Assignment 2: The Quest for the Holy Grail

Due: Thursday, October 29th at 11:59:59PM

Overview

Congratulations! You have embarked on a quest to find the famous Holy Grail. Along the way, you will have to write various Python scripts to assist you in your quest. I know you're up to the challenge. Your quest will draw on everything you have learned thus far, and may even take you around Stanford campus.

Godspeed.

Logistics

Download the starter files and use some of the hints below to solve three puzzles and make progress towards the larger meta-puzzle!

Puzzle Guidelines

Text Processing for the White Order

We've given you a `words.txt` file that has some peculiar modifications compared to the original `/usr/share/dict/words` dictionary that comes with Unix platforms. In particular, you'll notice that the first line of the file is the start of an incomplete bit.ly url. Finish out this bit.ly url by doing some text processing on the `words.txt` file.

To start you off with a hint, Monty Python battled a Black Knight in his quest for the Holy Grail. But for your quest, you'll need to investigate the White Order...

When you solve this quest, make sure to retain your solution to make progress towards the final meta puzzle!

Karel.py and the Ministry of Silly Walks
In this subproblem, you will be programming a python plugin that will allow Karel to join Monty Python's Ministry of Silly Walks. This plugin will give Karel the advanced ability to precompute a final location and direction based on a sequence of commands.

Karel will start facing North at the coordinates (0,0). You'll want to return a tuple containing an x,y coordinate tuple and a cardinal direction (N, NE, E, SE, S, etc…)

As input, this plugin accepts a single line of input, a string consisting of a series of letters:

- **A:** Karel rotates 45 degrees counterclockwise
- **C:** Karel rotates 45 degrees clockwise
- **S**: Karel steps in whichever direction he is facing

As output, this plugin returns an x,y tuple and a direction that Karel ends up in.

As examples, consider the following test cases:

Input	**Output**
S	(0,1), N
CS	(1,1), NE
AASA	(-1,0), SW
SSCSSASCSSSASSCCC	(5,10) SE

After verifying that your program works for the above test cases, you'll want to run it on our `ministry-silly-walks.txt` file that contains two such strings that will give you two separate coordinate pairs and directions.

To obtain the piece towards the meta-puzzle, add the x and y coordinates together obtain a pace count that should be used in tandem to the final cardinal direction that you obtained.
e.g. (10, 15) SE should result in 25 paces SE

Disclaimer: Karel may or may not have the same pace length that you do, so treat the number you calculate to be approximate!

Image Reassembly
You're almost there! After many quests, you've finally obtained a map to the Holy Grail! The journey is almost complete.

Unfortunately, during your post-quest celebration, Sir Belvedere accidently threw the map into the paper shredder! You really should fire that knight. And why did you bring a paper shredder on your travels anyway? Something about medieval taxes… Nonetheless, you will need to reconstruct the map to find your final destination.

Your objective for this sub-puzzle is to reconstruct an image from a collection of vertically shredded images. Specifically, you will be given a directory of k (m x n) images, and will need to reconstruct them into one (km x n) image. The vertical shredded images are mutually exclusive and collectively exhaustive - that is, they represent the entire image but they don't overlap.

We have provided a general utility library to interact with image files, called image.py, which exports some useful functions:

```
load_image(filename)
      """Returns a 2D array of pixels in column-major order
      from the image specified by filename which should include
      the full filepath."""

save_image(data, filename)
      """Saves a PNG of the image data to disk
      under the given filename"""

show_image(data)
      """Opens a preview of the image data
         -  useful for debugging"""

show_all_images(slice, *rest, buffer_width=1)
      """Draws all vertical slices in one image
      with a black buffer between slices buffer_width pixels wide
         -  Useful for debugging"""

files_in_directory(dirname)
      """Returns a list of filenames in the given directory"""
```

In order to use these functions in your program, you will need to include the following line at the top of your file:

```
from image import load_image, save_image, show_image,
show_all_images, files_in_directory
```

Note: We have adopted the convention that images will be represented in column major order - that is, the first element (at index 0) of the 2-dimensional array returned by load_image is a list representing the first column of the image, *not* the first row. Keep this in mind while writing your code.

Warning: The data you'll be working with in this problem represent very large lists, potentially on the order of 1000s x 1000s. As such, please don't try to print all of your image data to the console at once, or else your computer will freeze. You can print out slices of the data (for example, the first ten pixels in a given column) for debugging purposes, but be aware that these data structures are very large. If you need to regain control from a runaway expression, you can enter CTRL+C in the interactive interpreter. If that fails, quit Python and relaunch.

