Assignment Guidelines:

- This assignment covers material up to Module 10.
- Submission details:
  - Solutions to these questions must be placed in files `a09q1.py` `a09q2.py` and `a09q3.py` and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page
  - Helper functions need design recipe elements but not examples and tests.
- Download the testing module from the course web page. Include `import check` in each solution file.
  - When a function produces a floating point value, you *must* use `check.within` for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than `math` and `check`.
  - You are always allowed to define your own helper functions, as long as they meet the assignment restrictions. Do **not** use Python constructs not discussed in this course(e.g. `zip`, anything with `set`s or `enumerator`s, list comprehension, commands `continue` or `break`, `with`, etc.). Use only the functions and methods as follows:
    - `abs`, `len`, `max`, `min`, `sum`, `range`, `sorted`, `ord` and `chr`
    - Any method or constant in the `math` module
    - Type casting including `int()`, `str()`, `float()`, `bool()`, `list()`, `dict()`
    - The command `type()`
    - Any basic arithmetic operation (including `+, -, *, /, //, %, **`)
    - String or list slicing and indexing as well as string or list operations using the operators above
    - Any string or list methods.
    - `input` and `print` as well as the formatting parameter `end` and method `format`. Note that all prompts must match exactly in order to obtain marks so ensure that you do not alter these prompts.
    - Abstract list functions `map` and `filter`. Recursion is also permitted.
    - Loops, specifically `for` and `while` loops.
    - Dictionaries, classes and methods.
    - File IO commands. Specifically, the commands `open` and methods `close`, `readline`, `readlines`, `write`, `writelines`
  - While you may use global *constants* in your solutions, **do not** use global *variables* for anything other than testing.
  - Read each question carefully for additional restrictions.
  - **The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.**
  - **Reminder:** This assignment is worth twice as much as other assignments and cannot be dropped.

Design Recipe

For this assignment, we will not expect you to include **examples** or **tests** in any of the problems. However we strongly urge you to test your code! Feel free to create your own tests and use some of the publicly available tests we have provided to help ensure your code is correct.

Steganography

Steganography is the practice of hiding information in other forms of data. In this multi-part problem, you will learn about a simple picture file format called "Plain PPM" or "Plain Portable Pixel Map". This file format is an uncompressed format that is easy to read as file input and interpret. You will be asked to read in these picture files and modify them to hide secret images and messages. These files can be opened as text files by right clicking them and opening them in a basic text editor (like Notepad or TextEdit). Previewer or GIMP (or File Viewer Plus 3) should be able to open the file as an image file if you want to see the picture. If you are having problems viewing the pictures, let us know. The idea we will use is as follows.

A picture consist of an array/matrix/list of pixels. Pixels are triples of natural numbers. These values represent how much Red, Green and Blue (RGB) this particular pixel has. We will denote pixels by three space-separated natural numbers between 0 and 255 (similar to how is done in the ppm format). As an example, consider 255 100 0 as having a lot of red (255), a little green (100) and no blue (0). What we're going to do is hide messages by using very small alterations to the blue component.

It turns out our eye is not able to detect many subtle differences in colours. The pixels represented by 255 255 255 (which is white) and 255 255 254 (which is "almost white") are practically indistinguishable by a human. However, computers only process data and these two pixels are not mathematically equivalent so where our eye fails to recognize the difference, a computer can.

We're going to encode (hide) black and white images and later text messages using this "human flaw" above. To encode a black and white image, we need to first have an **original image** we would like to use to hide our black and white image in of the same size (note a black and white image can only have pixels 0 0 0 and 255 255 255). We will take the original image and make sure every blue component is divisible by 2. For each blue component in the original picture, if the blue component was divisible by 2, we leave it alone. Otherwise, we subtract 1 from it making it even. Let's call this the **zeroed out image**.

Now, look at the black and white **secret image** we want to encode. This image must have the same size as the original image we are trying to hide our secret image in. This gives us a correspondence of pixels for the two images. We hide the secret image as follows: If the pixel in the secret image is white (that is, corresponds to 255, 255, 255), add 1 to the blue component in the zeroed out image. Otherwise leave the zeroed out blue component untouched. In this way we hide the information in the black and white image by using the altered blue component in the original image. Decoding an encoded black and white image is done by reversing the above operation.

Next, we discuss how to encode a text message in an image. To do this, we first present an aside to binary numbers. What the above is really doing is much clearer when we think of a number in binary. We are accustomed to our base 10 representation where we have the digits from 0 to 9 and we have place holders for the tens, hundreds, thousands, and so on. What if we only had 2 digits, namely 0 and 1? These are referred to as bits, short for **binary digit**. We would not have a ones, tens, and hundreds and other columns anymore, we would have a ones, twos, fours, eights, and so on in this manner. In this way we could interpret binary numbers as decimal numbers. As an example, consider the number $(10011001)_2$ (the subscript denotes we're thinking of the number as a binary number and not as a decimal number). This would be and we can add these values up to get the equivalent decimal

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	1	0	0	1

number

$$(1)2^7 + (0)2^6 + (0)2^5 + (1)2^4 + (1)2^3 + (0)2^2 + (0)2^1 + (1)2^0 = 128 + 16 + 8 + 1 = 153.$$

When we make the blue component divisible by 2 in the way described above, we are actually making the last bit 0. When we add one to the component, we make the last bit 1. In the above, when we encode our secret image, think of black pixels as being 0 and white pixels as being 1.

