# PLAB-2 (TDT-4113) Project 5:
Quick Art

**Purpose:**

- Become familiar with the Python Imaging Library (PIL or PILLOW)
- Gain hands-on experience with basic image-processing operations.

**Practical Information:**

- The due date for the demo of this project is **Monday**, October, 30 (kl 15:00).
- This project will be done individually, not in groups.
- All of the code described in this document is available in the file imager2.py, which is included in a zip file (imager.zip) provided on BLACKBOARD. All of the images to be used in this project are also included in that zip file, in the directory named "images".

## 1 Introduction

In this and the next project (involving robotics), you will write Python modules that handle images. Implementing image-processing software from scratch can be quite demanding, but luckily, the Python Imaging Library (PIL) provides a wide array of procedures that support an equally wide array of applications. This project will get you started with PIL (now renamed PILLOW for Python 3.0) so that you are fully prepared to deal with camera images from the robot in the next project.

The gist of this project is very simple: given a few images, do something *artistic* with them. You can define "artistic" as you prefer, but the implicit assumption is that you will show some *technical creativity* by combining and transforming the images in various ways. The output of your system should then be a single new image that attests to your creativity. That image will be a collage of several images: various copies, rotations, resizings or other manipulations of the original images.

This document provides some background on PIL along with a few examples of image combination and manipulation using it. A basic object-oriented approach to the problem is also presented. You are strongly advised to consult the PIL documentation at locations such as the following:

- effbot.org/imagingbook/pil-index.htm
- pillow.readthedocs.org

# 2 Using PIL

PILLOW is labelled as the *Friendly PIL Fork*. You will need to install it, not PIL, for this (and the next) project. However, the name PIL is still prominent when using PILLOW. For example, the following are three PILLOW imports that are commonly used, though only the first (Image) is necessary for the examples in this document:

```
from PIL import Image
from PIL import ImageFilter
from PIL import ImageEnhance
```

Note that we are importing from PIL, not PILLOW. There are probably a few minor exceptions, but for the most part, anything that you read about in online PIL documentation should pertain to PILLOW as well. So once you've downloaded and installed PILLOW, you can safely forget the LOW suffix and use PIL as the main module name.

In the Python discussion that follows, you will see two different classes (**Image** and **Imager**) and many different methods. In the text, some of these methods will be referenced using the class as a prefix, as in *Image.open* and *Imager.resize*, while in other cases, the method name (open or resize) will appear alone, without the class prefix. This is just a notational difference: both descriptors refer to the same thing. In some situations, the prefix is included simply as a (class) reminder to the reader.

## 2.1 The Image Class

The first of the three imports above, Image, is the fundamental class in PIL. It includes a whole host of methods for reading and writing images to/from files (of many different standard image formats, such as GIF, JPEG, and PNG). [1]

To create an Image object based on an image file, do the following:

Image.open(filepath)

This will open the file and load its contents into a newly-created Image object, which is returned. That object becomes your handle on the image, which you can then begin to analyze and manipulate. None of those changes will affect the original image file unless you decide to write the contents of the Image object back to that same file (using Image.save(filepath)).

## 2.2 The Imager Class

There are several properties that are useful to associate with an Image object, such as the file that it comes from and various alternate representations of the image, such as an array containing only the red component of each pixel. I have chosen to encapsulate each Image object inside a wrapper object named **Imager**, (an abbrevation for "Image Manager") which houses all of this auxiliary information. Most of the methods in this document operate on Imager objects, and they often return Imager objects. As you will see, this

[1] PILLOW can write to but cannot read from PDF files.

*common currency* (exchanged between methods) can greatly simplify coding such that many operations can be performed in one or a few lines of code.

