Week 4: Bits, Bytes and Binary

Introduction

In this worksheet you will get familiar with working with binary numbers! All data in computers is stored and transmitted in terms of binary numbers so it is very useful to understand the basics of how it works - it is also a popular interview topic! You will look into how exactly data is transmitted across the internet (UTF-8 encoding) and build your own decoder so you can read a bit stream!

TASKS

TASK 1: Conversion

What are these binary numbers as integers? Note that the least significant bit of on the right-hand-side (this is the 'normal' way for data communication).

1. 10
2. 101101
3. 1001
4. 1101110

Convert these integers to binary:

1. 5
2. 12
3. 37

TASK 2: BinaryToInt

Complete the 'BinaryToInt(B)' function. This will input a binary number *as a string* and return an integer. Try code this from scratch rather than using Python libraries and built-in functions!

You can test this function using Python's own $bin(N)$ function. I would suggest using this in the future, instead of your own built function!

Hint: If you need to use it then L[::-1] will reverse the list L.
Hint: The symbol ^ is not the power operator in Python.

TASK 3: IntToBinary

Complete the 'IntToBinary(N)' function. This will input an integer and return a string containing its binary representation. If you would like a description of a method to help you, see the 'Hints' section at the end of the worksheet.

You can test your function by comparing it to Python's own $int(BinaryString, 2)$ function. Input a binary string and the number 2 (to represent the base number).

Hint: Look up the Python operators '//' and '%'. What do these do and how could they be useful?

TASK 4: ASCII

When computers want to transmit any data, it is always sent as **bits** (each bit is either a 0 or a 1), where 8 bits make up a **byte**. This task will walk you through how a computer may send a string in terms of bits.

First, let's look at something called ASCII conversion. This is a standard conversion, which gives **characters (text and symbols) a numbering**. To find the number of any character you can use the Python function 'ord'. Try running the following code:

```
print( ord('a') )
print( ord('z') )
print( ord('0') )
print( ord('9') )
print( ord('A') )
print( ord('#') )
```

You will notice that the digits (0-9) are not encoded as themselves, instead encoded from a range 48-57. Also, notice that capital letters have different encodings to lower case letters.

ASCII has 128 character encodings where only 96 of these are visible - for example, 7 represents an audible beep.

So now we have a way of turning text into numbers, all that is left to do is turn the numbers into binary form!

TO DO:

1. Write function called 'IntToByte' which will input an integer and return it in terms of its binary number, but always in a block of 8 bits (a byte). So, instead of 10 being represented as 1010, the function returns '00001010'.
2. Write a function called 'StringToBytes' that inputs a string and returns a list of binary numbers (in string form). You will need to first transform the letters into their encodings and then turn these into bytes. For example:
   ```
       StringToBytes('hi') = ['01101000', '01101001']
   ```
3. Write the opposite function 'BytesToString' which takes a list of binary inputs and returns a string. Note that the opposite of the 'ord' function is 'chr'.
4. Decode the list of bytes in the 'Message1.txt' file (given in the Week4 zip folder) to reveal a secret message and check that your function works!

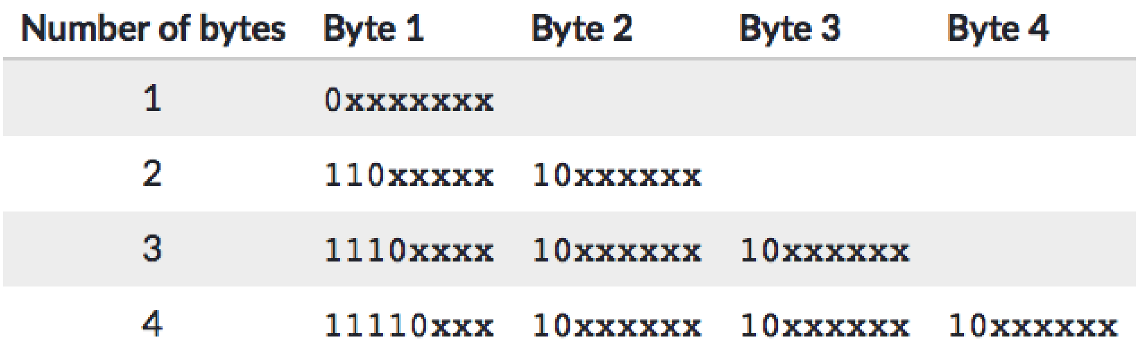
It must be noted that Python represents binary numbers by putting '0b' in front of them, e.g 3 in binary in Python is '0b11'. So, if you are planning on using binary inside Python for other projects then edit your code to concatenate this on the front of any binary number!

