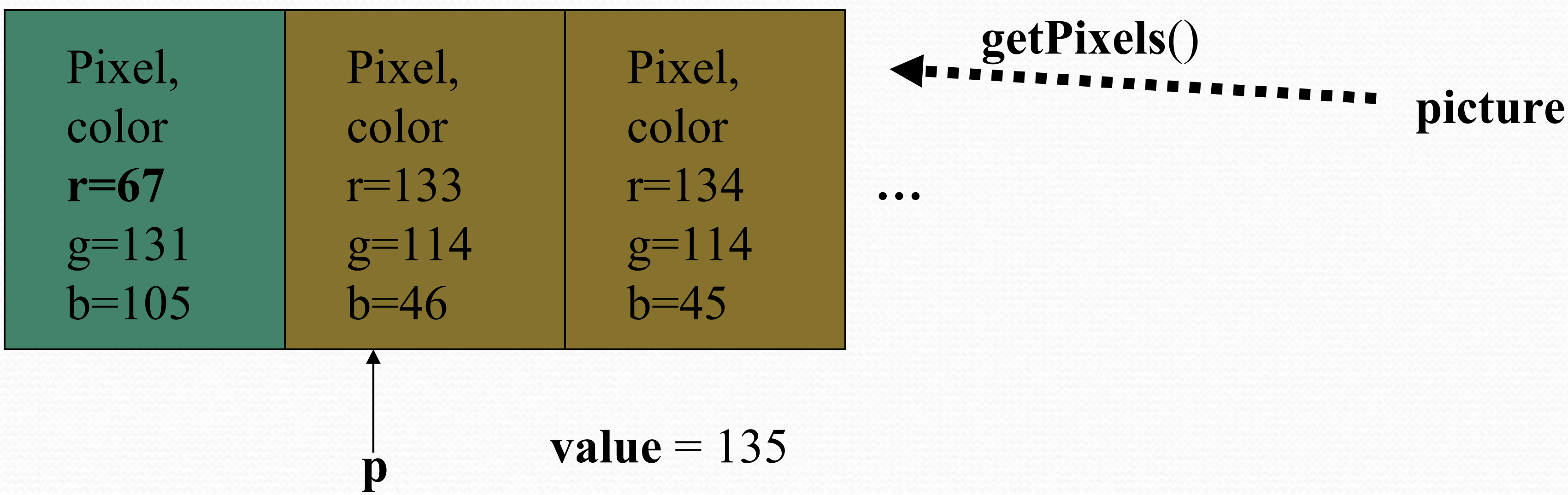
Set the red value of pixel **p** to 0.5 (50%) of **originalRed**



