Image Objects

To access the pixels in a real image, we need to first create an `Image` object. Image objects can be created in two ways. First, an Image object can be made from the files that store digital images. The image object has an attribute corresponding to the width, the height, and the collection of pixels in the image.

It is also possible to create an Image object that is “empty”. An `EmptyImage` has a width and a height. However, the pixel collection consists of only “White” pixels.

We can ask an image object to return its size using the `getWidth` and `getHeight` methods. We can also get a pixel from a particular location in the image using `getPixel` and change the pixel at a particular location using `setPixel`.

The Image class is shown below. Note that the first two entries show how to create image objects. The parameters are different depending on whether you are using an image file or creating an empty image.

Method Name	**Example**	**Explanation**
Image(filename)	img = image.Image(“cy.png”)	Create an Image object from the file cy.png.
EmptyImage()	img = image.EmptyImage(100,200)	Create an Image object that has all “White” pixels
getWidth()	w = img.getWidth()	Return the width of the image in pixels.
getHeight()	h = img.getHeight()	Return the height of the image in pixels.
getPixel(col,row)	p = img.getPixel(35,86)	Return the pixel at column 35, row 86d.
setPixel(col,row,p)	img.setPixel(100,50,mp)	Set the pixel at column 100, row 50 to be mp.

Consider the image shown below. Assume that the image is stored in a file called “luther.jpg”. Line 2 opens the file and uses the contents to create an image object that is referred to by `img`. Once we have an image object, we can use the methods described above to access information about the image or to get a specific pixel and check on its basic color intensities.

```
import image
img = image.Image("luther.jpg")

print(img.getWidth())
print(img.getHeight())

p = img.getPixel(45,55)
print(p.getRed(), p.getGreen(), p.getBlue())

```

ActiveCode: 1 (pixelex1)

Run

When you run the program you can see that the image has a width of 400 pixels and a height of 244 pixels. Also, the pixel at column 45, row 55, has RGB values of 165, 161, and 158. Try a few other pixel locations by changing the `getPixel` arguments and rerunning the program.

Check your understanding

itr-8: In the previous ActiveCode example, what are the RGB values of the pixel at row 100, column 30?

- a) 149 132 122
- [✓ b) 183 179 170]
- c) 165 161 158
- d) 201 104 115

Check Me Compare Me

Image Processing and Nested Iteration

Image processing refers to the ability to manipulate the individual pixels in a digital image. In order to process all of the pixels, we need to be able to systematically visit all of the rows and columns in the image. The best way to do this is to use **nested iteration**.

Nested iteration simply means that we will place one iteration construct inside of another. We will call these two iterations the **outer iteration** and the **inner iteration**. To see how this works, consider the iteration below.

```
for i in range(5):
   print(i)
```

We have seen this enough times to know that the value of `i` will be 0, then 1, then 2, and so on up to 4. The `print` will be performed once for each pass. However, the body of the loop can contain any statements including another iteration (another `for` statement). For example,

```
for i in range(5):
   for j in range(3):
        print(i,j)
```

The `for i` iteration is the outer iteration and the `for j` iteration is the inner iteration. Each pass thru the outer iteration will result in the complete processing of the inner iteration from beginning to end. This means that the output from this nested iteration will show that for each value of `i`, all values of `j` will occur.

Here is the same example in activecode. Try it. Note that the value of `i` stays the same while the value of `j` changes. The inner iteration, in effect, is moving faster than the outer iteration.

```
for i in range(5):
   for j in range(3):
        print(i,j)

```

ActiveCode: 2 (nested1)

Run

Another way to see this in more detail is to examine the behavior with codelens. Step thru the iterations to see the flow of control as it occurs with the nested iteration. Again, for every value of `i` , all of the values of `j` will occur. You can see that the inner iteration completes before going on to the next pass of the outer iteration.

```
for i in range(5):
    for j in range(3):
        print(i,j)
```

<< First   < Back   Step 1 of 41   Forward >   Last >>

➡ line that has just executed

➡ next line to execute

Frames   Objects

CodeLens: 1 (nested2)

Our goal with image processing is to visit each pixel. We will use an iteration to process each row. Within that iteration, we will use a nested iteration to process each column. The result is a nested iteration, similar to the one seen above, where the outer `for` loop processes the rows, from 0 up to but not including the height of the image. The inner `for` loop will process each column of a row, again from 0 up to but not including the width of the image.

The resulting code will look like the following. We are now free to do anything we wish to each pixel in the image.

```
for row in range(img.getHeight()):
    for col in range(img.getWidth()):
        #do something with the pixel at position (col,row)
```

One of the easiest image processing algorithms will create what is known as a **negative** image. A negative image simply means that each pixel will be the opposite of what it was originally. But what does opposite mean?

In the RGB color model, we can consider the opposite of the red component as the difference between the original red and 255. For example, if the original red component was 50, then the opposite, or negative red value would be `255-50` or 205. In other words, pixels with alot of red will have negatives with little red and pixels with little red will have negatives with alot. We do the same for the blue and green as well.