TASK 5: UTF-8

ASCII only encodes 128 characters and does not include many foreign symbols, i.e you could not encode an 'é'. A universal list of characters has been created called **unicode** (note that this is not the conversion). ASCII has been extended to include a conversion for all of the unicode characters - this is called **UTF-8**.

UTF-8 has the ability to encode 1,112,064 characters and you may be wondering how it does this when working with singular bytes (you can only encode up to 255 characters with 8 bits as the highest number is 11111111).

Well, UTF-8 let's us work with up to 4 bytes (32 bits), with the number of bytes being used for this number displayed at the front. A table for how this works can be seen below:



Number of bytes	Byte 1	Byte 2	Byte 3	Byte 4
1	0xxxxxxx			
2	110xxxxx	10xxxxxx		
3	1110xxxx	10xxxxxx	10xxxxxx	
4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

So, the first byte tells us how many bytes we have by how many 1's there are, which is then followed by a 0. Note that if our character only has length of 1 byte, then this is represented with a 0 at the beginning (probably because this take up less room in the 8 bits than '10').

The '10' at the beginning of a byte means that byte is still part of the encoding for the same character.

DISCUSSION: How many characters are in the following sequence? *In practice, the numbers would not be sent with spaces between the bits, this has just been done to help you visualise.*

$$01001001\ 11011001\ 10001001\ 00100111$$

TO DO:
Write a function called 'UTFDecoder'. This function should:

1. Input a string containing a stream of bits.

2. Split the string up into bytes (use a list of strings to store). It might be worth adding an error incase the input does not have a length of multiple 8.

3. Group the bits into characters (use a list of lists to store as in each element inside the character list contains a list of all the bytes in that character). You might find the 'string.find()' function helpful, e.g '11100101'.find('0') = 3.

4. Remove each of the prefixes of each byte. Remember that you will be removing a different number of bits on the first bytes of characters, and then only 2 bits for the remaining bytes. *Hint:* To split a list L by a certain index n you can use the following syntax:

```
    L = [0,1,2,3,4]
    L[None:2] = [0,1]
    L[2:None] = [2,3,4]
```

5. For each character, concatenate the remaining bits together to create one binary number.

6. Decode each character. The ’ord’ function should work for UTF-8 conversions too.
7. Return the message as a string!
8. Now, use your function to decode the **secret message**. This is a binary string representing a stream of characters which hopefully your function will be able to decode! You can find the message in the **’Message2.txt’ file** provided in the Week4 zip file!.

Discussion

On top of UTF-8, there is also UTF-16 and UTF-32. In UTF-8, a character has a minimum of 8 bits, whereas in UTF-16 a character has to have a minimum of 16 bits (similarly with UTF-32).

What are some pros and cons of these different versions?

When would you be more likely to use UTF-16 or UTF-32 or UTF-8?

Topic Suggestions

If you have any topics which you would like covered in these sessions then feel free to let me know by clicking here.
This may be content that isn’t covered within your units which you would find useful or a particular topic within your current units that you feel you would benefit from more help with!

Hints

Converting from Integer to Binary

If we take a number n, we want to see how many powers of 2 fit into it and if those powers of 2 exactly fit or have 1 left over. We will take n and continually divide by 2 (until we reach 0), looking at the quotients and remainders. We will recursively do this process on the quotient of each division. We will take note of the remainders in order, and then reverse this sequence to get our binary number.

Let’s walk though an example: 13

13 = (2*6) + 1
6 = (2*3) + 0
3 = (2*1) + 1
1 = (2*0) + 1

Walking back up through the remainders, we have the sequence 1101 - which is the binary number for 13.