IMAGE PROCESSING PROJECT

CS110 Introduction to computer science

Terence Parr

UNIVERSITY OF SAN FRANCISCO

September 2014

The goal of this project is to exercise your understanding of all of the major components in Python: **assignments, expressions, if and loop statements, functions, lists, and libraries**. To make things interesting, we will perform some cool image processing tasks: **flipping horizontally, blurring, removing salt-and-pepper image noise, finding edges within images, and image sharpening**. For example, Figure 1 demonstrates image sharpening.

Figure 1: Bonkers the cat sharpened using `sharpen.py`.

Assignment structure

To make this project more manageable, we'll break it down into three parts.

- Part I: *Tasks 1-3*
- Part II: *Tasks 4-6*
- Part III: *Tasks 7-8*

You will have one week to do each part, Thursday to Thursday. We will still have quizzes every Tuesday.

Getting started

To get started, let's review how to access the command-line arguments that every program running on a computer can accept. Here is a small program, `view.py`, that accepts a filename as a command-line argument.

```
import sys
from PIL import Image
if len(sys.argv) != 2:
    print "$ python view.py imagefilename"
    sys.exit(1)
filename = sys.argv[1] # get the argument passed to us by operating system
img = Image.open(filename) # load file specified on the command line
img = img.convert("L") # grayscale
img.show()
```

To run this program, launch a terminal on Mac or Linux. Then, move to the directory that contains `view.py` that you saved/typed-in using terminal command `cd` (change directory). For example, if you are using directory `cs110/proj1` under your home directory for this project, then type this for Mac:

```
cd /Users/YOURID/cs110/proj1
```

and this for Linux:

```
cd /home/YOURID/cs110/proj1
```

Now, we can execute our `view.py` script and pass it an argument of `obama.png`:

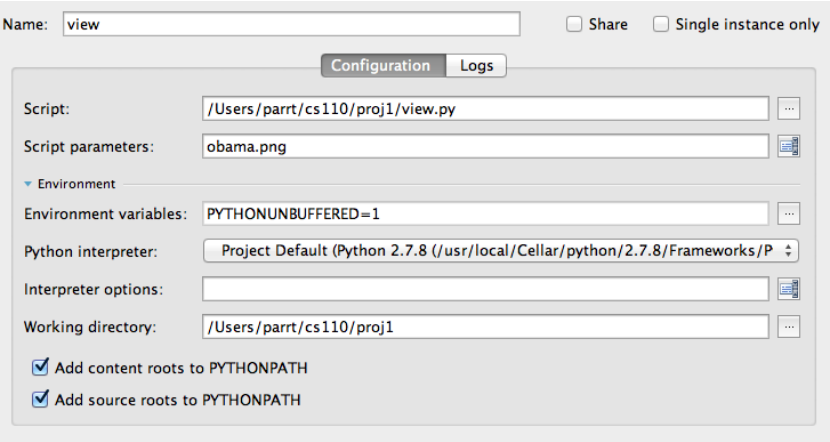
```
python view.py obama.png
```

For the remainder of this document, you'll see a $ on the left of the command, indicating you should execute it from the command line:

```
$ python view.py obama.png
```

You can download all of the image files I use from canvas in the files area under `projects/proj1`, but of course you can play around with whatever images you want.[1]

If you are using PyCharm, then you need to use the "`Edit configuration`" dialog under the Run menu. All file names that we specify in the script parameters area, or from the command line, are relative to the *current working directory*. That is why we used `cd` to change our working directory in the example above. For convenience, let's keep all of images and scripts in the same directory. For our purposes, let's assume that the directory is always `/Users/YOURID/cs110/proj1` (Mac) or `/home/YOURID/cs110/proj1` (Linux).



The "Script parameters" text field is where you provide the "command-line arguments," despite the fact we are not actually using a command-line. That is why PyCharm calls it script parameters instead of command-line parameters.

[1] Remember, however, that all images used in this class and those stored on University equipment must be "safe for work." Keep it G-rated please, with no offensive images popping up on your laptops or machines during lab etc.

1. Flipping an image horizontally

As a first task, you must create a script called `flip.py` that shows the image provided as a program (command-line) argument in its original form and then flipped horizontally. For example, Figure 2 shows the result of running script `flip.py` on image `eye.png`.

Figure 2: Flipping an image horizontally; the original is on the left.

Boilerplate code

Here's the boilerplate or "skeleton" code (save as `flip.py`) that we already know how to do but with a hole where you need to define a function called `flip` and a hole where you need to call that function to perform the flipping:

```
import sys
from PIL import Image

# define your flip function here
...
if len(sys.argv)<=1:
    print "missing image filename"
    sys.exit(1)
filename = sys.argv[1]
img = Image.open(filename)
img = img.convert("L")
img.show()

# call your flip function here
...
img.show()
```

If you are using PyCharm, your project file list area should look like the list in Figure 3. It assumes that you have downloaded two image files as well.