Below are some of the basic methods for the Imager class:

```
class Imager():

    _pixel_colors = {'red':(255,0,0), 'green': (0,255,0), 'blue': (0,0,255), 'white': (255,255,255),
                     'black': (0,0,0)}
    _image_dir_ = "images/"
    _image_ext_ = "jpeg"

    def __init__(self, fname=False, dir=None, ext=None, image=False, width=100, height=100, background=
                 mode='RGB'):
        self.init_file_info(fname, dir, ext)
        self.image = image  # A PIL image object
        self.xmax = width;
        self.ymax = height  # These can change if there's an input image or file
        self.mode = mode
        self.init_image(background=background)

    def init_file_info(self, fname=None, dir=None, ext=None):
        self.dir = dir if dir else self._image_dir_
        self.ext = ext if ext else self._image_ext_
        self.fid = self.gen_fid(fname) if fname else None

    def gen_fid(self, fname, dir=None, ext=None):
        dir = dir if dir else self.dir
        ext = ext if ext else self.ext
        return dir + fname + "." + ext

    def init_image(self,background='black'):
        if self.fid: self.load_image()
        if self.image: self.get_image_dims()
        else: self.image = self.gen_plain_image(self.xmax,self.ymax,background)

    # Load image from file
    def load_image(self):
        self.image = Image.open(self.fid)  # the image is actually loaded as needed (automatically by P
        if self.image.mode != self.mode:
            self.image = self.image.convert(self.mode)

    # Save image to a file.  Only if fid has no extension is the type argument used.
    def dump_image(self,fid,type='gif'):
        fname = fid.split('.')
        type = fname[1] if len(fname) > 1 else type
        self.image.save(fname[0]+'.'+type,format=type)

    def get_image(self): return self.image
    def set_image(self,im): self.image = im

    def display(self):  self.image.show()
```

```
def get_image_dims(self):
     self.xmax = self.image.size[0]
     self.ymax = self.image.size[1]

def gen_plain_image(self,w,h,color,mode='RGB'):
      return Image.new(mode,(w,h),self.get_color_rgb(color))

def get_color_rgb(self,colorname): return Imager._pixel_colors[colorname]

 def resize(self,new_width,new_height):
       return Imager(image=self.image.resize((new_width,new_height)))

 def scale(self,xfactor,yfactor):
       return self.resize(round(xfactor*self.xmax),round(yfactor*self.ymax))
```

Notice in the **__init__** method, a filename (*fname*), directory (*dir*) and extension (*ext*) can be supplied. The directory and extension, if given, become the defaults for all other files processed by this Imager object. The file id (*fid*) is the concatention of the directory, filename and extension. If no filename is given, then no fid is generated, and a blank image is created in the **init_image** method. Otherwise, **init_image** creates an Image object based on the contents of the file specfied by fid.

Instance variables such as *xmax* and *ymax* are largely convenience variables. As shown in the **get_image_dims** method, these make it a little easier to get the width (xmax) and height (ymax) of an image than to access the image's **size** property. However, they also allow us to associate a size (width and height) with the Imager's image before the actual Image object is created – a small, but sometimes important technical detail.

The methods **Imager.load_image** and **Imager.dump_image** provide access to methods **Image.open** and **Image.save**, respectively, thus simplifying file reads and writes. The **get_image** and **set_image** methods are simple interfaces to the Imager's image object, while **Imager.display** draws the image in a screen window.

**Imager.gen_plain_image** creates a new Image object that uses an RGB tuple for each pixel (when mode = 'RGB'), has the width and height specified by w and h, respectively, and initializes to a single-color background specified by the *color* argument, a string. **Imager.get_color_rgb** uses the dictionary (**Imager._pixel_colors**) to return the RGB tuple that corresponds to a color name. For example, 'blue' maps to the tuple (0,0,255): no red, no green, and the full amount of blue.

The **Imager.resize** method employs Image.resize provided by PIL. This creates a new Image with the new dimensions and then places it inside a new Imager object. The **Imager.scale** method enables easy resizing: the user need not specify the exact dimensions of the new image, just the fractional change from the old image, such as a 50% width reduction (xfactor = 0.5) and an 80% height increase (yfactor = 1.80) to simulate a fun-house mirror, for example.

### 2.2.1 Pixel Access and Manipulation

The actual picture associated with an Image object is essentially a 2-dimensional array of pixels, with each being represented as either a single value (for simple images, such as black-and-white pictures) or, for more complex (typically multi-color) images, as a 3-tuple consisting of the red (R), green (G) and blue (B)

components. Each of these R, G and B components is refered to as a **band** in PIL terminology, and each band is an integer between 0 and 255. A pixel that is very white will have high values in all 3 bands, whereas black has three low bands. Keep that in mind when writing code to find particular areas of an image that are very red (or green or blue). A white pixel will have a very high red band, whereas a red pixel will have a high red band AND comparatively lower blue and green bands.

To access the value of any Image pixel, the method **Image.getpixel(loc)** does the job: it returns the pixel at location, loc, which is a tuple (x,y,) where x (y) represents the horizontal (vertical) offset from the origin (0,0) in the upper left corner of the image.