Then move on to the next pixel

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

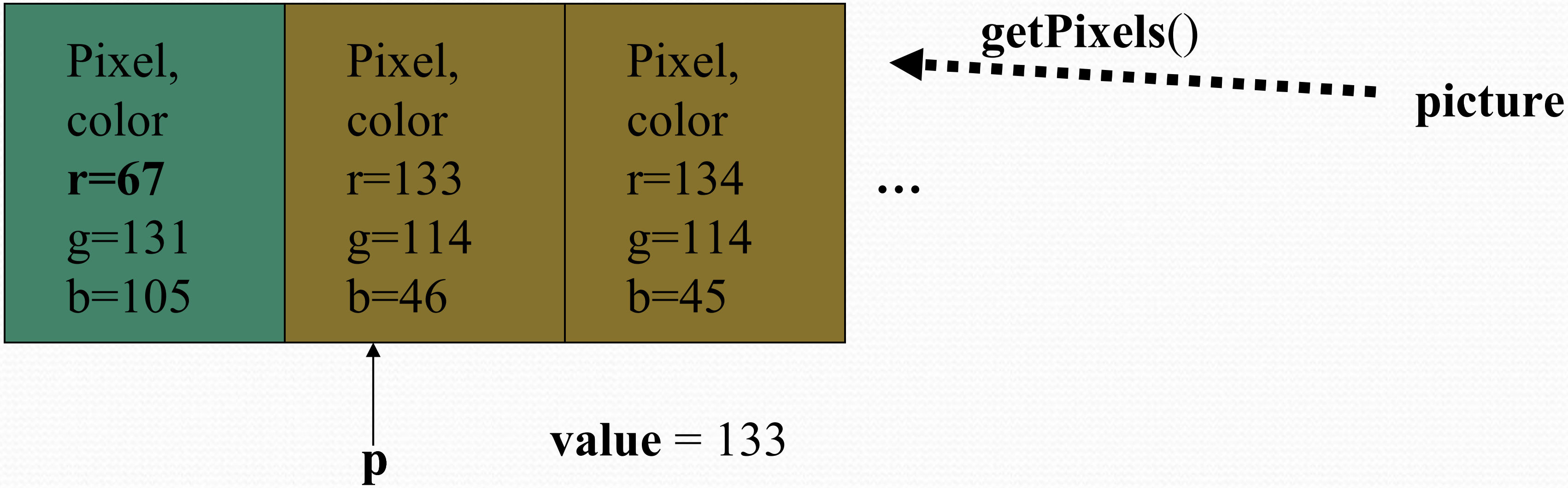
Move on to the next pixel and name *it* **p**



Get its red value

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

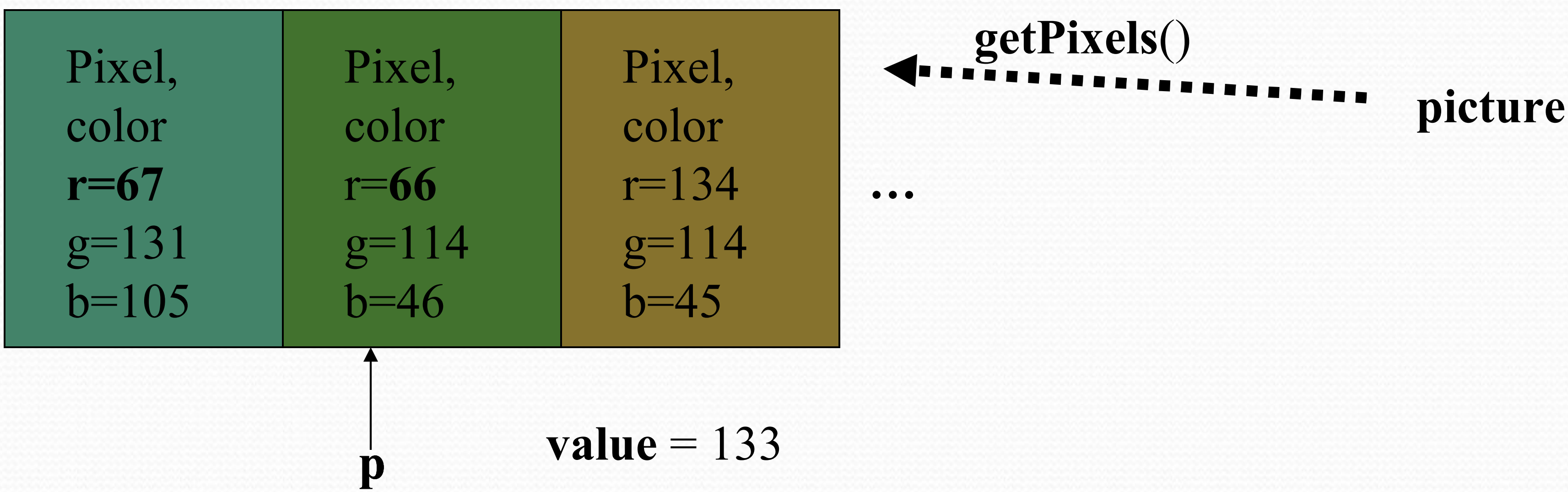
Set **originalRed** to the red value at the new **p**, then change the red at that new pixel.