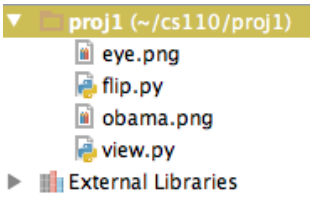


Figure 3: Project file list after building the flip task

Three new PIL pieces

To write your `flip` function, you will need **three pieces we have not covered yet**.

- `img.size` returns a tuple containing the width and height of image `img` so you can write code like this:

  ```
  width, height = img.size
  ```

  You'll need the width and height to iterate over the pixels of the image.

- `img.copy()` duplicates image `img`. For our `flip` function, it would be hard to modify the image in place because we would be overwriting pixels we

would need to flip later. It's easier to create a copy of the image in flipped position. You can write code like this:

```
imgdup = img.copy()
```

- `img.load()` is yet another weird name from PIL that actually returns an object that looks like a two-dimensional matrix. We have previously seen two-dimensional lists, which are really like lists of lists such as `m = [[1,2], [3, 4]]` or reformatted to look like the matrix:

  ```
  m = [[1, 2],
       [3, 4]]
  ```

  To get element 3, we would use list index expression `m[1][0]` because we want the list at index 1, `m[1]`, and then element 0 within that list. The two-dimensional object returned by `load()` uses similar notation. If we ask for the "matrix" with:

  ```
  m = img.load()
  ```

  then we would use notation `m[x,y]` to get the pixel at position (`x`, `y`).

  You will use these functions for the remaining tasks so keep them in mind.

Iterating over the image matrix

Define function `flip` using the familiar syntax and have it take a parameter called `img`, which will be the image we want to flip. The goal is to create a copy of this image, flip it, and return a copy so that we do not alter the incoming original image. To create `flip`, write code that implements the following steps.

1. Use `size` to define local variables `width` and `height`
2. Use `copy()` to make a copy of the incoming image `img` and save it in a local variable
3. Use `load()` to get the two-dimensional pixel matrix out of the incoming image and the copy of the image. Store these results in two new local variables.
4. To walk over the two-dimensional image, we've learned we need every combination of `x` and `y`. That means we need a nested `for` loop. Create a nested for loop that iterates over all `x` and all `y` values within the `width` and `height` of the image.
5. Within the inner loop, set pixels in the image copy to the appropriate pixel copied from the original image
6. At the end of the function, return the flipped image

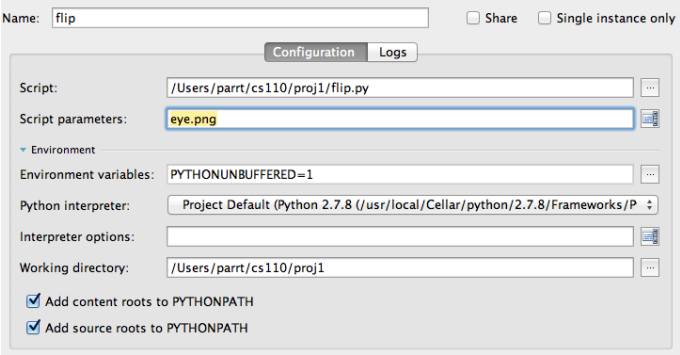
The only remaining issue is determining which pixel from the original image to copy into the (x, y) position in the image copy. The y index will be the same since we are flipping horizontally. The x index in the flipped image is index `width-x-1` from the original image. Trying out a few sample indexes shows that this works well. For example, a flipped image with `width=10` has its pixel at `x=0` pulled from index `x=10-0-1=9` in the original image. That's great, because it takes the image from all in the right in the original and copies it to the far left of the copy. Checking the opposite extreme, `x=9` in the flipped image should be `x=10-9-1=0` from the original image.

Running your flip script

To run the script from the command line, make sure you are in the `cs110/proj1` directory containing your scripts and images then do:

```
$ cd cs110/proj1
$ python flip.py eye.png
```

From PyCharm, I right-click and then select "Run." It will do nothing but print "`missing image filename`" because we have not given it a parameter yet, but we need to do this so that PyCharm creates a run configuration. Now, use the edit configuration dialog to specify `eye.png` as a parameter or any other image. (*Remember that the filename must refer to a file in the current working directory or must be fully qualified.*) Click "Run" now and it should bring out the two images.



What to do when the program doesn't work

If you have problems, follow these steps:

1. Don't Panic! Relax and realize that you will solve this problem, even if it takes a little bit of messing around. Banging your head against the computer is part of your job. Remember that the computer is doing precisely what you tell it to do. There is no mystery.

