Assignment 6: Images

(Source: Dept. of C.S., University of Cornell)

Submission deadline: April 16, 2023

In this assignment, you will learn how to make your own image filters.

An image is just a 2-dimensional list of pixels (which are themselves RGB objects). So most of the functions/methods that you write in this assignment will involve nested for-loops that read a pixel at a certain position and modify it. You will need to use what you have learned about multi-dimensional lists to help you with this assignment.

One major complication is that graphics cards prefer to represent images as a 1-dimensional list in a **flattened presentation** instead. It is much easier for hardware to process an 1-dimensional list than a 2-dimensional one. However, flattened presentation (which we explain below) can be really confusing to beginners. Therefore, another part of this assignment is learning to use classes to *abstract* a list of pixels, and present it in an easy-to-use package.

Finally, this assignment will introduce you to a Python *package*. While you are working on a relatively small section of the code, there are a lot of files that go into making this application work. Packages are how Python groups together a large number of modules into a single application.

Learning Objectives

This assignment is designed to give you practice with the following skills:

- How to implement a class from its interface.
- How to enforce class invariants.

- How to use classes to provide abstractions.
- How to write code for both 1-dimensional and 2-dimensional lists.
- How to manipulate images at the pixel level.
- How to write code for an underspecified function or method.
- How to program for unicode strings with foreign characters and emojis.

Academic Integrity and Collaboration

Any mutual discussions ae strongly discouraged. Please contact the instructor during contact hours for any help. Do not post your code to Pastebin, GitHub, or any other publicly accessible site.

Collaboration Policy

This assignment is to be done independently.

This assignment requires you to implement several parts before you have the whole application working. The key to finishing is to pace yourself, and make use of both the unit tests and the visualizer that we provide.

Assignment Source Code

To work on this assignment, you will need to download three files.

File	Description
imager.zip	The application package, with all the source code
samples.zip	Several sample images to test in the application
outputs.zip	The result of applying the filters to the sample images

You should download the folder `imager` from the link above and *put the contents in a new directory*. This folder contains a lot of files. You do not not need to understand most of these files. The are similar to `a3app.py` (Assignment 3) in that they provide the GUI interface for the application.

You only need to pay attention to the files that start with `a6`. There are five of these. Two are completed and three are only stubs, waiting for you to complete them.

File	Description
a6image.py	The `Image` class, to be completed in Task 1
a6editor.py	The `Editor` class, which is completed already
a6filter.py	The `Filter` class, to be completed in Task 2
a6encoder.py	The `Encoder` class, to be completed in Task 3
a6test.py	The test script for the assignment, which is completed already

You should skim all of these files before continuing with the assignment instructions.

The Imager Application

Because there are so many files involved, this application is handled a little differently from previous assignments. To run the application, keep all of the files inside of the folder `imager`. Do not rename this folder. To run the program, change the directory in your command shell to just *outside of the folder* `imager` and type

```
python imager
```

In this case, Python will run the *entire folder*. What this really means is that it runs the script in `__main__.py`. This script imports each of the other modules in this folder to create a complex application.

Right now, this application will not do anything. However, once you complete the `Image` class, it will display two images of Prof. Walker White, like this:

As you work with this application, the left image will not change; it is the original image. The right image will change as you click buttons. The actions for the buttons Invert and Rotate.., are already implemented. Once you have completed the `Image` class, you can click on them to see what they do.

You will notice that this takes a bit of time. The default image is 512x512. This is over 250 thousand pixels. The larger the image, the slower it will be. With that said, if you ever find this taking more than 30 seconds, your program is likely stuck and has a bug in it.

The effects of the buttons are cumulative. You can undo the last effect applied with Image.. Undo. To remove all of the effects, choose Image.. Reset. This will revert the right image to its original state.

You can load a new image at any time using the Image.. Load button. Alternatively, you can start up with a new image by typing

```
python imager myimage.png
```

where `myimage.png` is your image file. The program can handle PNG, JPEG, and GIF (not animated) files. You also can save the image on the right at any time by choosing **Image.. Save**. You can only save as a PNG file. The other formats cause problems with **Part 3** of the assignment.

The remaining buttons of the application are not implemented. **Reflect.. Horizontal** works but the vertical choice does not. In **Part 2** of the assignment, you will write the code to make them work.

