ASCII Art Generator Lab

Introduction:

In this lab, we are going to create a Python script that can take in image files and convert the images into ASCII art, stored as a .txt file on your computer. This works due to the limited resolution of the human eye; since our eyes have limited resolution, we see the average brightness of the characters in the ASCII art, but we cannot pick out the detail of the individual characters. This is good for us, because if our eyes had a high enough resolution to pick out that detail, the art wouldn't look as "real" since we would easily see it is just a big bunch of text. We have a few steps we need to follow to generate our art, which are summarized here:

1. Take the image and convert it to greyscale
2. Split up the image into m x n tiles
3. Correct m (# of rows) to match the image and font aspect ratio (ratio of width to height)
4. Calculate the average brightness for each image tile and look up an ASCII character for each tile
5. Create rows of ASCII strings and print them to a text file to form the final image

Now, doing this from scratch would be a very difficult problem indeed. Fortunately, programmers often create libraries and frameworks to make solving specific problems easier for others, and we will be using a few of these libraries to make our job much easier. Specifically, we will be using two external Python libraries called numpy and Pillow in our program. Numpy is a python library that contains some handy functions for handling linear algebra, and Pillow contains an Image class that we will be using for all our image-related needs. You are not expected to be familiar with these libraries, so instead of explaining the entire library, we will instead cover select functions and how they are used when they come up. Additionally, our program is going to be run from the command line, so we are going to use Python's built in argparse library to handle taking in arguments from the command line. Don't worry if you've never done this before and it sounds overwhelming; you will see these are fairly easy things to do, and we will create a few pieces of code to help us out first instead of trying to do the entire project at once. That brings us to the first step, where we consider defining our greyscale ramp and converting the image into greyscale.

Note: In the file provided, there are comments showing which step of the lab corresponds with which line in the file. This way you can check if you are working in the correct part of the file, and if you get lost you can find where you were working relatively quickly.

Part One: Converting Image to Greyscale and Picking Greyscale Levels

When you open the file provided for the lab, you'll notice a few things. First, there will be some import statements at the top of the file. These are for the libraries we are using, and you should not edit these or add any additional libraries.

```
import argparse
import numpy as np
from PIL import Image
```

Then, you should see three functions, getAverageL(image), convertImageToAscii (fileName, cols, scale, moreLevels), and main (). For this section, we are writing code for the convertImageToAscii function, so make sure you are writing your code in the correct spot:

```
def convertImageToAscii(fileName, cols, scale, moreLevels):
    """
    Given Image and dims (rows, cols) returns an m*n list of Images
    """
```

Step 1: You may notice that currently we do not have any values for some of the variables we are using, such as fileName or cols. We will be getting these values from the user's input to the command line at a later step, so for now, pretend that we do have these values to avoid confusion. The greyscale values are already defined for you at the start of the function:

```
# Step 1 ----------------------------------------------------------------------------
# 70 levels of grey
gscale1 = "$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\|()1{}[]?-_+~<>i!lI;:,\"^`'. "
# 10 levels of grey
gscale2 = '@%#*+=-:. '
```

Both values are stored as strings, and they progress from darkest to lightest. This is to save you from having to type them in yourself, and you can find the justification for using these two greyscale levels at the following website: http://paulbourke.net/dataformats/asciiart/. We will be using these strings as look-up tables once we start finding the brightness of our image tiles.

Step 2: Now that we have our greyscale values, we can set up our image. We need to open our image and split it up into a grid. To do this, we are going to use the Image class from the Pillow library that we imported at the top of the Python script. The Image class has an open() method and a convert() method that we can use together to open the image and convert it to greyscale like so:

```
# open image and convert to greyscale
image = Image.open(fileName).convert('L')
```

This opens the image with the name stored in the variable fileName, and we convert to greyscale by specifying 'L' in the convert method, which stands for luminance and can be thought of as the brightness of a pixel. Now we have an 8-bit greyscale image, so each pixel has a value from 0 to 255, with 0 being black and 255 being white.