2. Determine precisely what is going on. Did you get an error message from Python? Is it a syntax error? If so, review the syntax of all your statements

and expressions. PyCharm is your friend here and should highlight erroneous things with a red squiggly underline. If you got an error message that has what we call a stack trace, a number of things could be wrong. For example, if I misspell `show()` as `shower()`, I get the following message:

```
Traceback (most recent call last):
  File "/Users/parrt/cs110/proj1/flip.py", line 26, in <module>
    img.shower()
  File "/usr/local/lib/python2.7/site-packages/PIL/Image.py", line 605, in __getattr__
    raise AttributeError(name)
AttributeError: shower
```

In PyCharm, the "`/Users/parrt/cs110/proj1/flip.py`" part of the trace will be a blue link that you can click on. It will take you to the exact location in your script where there is a problem. Look for anything that refers to that file.

3. If it does not look like it some simple misspelling, you might get lucky and find something in Google if you cut-and-paste that error message.

4. If your script shows the original image but not the flipped image, then you likely have a problem with your `flip` function.

5. If your program is at least running and doing something, then insert print statements to figure out what the variables are and how far into the program you get before a craps out. That often tells you what the problem is.

6. Try using the debugger to step through your program in PyCharm. Set a breakpoint on for example the line `filename = sys.argv[1]` by clicking in the gutter to the left of that line. A red dot should appear, indicating there is a breakpoint there. Then click the little bug icon instead of the green triangle (which is run button). That will start execution and then stop at that line. You can look at all of the variables at that point. Then you can step forward line by line. Read how to use the debugger online.

7. Definitely try to solve the problem yourself, but don't waste too much time. The TAs or I can typically help you out quickly so you can move forward.

2. Blurring

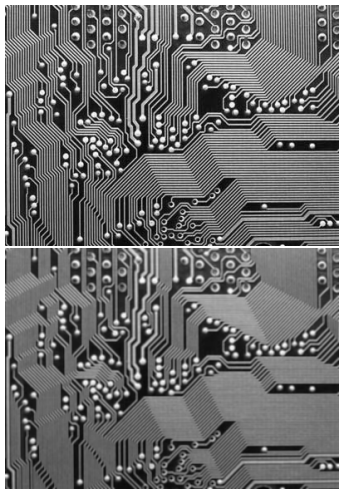


Figure 4: Blurring of a circuit board; the original is on top.

In this task, we want to blur an image by removing detail as shown in Figure . We will do this by creating a new image whose pixels are the average of the surrounding pixels for which we will use a 3x3 region as shown in Figure 5. The pixel in the center of the region is the region to compute as we slide the region around an image. In other words, `pixel[x,y]` is the sum of `pixel[x,y]` and all surrounding pixels divided by 9, the total number of pixels.

To implement this, start with the boilerplate from the previous section, which you should put into script `blur.py`. The only difference is that you must call soon-to-be-created function `blur` not `flip` as you had before. Now, let's start at the courses level of functionality and realize that we have to walk over every pixel in the image. (This is called *top-down design*.)

Blurring function

Define function `blur` to take an `img` parameter, just like the `flip` function in the previous task. In a manner very similar to `flip`, write code in `blur` to accomplish these steps:

1. Define local variables `width` and `height`.
2. Make a copy of the incoming image `img` and save it in a local variable.
3. Get the two-dimensional pixel matrix out of the image copy. Store it in a new local variable called `pixels`.
4. Create a nested for loop that iterates over all `x` and all `y` values within the `width` and `height` of the image.
5. Within the inner loop:
   (a) Call to-be-created function `region3x3` with arguments `img`, `x`, and `y` in store into local variable `r`.
   (b) Set `pixels[x,y]` in the image copy to the result of calling to-be-created function `avg` with an argument of `r`.
6. At the end of the function, return the flipped image.

Following the top-down design strategy, let's **define function** `avg` since it's the easiest. Define `avg` to take an argument called `data` or another of your choice. This will be the list of 9 pixels returned by function `region3x3`. The average of a set of numbers is their total divided by how many numbers there are. Python provides two useful functions here: `sum(data)` and `len(data)`. (Naturally, `sum` simply walks the list and accumulates values using a pattern we are familiar with.)