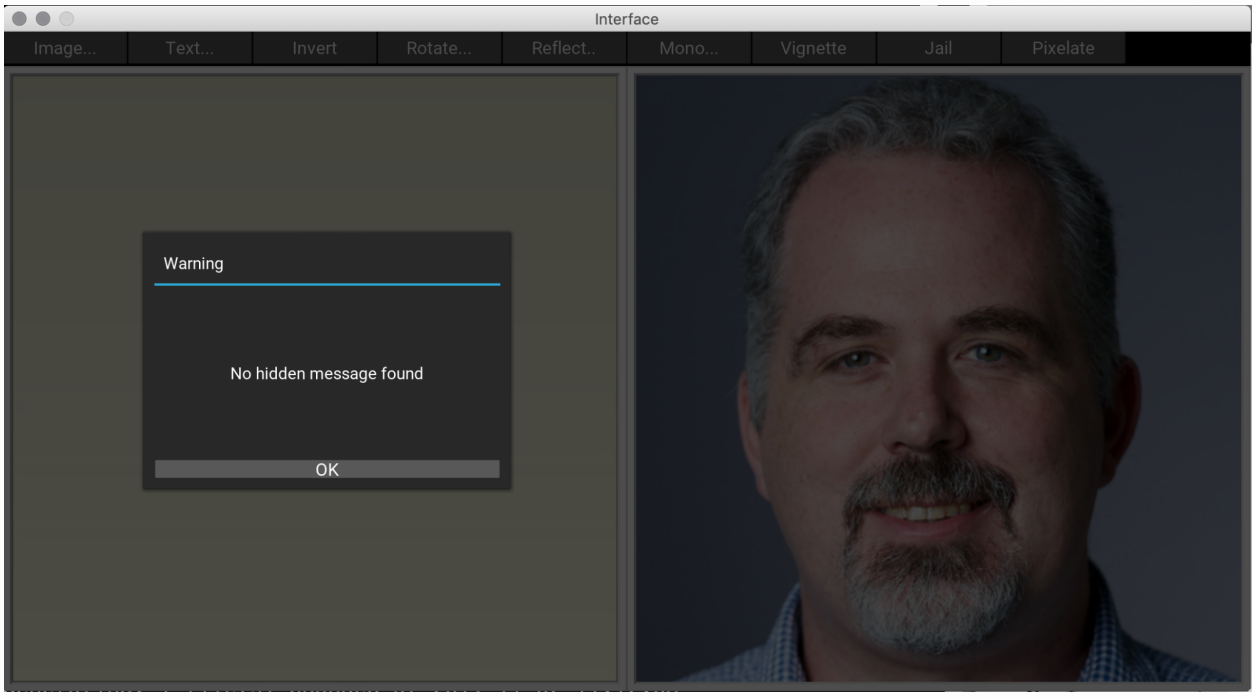
If you are curious about how this application works, most of the code is in `interface.py` and `filter.kv`. The file `filter.kv` arranges the buttons and colors them. The module `filter.py` contains Python code that tells what the buttons do. However, the code in this module is quite advanced and we do not expect you to understand any of it.

The Encoder Features

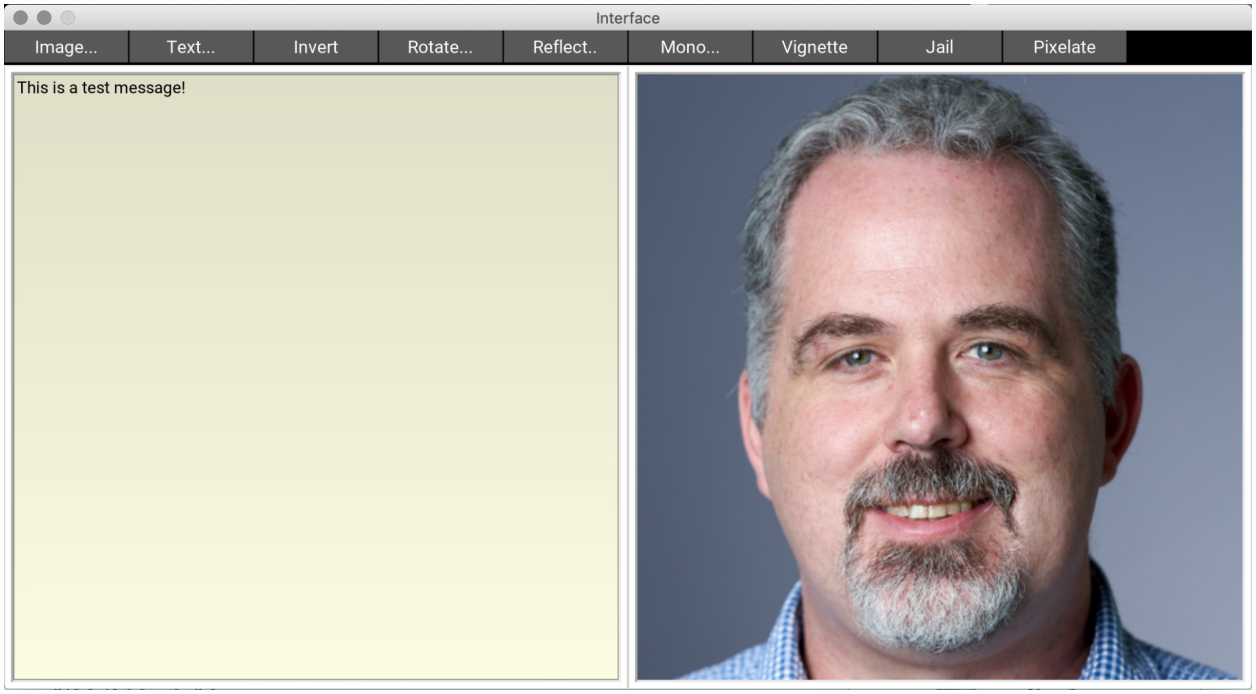
Next to the **Image..** button, you will see a button for **Text..**. This button is used by the **last part** of the assignment, to store secret messages in text. To access these features, choose **Text.. Show**.

