## Pictures and Loops

In the previous section we saw how to change the red, green, and blue values of one pixel. Now, suppose we want to do this for many pixels, or even all of the pixels in an image. If we have 640 x 480 = 307,200 pixels, do you want to type in 307,200 `setRed` statements? Don't worry, we're not going to do this! We are going to use a loop statement to do this. A loop is a way to tell the computer to do the same thing (or almost the same thing) over and over again. We want to do the same thing, but to a different pixel each time.

A `for` loop will execute a block of commands that you specify for each item in a list that you provide. As part of the loop statement, you will specify a variable that will get the value of a different item from your list each time the group of commands executes. The list is an ordered collection of data – it could be numbers, strings, pixels, or many other different kinds of collections.

A `for` loop in Python looks like the following:

```
for ___________ in ___________:
    # do some cool stuff here
    # do some more cool stuff here
```

The very first part of this statement is the keyword `for`. The first blank (after the keyword `for`) will be filled in with your choice of a variable name, typically a name that represents the elements in your list. This variable is followed by the keyword `in`. The second blank (after the keyword `in`) will be filled in with the list for which you want to repeat the code with. At the end of the line you *must* have a colon, to indicate that what comes next is the block of code you want to repeat. The lines of code that are to be repeated should all be indented, and indented the same amount.

Since we will be working with pictures, we would like to be able to loop through all of the pixels in a picture in order to manipulate their colors. There is a function in the jes4py library called `getAllPixels` that will return a list of all the pixels in a picture. (Note: You could also use the function `getPixels`, but `getAllPixels` is a more descriptive name.)

**Example 1:** The following code segment will change the red value of each pixel to 0.

```
myPict = makePicture(pickAFile( ))
for px in getAllPixels(myPict):
        setRed(px, 0)
```

The variable `px` was chosen to represent each of the pixels that get returned from the call to `getAllPixels`. The first time through the loop, `px` will represent the pixel at location (0, 0), and that pixel’s red value will be changed. The second time through the loop, `px` will represent the pixel at location (1, 0), and that pixel’s red value will be changed. The loop will continue executing until `px` has represented each of the pixels in the picture.

Setting all of the red values to 0 may seem a little drastic, although it may produce a cool result. Let’s consider a function to cut the red value of a picture by a quarter.

**Example 2:** Reducing red by 25%

```
# Function to reduce red values by 25%
def reduceRed(picture):
  # copy the original picture
  newPict = duplicatePicture(picture)

  # change red value of all pixels
  for px in getAllPixels(newPict):
    value = getRed(px)
    setRed(px, value * 0.75)

  # return the newly modified picture
  return newPict
```

Notice that the loop structure in the middle of this function looks exactly like the loop we used in the previous example. The code inside the loop is slightly different - here we get the current amount of red in a pixel and then change it. By multiplying the red value by a factor less than 1 we are making the red value smaller. Reducing the color by 25% means we have 75% of the color remaining. That’s why we multiply the current red value by 0.75.

We also see in this example that a picture gets passed in as a parameter, and then the first thing we do is duplicate it. We do this so that we don’t modify the original picture. We create a new picture identical to the original, and then make our changes to the new picture. At the end of the function, we return the new picture so that we can use or save our results.

In the main function, we would call this function with code something like the following:

```
If __name__ == ‘__main__’:
        myPict = makePicture(pickAFile( ))
        newPict = reduceRed(myPict)
        show(newPict)
        newPict.show( ) #Windows users
```

How could we modify Example 2 so that we make the red values larger? The only change we need to make is to change the multiplication factor to be a value larger than 1 instead of smaller than 1.

**Example 3:** Increase the amount of red in a picture

```
# Function to increase red values by 20%
def increaseRed(picture):
  # copy the original picture
  newPict = duplicatePicture(picture)

  # change red value of all pixels
  for px in getAllPixels(newPict):
    value = getRed(px)
    setRed(px, value * 1.2)

  # return the newly modified picture
  return newPict
```

What happens if you increase the red in a picture that has a lot of red? When you multiply the red values by something larger than 1, there is a chance that the resulting value will be greater than 255. So what should we do? One possibility is that the program could crash. (This is undesirable, so designers of jes4py have avoided this possibility!) The other options would be to clip (*i.e.*, cap) the value of red at 255, or to wrap it around using the **modulo** [%](remainder) operator. Wrapping around may give you unexpected (but possibly interesting) results. For example, if the red value was 150 and you tried to double the red, the new value would be 300.

We can't have a red value of 300, so we would either set it to 255, or let it wrap around to 44 (300 – 256). The jes4py library was designed to offer some protection. In this case, the colors are automatically capped at 255.

