Image Processing

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 **??** demonstrates image sharpening.

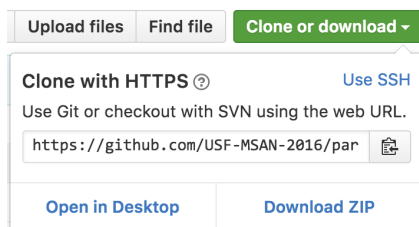Figure 1: Bonkers the cat sharpened using `sharpen.py`.

Git from the command line

Let's prepare our programming environment so that submitting this project is easy. This process is abbreviated here, but you can see it fleshed out in more detail in the section "Getting Started On The Images Project" of the [git lecture](#).

I will post to slack or via the students email list a URL for each project that represents an invitation to create a repository associated with a specific project for this class. Each student will have their own private repository per project. Once you have created the repository, you can proceed with this section to clone that empty repository onto your laptop disk.

Launch Terminal.app or whatever your bash shell is called on your system. Now, using a browser, go to the URL associated with your images project at github, which should be something like (YOURID is your github user/id not your laptop username):

```
https://github.com/YOURID/images-YOURID
```

and copy the `HTTPS not SSH` URL from here:

(That graphic is a bit out of date and the URL will look different in that little text field.) Then, use the `git clone` *URL* command as you see below, replacing `images-parrt` with `images-`*yourid*. You will have to type in your github user ID and password.

```
$ cd ~/msan501 # assuming this is where you want the root of all of 501 projects
```
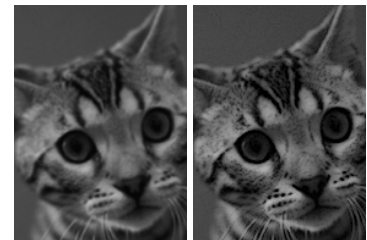
```
$ git clone https://github.com/USF-MSAN-2016/images-parrt.git
Cloning into 'images-parrt'...
Username for 'https://github.com': parrt
Password for 'https://parrt@github.com':
remote: Counting objects: 19, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 19 (delta 0), reused 0 (delta 0), pack-reused 13
Unpacking objects: 100% (19/19), done.
Checking connectivity... done.
$ cd images-parrt # after this command you are now in your repository
```

There is nothing in the repo yet, but we'll get to it!

Getting started

To get started, 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. The goal is really just to ensure that the PIL library is available for use, which you should be because you're using the Anaconda icon installation.

?? PythonTeX ??

(I provide this file and another in the starter kit.)

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 `msan501/images-yourid` under your home directory for this project, then type this:

```
cd ~/msan501/images-yourid; # ~ character is shorthand for /Users/yourid
```

You can get a bunch of sample images that I use for these notes from github, but of course you can play around with whatever images you want.[1]

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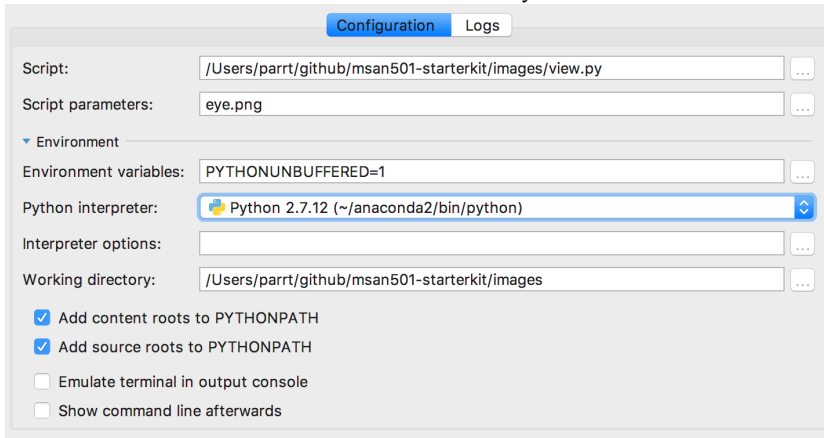
```
$ python view.py obama.png
```

If you are using PyCharm, you need to right-click in the code editor and say "run view.py," which will "fail" with this message in the Run tab:

```
/usr/local/bin/python /Users/parrt/github/msan501/projects/code/images/view.py
$ python view.py imagefilename
```

Now, 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*.

