Assignment 8

Objectives: Read and write binary files. Practice catching errors. Open files using their URL.

Note: Include DocStrings in each script you submit analogous to the following:

```
"""Retrieves the size of a bmp image and crops the image



Submitted by Mauricio Arias. NetID: ma6918
This script reads a bmp file and retrieves the size information and other
details. It then crops the image so that only the center is visible.
"""
```

Note: In all the scripts below, you are to catch the exceptions when opening files and tell the user there was a problem opening the file.

Part 1. Binary files. 5 points.
The purpose of binary files is to provide information that has specific meanings depending on the context of the file. For bitmap files the information in the binary file relates to the characteristics of the image. These characteristics are divided into parts: the general information like the size of the image and the size of the file, and the specific information about the colors in each pixel. The former is contained in a specific section of the file called the header; the header is at the beginning of the file. The latter is contained in the data section of the file, which comprises the rest of the file.

The interpretation of each piece of information in a binary file is very specific for the application. For example, colors are specified by at least three numbers (bytes): they represent the intensities of the colors red, green and blue in the RGB scheme. For bitmaps they are ordered as BGR in the file itself.

The header section comprises several fields with specific information about the image. Depending on the version of the specification they include different numbers of fields. However, they all start the same way. We will only use a few of the fields at the beginning of the file.

Descriptor:	2 bytes. “BM” for bitmaps.
File size:	4 bytes. Little Endian number representing the total size of the file in bytes.
Unused bytes:	4 bytes. Common in headers. They are usually reserved for future use.
Offset:	4 bytes. Position of the data section in reference to the beginning of the file.
Header size:	4 bytes. Size of the header. It usually leaves out some initial or unused bytes.
Image width:	4 bytes. Width of the image in pixels.
Image height:	4 bytes. Height of the image in pixels.
Color planes:	2 bytes. Number of color planes. It is usually 1. We will not pay attention to this.
Bits per pixel:	2 bytes. Number of bits used to specify the color characteristics of each pixel.
Other fields:	variable size. Other information in the header. It includes size of data section and others.

Tasks:
- Write a function `retrieve_int()` that takes two arguments: a file handle and a number of bytes to be read. This function should read the bytes (as read() provides them: type bytes) and return an integer representation of the bytes read. To accomplish this, the bytes read should be processed one at a time and combined to form a number. Importantly the numbers in this context are given in a peculiar order. Instead

of providing the most significant figure in the leftmost position as it is most often done, it is provided in the rightmost position. The other positions are filled likewise. This leaves the least significant figure in the leftmost position. For this reason, this approach to writing data is called the Little Endian approach. (The alternative is called the Big Endian approach.) For example if `bytes_read = b'\x01\x02\x05\x09'` the corresponding number is $1 + 2 * 256 + 5 * 256^2 + 9 * 256^3 = 151\,323\,137$. Each byte can be obtained in the usual pythonic way: `for byte in bytes_read: ...` . This would get '\x01' first and '\x09' last. Test it. Make sure it can handle different numbers of bytes.

- Read all the fields described for the header up to the bits per pixel field. Notice that with this information the whole structure of the bmp file can be deduced for processing. In particular note the meanings of offset, width and height. Also note the meaning of the bits per pixel. Convert it to bytes per pixel. (It is usually 3 or 4.)

- Go back to the beginning of the file and read the full header. Save a copy in a variable.

- Go to the beginning of the data section. Read all the pixels according to the information in the bit per pixel field. Take into account that there might be padding. Padding is calculated in the usual way: `- width * (bits_per_pixel//8) % 4`. If you need it, a padding byte can be `b'\x00'`. Check previous assignments if in doubt. Padding is only present at the end of horizontal lines but it might not be necessary.

- For this assignment, the pixel information will be stored in a two dimensional list. In this list, each horizontal line will be stored as a list of pixels: the rows. Read each pixel information in a horizontal line and store it in a list by appending it. When done with a horizontal line, read the padding bytes if necessary and discard them. Append the generated list to the main two dimensional list as a row.

- Make an output file by concatenating the word "inverted" and the original filename, including the bmp extension. Save an upside-down copy of the image in this file as if it were rotated 180 deg. For example, if an image containing only the letter "d" was inverted in this way, it would show the letter "p."

- Make sure you close the files once you are done with them.

Submit your script as `[NetID]_inverting_image.py`. Use the attached images to test your script: the "Tricky" one requires padding to work, the other one does not.

To explore further: If you want you can change the saturation of the image by taking each component of the pixel and multiplying it by say 3, with a max of 255. Also you can crop the image using specific coordinates: remember that the origin of the image is in the lower left corner. Alternatively you can switch the components of the colors to exchange the red and blue components. You don't need to submit any of this. Have fun.

Part 2. Multiple assignment. 2 points.
Make a script that requests some text and then separates it into words. Your script should then take each word and separate the first and last **letters** by using multiple assignment. Each word that has 3 or more characters should be replaced by the first letter and the last letter, with the number of characters removed placed in between those letters. Words with fewer characters should remain as they were. The punctuation marks should be kept.

As an example, consider the following text:

```
This is a simple, but useful example! However, there are many cases not
included here...
```

It should generate the following output:

```
T2s is a s4e, b1t u4l e5e! H5r, t3e a1e m2y c3s n1t i6d h2e...
```

Submit your script as `[NetID]_cryptic_text.py`.

Part 2. Internet. 4 points.
Using a script open the URL https://www.gutenberg.org/cache/epub/28/pg28.txt. It is in UTF-8. Search for the contents section. Read the information in that section. (Think of a strategy to know when it ends.) Use that information to make a dictionary for the file where each section in the contents has an entry: the title as the key and the text of fable as the value.

To help in deciphering the structure of the file and in extracting the appropriate information, some people add some marks inside the file. At the beginning of the actual text for the fables you find the mark:
Aesop's Fables

Near the end of this file you will find the following one:
*** END OF THE PROJECT GUTENBERG EBOOK THE FABLES OF AESOP ***

Use them to decide which items at the beginning of the contents table are not fables and to trim the last fable.

This process is called parsing a file...

After parsing, display a list of the fables available with a corresponding number. Ask the user to select a fable by number. Take the input and display the corresponding fable or provide a message indicating any problems.

After the fable is displayed, ask if the user wants to read another one until the user presses enter without making a selection. Leave a few blank lines, display the list and start the selection process anew. Don't let your script crash due to user input; catch all exceptions related to user input. Catch exceptions related to problems with retrieving the original file. Exit gracefully by providing a germane message if the file cannot be retrieved.

Submit your script as `[NetID]_fables.py`.

Try to come up with an approach to do this for a few minutes before reading what is below.

Possible approach: make a list of the elements in the contents. Look for text lines containing only the same text as an item in the list. Use those points to divide the contents. Use also the marks mentioned before to restrict the search. Only generate the dictionary for actual fables. Trim as desired.

When presenting the fables, dumping the text without blank lines in between is acceptable. The user would have to scroll. Provide an anchor to separate the current fable from the one before so that searching for the beginning is easy: say "\n"*2 + "="*20 + " + " + "="*20 + "\n"*2.