

# Python for Image Manipulation

---

## Topics

1. [Exploring Pictures](#)
  - [JES Code for Viewing Pictures and Properties](#)
  - [Exploring Pixels](#)
  - [Exploring Colors](#)
2. ["Negating" an Image](#)
  - [Going through a List of Pixels](#)
  - [Using a Nested For Loop](#)
  - [Using a Function](#)
3. [Changing the Color Values](#)
  - [Lightening](#)
  - [Reducing Red](#)
  - [Applying Grayscale](#)
  - [Blending with White](#)
4. [Copying Pixels](#)
  - [Duplicating an Image](#)
  - [Mirroring an Image](#)
5. [References](#)
6. [Exercise](#)

Please get the code examples and image used in this lab, by clicking [here](#)

---

## 1. Exploring Pictures

In order to understand, how to manipulate pictures, you first need to understand how pictures are represented on a computer. Pictures are two-dimensional arrays of pixels. Have you every zoomed in so close to an image that you see individual boxes? One of those "boxes" represents a **pixel** (short for "picture element"). And that pixel has a **color**. In summary, pictures are made up of (really) tiny "boxes" of color. In the next sections, we will take a closer look at pictures using JES/Python, and then examine pixels and the RGB color model.

### 1.1 JES Code for Viewing Pictures and Properties

You will find that some functions that we use for pictures are not part of Python; they only work inside of JES. Examples of JES specific functions are:

- `pickAFile`, which pops up a file choosing dialog and returns a string for the file chosen
- `makePicture`, which creates a picture object
- `getHeight`, which returns the number of pixels from top-to-bottom
- `getWidth`, which returns the number of pixels from left-to-right
- `getPixels`, which returns a *list* of all the pixels (Pixel objects) in the picture
- `getPixel`, which returns *one* Pixel object at x and y coordinates

If you are not sure which functions are part of Python or which are part of JES, you can look in the JES menu under [Help > Understanding Pictures](#). Click on "[Picture Functions in JES](#)". You might find some cool stuff in there.

The following function is a compilation of all of the JES functions mentioned above.

```
def viewPicture():
    file = pickAFile()
    pict = makePicture(file)
    pixel = getPixel(pict, 0, 0)
    pixels = getPixels(pict)
    print "width is", getWidth(pict), "height is", getHeight(pict)
    print "The value of the pixel at 0, 0 is: ", pixel
    print "A different way to get the pixel at 0, 0 is: ", pixels[0]
    explore(pict)
```

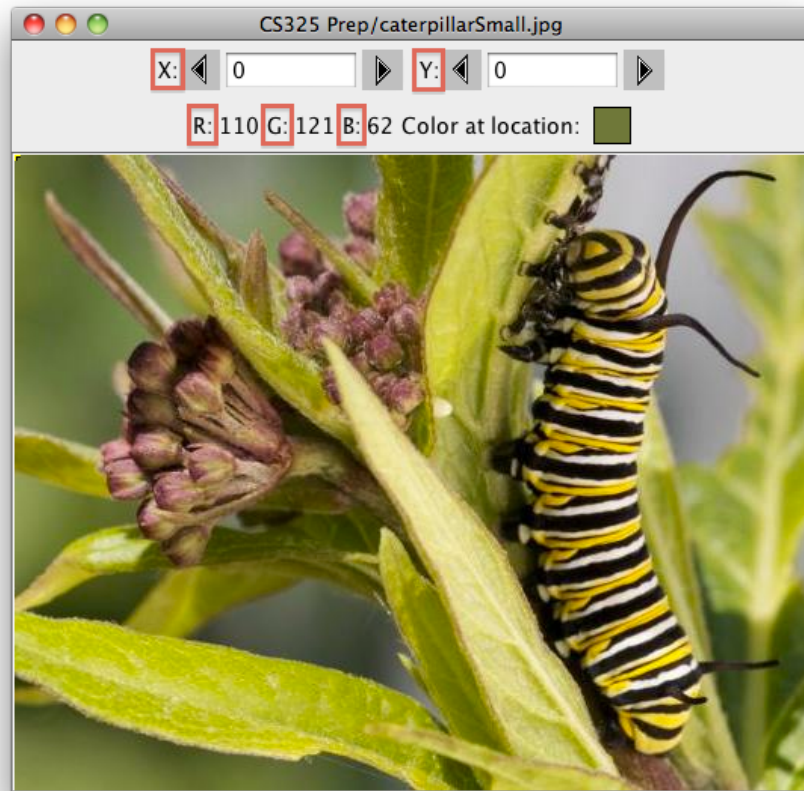
Notice that when you run it and select "`caterpillarSmall.jpg`" the following is output:

```
>>> viewPicture()
width is 526 height is 423
The value of the pixel at 0, 0 is: Pixel red=110 green=121 blue=62
A different way to get the pixel at 0, 0 is: Pixel red=110 green=121 blue=62
```

There are a couple of things to pay attention to: your pixel has a location and values for red, green, and blue. The next couple of sections will explain this in more detail.