The program below implements this algorithm using the previous image. Run it to see the resulting negative image. Note that there is alot of processing taking place and this may take a few seconds to complete. In addition, here are two other images that you can use. Change the name of the file in the `image.Image()` call to see how these images look as negatives. Also, note that there is an `exitonclick` method call at the very end which will close the window when you click on it. This will allow you to “clear the screen” before drawing the next negative.

cy.png

goldygopher.png

```
import image

img = image.Image("luther.jpg")
newimg = image.EmptyImage(img.getWidth(),img.getHeight())
win = image.ImageWin()

for row in range(img.getHeight()):
    for col in range(img.getWidth()):
       p = img.getPixel(col,row)

       newred = 255-p.getRed()
       newgreen = 255-p.getGreen()
       newblue = 255-p.getBlue()

       newpixel = image.Pixel(newred,newgreen,newblue)
```

ActiveCode: 3 (acimg_1)

Run

Lets take a closer look at the code. After importing the image module, we create two image objects. The first, `img`, represents a typical digital photo. The second, `newimg`, is an empty image that will be "filled in" as we process the original pixel by pixel. Note that the width and height of the empty image is set to be the same as the width and height of the original.

Lines 7 and 8 create the nested iteration that we discussed earlier. This allows us to process each pixel in the image. Line 9 gets an individual pixel.

Lines 11-13 create the negative intensity values by extracting the original intensity from the pixel and subtracting it from 255. Once we have the `newred`, `newgreen`, and `newblue` values, we can create a new pixel (Line 15).

Finally, we need to insert the new pixel into the empty image in the same location as the original pixel that it came from in the digital photo.

Other pixel manipulation

There are a number of different image processing algorithms that follow the same pattern as shown above. Namely, take the original pixel, extract the red, green, and blue intensities, and then create a new pixel from them. The new pixel is inserted into an empty image at the same location as the original.

For example, you can create a **gray scale** pixel by averaging the red, green and blue intensities and then using that value for all intensities.

From the gray scale you can create **black white** by setting a threshold and selecting to either insert a white pixel or a black pixel into the empty image.

You can also do some complex arithmetic and create interesting effects, such as Sepia Tone (http://en.wikipedia.org/wiki/Sepia_tone#Sepia_toning)

You have just passed a very important point in your study of Python programming. Even though there is much more that we will do, you have learned all of the basic building blocks that are necessary to solve many interesting problems. From an algorithm point of view, you can now implement selection and iteration. You can also solve problems by breaking them down into smaller parts, writing functions for those parts, and then calling the functions to complete the implementation. What remains is to focus on ways that we can better represent our problems in terms of the data that we manipulate. We will now turn our attention to studying the main data collections provided by Python.

Check your understanding

itr-9: What will the following nested for-loop print? (Note, if you are having trouble with this question, review CodeLens 3).

```
for i in range(3):
  for j in range(2):
    print(i,j)
```

```
a.

0 0
0 1
1 0
1 1
2 0
2 1

b.

0   0
1   0
2   0
0   1
1   1
2   1

c.

0   0
0   1
0   2
1   0
1   1
1   2

d.

0   1
0   1
0   1
```

[✓ a) Output a]

b) Output b

c) Output c

d) Output d

Check Me Compare Me

Correct!! i will start with a value of 0 and then j will iterate from 0 to 1. Next, i will be 1 and j will iterate from 0 to 1. Finally, i will be 2 and j will iterate from 0 to 1.

itr-10: What would the image produced from ActiveCode box 16 look like if you replaced the lines:

```
newred = 255-p.getRed()
newgreen = 255-p.getGreen()
newblue = 255-p.getBlue()
```

with the lines:

```
newred = p.getRed()
newgreen = 0
newblue = 0
```

[✓ a) It would look like a red-washed version of the bell image]

b) It would be a solid red rectangle the same size as the original image

c) It would look the same as the original image

d) It would look the same as the negative image in the example code

Check Me Compare Me

Correct!! Because we are removing the green and the blue values, but keeping the variation of the red the same, you will get the same image, but it will look like it has been bathed in red.

Image Processing on Your Own

If you want to try some image processing on your own, you can do so using the cImage module. You can download `cImage.py` from The github page (https://github.com/bnmnetp/cImage) . If you put `cImage.py` in the same folder as your program you can then do the following to be fully compatible with the code in this material.

```
import cImage as image
img = image.Image("myfile.gif")
```

Note

One important caveat about using `cImage.py` is that it will only work with GIF files unless you also install the Python Image Library. The easiest version to install is called `Pillow`. If you have the `pip` command installed on your computer this is really easy to install, with `pip install pillow` otherwise you will need to follow the instructions on the Python Package Index (https://pypi.python.org/pypi/Pillow/) page. With Pillow installed you will be able to use almost any kind of image that you download.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```


```

ActiveCode: 4 (scratch_07_05)

Run