When you do this for the first time, you will see an error message that says **"No hidden message found"**. This is perfectly normal. You have not encoded anything yet, so there

is nothing to decode. As a rule, the application will always decode the message hidden in the file at start-up, and you will see that error if there is no message.

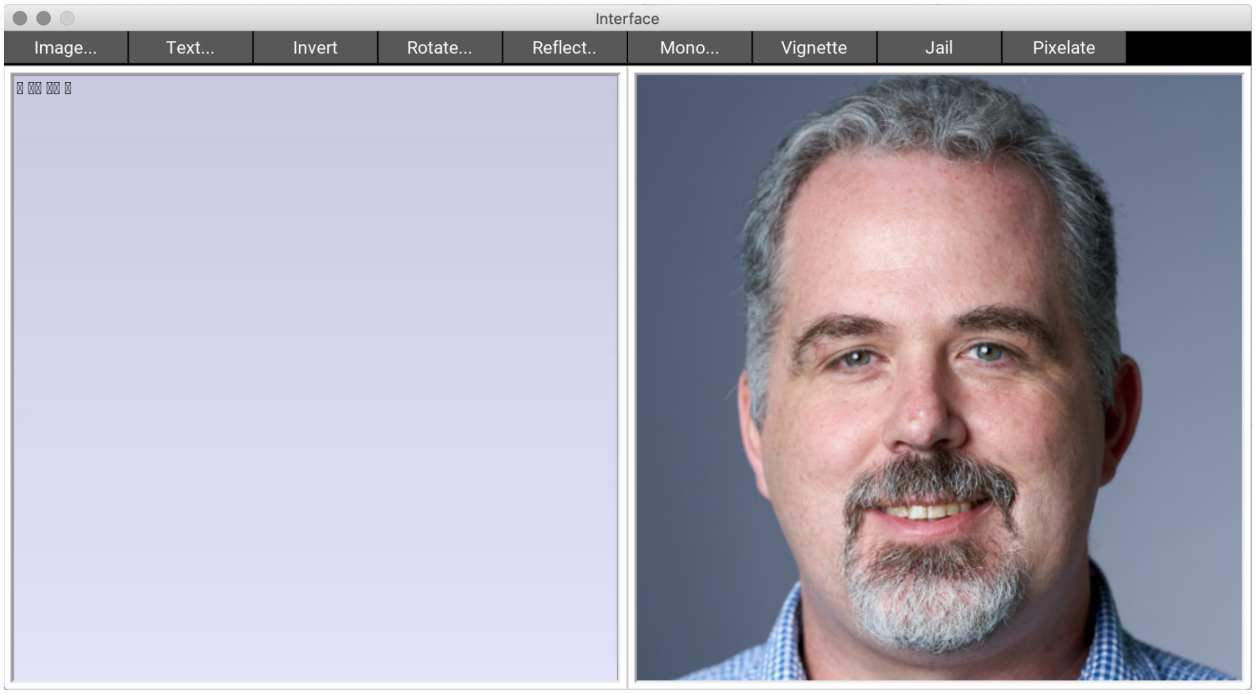


To encode a message (once you have completed **Part 3**), type text in the box on the left. The box will change color to blue if you have unsaved changes. Once you press the **Encode** button, it will write the message inside the image and change color back. At this point, you will probably want to save the image to a PNG file to decode later. Applying any of the image filters (invert, reflect, etc.) will corrupt the hidden message and cause it to be lost.



Typing text in the box is a lot of work. Therefore, you can always read from a text file (or even a .py file). If you chose **Text.. Import**, this will load the file into the box on the left. However, it will not encode the text until you chose to do so. Hence, the text box will be blue after the file is loaded. Similarly, you can save the decoded text to a file with **Text.. Export**. These file features will be very useful for debugging more complex messages.