Step 3: All that's left for now is to break up the image into a grid. To do so, we will use the size method to figure out how large our image is first. As we stored our picture in the variable "image", we can call another method image.size, which returns a list of the width and height of the image as a list, [width, height]. You do not need to call image.size using round brackets. Store the width and height in the variables W and H by calling image.size. Hint: Remember that you can index a list using square brackets like so: myList[0] would give the first element stored in myList.

```
# Step 3 ------------------------------------------------------------------------
# store dimensions
W, H =
print("input image dims: %d x %d" % (W, H))
```

The print statement will display the total image size to the user, as they may find that useful. Now we have the width and height of the entire image, but it will be important to know the dimensions of the tiles we are working with as well. We already have the number of columns from the user, stored in a variable named "cols". This might not seem like much, but it gives us everything we need to calculate the missing values tile width, tile height, and number of rows.

```
# compute width of tile
w =
# compute tile height based on aspect ratio and scale
h =
# compute number of rows
rows =

print("cols: %d, rows: %d" % (cols, rows))
print("tile dims: %d x %d" % (w, h))
```

Calculating the width of a tile should be straightforward as we know the number of columns required, but calculating the tile height is a bit trickier. This depends on the aspect ratio of the image, and the scale for the font we are using. The aspect ratio is the ratio of the width of an image to its height. A common aspect ratio you may have seen is 16:9, which follows the common x:y format. For an x:y aspect ratio, no matter how big or small the image is, if the width is divided into x units of equal length and the height is measured using this same length unit, the height will be measured to be y units. The value stored in the variable "scale" represents the aspect ratio, so if we divide the width of a single tile by the value in scale, we will get the height of all our tiles (as all tiles will be the same size). Once we know the height of a tile, finding how many rows we have in our image is straightforward. Make sure the value stored in rows is an integer (it would not make sense to have 0.3 of a row, for example).

Lastly, we implement a quick check that the image is not too small for our program to convert it to ASCII art. An image is too small for our program if the number of columns is greater than the width of the image, or if the number of rows is greater than the height of the image. This comes from the simple fact that we can't have more columns or rows than pixels, as there is no way to split the image up in to units smaller than one pixel. Implement the check in the code below so that it will print an error message to the user to let them know what went wrong if the image is too small:

```
# check if image size is too small
if
    print("Image too small for specified cols!")
    exit(0)
# END OF PART ONE
```

Now we are done with Part One, and we can move on to the next important step: computing the average brightness of a tile.

Part Two: Finding the Average Brightness

To recap, we've converted our image to greyscale, split the image up into m x n tiles, and we've corrected m to match the image and font aspect ratio. In this part of the lab, we're going to write the code for the helper function "getAverageL" at the top of the script, pictured below:

```
# START OF PART TWO
def getAverageL(image):
    """
    Given PIL Image, return average value of greyscale value
    """
```

The L stands for luminance, which for our purposes is another word for the brightness of the image in greyscale. Our function will be quite short, as you can see from the image below we only need three lines to make this function work.

```
# get image as numpy array
im =
# get shape
w,h =
# get average
return
```

Step 4: Here is where the numpy library we imported will come in handy. You may not have learned linear algebra yet, but we can think of an image as a matrix of values, one value for each pixel. If we had a 3x3 image tile that we passed into the function, it would consist of 9 pixels and it may look something like this:

```
# Each list inside of image_matrix is a row, each integer is a pixel's brightness
image_matrix = [[87,90,102],[40,34,60],[12,208,255]]
```

This represents the square image tile we received, and stores all the brightness values for each pixel as an integer from 0 to 255. This might be difficult to picture, so here is what the image would look like if we replaced each pixel with it's brightness value:

```
image_matrix =

    87    90   102
    40    34    60
    12   208   255
```

We want to get our image into this format so that we can compute the average brightness. Fortunately, numpy makes this easy with the array function, which will automatically convert an image into an array like the one above. The syntax is "np.array()", with the brackets containing the item we wish to convert to an array, our image in this case. Once we have the image stored as an array in the variable "im", we can call the "shape" method on "im" using dot notation to give us the width and height of our array, which we will need in the last part of the function. You do not have to call the shape method with round brackets.