[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.

That is why we used `cd` to change our working directory in the example above. For convenience, keep all of images and scripts in the same directory. For our purposes, let's assume that the directory is always `/Users/YOURID/msan501/images-YOURID` (Mac) or `/home/YOURID/msan501/images-YOURID` (Linux). Here is the dialogbox for editing the run configuration, where my `view.py` file is in my `/github/msan501-starterkit/images` directory:

Configuration	Logs
Script:	/Users/parrt/github/msan501-starterkit/images/view.py
Script parameters:	eye.png
▾ Environment	
Environment variables:	PYTHONUNBUFFERED=1
Python interpreter:	Python 2.7.12 (~/anaconda2/bin/python)
Interpreter options:	
Working directory:	/Users/parrt/github/msan501-starterkit/images
☑ Add content roots to PYTHONPATH	
☑ Add source roots to PYTHONPATH	
☐ Emulate terminal in output console	
☐ Show command line afterwards	

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. Also note that it is running with anaconda's Python, which has the imaging library preinstalled!

Now, if you click the "run" button, you will see Obama's image appear on your screen or the "eyes" if you use `eye.png`.

Task 1. Flipping an image horizontally

As a first task, 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 **??** shows the result of running script `flip.py` on image `eye.png`.

Boilerplate code

Here's the boilerplate or "skeleton" code from:

https://github.com/parrt/msan501-starterkit/tree/master/images/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:

?? PythonTeX ??

An explanation of how to fill in those holes appears shortly.

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

Adding files to your repo

After creating the `flip.py` file and copying over an image or two, your PyCharm project file list area should look like the list in Figure **??**. So that git will manage our project, we need to announce that git should track `flip.py`:

```
$ cd ~/msan501/images-yourid
... copy image files, create flip.py ...
$ git add flip.py
```


Figure 3: Project file list after creating the flip.py file

Note that, if you are using PyCharm, when you open the dir in PyCharm, it will notice that the dir is a git repo. You can use PyCharm to manage the git repository later when you get more experience, but let's do things explicitly now from the command line.

Even though we haven't done any work on `flip.py` yet, let's commit the file to the repository and then push the repository back to github:

```
$ git commit -a -m 'add eye flip boilerplate code'
[master adcf1df] add eye flip boilerplate code
...
$ git push origin master
...
```

Now, go check out that github has the files. The web page for your repository should look something like the following, but without the images:

Three new Pillow pieces

To write your `flip` function, you will need three basic pieces:

- `img.size` returns a tuple containing the width and height of image img so you can write code like this: **?? PythonTeX ??**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 over-writing 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, which is really just a lists of lists such as `m = [[1,2], [3, 4]]`. When printed, the output looks like a matrix:

  ?? PythonTeX ??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: **?? PythonTeX ??**then we 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 function definition 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.
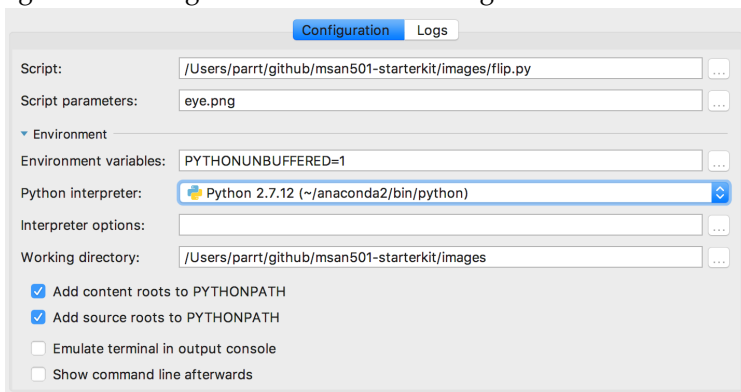
For example, a flipped image with `width=10` has its pixel at `x=0` pulled from index `x=10-0-1=9` in the original image. That's great, because it takes the image from all in the right in the original and copies it to the far left of the copy. Checking the opposite extreme, `x=9` in the flipped image 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 `msan501/images-userid` directory containing your scripts and images then do:

```
$ cd msan501/images-userid
$ 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. The dialog box will look something like this:

	Configuration	Logs
Script:	/Users/parrt/github/msan501-starterkit/images/flip.py	...
Script parameters:	eye.png	...

▼ Environment

Environment variables:	PYTHONUNBUFFERED=1	...
Python interpreter:	Python 2.7.12 (~/anaconda2/bin/python)	
Interpreter options:		...
Working directory:	/Users/parrt/github/msan501-starterkit/images	...

☑ Add content roots to PYTHONPATH
☑ Add source roots to PYTHONPATH
☐ Emulate terminal in output console
☐ Show command line afterwards

Again, please make sure that you are using the Anaconda Python so that it has the appropriate image processing libraries.

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/msan501/images-parrt/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/msan501/images-parrt/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. I can typically help you out quickly so you can move forward.

> **Deliverables**. Make sure that
> `images-`*userid*`/flip.py` is correctly committed to
> your repository and pushed to github.

Task 2. Blurring

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 **??**. 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 coarsest-level of functionality and realize that we have to walk over every pixel in the image. (This is called *top-down design*.)

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

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 blurred 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
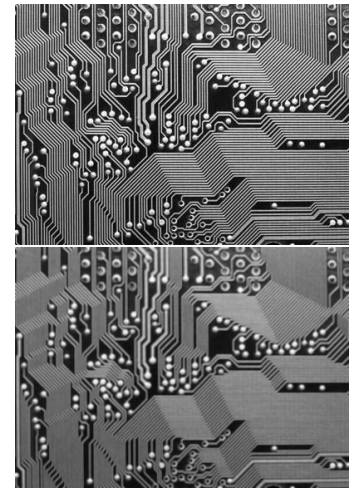
function `region3x3`. The average of a set of numbers is their total divided by how many numbers there are. Python provides two useful functions here: `sum(data)` and `len(data)`. (Naturally, `sum` simply walks the list and accumulates values using a pattern we are familiar with.)

Image regions

Now we need to **define function** `region3x3`. Have it take three parameters as described above. This function creates 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 **??**. Create a series of assignments that look like this:

   ?? PythonTeX ??

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 using `[a,b,c,...]` list literal notation and return it. Make sure that this list is a list of integers and exactly 9 elements long and that you keep in mind the order in which you add these pixels to the list. Any function that we create to operate on a region naturally needs to know the order so we can properly extract pixels from the list. For example, my implementation always puts the pixel at x and y first, then North, etc...

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

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 **??**. 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.

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).
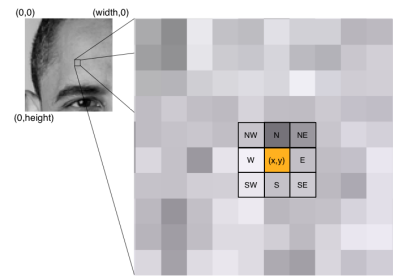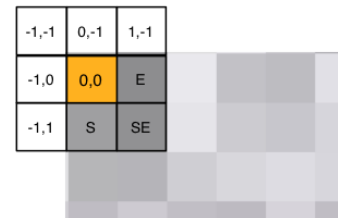
4. If the y value is less than 0, set it to 0.

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

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. You will need to use the `img.load()` function again to get the 2D `pixels` matrix as you did in function `blur`. Make sure you returning pixel and not the coordinates of the pixel from `getpixel`.

Testing your blur code

That is a lot of code to enter and so more than likely it won't work the first time.[2] That means we should test the pieces. It's generally a good idea to do top-down design but *bottom-up testing*. In other words, let's test the simple low-level functions first and make sure that works before testing the functions that call those functions and so on until we reach the outermost script.

With that in mind, lets test `avg` by passing it a fixed list of numbers to see if we get the right number. Add this to your script before it does any of the file loading stuff:

?? PythonTeX ??

Then run `blur.py` with any old image; the `apple.png` file is a good one because it's small (it's in the `projects/figures` dir of the `msan501` repo):