As we describe in the instructions below, your encode and decode operations will support full Unicode (Asian characters, emojis, etc.). However the font that Kivy uses cannot support emojis; it will display them as boxes with a cross in them (to indicate "missing"). For example, if you import **this text** into the application, you will see this



But if you export that text to a file, and open it up with an editor that can display emojis, it will look correct. If you want to encode emojis, write them in a text editor, import them, and encode. To get emojis back from a message, you decode and export.

The undo functionality works for all of these features as well. Choosing to undo will remove the most recently encoded message, restoring to a previously encoded message.

The Integrated Test Script

As with A4 (**Turtles**), debugging everything visually can be tricky. That is why we have provided you with a (partial) test script to help you with this assignment. This test script is integrated into the Imager application. To run it, type

```
python imager --test
```

The application will run test cases (provided in `a6test.py`) on the classes `Image`, `Filter`, and `Encoder`, in that order. This is incredibly useful, since you cannot even use the Imager app until you finish the `Image` class.

These test cases are designed so that you should be able to test your code in the order that you implement it. Howevever, if you want to "skip ahead" on a feature, you are allowed to edit `a6test.py` to remove a test. Those tests are simply there for your convenience.

This test script is fairly long, but if you learn to read what this script is doing, you will understand exactly what is going on in this assignment and it will be easier to understand what is going wrong when there is a bug.

As with the Turtles assignment, **this test script is not complete**. It does not have full coverage of all the major cases, and it may miss some bugs in your code. It is just enough to ensure that the GUI application is working correctly. Therefore, you may want to add additional tests as you debug. With that said, we do not want you to submit the file `a6test.py` when you are done, even if you made modifications to the file.

Assignment Instructions

There are so many parts to the Imager application that this assignment can feel very overwhelming. But in these instructions we take you through everything step-by-step. As long as you pay close attention to the specifications, you should be able to complete everything. This assignment may take longer than the others, but it is well within your ability.

Task 0: Pixel Representation

You do not ever need to worry about writing code to load an image from a file. There are other modules in `imager` that handle that step for you. Those modules use the **PIL**

module to extract pixel data from a file. The functions in this module return the image as a *flattened list of pixels*.

To understand what we mean by this, let's talk about pixels first. A pixel is a single RGB (red-green-blue) value that instructs you computer monitor how to light up that portion of the screen. Each RGB component is given by a number in the range 0 to 255. Black is represented by (0, 0, 0), red by (255, 0, 0), green by (0, 255, 0), blue by (0, 0, 255), and white by (255, 255, 255).

In previous assignments, we stored these pixels as an `RGB` object defined in the `introcs` module. These were mutable objects where you could change each of the color values, and these objects would automatically enforce the 0..255 invariant. However, the pixels in this assignment will be 3-element tuples of integers. That is because they are faster to process, and Kivy prefers this format. Because image processing is slow enough already, we have elected to stick with this format. In addition, this means that you get some experience checking and enforcing that the pixels are in the correct format.

So if that is what we mean by a pixel, what is a "flattened list of pixels"? We generally think of an image as a rectangular list of pixels, where each pixel has a row and column (indicating its position on the screen). For example, a 3x4 pixel art image would look something like the illustration below. Note that we generally refer to the pixel in the top left corner as the "first" pixel.

However, graphics cards really like images as one-dimensional list. One-dimensional lists are a lot faster to process and are more natural for custom hardware. So a graphics card will flatten this image into the following one-dimensional list.

If you look at this picture carefully, you will notice that is it is very similar to *row-major order* introduced in class. Suppose we represented the 3x4 image above as follows:

```
Unset
E00  E01  E02  E03

E10  E11  E12  E13

E20  E21  E22  E23
```

The value `Eij` here represents the pixel at row `i` and column `j`. If were were to represent this image as a two-dimensional list in Python, we would write.

```
[[E00, E01, E02, E03], [E10, E11, E12, E13], [E20, E21, E22, E23]]
```

Flattened representation just removes those inner brackets, so that you are left with the one-dimensional list.

```
[E00, E01, E02, E03, E10, E11, E12, E13, E20, E21, E22, E23]
```

Precondition Enforcement

Throughout this assignment, you will be asked to enforce preconditions. A common precondition that will come up over and over again is that a value is a pixel, or a value is a pixel list. Inside of the file `a6image.py` are two helper functions to help you enforce these preconditions: `_is_pixel` and `_is_pixel_list`. The first has been completed for you. The second is unfinished.

Before you do anything else, complete the function `_is_pixel_list`. Despite the fact that this is a hidden function, we do test it in `a6test.py`. So you should run the **test script** to verify that your implementation is correct.

Task 1. The `Image` Class

For some applications, flattened representation is just fine. For example, if you want to convert an image to greyscale, you do not need to known exactly where each pixel is inside of the file. You just modify each pixel individually. However, other effects like rotating and reflecting require that you know the position of each pixel. In those cases you would rather have a two-dimensional list.