Step 5: Lastly, now that we have the array dimensions we can find the average of all the pixel brightness values. In order to do this, we need two functions: np.average(), which returns the average value of a one-dimensional array, and im.reshape(), which will convert our two dimensional image array into a one dimensional array. In order to use im.reshape(), we must enter the length of the array into the round brackets as an integer. This length should be equal to the number of pixels in a tile. Then, we can call a function using another function as input, so we can take the average of the reshaped array. Return the average of the reshaped array, and you have finished the helper function! This will return a number between 0 and 255 so that we can use it to look up a proper ASCII character.

Part Three: Generating the ASCII content from the Image

Now we are going to finish the main part of the program, which generates a list of strings of ASCII art that we will print to a text file. As our image is going to be a list of character strings, we initialize an empty list for us to store our strings in:

```
# START OF PART THREE
# ascii image is a list of character strings
aimg = []
```

Now, we know how many rows and columns there are for our grid of tiles. This means we can set up a nested for loop structure, which is a for loop inside of another for loop. The first loop will cycle the rows, and the second loop will cycle the columns. So, we will start at the first row, process column one, column two, and so on until all the columns are processed. Then we will move on to the second row and repeat the process, finishing once we have computed the average brightness of every tile. The general

framework of the for loops have been filled out for you, but it will be up to you to do some math to calculate the variables required (there will be hints).

Step 6:

```
# generate list of dimensions
for j in range(rows):
    y1 =
    y2 =
    # correct last tile
    if j == rows-1:
        y2 = H
    # append an empty string
    aimg.append("")
    for i in range(cols):
        # crop image to tile
        x1 =
        x2 =
        # correct last tile
        if i == cols-1:
            x2 = W
        # crop image to extract tile
        img = image.crop((x1, y1, x2, y2))
```

As we can see, we will need to calculate x1, y1, x2, and y2. These are the coordinates of the image tile we want to crop and send to our getAverageL function. Since we are dealing with an integer coordinate system, we should make sure the final values of x and y are integers using the int() function. We'll start with y1 and y2, and if you can get these calculated then x1 and x2 should be straightforward.

We want to calculate the starting and ending y coordinates of the image tile to be cropped. We know the height of a single tile is stored in the variable "h" from part one of the lab. We can use that here to calculate our coordinates. If we imagine the image as a graph, we start at the origin (0,0). We want to scan across the image row by row, so our first two y coordinates would be 0 and h. If we then move on to the second row, y1 would be h and y2 would be 2h. If we continue this pattern for y1 and y2: (2h, 3h), (3h, 4h), (4h, 5h), … we should be able to come up with an equation to calculate y1 and y2. As a hint, you should be using the variable j and the variable h in your equation for both y1 and y2. After calculating y1 and y2, we append an empty string to the list of strings "aimg" so that we can add ASCII characters into it later. Once you think you have the formula correct, you should be able to use a very similar formula to calculate x1 and x2 using the variables "i" and "w". Now that we have all the necessary coordinates, we can move on to the next step of finding the tile's luminance and picking an appropriate ASCII character.

Step 7:

```
# get average luminance (it should be an integer)
avg =
# look up ascii char
if moreLevels:
    gsval = gscale1[int((avg*_)/255)]
else:
    gsval = gscale2[int((avg*_)/255)]
# append ascii char to string
aimg[j] += gsval
```

In the last section, we want to use our getAverageL function to store a value in avg, and we want to replace the two underscores with appropriate values to multiply the variable "avg" with. That can be a little tricky, but all we are trying to do is scale the value in "avg" so that we end up with an integer from 0 to 69 for gscale1 (the 70-character greyscale string) and an integer from 0 to 9 for gscale2 (the 10-character greyscale string). This calculation gives us the index for the strings gscale1 and gscale 2, and it will return an ASCII character from those strings and store it in gsval. Once gsval is calculated, we append the character stored in gsval to the j^{th} string in aimg (remember j represents which row we are currently on), and we've calculated the ASCII character for one tile! Fortunately, the for loops will take care of the rest of the work, cycling through all the tiles in the image. We just have one part of the lab left: getting command line input from the user and printing the text to a file for viewing.