## 1.2 Exploring Pixels

After you ran the code above, you will have noticed that an additional window has popped up. This is the "Explorer" tool, which is bundled together with JES as part of the MediaTools application. This is what it should look like for you:

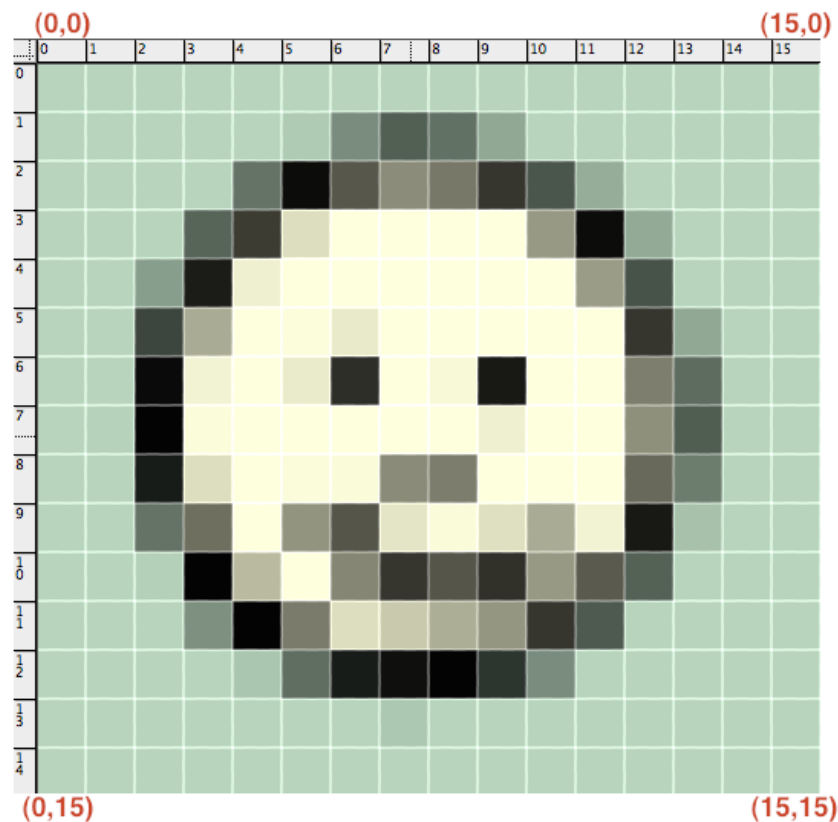


Let us try the following:

- click anywhere on the picture and notice how the X , Y and R:, G:, B: values change
- try clicking on the upper right-hand corner. Notice the value of X is close to width.
- try clicking on the bottom left-hand corner. Notice the value of Y is close to height.
- what do you notice about the X and Y coordinates when you click near the bottom right-hand corner?
- zoom into 500% (from the [Zoom](#) menu option). Do you see squares of color? You can click on the individual squares and see the RGB color values.

From exploring this picture, you may have an intuitive understanding of pixels. To give you a little more information, the pixels are stored in a two dimensional array. The x values represent the columns and the y values represent the rows. When we want to access an individual pixel (at a specific column and row), we can use the function `getPixel(picture, x, y)`.

The below diagram shows a small picture (16 x 16 pixels) and the representation of rows and columns with the corner coordinates annotated in red. Notice how the x values range from 0 to (width-1) and the y values range from 0 to (height-1).



Notice also how the order of the coordinates is (column, row). Be careful: this is different from two dimensional arrays in C++.

### 1.3 Exploring Colors

An individual pixel contains information about the color at that location. There are different color models; we will only focus on the RGB color model. This model corresponds to how we see color. Our eyes have three sensors that are triggered by light wavelengths that correspond to red, green, and blue. Combinations of (intensities of) these wavelengths are picked up by our sensors and allow us to perceive color.