We can convert to and from 8 bit binary using the command `format(n,'08b')` to convert a natural number `n` that corresponds to a decimal number to an 8 bit binary number (as a string) and the command `int(s, 2)` to convert a binary number that is stored as a string `s` to a decimal number.

Using the binary interpretation above, we can also encode text messages in images in a similar way. Again, zero out the original image so that the last bits in every blue component are 0. Then, take a character, say `'a'` and convert it using the `ord` command giving you 97. Writing 97 in binary using 8 bits and the commands above gives

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	0	0	1

How can we use the above then to encode a **text message** into an image? Suppose we want to encode the character `a`. Get the binary encoding as described above. Then, in eight consecutive blue components, take the numbers $0, 1, 1, 0, 0, 0, 0, 1$ and add them to the eight blue components in order. For longer strings, continue this way for each character until you have encoded an entire string! When decoding, you have reached the end of the string if you decode from the image but get the binary number 0 which is our null character (and there is no corresponding displayable character for the null character). The reason for using 8 bits is related to the ASCII encoding of a character which is left as additional reading for the interested reader. For the purposes of this problem, you may assume that the message will always fit into the picture with room for our final 8 bits of 0 to indicate the end.

The PPM File Format

A Plain PPM File consists of the following specifications[1]:

- A "magic number" for identifying the file type. For us, this will be "P3" (and yes, we are aware of the irony that the magic number is actually a string and not a number!)
- A newline character.
- A `width`, as a natural number (in base 10) followed by a single whitespace followed by a `height`, again as a natural number (in base 10).
- A newline character.
- The maximum colour value `maxval`, again as a natural number. Must be less than 65536 and more than zero. For us, we will always use 255.
- A newline character.
- There are then a total of `height` lines of pixels, each of which has `3 * width` of natural numbers. Every three numbers represents the colour of one pixel in the current line. Each of these lines ends in a single newline character and each number is separated by a single space. Each of the number is between 0 and `maxval`.

[1]The actual specifications are a bit more general than these but you may assume our tests will always follow this format

There will be some sample pictures posted with this assignment that we encourage you to open (as a text file) to better clarify the above format. Please note that solving problems relating to `a09q1.py` will be vital for ensuring that the other two problems work correctly.

1 PPM Class Part 1 in `a09q1.py`

In your starter file, you are given the following class definition:

```
class PPM:
    def __init__(self, file_name):
        '''
        Fields: magic_number (Str), width (Nat), height(Nat),
                maxval (Nat), image (listof Nat)
        requires: 0 <=  maxval  <= 65536
                  0 <= image[i] <= maxval  for all i in range(len(image))
        '''
        #YOUR CODE GOES HERE
        pass

    def __eq__(self, other):
        '''
        Returns true if and only if self and other have the same fields
        '''
        return isinstance(other, PPM) and \
               self.magic_number == other.magic_number and\
               self.width == other.width and\
               self.height == other.height and\
               self.maxval == other.maxval and\
               self.image == other.image

    def __repr__(self):
        '''
        Returns a string of the image
        '''
        return "Dimensions: ({0.width}, {0.height})\nImage: {0.image}".
            format(self)
```

Complete the initialization method above. It will consume a `self` object and a `file_name` of a valid ppm file in the current folder and initializes the fields of the PPM class object to be the values as specified in the file format given the file at `file_name`. You may assume the file exists.

2 PPM Class Part 2 in `a09q1.py`

Write the `PPM` class method

`zero_out(self)`

which consumes `self`, return `None` and mutates `self.image` so that every third entry (starting with the third value; the blue components) is divisible by 2 by performing the following. If the number was already divisible by 2, leave it alone. If the number was not divisible by 2, then subtract 1 from it.

3 Encoding an Image in `a09q2.py`

In the next four sections, you will no longer write class methods but rather functions. In each of the next four parts, you should be importing the `PPM` class from `a09q1.py` to use with the command

`from a09q1 import PPM`

Note that this is similar to how the `check` module was imported (but allows us to forgo typing `a09q1.PPM` and we can instead type `PPM` when we need to use this class). Write the function

```
encode_img(file_name, file_name_secret, file_name_out)
```

which consumes three file names as strings; the first two corresponding to existing `.ppm` files in `file_name` (any picture) and `file_name_secret` (your black and white image you wish to hide) which your function should load using the `PPM` class above. The function also consumes a destination file name in the string `file_name_out`. Your function should encode the secret image in the original image file and then create a `.ppm` file at the destination given by `file_name_out` in the proper format as described above. The function should return `None`. You may assume that the height and width of the corresponding `.ppm` files are the same.

4 Decode an Image in `a09q2.py`

Write the function

```
decode_img(file_name, file_name_out)
```

which consumes a `.ppm` file name in the string `file_name` that was encoded using the algorithm above, load this file using the `PPM` class, decodes the secret black and white image and saves it to the `.ppm` file with file name as specified by the string `file_name_out`. The function should return `None`.

5 Encode a Message in `a09q3.py`

Write the function

```
encode_msg(file_name, msg, file_name_out)
```

which consumes a string `file_name` corresponding to an existing `.ppm` file which should be opened using the `PPM` class methods and a string `msg` to be encoded. It then encodes the message into the picture as described above and saves this image as a `.ppm` file with name `file_name_out`. The function should return `None`.

6 Decode a Message in `a09q3.py`

Write the function

```
decode_msg(file_name)
```

which consumes a `.ppm` file in the string `file_name` that was encoded using the algorithm above, opens this using the `PPM` class, decodes the hidden message and returns this string.