```
$ python blur.py apple.png
3
$
```

If it does not print $(1+2+3+4+5)/5 = 3$, then you know you have a problem in `avg`.

Now test `getpixel`. You will have to insert some code after loading and converting the image to grayscale because `getpixel` takes an image parameter:

?? PythonTeX ??

That should print: 96, 96, and 255. The upper left corner is gray and pixel 10, 20 is somewhere in the middle of the white Apple logo. If you don't get those numbers, then you have a problem with `getpixel`. Worse, if you don't get simple numbers, then you really have a problem with `getpixel`.

Before getting to `blur`, we should also **test** `region3x3` to ensure it gets the proper region surrounding a pixel. Replace those `getpixel` calls in the `print getpixel` statements with calls to `region3x3`. Use the x, y of the upper left-hand corner and somewhere near the upper left of the white section of the logo such as:

?? PythonTeX ??

That checks whether we get an out of range error at the margins

and that we get the correct region from somewhere in the middle. Running the script should give you the following numbers:

```
$ python blur.py apple.png
[96, 96, 96, 96, 96, 96, 96, 96, 96]
[255, 176, 255, 255, 215, 96, 245, 255, 255]
$
```

That assumes order: current pixel, N, S, E, W, NW, NE, SE, SW.

When you have verified that all of these functions work, it's time to check function `blur` itself. Try the printed circuit board image:

```
$ 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.

> ⚠️ Make sure to remove all of your debugging code before submitting your scripts. Submitting a project that prints out a bunch of random debugging output is considered sloppy, like submitting an English paper with a bunch of handwritten edits.

> 🖋 **Deliverables**. Make sure that images-*userid*/`blur.py` is correctly committed to your repository and pushed to github.

Task 3. Removing noise

For our next task, we are going to de-noise (remove noise) from an image as shown in Figure **??**. 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 de-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 `median` 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, which
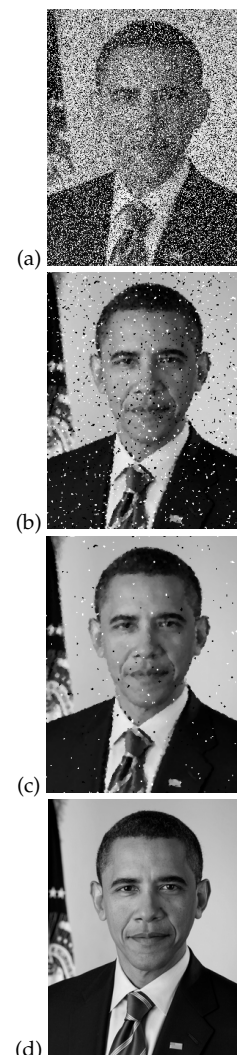

(a)

(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.

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 even, 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.py 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.[3] Running `denoise.py` twice, gives the cleaned up image (c) from Figure **??**.

[🖋 **Deliverables**. Make sure that `images-`*userid*`/denoise.py` is correctly committed to your repository and pushed to github.]

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

Task 4. Re-factoring to improve code quality

As I mentioned in the last task, `blur.py` and `denoise.py` are virtually identical, meaning that we have a lot of code in common. Figure **??** demonstrates this visually. One of the most important principles of computer science is to reduce code duplication. We always want exactly one place to change a particular bit of functionality. In this case, we have the following common code:

1. Functions `getpixel` and `region3x3`.

2. The "main" part of the script that loads the original image.

3. Functions `blur` and `denoise` are identical except for the function called to compute a new pixel in the image copy from a 3x3 region in the original (`avg` or `median`).

The goal of this task is to make new versions, `blur2.py` and `denoise2.py`, that share as much code as possible. The functionality will be the same, but they will be much smaller and we will get warm feeling that our code is well structured. To share code, we need to (a) put all common code in a file that these new scripts can import and (b) create a generic function called `filter` that will work for both blurring and de-noising in an image. Once we get the common code into a single file, which we will call `filter.py`, we can import it into `blur2.py` and `denoise2.py` like this:[4]

Figure 8: Visual difference between scripts `blur.py` and `denoise.py`. The files are identical except for orange marks.