As shown below, **Imager.get_pixel** uses **Image.getpixel** to simplify access to the pixel's of an Imager's image. This returns a band value for simple images, and an RGB tuple for complex images.

To modify a pixel in an Image object, use **Image.putpixel(loc,p)**, where p is the new pixel value and should be a band or an RGB tuple for a simple or complex image, respectively. This is used in **Imager.set_pixel** and in **Imager.map_image2**, both shown below.

The method **combine_pixels(p1,p2,alpha)** takes two pixels (represented as RGB tuples) and generates a third pixel in which the bands of p1 and p2 are combined in a manner dictated by the parameter *alpha*. If alpha=0.5, then each pair of bands is simply averaged, but if alpha=0.25, then if, for example, the red bands of p1 and p2 are 200 and 40, respectively, then the red band of the new pixel will be $200(0.25)+40(0.75) = 80$. In short, this computes the weighted average of each pair of bands. This method will come in handy in the next section, when we morph one image into another.

The two mapping methods (shown below) allow the application (mapping) of a function to each pixel of an image. We need different methods, since mapping can be done in different ways.

First, **Imager.map_image** takes advantage of Image.eval(image,func). The Image object's eval method applies the given function (func) to each pixel of the image. The key point is that it actually applies the function to each BAND of each pixel. Hence, the operation embodied in func is performed on the red, green, and blue components independently. So a function such as (lambda v: v * 2) would multiply every band by 2. Hence, the RGB tuple (100, 50,75) would be converted to (200, 100,150). **Image.eval** (used in Imager.map_image) creates a new Image object that incorporates all of these modified pixel bands, and Imager.map_image then returns a new Imager object housing this new Image object.

In other situations, you may want to apply a function to each RGB tuple as a unit: you do not want to treat the bands independently. For instance, you may want to perform a "winner take all" transformation wherein each RGB tuple is converted to a simplified tuple in which only the maximum band has a non-zero value. Hence, the tuple (200, 150, 180) would be converted to (200, 0, 0). This requires a function that **compares** the three bands, i.e., it does not treat them independently. The method **Imager.map_image2** allows this type of mapping. Note that this method needs code – nested **for** loops – to iterate through the entire pixel array, whereas, in Imager.map_image, we simply called Image.eval (which handles the iteration).

Notice that map_image2 creates a copy (im2) of the original image and then modifies each pixel in im2. It does this by accessing each RGB pixel (using im2.get_pixel), and then applying the function (*func*) to it. The returned value is then put back into the pixel's location in the pixel array (using im2.putpixel). This requires that func is designed to return an RGB tuple.

To illustrate the use of map_image2, consider the method **Imager.map_color_wta**, which performs a "winner take all" competition among the bands of each pixel (as mentioned above) and gives the losing two bands a zero value. However, it only does so if the highest band takes up a large enough fraction of the sum total of bands – the fraction must equal or exceed the value of the argument **thresh** (for "threshold"). Figure 1

shows how winner-take-all can alter an image.

Dissecting the map_color_wta code a little more, it first defines a local function, **wta**, that will perform a winner-take-all operation on each pixel. It then calls map_image2 on the Imager's image with the func argument bound to the wta function. This is another nice feature of Python: it is easy to send functions as arguments to other functions or methods.

```
def get_pixel(self,x,y): return self.image.getpixel((x,y))
def set_pixel(self,x,y,rgb): self.image.putpixel((x,y),rgb)

def combine_pixels(self,p1,p2,alpha=0.5):
       return tuple([round(alpha*p1[i] + (1 - alpha)*p2[i]) for i in range(3)])

def map_image(self,func):
       return Imager(image = Image.eval(self.image,func)) # eval creates a new image.

def map_image2(self,func):
       im2 = self.image.copy()
       for i in range(self.xmax):
           for j in range(self.ymax):
               im2.putpixel((i,j),func(im2.getpixel((i,j))))
       return Imager(image = im2)

 def map_color_wta(self,thresh=0.34):
       # Local function
       def wta(p):  # p is an RGB tuple
           s = sum(p); w = max(p)
           if s > 0 and w/s >= thresh:
               return tuple([(x if x == w else 0) for x in p])
           else:
               return (0,0,0)

       return self.map_image2(wta,self.image)
```