When we use the RGB color model, there are three values: red (R), green (G), and blue (B). If each of these is stored in one byte (or 8 bits) of memory, then 0 represents no intensity and 255 represents full intensity.

Some examples:

- pure red is R: 255, G: 0, B: 0
- pure blue is R: 0, G: 0, B: 255
- pure white is R: 255, G: 255, B: 255

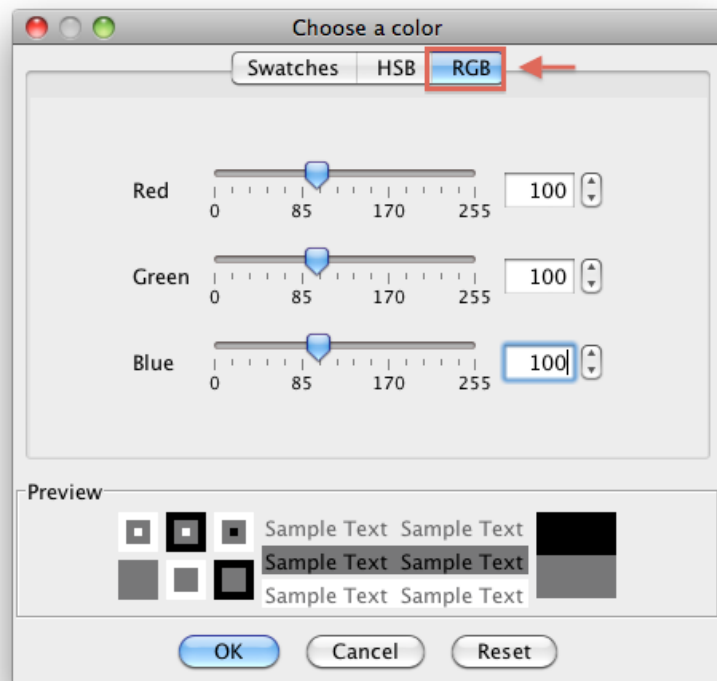
- pure black is R: 0, G: 0, B: 0
- a shade of grey is R: 100, G: 100, B: 100

Make a mental note that shades of grey have equal intensities for red, green, and blue components.

If you would like to play with colors and see their values, JES has a function called `pickAColor()`. The following implementation of a function makes use of `pickAColor()` and displays the RGB values that you have chosen.

```
def printColor():  
    color=pickAColor()  
    print color
```

The dialog box (shown below) is displayed for the `pickAColor()` function. Notice that you can select the RGB color model (outlined in red).



## 2. "Negating" an Image

The next two sections focus on changing the color values of a pixel. Remember, each pixel has a corresponding color value (red, green, and blue). If you move through the pixels one by one and change the color at that location, then you will end up modifying the overall picture.

So the question becomes, "how do we move through the picture one pixel at a time?" Well, we are going to use `for` loops. As it turns out, using JES, there are two approaches to moving through the pixels: the first approach generates a list of all the Pixel objects, and the second approach gets one pixel at a time based on a row and column value. Why are you learning both approaches? Well, if you want to work with part of the image or "flip" an image, it is easier to access individual pixels in specific rows and columns. By contrast, if you are making global changes in color to your entire image, you can grab all of the pixels (in a list). Enough theory, let us look at the code!

## 2.1 Going through a List of Pixels

The following code generates a list of all the Pixel objects (using the `getPixels()` function). Yes, a list of ALL of the pixels in the picture:

```
#program 16, page 69
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)
    show(picture)
    repaint(picture)
```

The `for` statement iterates through the list one Pixel object (`px`) at a time. Using this one Pixel object (`px`), we can:

- get the red, green and blue components (using `getRed()`, `getGreen()`, and `getBlue()`)
- create a new color (using `makeColor()`)
- modify the color at the current Pixel (using `setColor()`)

Voila, new image! And what do you think `makeColor(255-red, 255-green, 255-blue)` does? Well, if you have black (0, 0, 0), it will turn white (255, 255, 255). If you have red (255, 0, 0), what color will it turn?

You can try running the code by typing the following in JES:

```
>>> pict=returnPicture()
>>> negative(pict)
```

Where, `returnPicture()` is a helper function, included in this lab's sample code file. The helper function calls `pickAFile()` and `makePicture()` and returns the picture object.