And change *this* red value

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

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



And eventually, we do all pixels

- You can see the difference in this demo!

“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.

Making it work for all pictures!

- Do we change the program at all?
- Does it work for different 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)
```

NOTE: If you have a variable *picture* in your Command Area, that's *not the same* as the *picture* in *decreaseRed*.

Read it as a Recipe

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

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

Let’s use something with known red to manipulate: Santa Claus

What if you decrease Santa's red again and again and again…?

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

(That's the first one)

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

(That's the second)

If you make something you like…

- `writePictureTo(picture,"filename")`
- Writes the picture out as a JPEG
- Be sure to end your filename as ".jpg"!
- If you don't specify a full path,
  will be saved in the same directory as JES.

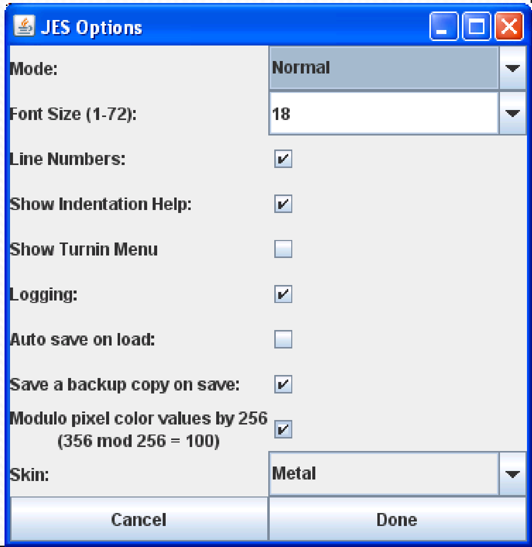
Increasing Red

```
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.



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 a specific 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.

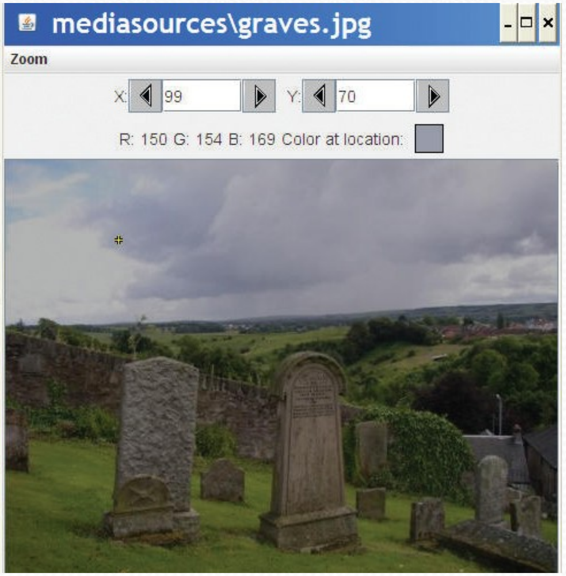
A Sunset-generation Function

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

Lightening and darkening an image

```
def lighten(picture):
 for px in getPixels(picture):
  color = getColor(px)
  color = makeLighter(color)
  setColor(px ,color)
```

```
def darken(picture):
 for px in getPixels(picture):
  color = getColor(px)
  color = makeDarker(color)
  setColor(px ,color)
```



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.

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)
```

Original, negative, negative-negative

Converting to greyscale

- 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 greyscale

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

Can we get back again?
Nope

- 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.

But that’s not really the *best* greyscale

- 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 greyscale 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 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))
```

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

Let’s use a black cat to compare

Average on left, weighted on right

If you make something you like…

- `writePictureTo(picture,"C:/filename.jpg")`
- Writes the picture out as a JPEG
- Be sure to end your filename as ".jpg"!