Each pixel is just a 4-tuple (R, G, B, A), where the four values respectively represent the intensity of the red, green, blue, and alpha channels.

General Algorithmic Hints
A reasonable approach is to look at all pairs of shredded image slices, decide which pair matches up best, and glue those two slices together into a slightly bigger slice. Rinse and repeat until there is just one piece left - it's the reconstructed image!

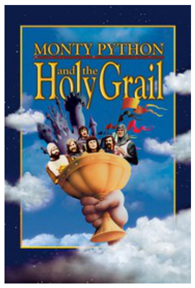
How do we determine the similarity between two vertically shredded image slices? There are many different approaches, but one that seems to work is to look at the rightmost column of the left piece and the leftmost column of the right piece to see how well they match up.

How can we decide how well two columns match up? We can look at pixels componentwise. If that's not quite enough, we can look around a given pixel and incorporate the values of its neighbors. The details are up to you.

How similar are pixels? Again, there are many different ways to approach this problem, each of which yields varying degrees of success. One approach that works well is to sum the squares of the absolute difference between each color channel.

Starter Files

```
image-starter/
      ├── image.py
      ├── shredded
      │   ├── grail4/
      │   ├── grail20/
      │   └── destination/
      └── PIL/
```



In addition to the `image.py` utility file, you will find a number of vertically shredded images in `shredded/`. The images in `grail4/` represent an image (seen at right) that has been shredded into four pieces, each of which is wide enough to reassemble by hand. The images in `grail20/` represent the same image, sliced into 20 pieces.

Additionally, you will see a folder called `PIL/`. This folder is a third-party library for image-processing - please don't modify it.

If you want additional shredded images to play around with, consider building your own, or email us for another sample.

Challenge
Ultimately, your goal (and success in this part) will be to reassemble the slices in the `destination/` folder. These image slices are a lot larger than the in `grail4/` or `grail20/`, and there are a lot more of them, so you will need to think about efficiency.

This is a hard problem, so please post questions on Piazza if you are stuck. Furthermore, we want to remind everyone that it's okay to not finish this problem if you're spending an exorbitant amount of time and effort. You can also come visit us during office hours for more help.

For reference, our solution is about 50 lines of code, not counting comments or whitespace.

Grading

Style (50%)

Your style grade is determined by the quality and reusability of your submitted code. We know that there are many ways to solve each of the challenges, so spend time thinking about the best approach before beginning.

Functionality (50%)

Your functionality grade is determined by your progress in the Quest. Each challenge is worth the same number of points, so if you complete the quest successfully you are guaranteed full functionality credit. If you complete only half of the challenges, you will receive half-credit for functionality.

Submission Guidelines

Please zip up the folder with our starter files along with any code that you wrote and submit it to our submission portal. As a friendly reminder, please help us out by naming your zipped folder <SUNetID>-assign2.zip before submitting it.

Bonus

Mystery Function

As an optional addendum to this assignment, you can attempt to glean a useful hint towards the meta-puzzle by solving this last sub-problem.
We've given you a compiled python file, `mystery.pyc` that can be imported into the python interpreter or your own `.py` script. This compiled script contains a `mystery` function that takes an unknown number and type of parameters. You'll want to use hints given by the python compiler and by the function itself to figure out what to pass into this mystery function to obtain the solution to this puzzle.

To work within the Python interpreter, you'll want to execute the following lines from the same directory that contains `mystery.pyc`:

```
python3
>>> from mystery import mystery
>>> mystery()
```

You’ll likely want to use the interpreter to figure out what to pass into the mystery function, but it may prove advantageous to start working within your own `.py` file once you figure out what variables you want to pass into the function.

To import the mystery function into a file, you’ll need to include the following line at the top of your file:

```
from mystery import mystery
```

If you get stuck, post general inquiries on Piazza. If you’re blocked on a particular hint given by the compiled python file, please send us a private note on Piazza!

Credit

Inspiration for this assignment comes from the fantastic 1975 British Masterpiece, Monty Python and the Holy Grail. We’d also like to credit Salesforce IQ Coding Challenge for our subpuzzle in karel.py and Nick Parlante’s Image Puzzle assignment.