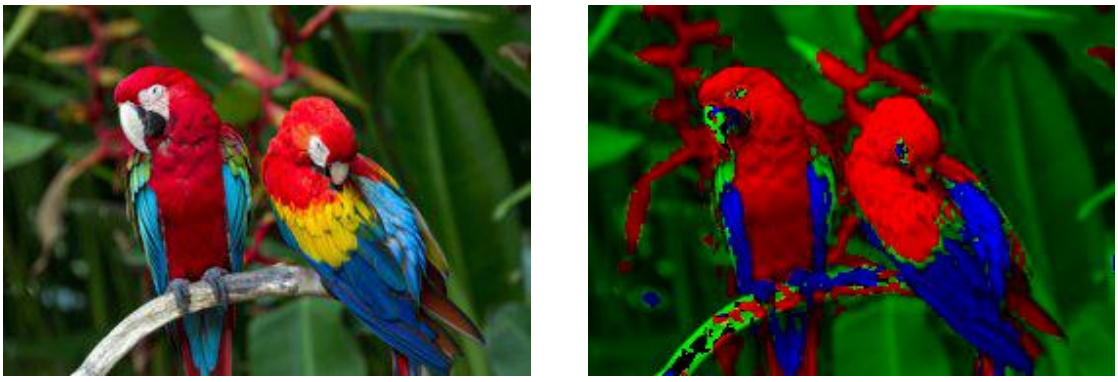

Figure 1: (Left) Original image. (Right) The result of applying map_color_wta to the original image. Note how the colors become more pure, and the white portions of the original are replaced by red, green or blue.

### 2.2.2 Combining Images

One of PIL's key primitives for combining images is the method **Image.paste**, which produces a new image (im3) resulting from overwriting a portion of the original image (im1) with a second image (im2). In the code below, **Imager.paste** is defined as the corresponding paste operator for Imager objects. Note that it requires only 3 parameters: the 2nd image, and the x and y coordinates of the first image that will serve as the upper left corner of the second image. The far right and bottom coordinates are simply derived from the width and height of the second image. Any compressions or other modifications of the second image need to be done before pasting it into the first.

Note that the call to Image.paste takes a single argument, a quad, consisting of the left, top, right and bottom coordinates that define the box (within the first image) where the second image will be placed.

Using Imager.paste, we can perform all sorts of image combinations. As simple examples, the **Imager.concat_vert** and **Imager.concat_horiz** methods (shown below) combine two images vertically and horizontally, respectively. Both methods create a third Imager (im3) whose dimensions fit those of the desired combination. Images need not have matching dimensions to be concatenated; the use of the **max** function insures that im3 will be large enough to accomodate the larger of the two images. Note that if a second image is not supplied, the first image is concatenated with a copy of itself.

```
def paste(self,im2,x0=0,y0=0):
     self.image.paste(im2.image,(x0,y0,x0+im2.xmax,y0+im2.ymax))

 def reformat(self, fname, dir=None, ext='jpeg', scalex=1.0, scaley=1.0):
     im = self.scale(scalex, scaley)
     im.dump_image(fname, dir=dir, ext=ext)

def concat_vert(self,im2=False,background='black'):
     im2 = im2 if im2 else self # concat with yourself if no other imager is given.
     im3 = Imager()
     im3.xmax = max(self.xmax,im2.xmax)
     im3.ymax = self.ymax + im2.ymax
     im3.image = im3.gen_plain_image(im3.xmax,im3.ymax,background)
     im3.paste(self,0,0)
     im3.paste(im2, 0,self.ymax)
     return im3

 def concat_horiz(self,im2=False,background='black'):
     im2 = im2 if im2 else self # concat with yourself if no other imager is given.
     im3 = Imager()
     im3.ymax = max(self.ymax,im2.ymax)
     im3.xmax = self.xmax + im2.xmax
     im3.image = im3.gen_plain_image(im3.xmax,im3.ymax,background)
     im3.paste(self, 0,0)
     im3.paste(im2, self.xmax,0)
     return im3
```

Using the method **Imager.combine_pixels** introduced earlier, we can easily morph one image into another by creating a series of images, each of which combines the pixels of the two original images, but using different

values of *alpha*: higher values give images closer to image 1, whereas lower alpha values give images that most resemble image 2. Below, the method **Imager.morph** creates a single morph image, while **Imager.morph4** creates two morphed images and combines them with the original two images into a rectangular set of 4 images. These morphing methods assume that the original two images have equal heights and equal widths.