A different sunset-generation function

```
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.
- ***Programs are written for people, not computers.***

Let's talk about functions

- How can we reuse variable names like **picture** in both a function and in the Command Area?
- Why do we write the functions like this? Would other ways be just as good?
- Is there such a thing as a better or worse function?
- Why don't we just build in calls to **pickAFile** and **makePicture?**

One and only one thing

- We write functions as we do to make them *general* and *reusable*
  - Programmers hate to have to re-write something they've written before
  - They write functions in a general way so that they can be used in many circumstances.
- What makes a function *general* and thus *reusable*?
  - A reusable function does *One and Only One Thing*

Contrast these two programs

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

```
def makeSunset(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)
```

Yes, they do the exact same thing!

makeSunset(somepict) works the same in both cases

Observations on the new makeSunset

- It's okay to have more than one function in the same Program Area (and file)
- makeSunset in this one is somewhat easier to read.
  - It's clear what it does "reduceBlue" and "reduceGreen"
  - That's important!

```
def makeSunset(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)
```

Programs are written for people, not computers!

Considering variations

- We can only do this because **reduceBlue** and **reduceGreen**, do *one and only one thing*.
- If we put **pickAFile** and **makePicture** in them, we'd have to pick a file twice (better be the same file), make the picture—then save the picture so that the next one could get it!

```
def makeSunset(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)
```

Does makeSunset do *one and only one thing*?

- Yes, but it's a higher-level, *more abstract* thing.
  - It's built on lower-level *one and only one thing*
- We call this *hierarchical decomposition.*
  - You have some *thing* that you want the computer to do?
  - Redefine that *thing* in terms of smaller *things*
  - Repeat until you know how to write the smaller things
  - Then write the larger things in terms of the smaller things.

Are all these *pictures* the same?

- What if we use this like this in the Command Area:

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

```
def makeSunset(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)
```

What happens when we use a function

- When we type in the Command Area

```
makeSunset(picture)
```

- Whatever object that is in the *Command Area* variable **picture** becomes the value of the *placeholder (input) variable* **picture** in

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

- **makeSunset**’s picture is then passed as input to **reduceBlue** and **reduceGreen**, but their input variables are completely different from **makeSunset**’s picture.
  - For the life of the functions, they are the same *values (picture objects*)

Names have contexts

- In natural language, the same word has different meanings depending on *context*.
  - I'm going to fly to Vegas.
  - Would you please swat that fly?
- 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*

Input variables are placeholders

- Think of the input variable as a placeholder
  - It takes the place of the input object
- During the time that the function is executing, the placeholder variable *stands for* the input object.
- When we modify the placeholder by changing its pixels with **setRed**, we actually change the input object.

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)
```

Writing *real* functions

- Functions in the mathematics sense take input and usually return *output*.
  - Like ord() or makePicture()
- What if you create something inside a function that you *do* want to get back to the Command Area?
  - You can **return** it.
  - We'll talk more about return later—that's how functions *output* something

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%

Names are important

- This function should probably be called **changeRed** because that's what it does.
- Is it more general? Yes.
- But is it the one and only one thing that you need done?
  - If not, then it may be less understandable.
  - You can be *too* general

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

Understandability comes first

- Consider these two functions below
- *They do the same thing!*
- The one on the right *looks like* the other increase/decrease functions we've written.
  - That may make it more understandable for you to write first.
  - But later, it doesn't make much sense to you
    - "Why multiply by zero, when the result is always zero?!?"

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

```
def clearBlue(picture):
  for p in getPixels(picture):
    value = getBlue(p)
    setBlue(p,value*0)
```

Always write the program understandable *first*

- Write your functions so that *you* can understand them *first*
  - Get your program *running*
- ***THEN*** make them better
  - Make them more understandable to others
    - Set to zero rather than multiply by zero
    - Another programmer (or you in six months) may not remember or be thinking about increase/decrease functions
  - Make them more efficient
    - The new version of **makeSunset** takes twice as long as the first version, because it changes all the pixels *twice*