The `Image` class has attributes and methods that allow you to treat the image either as a flattened one-dimensional list or as a two-dimensional list, depending on your application. This is what we mean by an *abstraction*. While the data is not stored in a two-dimensional list, methods like `getPixel(row,col)` allows you to pretend that it is.

The file `a6image.py` contains the class definition for `Image`. This class is fully specified. It has a class specification with the class invariants. It also has specifications for each of the methods. All you have to do is to write the code to satisfy these specifications.

As you work, you should run the **test cases** to verify that your code is correct. To get the most use out of the testing program, we recommend that you implement the methods in the same order that we test them.

The Initializer, Getter and Setter Methods

To do anything at all, you have to be able to create an `Image` object and access the attributes. This means that you have to complete the initializer and the getters and setters for the three attributes: data, width and height. If you read the specifications, you will see that these are all self-explanatory. Note that these attributes are hidden, so the class invariant is given by (hidden) single line comments.

The only challenge here is the width and height. Note that there is an extra invariant that the following must be true at all times:

```
width*height == # of pixels
```

You must ensure this invariant in both the initialiers and the setters. In addition, we expect you to enforce all preconditions with asserts.

The One-Dimensional Operators

The getter `getData` already returns the image data as a flattened list of pixels. So you might think we do not need to do anything more here. However, notice that `getData` returns a copy of the pixel list. So it is not useful if you want to *modify* the image. Instead, the class `Image` has methods to allow modification of the image, while still enforcing the class invariant.

The methods for one-dimensional access are special methods (discussed in class) that begin and end with double-underscores. The methods to implement are `__len__`, `__getitem__` and `__setitem__`.

As a reminder, recall that `__len__` is a helper method for the `len` function. So the following two lines of code are identical.

```
x = image.__len__()
```

```
x = len(image)
```

The `__getitem__` method allows you to use square brackets to access a pixel in an `Image` object (just one pixel, not a slice). Once you implement it, the following two lines of code are identical.

```
x = image.__getitem__(i)
```

```
x = image[i]
```

Finally, the `__setitem__` method allow you to use brackets on an `Image` object to replace a pixel. Once you implement it, the following two lines of code are identical.

```
image.__setitem__(i,p)
```

```
image[i] = p
```

To see more, look at the tests in `a6test.py`.

The code for all of these methods is incredibly simple. For each method you just have one line to access or update the `_data` attribute. So why have these methods? To enforce the preconditions, of course. A simple list does not care whether its contents are valid pixels. But this is required by the class invariant and you must enforce this to prevent the user from adding invalid data to the list (do not assert anything about `_data`; just assert the preconditions).

The Two-Dimensional Methods

The `getPixel` and `setPixel` methods present the image as a two-dimensional list. This is a little trickier. You have to figure out how to compute the flat position from the row and column. Here is a hint: it is a formula that requires the row, the column, and the width. You do not need anything else. Look at the illustrations in our discussion of **flattened representation** to see if you can figure out the formula.

Figuring out the conversion formula is the only hard part of this exercise. Otherwise it is the same as for the one-dimensional operators. Make sure to enforce all of the preconditions.

The `__str__` Method

This method is arguably the hardest one in the entire `Image` class. We want you to create a string that represents the pixels in two-dimensional row-major order. As a hint, this is a classic for-loop problem. You will need an accumulator to store the string, and you will build it up via concatenation. You will also want to use previous methods that you wrote as helpers.

You might think that all you need to do is to accumulate the pixels into a two-dimensional list and convert that list to a string. This would be correct **except** for the newlines between rows. You need to be a little more clever here. Again, use the the **test script** to verify that your code is correct.

The Remaining Methods

You will notice that there are two other methods in class `Image`: `swapPixel` and `copy`. We need these methods for the filtering application, and we have completed them for you.

Task 2. The `Filter` Class

The module `a6filter.py` contains the `Filter` class. You will notice that it is a subclass of the `Editor` class in `a6editor.py`. The `Editor` class is complete; you do not have to do anything with this class. It implements the **Undo** functionality in the `imager` application. This class implements an edit history and the getter `getCurrent` accesses the most recent update of the image.

You do not need to understand the `Editor` class at all, but you should read its specification. Since `Filter` is a subclass, it will need to access the inherited methods from `Editor`. In particular, you will notice that *none* of the methods in `Filter` take an image as an input. Instead, those methods are to work on the current image, which they access with the method `getCurrent`.

To make it easier to follow all this, we have provided you with several example methods to study. You will notice that some filters – like `invert` – modify the image with the one-dimensional operators. Others – like `transpose` and `reflectHori` – modify the image with the two-dimensional operators. Use this code as a guide for implementing the unfinished methods.