[4] Do not confuse script names with function names; `filter.py` can contain anything we want and it so happens that we will also put a function in there with the same name, `filter`.

?? PythonTeX ??

That statement asks Python to look in the current directory (among other places we don't care about for now) for file called `filter.py` and import all of the functions. You can think of it as a formalized cut-and-paste.

Let's start by creating new "library" file `filter.py` and placing our usual imports at the top:

?? PythonTeX ??

Now, copy functions `getpixel` and `region3x3` into it.

We can also create a function called `open` to hide all of the messiness that checks for command-line arguments and opens the indicated image:

?? PythonTeX ??

The only tricky bit[5] is to create a single generic `filter` function that can reproduce the functionality we have in functions `blur` and `denoise`.

1. Define function `filter` to take `img` and `f` parameters.

2. Copy the body of function `blur` into your new `filter` function.

3. Replace the call to `avg(r)` with `f(r)`.

As we discussed in class, functions are objects in Python just like any strings, lists, and so on. That means we can pass them around as function arguments.[6] To use our new generic `filter` function, we pass it an image as usual but also the name of a function:

?? PythonTeX ??

In the end, your `filter.py` script file should have 4 functions: `getpixel`, `region3x3`, `filter`, and `open`.

Armed with this awesome new common file, our entire `blur2.py` file shrinks to a tiny script:[7]

?? PythonTeX ??

The `denoise2.py` script is also tiny:[8]

?? PythonTeX ??

Before finishing this task, be a thorough programmer and test your new scripts to see that they work:

```
$ python blur2.py pcb.png
$ python denoise2.py guesswho.png
```

They *should* work, but unfortunately that is never good enough in the programming world. Lot of little things can go wrong. *Certainty* is always better than *likelihood*.

[5] Pun intended

[6] Don't confuse the name of a function with an expression that calls it. Assignment `f = avg` makes variable `f` refer to function `avg`. `f = avg()` **calls** function `avg` and stores the return value in variable `f`. Using `f = avg`, we can call `avg` with expression `f()`. You can think of `f` as an alias for `avg`.

[7] You might be wondering why we don't have to include the usual `sys` and `PIL` imports at the start of our new files. That is because we import our `filter.py` file, which in turn imports those files.

[8] Yep, these files are identical except for the fact that we call `filter` with different function names. If you wanted to get really fancy, you could replace both of these scripts with a single script that took a function name as a second argument (after the image filename). With some magic incantations, you'd then ask Python to lookup the function with the indicated name and pass it to function `filter` instead of hard coding.

We will import file `filter.py` into the future scripts in this project. You have created your first useful library. **Good job!** 🙂

> **Deliverables**. Make sure that
> `images-`*userid*`/blur2.py`,
> `images-`*userid*`/denoise2.py`, and
> `images-`*userid*`/filter.py` are correctly committed to
> your repository and pushed to github.

(a)

(b)

Figure 9: Edges of an old photograph from World War II. (a) original, (b) edges as computed by `edges.py`.

Task 5. Highlighting image edges

Now that we have some basic machinery in `filter.py`, we can easily build new functionality. In this task, we want to highlight edges found within an image. It is surprisingly easy to capture all of the important edges in an image, as shown in image (b) from Figure **??**.

The mechanism we're going to use is derived from some serious calculus kung fu called the *Laplacian*, but which, in the end, reduces to 4 additions and a subtraction! The intuition behind the Laplacian is that abrupt changes in brightness indicate edges, such as the transition from the darkness of a uniform to the brightness of a windshield edge. As we did for blurring and denoising, we are going to slide a 3x3 region around the image to create new pixels at each x, y. That means we can reuse our `filter` function—we just need a `laplace` function to pass to `filter`.

To get started, here is the boilerplate code copied from `denoise2.py` but with function name `laplace` (the object of this task) passed as an argument to function `filter`:

?? PythonTeX ??

