Challenge 02: Run-length encoding

September 23, 2019

The challenge

Data is the new ~~gold~~ ~~oil~~ ~~buzzword~~ — err money-maker. There is so much data running through the tubes of the internet every millisecond! It is of great interest to computer scientists to increase the throughput of this data to feed the many, many systems that power our emails, search engines, web trackers, social media networks, targetted advertising platforms, etc. etc. etc. etc.

DATA IS BIG.

Let's make it smaller.

Maybe.

A friend of yours has an idea for a new compression algorithm that takes care of repeating characters in files. They think it will give the Pied Piper folks a run for their money!

Your task will be to create a *run-length* encoding scheme for plain-text data. This scheme will *encode* the data *losslessly* such that when *decoded* the original message is extracted, unchanged, in full.

The algorithm

Given a *plain text* file, we will call a subset of the text that consists of repeating symbols a *block*.

A block will be *encoded* by counting the block size and recording the *size* followed by the block *symbol*. A block symbol is a single character. You may assume that we are dealing only with ASCII, and that each character is only one (1) byte.

It is possible the original text will contain numbers. You might notice this is problematic since numbers denote *size* in our encoding scheme. Your friend knows about non-printable ASCII characters and has cleverly designed a mitigation. They say that in the case of a numeric *symbol*, you should separate the *size* from the *symbol* with the ASCII character `0x2` (STX, start of text).

For non-numeric compression blocks, no STX character will be used to delimit the *size* and the *symbol*.

In python, you can add a STX character to a string with the escape sequence `\x02`.

An *encoded block* will therefore match one of the following patterns:

- `[0-9]+<STX>[0-9]`
- `[0-9]+[a-zA-Z]`

defined as (*size*, *possible STX*, *symbol*).

The encoded output will be a sequence of these blocks.

Assumptions and constraints

- You may assume that input to your programs will always be valid. You do not need to write any error checking code.
- You *must* write your solution using Python 3 code. Yes, even if you don't know it. See the syntax primer.
- You *may not* use the Internet! Moderators will be spying on you to make sure you don't cheat!
- Every 20 minutes there will be a rotation within your group. Make sure your teammates can pick up where you left off! There will be a total of two (2) rotations, so that you will work on each project for 20 minutes.

Operating modes

Your software will need to operate in two modes:

1. `c`: compression
2. `x`: decompression

In *compression* mode, your software will need to parse plain text files, encoding them as described by the algorithm above.

In *decompression* mode, your software will need to parse encoded text, and decode it to output the original plain text.

Your program should be run like this for compression mode:

```
./rle.py c file.txt
```

and display the encoded output to the console (`stdout`).

In decompression mode, your program should be run like this:

```
./rle.py x encoded.txt
```

and display the decompressed (original) text to the console.

If an optional output file is specified as a third argument, then the output should recorded in that file instead of printed to the console.

Sample inputs and outputs

In compression mode, your program will run-length encode text.

Sample input 1

```
Aaaaaaa a bee! Heeeeeeeeeeeeeeeeeeeellp!!!!!!
```

Sample output 1

```
1A6a1 1a1 1b2e1!1 1H20e2l1p6!
```

Sample input 2

```
++==!!!!!777777.......__________(AABBC)
```

Sample output 2

In the text below, `<STX>` means the non-printable character usually represented as `\x02` in code. When you actually print the compressed output to console, the `<STX>` should not show up, rather no space will appear between the `6` and the `7`.

```
2+2=5!6<STX>76.10_1(2A2B1C1)
```

In decompression mode, your program will decode the text.

The sample inputs should each be able to be recovered from the sample outputs.

Running the program

Ensure that your python program is executable and accepts the following options.

```
./rle.py MODE INPUT [OUTPUT]
```

Where MODE is a single character, with valid options `c` and `x`, INPUT is the path to an input text file, and OUTPUT is an optional path to the output file.

If no output file is supplied, then your program should print the results to the console (`stdout`).

If an output file is specified and exists, your program should **overwrite** the specified file. Otherwise, it should create it.

In other words, your program should be able to run in each of the following ways:

```
./rle.py c input.txt
./rle.py c input.txt output.txt
./rle.py x encoded.txt
./rle.py x encoded.text decoded.txt
```