## 2.2 Using a Nested For Loop

The above subsection generated a list of all the pixels, which is good for making global changes. However, if we want to work with specific rows and columns, it is better to use a nested `for` loop as below:

```
def negative2(picture):
    for x in range(0,getWidth(picture)):
        for y in range(0,getHeight(picture)):
            pixel = getPixel (picture, x, y)
            red = getRed(pixel)
            green = getGreen(pixel)
            blue = getBlue(pixel)
            negColor = makeColor(255 - red, 255 - green, 255 - blue)
            setColor(pixel, negColor)
        show(picture)
    repaint(picture)
```

You might have noticed the range function. For instance, `range(0, getWidth(picture))`, generates a list starting at 0 (the first argument) to one less than the width of the picture (the second argument). The inner loop (associated with `y`) will change the row. The outer loop (associated with `x`) will change the column. Effectively, this code will move through the picture down the columns one at a time.

Notice the difference in this code: we have to get individual pixels by their `x` and `y` location, using `getPixel(picture, x, y)`. Once we have an individual pixel, the code is the same as the previous subsection and will produce the same picture.

For something different, how would you only apply the pixel changes to the top half of picture?

## 2.3 Using a Function

This subsection is thrown in here just to show you that once you have an individual pixel, you can call a function (in this case, `negatePixel()`) that will change its color value(s):

```
def changePixel(picture):
    for px in getPixels(picture):
        negatePixel(px)
    show(picture)
    repaint(picture)

def negatePixel(pixel):
    red = getRed(pixel)
    green = getGreen(pixel)
    blue = getBlue(pixel)
    negColor = makeColor(255 - red, 255 - green, 255 - blue)
    setColor(pixel, negColor)
```

The purpose of this is to show you the "generic" concept behind modifying the colors in an image. Instead of calling the `negatePixel()` function, we could call another function that would modify the colors in a different way.

Syntactically, pay attention to how the function is defined:

- use the keyword `def` followed by the name of the function and any arguments in brackets
- do not forget the full colon (`:`) after the arguments

Can you tell from this code if arguments are passed by reference or by value? Why?

---

### 3. Changing the Color Values

The previous subsection was a warm up for this section. Through the following subsections, we are going to take a look at other color changes we can make to the pixels. To do this, we have created a function called `changePixel2` that controls what color change function will be invoked based on the second argument (`option`):

```
def changePixel2(picture, option):
    # valid option values are:
    # 1. Negate the image
    # 2. Lighten the picture
    # 3. Reduce the red
    # 4. Convert to Grayscale
    # 5. Blend with white
    if int(option)==1:
        for px in getPixels(picture):
            negatePixel(px)
    elif int(option)==2:
        for px in getPixels(picture):
            lightenPixel(px)
    elif int(option)==3:
        for px in getPixels(picture):
            reduceRedPixel(px)
    elif int(option)==4:
        for px in getPixels(picture):
            grayScalePixel(px)
    elif int(option)==5:
        for px in getPixels(picture):
            blendWhite(px,.30) #second argument specifies proportion of white
    else:
        print "not implemented yet"
    show(picture)
    repaint(picture)
```








Notice the syntax of the `if/elif/else` statements in the code. There are no brackets around the condition(s) and each condition has a full colon (`:`) after it.

To summarize the code, the options allow us to: negate the image (already discussed above), lighten the pixels, reduce the red in the image, turn the picture into gray scale, and blend the image with white. The format of all of these options is the same:

- go through the image one pixel at a time
- for each pixel, call a function to modify the color value of that pixel

The following table summarizes the results of each of these color changing functions and their corresponding number:

1. Negate	2. Lighten	3. Reduce Red	4. Grayscale	5. Blend with White
				

You can compare these results to the original:



You already know about the function to negate an image, but you might be curious about the other functions. The following subsections discuss the additional pixel modifying functions: `lightenPixel()`, `reduceRedPixel()`, `grayScalePixel()`, and `blendWhite()`.

### 3.1 Lightening

The lightening function is below:

```
def lightenPixel(pixel):
    color=getColor(pixel)
```

```
color=makeLighter(color)
setColor(pixel, color)
```

The code can be summarized as follows:

- get the Color object for the current pixel (using the JES function `getColor()`)
- return a lighter version of the original color (using the JES function `makeLighter()`)
- modify the current pixel to the lighter color (using `setColor()`)

## 3.2 Reducing Red

The function to reduce the red intensity by 50% is below:

```
def reduceRedPixel(pixel):
    value=getRed(pixel)
    setRed(pixel,value*0.5)
```

The code can be summarized as follows:

- get the red component for the current pixel (using the JES function `getRed()`)
- modify the current pixel's red value by 50% (using the JES function `setRed()`)

Notice how we didn't have to modify any of the green or blue, so the code does not touch those components.