Create function `laplace` that takes region `data` as an argument as usual. Have the function body return the sum of the North, South, East, and West pixels minus 4 times the middle pixel from our usual region:

NW	N	NE
W	(x,y)	E
SW	S	SE

That computation effectively compares the strength of the current pixel with those around it.

> For those familiar with calculus, we are using the second partial derivative (i.e., acceleration) in x and y directions. The first derivative would detect edges even for gradual changes but the second derivative detects only really sharp changes. For a pixel fetching function f operating on a 3x3 region around (x, y), "applying the *Laplacian*" means computing a filtered image pixel at x, y as:
>
> $$f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$
>
> where $f(x, y)$ is equivalent to our `pixels[x,y]`.

For example, imagine a region centered over a vertical white line. The region might look like:

	255	
0	255 (x,y)	0
	255	

The `laplace` function would return $255 + 255 + 0 + 0 - 4 \times 255 = -510$.

Compare that to the opposite extreme where values are almost the same:

	18	
21	10 (x,y)	15
	19	

The `laplace` function would return $18 + 19 + 15 + 21 - 4 \times 10 = 33$.

Once you have implemented your `laplace` function, give it a try with some of the sample images you have such as the jeep or Obama:

```
$ python edges.py obama.png
```

It actually does a really good job capturing Obama's outline:

Be aware of something that Pillow is doing for us automatically when we store values into an image with `pixels[x,y] = v`. **If `v` is out of range 0..255, Pillow clips `v`.** So, for example, `pixels[x,y] = -510` behaves like `pixels[x,y] = 0` and `pixels[x,y] = 510` behaves like `pixels[x,y] = 255`. It doesn't affect edge detection or any of our other operations in future tasks but I wanted to point out that in a more advanced class we would **scale** these pixel values instead of clipping them. Clipping has the effect of reducing contrast.

> **Deliverables**. Make sure that
> `images-`*userid*`/edges.py` is correctly committed to
> your repository and pushed to github.

Task 6. Sharpening

Sharpening an image is a matter of highlighting the edges, which we
know how to compute from the previous task. Script `edges.py` com-
putes just the edges so, to highlight the original image, we *subtract*
that white-on-black edges image from the original. You might imag-
ine that *adding* the edges back in would be more appropriate and it
sort of works, but the edges are slightly off. We get a better image
by subtracting the high-valued light pixels because that darkens the
edges in the original image, such as between the uniform and the
windshield. Let's start with the easy stuff:

1. Copy your previous `edges.py` file to new script `sharpen.py`.

2. After `edges = filter(img, laplace)`, add a line that calls a
   function we'll create shortly called `minus`. `minus` takes two image
   parameters, `A` and `B` and returns `A-B`. In our case, pass in the orig-
   inal image and the image you get back from calling `filter(img,`
   `laplace)`.

3. Show the result of the `minus` function.

That only leaves the task of **creating function** `minus` to subtract
the pixels of one image from the pixels of another image like a 2-D
matrix subtraction. As we did before, we will return a modified
version of a copy of an incoming image parameter. (In my solution,
I arbitrarily chose to create and return a copy of `A`.) Because you are
getting really good at creating functions to manipulate images, the
instructions for creating `minus` in this task are less specific than in
previous tasks. You need to fill in the body of this function:

    ?? PythonTeX ??

The mechanism is the same as before: iterating loop variables
`x` and `y` across the entire image and processing the pixel at each
location. The only difference between this function and `filter` is that
we want to operate on individual pixels not 3x3 regions. In the inner
loop, set `pixels[x,y]` to the value of pixel `A[x,y]` minus pixel `B[x,y]`.
Don't forget to return the image you filled in.

Here's how to run `sharpen.py` on Bonkers the cat:
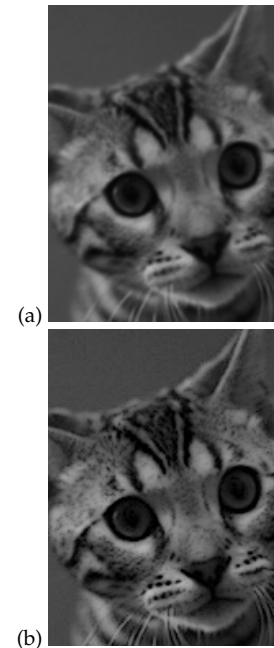
```
$ python sharpen.py bonkers-bw.png
```


(a)

(b)

Figure 10: Bonkers the cat portrait. (a)
original and (b) sharpened as computed
by `sharpen.py`.