Part Four: Command Line Parsing and Writing to a Text File

We are at the final step before we will have a working ASCII art generator. This part of the program is very important, as it sets up many of the variables we have been using in the rest of the lab. When writing Python scripts, it can be handy to have the user set variables from the command line instead of having to open the script up and manually change the variables themselves. Among other things, this lets the user call your script from the command line and try a variety of variable inputs without having to open your file at all. This also makes it easier to migrate the Python script to an operating system such as Linux, which runs primarily through the command line.

Step 8:

```
# START OF PART FOUR
def main():
    # Step 8 ------------------------------------------------------------------------------
    # create parser
    descStr = "This program converts an image into ASCII art."
    parser = argparse.ArgumentParser(description=descStr)
    # add expected arguments
    parser.add_argument('--file', dest='imgFile', required=True)
    parser.add_argument('', dest='', required=)
    parser.add_argument('', dest='', required=)
    parser.add_argument('', dest='', required=)
    parser.add_argument('',dest='',action='store_true')

    # parse args
    args = parser.parse_args()
```

The first step is to create an Argument Parser object. To do so, we must pass in a description for the parser, which we set up on the previous line as a string. This will tell the user what the program is doing, which is usually nice to know. Put simply, an argument parser object exists to take in arguments from the user, and then translate them into variables in our code. After creating a parser, we need to specify how many arguments we want using parser.add_argument for each argument required. In our case, we will need five arguments. The first one has been specified for you as an example.

This may be new to you so let's break down what is happening in the first add_argument call. The first argument '—file' is what the user will have to enter into the command line to specify the file path containing the image to be converted. You will see an example of what this looks like later if you missed the one at the beginning of the lab. The second variable, "dest", is a string specifying the name of the variable where we wish to store the user's input. Here, we have specified the file path should go into the variable "imgFile". Lastly, the required variable must be set to True or False, depending on whether the input is required for the program to function properly. So, that leaves the other four arguments to be filled in by you. To give a little help, the four other arguments should be specified as –scale, --out, --cols, and –morelevels (they all start with a double hyphen). You can choose other names if you like, these are just the ones I've chosen, but they should start with a double hyphen. Then, the variable names you should be using for each argument can be found throughout the code so far. Finally, the three arguments containing "required" are not necessary for the program to run. In the fourth argument, we are only concerned with a binary yes or no, so if the user specifies this argument in the command line, we will store True in this variable, and if they do not, we will store False (Hint: which of the four arguments is a binary yes or no? i.e it cannot take more than two values, yes or no). Once we finish adding all our arguments, we use the parse_args function to parse all the arguments passed in by the user. This will let us

tell if the user entered a value for each of the arguments, which is important since not all the arguments are required and so we may not get an input for each one.

Step 9:

```
imgFile = args.imgFile
# set output file
outFile = 'out.txt'
if args.outFile:
    outFile = args.outFile
# set scale default as 0.43 which suits a Courier font (scale should always be a float)
scale = 0.43
if :
    scale =
# set cols (cols should always be an int)
cols = 80
if :
    cols =

print('generating ASCII art...')
# convert image to ascii txt
aimg = convertImageToAscii(imgFile, cols, scale, args.moreLevels)
```

In this step we are setting up default values for any variables the user may have decided not to enter, and we also implement the check to see if the user entered anything at all for each argument. To start, we know that the user had to set something for imgFile, as it was required for the program to run. We store the user input (which was kept in args.imgFile for us to use thanks to the parser) in the variable imgFile as a result. Now we get into the non-required arguments. The first argument, the output file, is handled for you as an example. First, we set up a default value in the variable outFile saying that the default name of the text file we write to should be out.txt. Then, we check if there is anything inside of the outFile argument from the parser. In Python, any non-zero and non-empty value is considered True, so if the user did enter something for outFile, the if condition will be true and we enter the if statement, where we set the value of outFile to the argument entered by the user. The process is similar for the scale and cols variables, so you should be able to fill in the rest. Just note that scale must always be a float value and cols must always be an integer. Then, we print a message for the user to let them know we are creating the ASCII art, and we call our convertImageToAscii function with all of the proper inputs so that we can get aimg, our list of strings. As a note, args.moreLevels will only ever be True or False, which is what is getting passed in to the function. If you like, go back to the convertImageToAscii function and see where each of the variables are used now that we have them from the user. We're almost done, we just need to write the output to a text file so we can look at our art!