While working on these methods, you may find that you want to introduce new helper methods. For example, `jail` already has a `_drawHBar` helper provided. You may find that you want a `_drawVBar` method as well. This is fine and is actually expected. However, *you must write a complete and thorough specification of any helper method you introduce*. It is best to write the specification before you write the method body, which is standard practice in this course. It is a **severe error not to have a specification, and points will be deducted for missing or inappropriate specifications**.

We have provided you with several **test cases** for these filters. But the output of these test cases are limited and not always useful. A better way to test is just to load the sample images and try them out. We have provided you with the **correct outputs** for each filter applied to each sample image.

Method `reflectVert`

This method should reflect the image about a horizontal line through the middle of the image. Look at the method `reflectHori` for inspiration, since it does something similar. This method should be relatively straightforward. It is primarily intended as a warm-up to give you confidence.

Method `monochromify`

In this method, you will change the image from color to either grayscale or sepia tone. The choice depends on the value of the parameter `sepia`. To implement this method, you should first calculate the overall brightness of each pixel using a combination of the original red, green, and blue values. The brightness is defined by:

```
brightness = 0.3 * red + 0.6 * green + 0.1 * blue
```

For grayscale, you should set each of the three color components (red, green, and blue) to the same value, `int(brightness)`.

Sepia was a process used to increase the longevity of photographic prints. To simulate a sepia-toned photograph, darken the green channel to `int(0.6 * brightness)` and blue channel to `int(0.4 * brightness)`, producing a reddish-brown tone. As a handy quick test, white pixels stay white for grayscale, and black pixels stay black for both grayscale and sepia tone.

To implement this method, you should get the color value for each pixel, recompute a new color value, and set the pixel to that color. Look at the method `invert` to see how this is done.

Method `jail`

Always a crowd favorite, the `jail` method draws a a red boundary and vertical bars on the image. You can see the result in the picture below. The specification is very clear about how many bars to create and how to space them. Follow this specification clearly when implementing the function.

We have given you helper method `_drawHBar` to draw a horizontal bar (note that we have hidden it; helper functions do not need to be visible to other modules or classes). In the same way, you should implement a helper method `_drawVBar` to draw a vertical bar. Do not forget to include its specification in your code.

This method is one where you have to be *very careful with round-off error*, to make sure that the bars are evenly spaced. You need to be aware of your types at all times. The number of bars should be an integer, not a float (you cannot have part of a bar).

However, the distance between bars should be a float. That means your column position of each bar will be a float. Wait to turn this column position into an `int` (by rounding and casting) until you are ready to draw the bar.

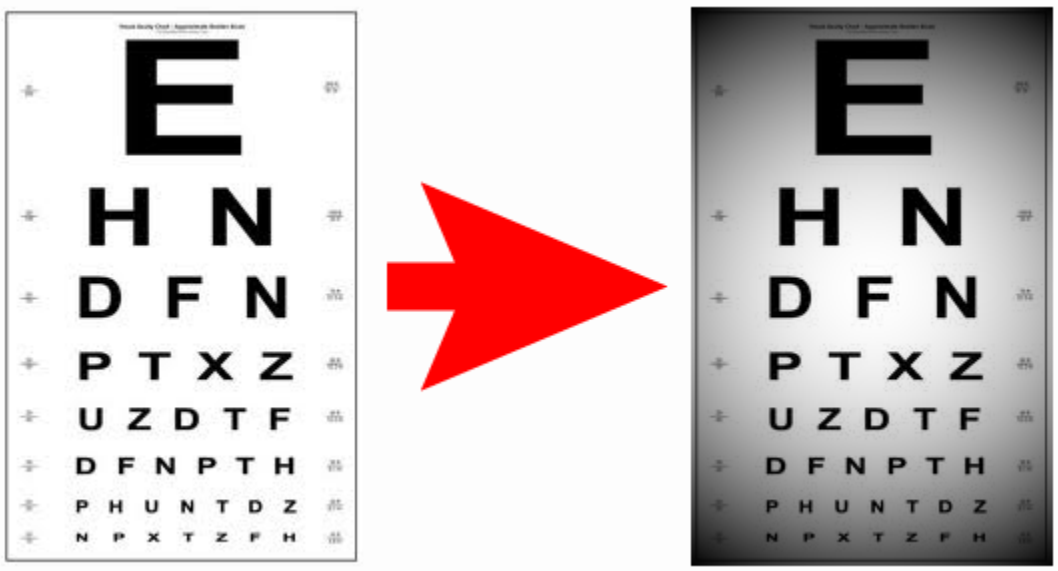
When you are finished with this method, open a picture and click the buttons **Jail**, **Transpose**, **Jail**, and **Transpose** again (in that order) for a nice effect.