In **Imager.morph4**, notice the final return line. There, several different concatenations are performed, and since each concatenation method returns an Imager object, that object can immediately call another method. The use of this common object type as the input and output of many methods allows us to string together (and nest) many method invocations. Figure 2 provides a sample image created using morph4.

```
def morph(self,im2,alpha=0.5):
      im3 = Imager(width=self.xmax,height=self.ymax) # Creates a plain image
      for x in range(self.xmax):
          for y in range(self.ymax):
              rgb = self.combine_pixels(self.get_pixel(x,y), im2.get_pixel(x,y), alpha=alpha)
              im3.set_pixel(x,y,rgb)
      return im3

def morph4(self,im2):
      im3 = self.morph(im2,alpha=0.66)
      im4 = self.morph(im2,alpha=0.33)
      return self.concat_horiz(im3).concat_vert(im4.concat_horiz(im2))
```

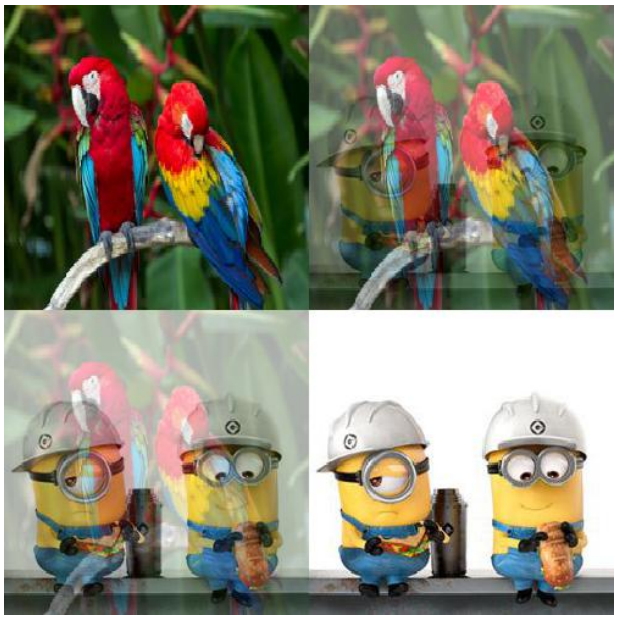

Figure 2: Example of a simple 3-step morph from two birds to two minions using the method **Imager.morph4**

As another trick, we can paste smaller and smaller copies of an image inside of itself to produce a tunnelling effect. The method **Imager.tunnel** employs recursion (via child.tunnel) to compactly implement

this entertaining procedure, an example of which appears in Figure 3.

```
def tunnel(self,levels=3, scale=0.75):
      if levels == 0: return self
      else:
          child = self.scale(scale,scale) # child is a scaled copy of self
          child.tunnel(levels-1,scale) # Recursion
          dx = round((1-scale)*self.xmax/2); dy = round((1-scale)*self.ymax/2)
          self.paste(child, dx,dy)
          return self

 def mortun(self,im2,levels=5,scale=0.75):
      return self.tunnel(levels,scale).morph4(im2.tunnel(levels,scale))
```

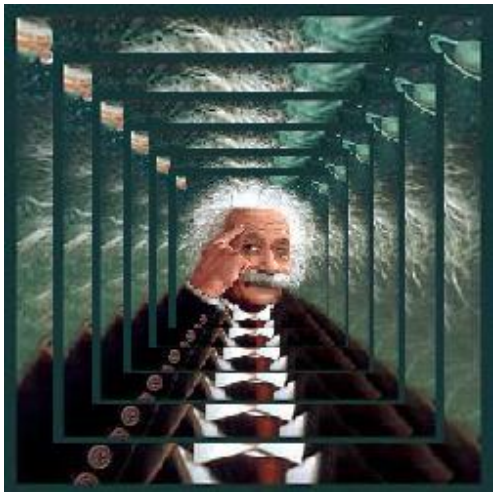

Figure 3: Recursive Einsteins produced by **Imager.tunnel**

Finally, tunnelling and morphing are easily combined (using method **Imager.mortun**) to produce some really strange images (Figure 4).

## 2.3 Tip of the Iceberg

The code above is a very small sampling of the possibilities that PIL affords. You are strongly advised to use the Imager class, or something similar, as the basis for your art-producing system. As is evident from methods such as Imager.mortun and Imager.morph4, once you have established a *common currency* data structure (e.g. object type) that many methods will exchange, the options for mixing and matching methods (for image manipulation and combination) increase dramatically. If you build your base of primitive methods correctly, you can easily explore wide expanses of artistic design space using relatively simple high-level methods.