Step 10:

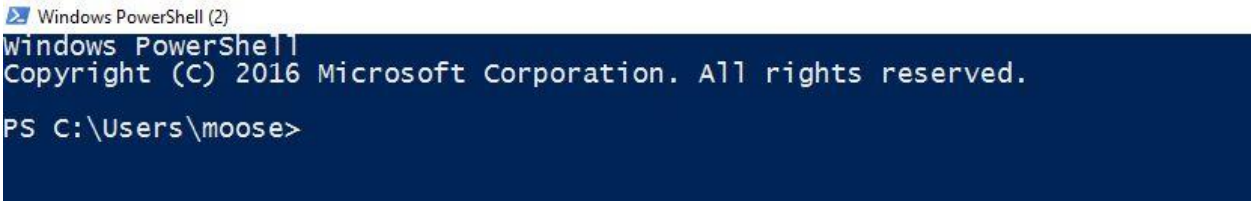
In the final step of the lab, we are looking to fill in some values so that we can write our list of strings, aimg, to a text file and look at our ASCII art that we've worked so hard for.

```
# Step 10 ------------------------------------------------------------------------
# open file
f = open(filename, 'w')
# write to file
for ___ in ____:
    f.write(___ + '\n')
# cleanup
f.close()
print("ASCII art written to %s" % outFile)
```

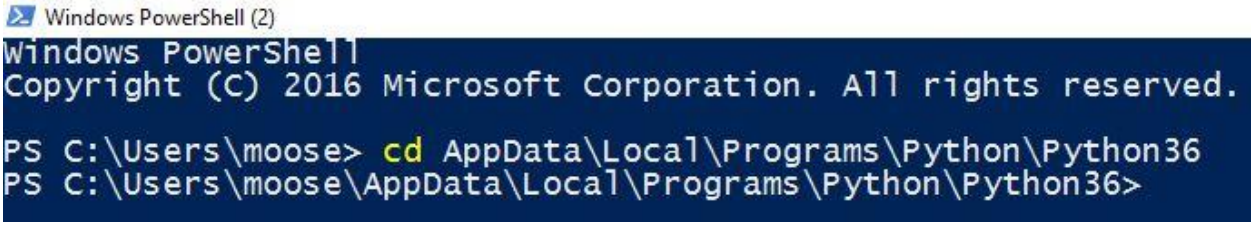
We have to replace "filename" with the name of the file we want as our output; this was stored as a variable in the previous step. We are using a built in function called open(), and we have to specify a filename and a mode to open the file in. We are specifying 'w' for write mode. Being in write mode means that even if the file we want to open does not exist, Python will just create a new one and write the contents to that file instead. If the file does exist, it will be completely overwritten and all the old file contents will be gone, so make sure you don't use the name of a text file you already have that you'd like to keep. Next, we use a bit of a shortcut to write the contents of aimg to our file. Since aimg is a list, we can iterate through it with a for loop to take each row, append a newline character to it, and then write it to our text file. This makes sure that each row is on a separate line of the text file and that everything will look normal when we open the file. Replace the underscores with appropriate variable names (Hint: one of the underscores should be aimg, and two of the underscores will be the same variable – I recommend using row as the variable but you can use whatever you like). Lastly, we close the file we opened and print a message to let the user know where the ASCII art was stored. And there you have it, a fully functioning ASCII art generator! The only thing left to do is show you how to run your program and open your ASCII art.