Image regions

Now we need to **define function** `region3x3`. Have it take three parameters as described above. This function create and return a list of nine pixels. The list includes the center pixel at x, y and the 8 adjacent pixels at N, S, E, W, ... as shown in Figure 5. Create a series of assignments that look like this:

```
me = getpixel(img, x, y)
N = getpixel(img, x, y - 1)
...
```

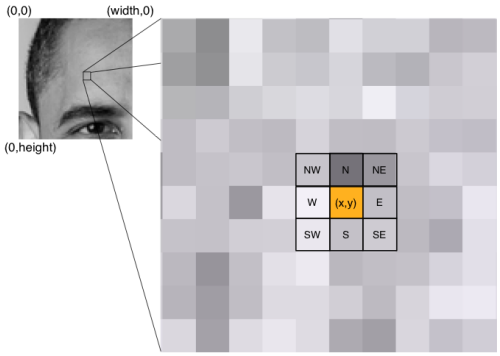


Figure 5: Hyper-zoom of Obama's forehead showing 3x3 region.

where function `getpixel(img, x, y)` gets the pixel at x, y in image `img`. We can't use the more readable expression `pixels[x,y]` in this case, as we'll see in a second. Collect all those pixel values into a list and return it. Make sure that this list is a list of integers and exactly 9 elements long.

Safely examining region pixels

We need to `define a function getpixel` instead of directly accessing pixels because some of the pixels in our 3x3 region will be outside of the image as we shift the region around. For example, when we start out at `x=0`, `y=0`, 5 of the pixels will be out of range, as shown in Figure 6.

Accessing `pixels[-1,-1]` will trigger:

```
IndexError: image index out of range
```

and stop the program. To avoid this error and provide a suitable definition for the ill-defined pixels on the edges, we will use a function that ensures all indices are within range.

Define function `getpixel` with the appropriate parameters. Its functionality is as follows:

1. Get the width and height into local variables.
2. If the x value is less than 0, set it to 0.

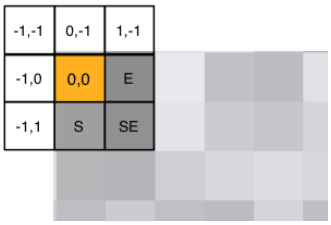


Figure 6: Our 3x3 region has pixels outside of the image boundaries as we slide it around the image along the edges.

3. If the x value is greater than or equal to the width, set it to the width minus 1 (the last valid pixel on the far right).
4. If the y value is less than 0, set it to 0.
5. If the y value is greater than or equal to the height, set it to the height minus 1 (the last felt pixel on the bottom).
6. Return the pixel at x, y.

Testing your blur code

That is all the code we need, so let's give it a test.

```
$ python blur.py pcb.png
```

That should pop up the original circuit board and the blurred version. It might take 10 seconds or more to compute and display the blurred image, depending on how fast your computer is.

3. *Removing noise*

For our next task, we are going to de-noise (remove noise) from an image as shown in Figure 7. It does a shockingly good job considering the simplicity of our approach. To blur, we used the average of all pixels in the region. To denoise, we will use the median, which is just the middle value in a list of ordered numbers.

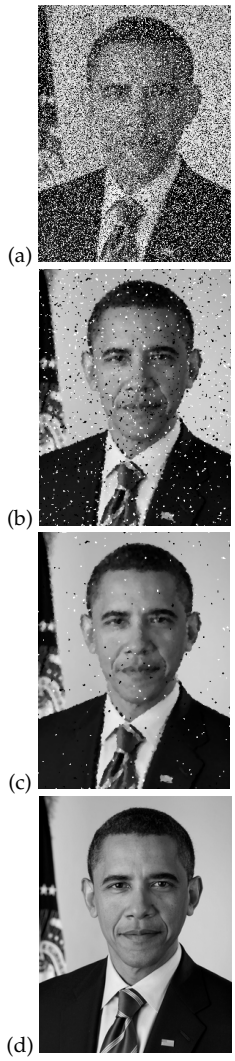
Believe it or not, we can implement the noise by copying `blur.py` into a new script called `denoise.py` and then changing a few lines. We also have to remove the no-longer-used `avg` function and replace it with a `denoise` function. Of course, instead of calling `blur`, we'll call function `denoise` with the usual `img` argument. The only difference between `denoise` and `blur` is that you will set the pixel to the `median` not `avg`. Hint: you need to tweak one statement in the inner loop that moves over all pixel values.

Now define function `median` that, like `avg`, takes a list of 9 numbers called `data`. Sort the list using Python's `sorted` function that takes a list and returns a sorted version of that list. Then compute the index of the middle list element, which is just the length of the list divided by two. If the length is odd, dividing by 2 (not 2.0) will round it down to the nearest index. Once you have this index, return the element at that index.

Let's give it a test:

```
$ python denoise.png guesswho.png
```

That should pop up the noisy Obama and the cleaned up version. You can save the cleaned up version and run `denoise.py` on that one to really improve it.[2] Running `denoise.py` twice, gives the cleaned up image (c) from Figure 7.



(a) (b) (c) (d)

Figure 7: Denoising an image of President Obama with lots of salt-and-pepper noise. (a) the noisy image, (b) denoised as computed by `denoise.py`, (c) denoised 2x, (d) original.

[2] Hint: **To save an image with PIL**, use `img.save("filename.png")`.