So far, we have only done one color manipulation in the loop. There is nothing preventing us from doing more than one color manipulation at time. Suppose we want to add a sunset effect to our picture. When you view a sunset, the sky seems to redden, while everything gets darker. One way to do this could be to reduce the amount of green and blue in the picture. By doing this, the red will stand out more, and the other colors will get darker.

**Example 4:** Sunset Effect

```
# This function creates a sunset effect
def makeSunset(picture):
  # make a copy of the original picture to work with
  newPict = duplicatePicture(picture)

  # reduce the green and blue in all pixels
  for px in getAllPixels(newPict):
    gvalue = getGreen(px)
    bvalue = getBlue(px)
    setGreen(px, gvalue * 0.70)
    setBlue(px, bvalue * 0.70)

  # return the new picture
  return newPict
```

Notice, again, in this example, we are using the same loop structure to iterate through and change each of the pixels in the pictures. The big difference here is that we are getting and changing both the green and the blue values inside the loop.

## Creating Negatives

To create the negative of an image, we want the opposite of each the current values of red, green, and blue. So what does this mean? If we have no red (*i.e.*, the red value is 0), we need all red (*i.e.*, a red value of 255). If there is a lot of red, we need a little bit of red, and vice versa. So, say the red value of a pixel in a picture is 60. The red value in the corresponding pixel in the negative image would be 255 – 60 = 195. So, to create a picture that is the negative, we compute the negative value (255 – original value) of each of the red, green, and blue components for each pixel in the original picture. We then set the colors of the pixels in the new picture to these new colors. Our function would look like:

**Example 5:** Creating a negative image

```
# This function creates a negative image
def negative(picture):
  # make a copy of the original picture to work with
  newPict = duplicatePicture(picture)

  # change the pixel values to be the negative values
  for px in getAllPixels(newPict):
    rvalue = getRed(px)
    gvalue = getGreen(px)
    bvalue = getBlue(px)
    setRed(px, 255 − rvalue)
    setGreen(px, 255 − gvalue)
    setBlue(px, 255 − bvalue)

  # return the new picture
  return newPict
```

We could modify this slightly by using the `makeColor` function. This function takes numerical values for red, green, and blue, and creates a new color with these values. Our function then would look like:

```
# This function creates a negative image
def negative(picture):
  # make a copy of the original picture to work with
  newPict = duplicatePicture(picture)

  # change the pixel values to be the negative values
  for px in getAllPixels(newPict):
    rvalue = getRed(px)
    gvalue = getGreen(px)
    bvalue = getBlue(px)
    negColor = makeColor(255 − rvalue, 255 − gvalue, 255 − bvalue)
    setColor(px, negColor)

  # return the new picture
  return newPict
```

## Creating Grayscale Images

Creating grayscale images is also not too difficult. When the red component, green component, and blue component all have the same value, the resultant color is gray. This means that we will have 256 different shades of gray, ranging from black at (0, 0, 0), to white at (255, 255, 255). The only tricky part is figuring out what the replicated value should be. We want a sense of the intensity of the color, or the luminance. One way to compute this is to compute the average of the red, green, and blue component colors. Our function would look like the following:

**Example 6:** Creating a grayscale image

```
# This function creates a grayscale version of an image
def grayscale(picture):
  # make a copy of the original picture to work with
  newPict = duplicatePicture(picture)

  # change the pixel values to be the grayscale values
  for px in getAllPixels(newPict):
    rvalue = getRed(px)
    gvalue = getGreen(px)
    bvalue = getBlue(px)
    intensity = (rvalue + gvalue + bvalue)/3
    newColor = makeColor(intensity, intensity, intensity)
    setColor(px, newColor)

  # return the new picture
  return newPict
```

As it turns out, this method really oversimplifies the notion of grayscale. We can actually take into account how the human eye perceives luminance – we consider blue to be darker than red, even if there is the same amount of light reflected off. So we will weight blue lower, red and green higher when we compute the average. Our modified function would look like:

**Example 7:** Grayscale with weights

```
def weightedGrayScale(picture):
  # make a copy of the original picture to work with
  newPict = duplicatePicture(picture)

  # change the pixel values to be the negative values
  for px in getAllPixels(newPict):
    newRValue = getRed(px) * 0.299
    newGValue = getGreen(px) * 0.587
    newBValue = getBlue(px) * 0.114
    luminance = newRValue + newGValue + newBValue
    newColor = makeColor(luminance, luminance, luminance)
    setColor(px, newColor)

  # return the new picture
  return newPict
```

We will now test some of these functions and write some of our own functions to manipulate pixel colors in the next mini-lab and lab.

Mini-Lab: For loops for manipulating pixels in a picture

Lab: Simple Picture Manipulation