Part Five: Running the Script and Viewing your Art

Up until now, the command line has been mentioned repeatedly, so we're going to go through how to use it to run your Python script and open the .txt file generated. These instructions will cover how to do so on a Windows machine, and as a result we will be using Windows Powershell. To open Powershell, go to the start menu and type in powershell; it is automatically installed on all Windows laptops so it should be the first thing to pop up. Once you open Powershell, you should see a window like this one:



This is the prompt for the command line. It is telling us that currently, powershell (PS) is in the C:\Users\moose directory (this is from my computer, which I named moose, but you will likely see something different. That is okay, so long as your prompt is similar to mine. If not, raise your hand and someone will come check what the problem is.). We want to change directories to where our Python was installed. The path is rather long, but it should look like the one below:



IMPORTANT NOTE: On my computer, the final folder in the path I want to use is Python36, but for your computer it will likely be Python36-32. It does not matter if your folder is different, but it has to be the exact file path for this to work.

Here, we used the cd command (which stands for change directory) to go to the file path specified. Now that we are in the proper directory, we can run our python script directly from the command line. You'll want to enter the following command, which will ONLY work if you have completed the rest of the lab up to this point.

As the image is small, the command is:

./python ascii_lab.py –file C:\gandalf.png –cols 600

A few things to point out. First, we are saying we wish to run a python script, and so we call the python command first. The ./ specifies that we wish to use the python installed in this directory, which avoids us accidentally using a different version of Python if there is more than one version installed on the computer. Next, we specify the name of the python script we want to run, which should always end in a .py extension. I've specified the file, which is using an image stored directly on my C drive called Gandalf.png. Your image path does not have to be the same, in fact it should be the path to wherever you saved the image you want to convert in to ASCII art. Next, I've specified I want the number of columns to be 600. This is a value that worked well for the Gandalf photo I used, but you should experiment with different values to see what works best for your

photo. Remember that we can also specify –morelevels, the name of the output file –outFile, and the scale we want to apply to our tiles –scale. You can play around with these values as much as you like, the program will simply overwrite the old text file everytime you create new ASCII art (so make sure to save any art you really like first!). If your code is correct, and you entered everything correctly on the command line, you should see something like this:

```
generating ASCII art...
input image dims: 1024 x 430
cols: 600, rows: 108
tile dims: 1 x 3
ASCII art written to out.txt
PS C:\Users\moose\AppData\Local\Programs\Python\Python36>
```

We just have one final step, opening the text file! Now, we could use the file explorer to navigate to that path and open the file manually, but that is a long path and that sounds like a lot of work. One of the most useful things about the command line is that it lets us do just about anything we could normally do on our computers by typing out commands instead of having to click through multiple folders. We are going to use a command “ii”, which is short for Invoke-Item. This will open a file in the specified path, if there is a file to be opened. My command line looked like this:

```
ASCII art written to out.txt
PS C:\Users\moose\AppData\Local\Programs\Python\Python36> ii $pwd\out.txt
```

First, the ii command to open the file. Then, we have to specify where the file is. The output file will always be in the same directory as our python installation, and we know we are already in the same directory as our python version (we had to be here to run the code). As a result, we can take a shortcut, and instead of having to type out C:\Users\moose\AppData\Local\Programs\Python\Python36 again, we can simply say $pwd for short. This uses another command, print working directory (pwd) which prints out our current path. We put the dollar sign in front of it to specify that we want to use it as a variable, and then we add \out.txt at the end so that Invoke-Item knows what file to open. If you made it this far, everything should be correct in your code and we should be in the correct directory, and so you should see a notepad file pop open on your screen containing your ASCII art! One tip, usually the font size in notepad defaults to 12, but you should change it to something much smaller like 3 or 4 by going to Format > Font at the top of the notepad file. This will make it easier to fit your ASCII art onto one screen.

There you have it, you’ve just created an ASCII art filter that should work on just about any image (with some tweaking to the variables in the script of course). The principles you learned here can be applied to many image processing problems, and you may find yourself using them in the future whether its for a school project or for a hobby you work on from time to time. For now, pick an image you want to convert and see what you can do with the filter!