What would we change if we wanted to reduce the red by 25%?

## 3.3 Applying Grayscale

The function to apply grayscale to the image is below:

```
def grayScalePixel(pixel):
    newRed = getRed(pixel)*0.299
    newGreen = getGreen(pixel)*0.587
    newBlue = getBlue(pixel)* 0.114
    luminance = newRed+newGreen+newBlue
    setColor(pixel, makeColor(luminance,luminance,luminance))
```

Remember from a previous section that gray is made by equal intensities of red, green, and blue; notice that `makeColor` has been called with the same value for red, green, and blue. You might be wondering how we got the calculation for the `luminance` value. Logically, the idea is that luminance will be calculated by the average of red, green, and blue intensities, which would give us a formula like one of the two below:

- `luminance=(red+green+blue)/3`
- `luminance=(red*0.3333)+(green*0.3333)+(blue*0.3333)`

However, the eye perceives blue to be darker than red so the formula for luminance places a lower weight on blue. Notice how  $0.299+0.587+0.114$  is 1.0 (a good weighting function). Once we have a luminance value from these weighted RGB values, we can then create a new gray color.

### 3.4 Blending with White

The function to blend the colors with `amount` of white is below:

```
def blendWhite(pixel, amount):  
    newRed = 255*amount + getRed(pixel)*(1-amount)  
    newGreen = 255*amount + getGreen(pixel)*(1-amount)  
    newBlue = 255*amount + getBlue(pixel)*(1-amount)  
    setColor(pixel, makeColor(newRed, newGreen, newBlue))
```

Yes, I know; it is more math. Let us examine how the `newRed` value is calculated. First, remember that white has RGB values of 255, 255, 255. If we want to blend with 20% white, then we should use 80% ( $1-0.20$ ) of the current red intensity. The first part of the formula (`255*amount`) is for controlling the "whiteness", and the second part (`getRed(pixel)*(1-amount)`) is for controlling how much of the original color comes through.

- If amount is 0.5, how much white and how much original color are used?
- If amount is 0.95, how much white and how much original color are used?
- If amount is 1, how much white and how much original color are used?

We are going to make use of blending with white in this lab's exercise!

---

## 4. Copying Pixels

The previous section used one `for` loop that cycled through a list of all of the pixels and made global changes to color. Sometimes (such as when we copy or mirror an image), we just want to use part of an image or access individual rows and pixels. To do that, we need to use a nested `for` loop. The following two sections examine how to create code that will duplicate an image and mirror an image.

### 4.1 Duplicating an Image

The general idea behind duplicating an image is to copy the pixels from one image to the exact location in another picture. Notice that the code below makes use of a JES function called `makeEmptyPicture()` to create an "empty" canvas with a default color of white and width and height equal to the original image:

```
#modified from page 86 program 24
#use setMediaPath() before calling this function
def copyCaterpillar():
    catFile=getMediaPath("caterpillarSmall.jpg")
    catPict=makePicture(catFile)
    width=getWidth(catPict)
    height=getHeight(catPict)
    canvas=makeEmptyPicture(width,height)
    #Now, do the actual copying
    for x in range(0, width):
        for y in range(0, height):
            color=getColor(getPixel(catPict, x, y))
            setColor(getPixel(canvas, x, y), color)
    show(catPict)
    show(canvas)
    return canvas
```

There are a couple of additional functions:

- `setMediaPath()` - displays a file picker dialog box. You can select a directory (or folder) where your pictures are stored. Call this function before using the code above.
- `getMediaPath()` - uses the directory that you have selected with `setMediaPath` and prepends it to filename. These two are super handy functions!

