SE1-SMP Lab Final

1. Network Scanner

In this exercise we will create a simple network scanner which implements a small part of the functionality of Wireshark

- First, create a text file called 'scan'. (So far we called our files scan.py. For executable files it is common to leave the .py away, but then you must tell your editor it is editing Python. You could also just call it scan.py)
- Insert the first line to identify YOUR Python executable. Mine is:
    - #!/usr/bin/env python3
- In your Linux Shell, make the file executable
    - (find out how using Google: chmod)

The beauty of Linux is that it is easy to combine commands, even in Python. We use the Python library subprocess to call the Linux ping command as if it were a function.
Using 'man ping' in your shell you find out what the command line arguments can be. We want to send one ping only, so there are at least three arguments to ping: the command-line switch -c, a number and the ip address to ping.

- Write a onePing function which takes an ip address (string) as argument, uses .run and returns the returncode of the CompletedProcess returned.
- Now write a loop in main which generates and pings -- say -- 100 different IP addresses
    - Look at the IP of your laptop to see in what range you are.
    - Don't scan more than 100 IP's or so. Network admins dislike and distrust scans.
- Find out which ping arguments work for you. For me, on a Mac, ['ping', '-c', '1', '-q','-t', '1','-i', '0.2', ip] work
- Now improve your output using str.format()

It's starting to look right, but my output thrown together with ping output is annoying.
Subprocess can be used to capture all ping output by using the method 'check_output' or the flag captureOutput

- Change your program to use either method to prevent ping's own output

That's straightforward, but now the program still throws errors.

- Use 'try' to capture errors of type 'CalledProcessError'.
- Adapt your function ping to return False for non-responding ip's and True otherwise and process that value appropriately.

Note: yes, we know this implementation is very slow

2. Word Count

In this exercise we make a word counting (histogram) tool

- Find out how a Python script can use command-line arguments
- Create an executable script that takes a single argument (a file name)
- Open the file, read all the lines, separate in words (separated by whitespace, comma's, periods, etc.)
- Use a dict to count frequencies of all words
- Print a wordlist showing words and frequency ordered by frequency.
- Using .format(), make sure the colons between words and frequency are lined up

3. File Tree

Make an executable Python script called ‘ftree’ which prints a file tree as follows:

- Starting from the current directory, for each file print a single line:
  - That is indented to the current depth (as explained below)
  - The letter F for a file, or D for a directory
  - Followed by that file/directory’s name
  - Followed by the file size or number of files in the directory
- After the line for a directory, the files in that directory are shown at an indentation level one deeper than the directory

4. Image Manipulation

In this assignment we will write a filter for small images. Here, an image is represented as a 2-dimensional array of pixels. To be precise: an image is a list of rows, a row is a list of pixels and each pixel is a tuple of three RGB numbers from 0 to 255 (inclusive). (In reality, images wouldn’t be represented in this way)
For instance,

```
[[(255,0,0),(255,0,0),(255,0,0)],
 [(255,0,0),(255,0,0),(255,0,0)],
 [(255,0,0),(255,0,0),(255,0,0)]]
```

Is a 3x3 pixel red square.

- Write a function ‘grayscale’ with one argument (an image) that computes a grayscale version of the image by making the three rgb values in every pixel the average of the three colors in that pixel. For the example a pixel which is (255,0,0) becomes (85,85,85) (below, this value 85 is called the grey-scale value). In reality the average isn’t ideal because our eyes perceive colours differently. The luma/luminocity can be approximated by the following formula: Y = (R+R+B+G+G+G)/6; that is, green ‘weighs’ three times as much as blue since we’re better at seeing greens. For now you can just use the average).
- Now we will make a monochrome filter which uses the grey-scale value as a percentage factor of a monochrome color (expressed as a pixel). So, if the monochrome color is (0,128,255) (blueish) and we are looking at a red pixel, which has grey-scale 85, then the result is (0*85/255,128*85/255,255*85/255) = (0,43,85).
  Write a function ‘monochrome’ that takes one pixel (rgb-tuple) as an extra argument called *mono*. The result for each pixel is the grey-scale value of that pixel multiplied by that extra argument. For instance, suppose mono is (0,255,255) and an input pixel is (255,0,0). Then the grey-scale value of that pixel is 85, so that pixel becomes (0,85,85).
- Write a function ‘blur’ in which every color in every pixel is recomputed as follows. For each color in each pixel p with value c5, if the surrounding pixels have values c1, c2, c3, c4, c6, c7, c8, c9 for that color, the new value for that color in p becomes: (8*c5+c1+c2+c3+c4+c6+c7+c8+c9)/16.
  So
  ```
  1  2  3      ..            ..                 ..
  4  5  6 =>   ..  (8*5+1+2+3+4+6+1+2+3)/16  ..
  1  2  3      ..            ..                 ..
  ```
  Leads to: a value ~2.2 in the centre pixel.)

5. RPN Calculator

In this exercise we will make an RPN calculator.

- Create an executable script called ‘pdc’ (after Mr Watson’s favorite calculator dc, but written in Python)
- The calculator accepts strings of ‘words’ separated by whitespace
- A word is a number or an operator
- Numbers: Hexadecimal numbers prepended with 0x, binary numbers prepended with 0b, and decimal numbers
- Words include:
    - P: prints the value on the stack, without altering the stack
    - F: prints the entire stack (without altering it)
    - +, -, *, /, %, ~, ^, |, & [Note: use the Python-meaning of these operators; in Linux dc they mean something else).
    - D: duplicates the top value on the stack
    - sN (e.g. s1, s2, …) stores the top of stack in a register
    - lN (e.g. l1, l2, …) loads the value of a register on the stack

6. Paging

In this exercise we will model simple paging.

- Create a class Mem as an abstraction of memory. It has
    - Methods ‘get’ and ‘put’ which read one value from an address and write one value to an address. We will abstract from bytes and use memory cells which hold one number
    - An instance variable ‘pages’ which contains a list of, say, 16 objects of class Frame (these represent physical memory pages)
    - An instance variable ‘pagefile’ which contains a list of, say, 64 objects of class Page (these represent virtual memory pages)
    - An instance variable ‘pageTable’ which contains a list of 64 objects of class PageTableEntry
- For testing purposes we don’t need 4k-sized pages, so make every Page and Frame a list of 16 numbers
- Class Frame has ‘get’ and ‘put’ methods which get/put a number from/to the offset using the lower 4 bits in the address (why?)
- Every PageTableEntry knows if
    - A page is in physical memory or not
    - Whether the page is dirty or not
- Method get in class Mem checks for the given address (using all bits except the lower 4 as index) if
    - It is in physical memory: then it returns the corresponding ‘get’ on that frame
    - It is in virtual memory: then it looks at the least recently used page:
        - If it is dirty, write the contents back to virtual memory
        - After the write (or immediately if it is not dirty) use it to load the page
        - Access the value
    - It is outside the allowed range: generate a segmentation fault.
- Method put is similar except it must be flagged ‘dirty’ after changing it.