Method `vignette`

Camera lenses from the early days of photography often blocked some of the light focused at the edges of a photograph, producing a darkening toward the corners. This effect is known as `vignetting`, a distinctive feature of old photographs. You can simulate this effect using a simple formula. Pixel values in red, green, and blue are separately multiplied by the value

$$1 - d^2/h^2$$

where `d` is the distance from the pixel to the center of the image and `h` is the distance from the center to any one of the corners. The effect is shown below.



Like monochromification, this requires unpacking each pixel, modifying the RGB values, and repacking them (making sure that the values are ints when you do so). However, for this operation you will also need to know the row and column of the pixel you are processing, so that you can compute its distance from the center of the image.

For this reason, we `highly` recommend that you use the method `getPixel` and `setPixel` in the class `Image`. These methods treat the image as a two-dimensional list. Do not use the one-dimensional operators. That was fine for `invert` and `monochromify`, but that was because the row and column did not matter in those methods. Take `h` to be the distance from the center to (0,0).

Task 3. The `Encoder` Class

The last official part of the assignment is the most involved. The `Encoder` class is built on top of the `Filter` class. That is because it is adding new functionality to the `Filter` class and it needs access to the current image via `getCurrent`. In fact we could have combined these two classes, but we separated them for reasons of readability.

This class allows us to support steganography, which is "the art and science of writing hidden messages in such a way that no one apart from the intended recipient even realizes there is a hidden message." This is different from cryptography, where the existence of the message is not disguised but the content is obscured. Quite often, steganography deals with messages hidden in pictures.

This task is much more open-ended than anything we have done in the course before. There is no right way to hide a message, and the way that you choose to hide messages might be different than ours. All that matters is your `decode` method can extract messages hidden by your `encode` method.

To decide how to best hide (and reveal) messages, you should read all of the instructions below before starting on this class.

Encoding Bytes

A `byte` (comprising 8 bits) is an integer between 0 and 255. Computers are specifically designed to work with data in byte-sized chunks, which is why they are so common. The American Standard Code for Information Interchange (ASCII) is a way to represent Python strings as bytes. Each character corresponds to a number 0..255. You can use the function `ord` to convert a character to a byte and `chr` to convert a byte back to a character (see python documentation for more details).

It is very easy to encode a byte in a pixel. Suppose we have a pixel whose RGB values are 199, 222, and 142 and we want to store the byte 107 (which corresponds to the character `'k'`). We change the least significant digit of each color component to one of the digits of 107, as shown below.

Original Pixel				**Pixel with byte 107 hidden**		
Red	**Green**	**Blue**	**hide byte 107**	**Red**	**Green**	**Blue**
19**9**	22**2**	14**2**	→	19**1**	22**0**	14**7**

This change in each pixel is so slight that it is imperceptible to the human eye (unless the image is a rectangle of just one color).

Decoding the message, the reverse process, requires extracting the last digit of each color component of the pixel and forming a byte from the three extracted digits, then converting that byte back to a character. In the above example, we would extract the digits 1, 0, and 7 from the RBG components of the pixel (using `% 10`) and put them together to form 107, which is the ASCII value for `'k'`. Extracting the message *does not change the image*. The message stays in the image forever.

Unfortunately, all modern text is **Unicode**, not ASCII. Unicode supports all possible characters, including Asian characters and emojis. And Unicode strings are supported in Python. Try this out in the interactive shell:

```
>>> s = '😀 World!'
>>> s[0]
'😀'
```

To keep the international students from feeling left out, we will be using Unicode instead of ASCII.

Unicode characters are not represented by bytes. Emojis require much larger integers than 255. But the UTF-8 *encoding* is a simple way to convert an unicode string into a list of bytes. All strings have an `encode` method that allows you to do this conversion. Take the string `s` above and try the following:

```
>>> b = s.encode('utf-8')
>>> list(b)
[240, 159, 152, 128, 32, 87, 111, 114, 108, 100, 33]
```

You will note that the number of bytes is longer than the length of the string. In UTF-8, all ASCII characters are a single byte, but other characters can be anywhere from one to four bytes.

You can also convert a list of byte-size integers back into a unicode string. You need to use the Python `bytes` function to covert the list into a bytes-only sequence, and then use the `decode` method as follows:

```
>>> c = bytes([240, 159, 152, 128, 32, 87, 111, 114, 108, 100,
33])
```

```
>>> c.decode('utf-8')
'😀 World!'
```

With this information, you can now use the technique shown above to hide or reveal any Unicode string.

Designing an Encoding

You are to write code to hide characters of a message `text` in the pixels of an image in flattened representation, starting with pixel 0, 1, 2, … Before you write any code at all, you need to think about the following three issues and solve them.

1. Indicating an Encoding

First, you need some way to recognize that the image actually contains a message. You need to hide data in the initial pixels 0, 1, 2, … that has little chance of appearing in a real image. That way, when program detects the data in those first few pixels, it knows there is a message there. You cannot be complete sure that an image without a message does still contain that data, but the chances should be extremely small.