Figure **??**, Figure **??**, and Figure **??** show some sample transformation sequences with original, *Laplacian*, and sharpened images.

Deliverables. Make sure that `images-`*userid*`/sharpen.py` is correctly committed to your repository and pushed to github.

Figure 11: Sharpening of a Bonkers the cat. Clockwise: (a) original, (b) edges as computed by `edges.py`, (c) the sharpened image as computed by `sharpen.py`.
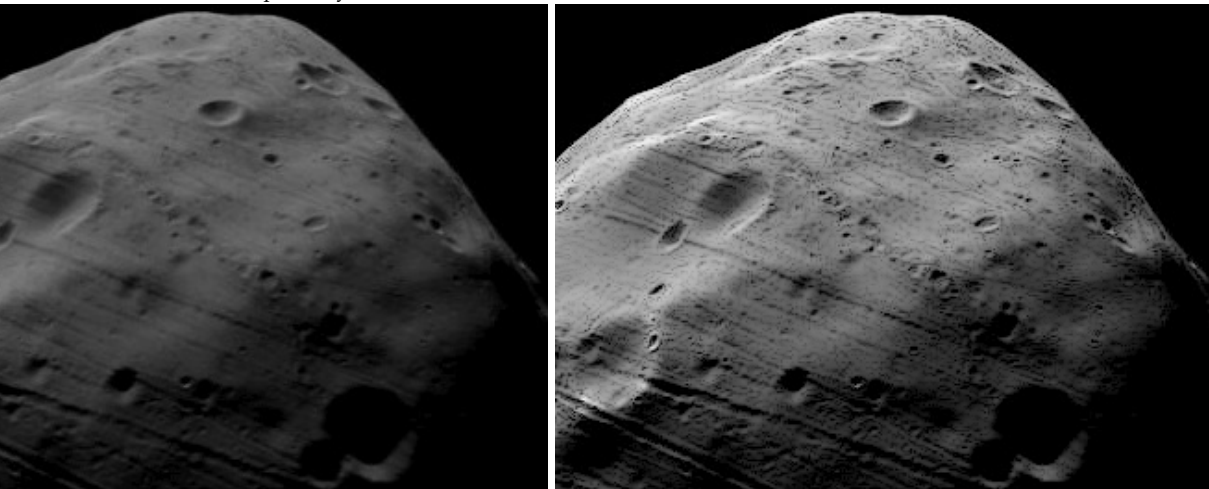

Figure 12: Sharpening of Phobos asteroid from NASA. Clockwise: (a) original, (b) edges as computed by `edges.py`, (c) the sharpened image as computed by `sharpen.py`.
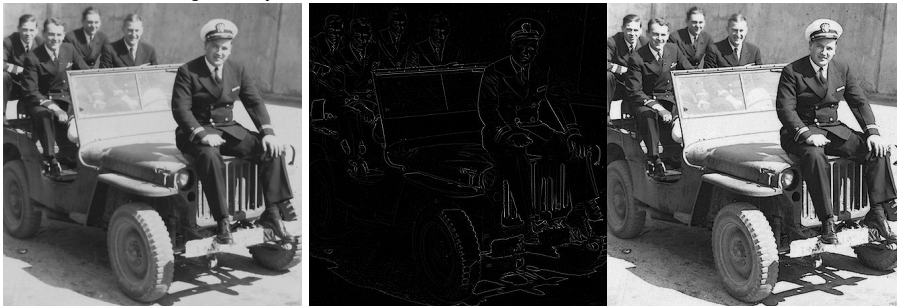

Figure 13: Sharpening of an old photograph from World War II. Clockwise: (a) original, (b) edges as computed by `edges.py`, (c) the sharpened image as computed by `sharpen.py`.