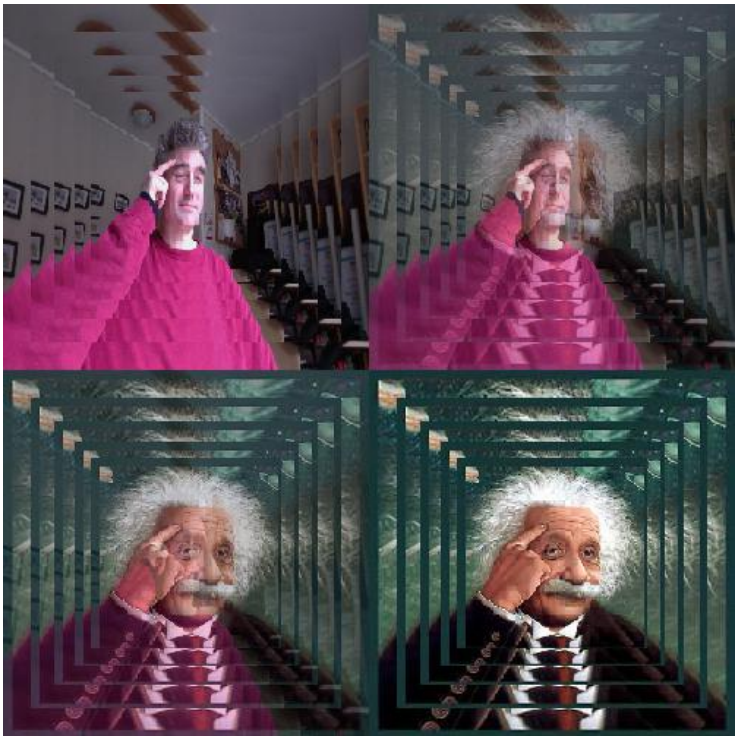

Figure 4: A combination of tunnelling and morphing produced by **Imager.mortun**

The code above uses only the Image Module, which you must also use. In addition, you **must** use image-manipulating tools from any **two** of the following additional PIL modules in order to receive a passing mark on this project:

- ImageEnhance
- ImageFilter
- ImageDraw
- ImageOps

As you will find by examining the PIL manual, each of these modules offers interesting new possibilities for image manipulation. During the demonstration, you will need to clearly explain how tools from two or more of these modules contribute to the final piece of artwork displayed on your laptop screen.

# 3 Demonstration

Along with a file containing the code above, you will receive a set of image files in both GIF and JPEG formats. These images have different sizes, so part of your preprocessing will probably involve resizing, since some procedures, such as morphing, typically require images with the same horizontal and vertical dimensions.

At the demonstration session, you will be given the names of **three** of those image files. You will then have **approximately 10 minutes** to examine the images and feed them into their system to produce a masterpiece.

Ten minutes will give you a little time to decide exactly how to arrange and modify the images. Depending upon your programming proficiency, this may be enough time to write a new high-level method; but you will certainly want to code up (**and debug**) all of the lower-level methods ahead of time, such as, for example, a method for removing all of the red from an image, or rotating it 90 degrees, or replacing all objects by simple black outlines on a white background, or progressively blurring an image from the inside out (i.e., the center is normal but blurring gets worse with radial outward movement), or.... use your imagination!

## 3.1 Essential Requirements

In summary, **to receive a passing mark on this project, all of the following criteria must be satisfied:**

1. Your code must be objected-oriented, with your classes using those provided by PILLOW. A class such as Imager is recommended but not required; you are free to explore other object-oriented approaches. For example, you may want to declare your main class as a subclass of Image instead of as a wrapper around an Image object.
2. Your code must employ PIL's Image module along with at least two of the other four PIL modules mentioned above, and these must clearly contribute to your resulting piece of art. You cannot merely import a module but never actually use it.

3. Your system must be able to make a piece of art out of the three images that are given to you at the start of the demo session. It must use all and only these three images as a starting point, though it may produce many copies of each.

   **Warning:** If your demo session turns into a debugging session, then you may fail this assignment. **Come to the demo session fully prepared.** Once you receive the names of three files, you will be expected to produce your final result in 10 minutes or less.