This beginning marker should be at least two pixels. If it is only one pixel, then we can corrupt your message by transposing the image (think about this). You can use more than two pixels, but the specification states that no more than 10 pixels may be used for encoding information other than the message text.

2. Indicating the Message Length

Next, you have to know where the message ends. You can do this in several ways. You can hide the length of the message in the first pixels in some way (how many pixels can that take?). You can also hide some unused marker at the end of the message. Or you can use some other scheme. You may assume that the message has fewer than one million bytes (e.g. the specification says that you can refuse to encode longer strings),

but you *must be prepared for a message with any sequence of bytes, including those produced by punctuation, emojis, and foreign characters.*

3. Staying within Color Bounds

Finally, the largest value of a color component (e.g. blue) is 255. Suppose the blue component is 252 and you want to hide 107 in this pixel. In this case, the blue component would be changed to 257. But this is impossible because a color component can be at most 255. Think about this problem and come up with some way to solve it. There is a way to do this *without* having to change the `_decode_pixel` helper that we have provided for you. However, you are allowed to change `_decode_pixel` if you figure out another way to do it.

As you can see, this part of the assignment is less defined than the previous ones. You get to come up with the solutions to some problems yourself.

Complete the Methods `encode` and `decode`

You should complete the body of the methods `encode` and `decode` in the class `Encoder`. These two methods should hide a message and reveal the message in the image. When you design `decode`, make sure it attempts to extract the message only if its presence is detected.

Feel free to introduce other helper methods as needed. For example, we have provided a helper method called `_decode_pixel`, which takes a pixel position `pos` and extracts a 3-digit number from it, using the encoding that we suggested above. This suggests that you might want to create a helper method called `_encode_pixel`, which encodes a number into a pixel. The exact specification of such a helper is up to you (if you start with our specification, be sure to modify it as appropriate for your code).

Note that the answers to the three problems above greatly influence how you write all of these methods. Therefore, the the specifications of these methods must include any

description of how you solved the three problems listed above. For example, the specification of `_encode_pixel` must describe how you handle the pixel overflow problem. In some case this may require modification of specifications written by us. You can change anything you want in the specification *except* the one line summary, the preconditions, and the last paragraph (the one that describes the only cases in which the method may fail).

As an aside, you will also notice that we use the operator `__getitem__` in `_decode_pixel`. That is because it is very natural to think of an image as one-dimensional list when encoding text. While an image is 2-dimensional arrangment of values, text is not. Hence, once again, we see the advantage of abstraction in `Image`, allowing us to access the data how we want for the particular application.

Test the Methods `encode` and `decode`

We have provided some simple tests (including emoji) for these methods in `a6test.py`. So you should run the provided **test script** to check your answers. However, the test script is not very useful when you have bugs and need to find them.

Debugging `encode` and `decode` can be difficult. Do not assume that you can debug simply by calling `encode` and then `decode` to see whether the message comes out correctly. Instead, write and debug `encode` fully before going on to debug `decode`.

How can you debug `encode` without `decode`? Start with short messages to hide (up to three ASCII characters, each a single byte). Use the method `getData()` in `Image` and slice off that many pixels from the list. Print these out and verify that they are what you expect.

When `encode` and `decode` are both done, try hiding and revealing a long message (e.g. 1000, 2000, or 5000 characters). This is where you really make use of the Imager application. Use the Text.. Import feature to load in a text file and try to encode that. We

highly recommend that you try to encode a Python program, as that is a good test of punctuation characters.

Save your messages

Clicking Image.. Save saves the image in the specified directory with the filename you enter. The message is saved in `.png` format, which is a lossless format. Saving in `.jpg` format would not work, because doing so tries to compress the image, which would result in less space but would also clobber your hidden message.

With `.png`, you can hide a message, save, quit, and then restart the application with the message still there. If the message contained emojis or foreign characters, they will not display in the Imager application, but you can still export the result to a file with Text.. Export. You should try to do that.

Finishing Touches and Submission

Once you have everything working you should go back and make sure that your program meets the class coding conventions. In particular, you should check that the following are all true:

- Functions are each separated by two blank lines.
- Methods are each separated by one blank line.
- Lines are short enough (~80 characters) that horizontal scrolling is not necessary.
- Docstrings are only used for specifications, not general comments.
- Specifications for any new methods are complete and are docstrings.
- Specifications are immediately after the method header and indented.
- Your name is in the comments at the top of the modules.

You will submit only three files for this assignment: `a6image.py`, `a6filter.py`, and `a6encoder.py`. Please ensure that you add your name as suffix to the submitted files. For instance, `a6image.py` should be named as `a6image_yourname.py`.

Email these files to sonika@sitare.org by the due date: Sunday, April 16. **Do not** submit the file `a6test.py`.