To summarize the code from above

- create a Picture object out of the caterpillar picture; use the JES functions `getMediaPath()` and `makePicture()`
- make an empty "canvas" (a second Picture object) to use for copying the image; use the JES function `makeEmptyPicture()`
- loop for all columns and all rows
  - get the current pixel's color for the caterpillar picture; use the JES function `getColor()`
  - copy that color onto the canvas (the copy picture); use the JES function `setColor()`

## 4.2 Mirroring an Image

This section talks about mirroring an image. The left-hand side is mirrored onto the right-hand side. Only half of the image is copied; but each pixel is copied twice. The second copy of the pixel is "flipped" to look like a reflection. The code is provided below:

```
#use setMediaPath() before calling this function
def copyAndMirrorCat():
    catFile=getMediaPath("caterpillarSmall.jpg")
    catPict=makePicture(catFile)
    width=getWidth(catPict)
    height=getHeight(catPict)
    canvas=makeEmptyPicture(width,height)
```

```
#Now, do the actual copying
for x in range(0, width/2):
    for y in range(0, height):
        color=getColor(getPixel(catPict, x, y))
        setColor(getPixel(canvas, x, y), color)
        setColor(getPixel(canvas,width-x-1, y),color)
show(catPict)
show(canvas)
return canvas
```

The first part of the code looks familiar:

- create a Picture object out of the caterpillar
- create a canvas (a second Picture object) of the same width and height as the caterpillar image
- now loop

Notice how you are only looping for half the width of the image. After you get a color at the current pixel (in the caterpillar image), you copy that color twice:

- one copy of the color is at the same pixel location as the original image
- the second copy is made starting from the right edge of the image (`width - x - 1`)

The resulting image looks like this:



What would you change to get an image that looks like this?



---

## 5. References

- Python Code and Concepts from: ***Computing and Programming in Python, a Multimedia Approach***, by Mark J. Guzdial and Barbara Ericson (Chapters 3 and 4)
  - S-curve Function from: [http://en.wikipedia.org/wiki/Logistic\\_function](http://en.wikipedia.org/wiki/Logistic_function)
  - Python Power Function from: <http://docs.python.org/library/functions.html>
- 

## 6. Exercise

### General Idea

A trendy thing to do is create a mirror image of a picture and then fade it to white using some application software. You can create the same effect in JES. The idea is that you will duplicate and mirror the image across the bottom. As you are duplicating the "flipped" image, you will calculate how much white blend you need to make. Yes, this does involve some math. You can choose a linear blend or an S-curve blend. Both results are shown below.

Linear Blend	S-curve Blend





## More Details

## Step 1

Create a function that will:

- Make a picture object out of an existing file. You can choose your own or use the caterpillar image that is in this lab.
- Create a "canvas" that is the same width and twice the height. If you like the white bar that separates the image from the reflection, then add approximately **three** pixels to the height of the canvas when you are creating it.
- Loop through column by column and row by row. Copy the pixel from the existing image twice:
  - one copy of the existing image
  - one copy to "reflect" the image (hint: the y value will start from the bottom edge of the canvas)
- Display (show) the canvas
- Return the canvas

At this point there will be no white blend but the image will be duplicated with one copy upside down.

## Step 2

Create a second function (to be called by the first) that will calculate and **return a "fade" amount**:

- Pass two arguments: the current y ([y](#)), the height of the image ([h](#))
- Choose one of these two:
  1. For the linear fade, the formula is:  
`fadeAmount=(h-y)/float(h)+0.15`
    - The idea of this formula is that the [y](#) will be mapped to a value between 0 and 1 (perfect for blend percentages). The [+0.15](#) is to give an instant "faded" look. Otherwise, the change is too gradual.
  2. For the S-curve fade, the formula is:  
`fadeAmount=1/(1+pow(2.71828, -(((h-y)/float(h))*8 -1)))`
    - this formula is a variation on the one in [wikipedia](#). Notice how [\(h-y\)/float\(h\)](#) is shared with the linear fade function to map [y](#) to a value between 0 and 1. The `*8-1` comes from experimenting with different values to see what looked good. Originally, `*12 - 6` was used to map the y value to the range of -6 and +6; but the "fade" did not happen fast enough or soon enough. You can try different values in the place of `*8-1` and see if you like the results.
- Check if `fadeAmount` is greater than one. If it is, then `fadeAmount=1`



- Return the `fadeAmount`

Use this function with your Step 1 function:

- Before you make a copy of the pixel for the "reflected" image, you can call this function to calculate your "faded" amount.
- The `blendWhite()` function might come in handy here too. Feel free to modify it.

---

The following function is included in the sample code for this lab and may come in handy if you want to save your results:

```
def writePict(pict,name):  
    file=getMediaPath(name)  
    writePictureTo(pict,file)
```

This code allows you to save your image to the specified `name` (in your current media path). The name, a string, should end in a .jpg, .png, or .bmp so that JES knows what format to apply.