15-110: Principles of Computing, Spring 2018

Programming Assignment 7

Due: Tuesday, March 20 by 9PM

Note: You are **responsible for protecting your solutions** to the following problems from being seen by other students both physically (e.g., by looking over your shoulder or verbal discussion) and electronically. In particular, since the lab machines use the Andrew File System (AFS) to share files worldwide, you need to be careful that you do not put files in a place that is publicly accessible.

If you are doing the assignment on the Gates-Hillman Cluster machines we use in the lab or on `unix.andrew.cmu.edu`, please remember to have your solutions inside a `private` folder (which is under your home directory). Our recommendation is that you create a `pa7` folder under `~/private/15110` for this assignment. That is, the new directory `pa7` is inside the directory named `15110`, which is inside the `private` directory.

Overview

For this assignment, you will create a Python file for each of the problems below. You should save all of these files in a folder named `pa7`. Once you have every file, you should zip up the `pa7` folder and submit the zipped file on Autolab.

This assignment will help you explore binary and hexadecimal notations and how characters are stored as ASCII codes. In addition to writing the code, you should test your code on multiple inputs to check its correctness.

Problems

1. [2 points] An unsigned 16-bit binary value can be converted to an integer simply by creating a variable to hold the total (What do you initialize this variable to?) and examine each bit one at a time using a loop: if any bit is 1, then add the corresponding power of 2 to the total.

For this problem, let's store an unsigned 16-bit binary value as a string. For example, the 16-bit unsigned integer 0101100110100101 would be stored in a string as '0101100110100101'.

Be careful! In our binary string, the indexing increases left to right but the powers of 2 increase right to left. Here's a hint:

> if *bitstring[0]* is '1', then you add 2^{15} to the *total*.
> if *bitstring[1]* is '1', then you add 2^{14} to the *total*.
> if *bitstring[2]* is '1', then you add 2^{13} to the *total*.
> etc.

Do you see the pattern? If *bitstring[i]* is '1', what do you add to the *total* as a function of *i*?

Write a function `unsigned_value(bitstring)` (in `unsigned_value.py`) which takes a string named `bitstring` consisting of some combination of sixteen 1s and 0s representing an unsigned 16-bit integer. This function should return the decimal (base 10) value of the 16-bit unsigned integer. Remember to use a loop. Your function should not have 16 separate tests.

Example usage:

```
> python3 -i unsigned_value.py
>>> unsigned_value('0000000000000000')
0
>>> unsigned_value('0000000000000001')
1
>>> unsigned_value('0000000000000010')
2
>>> unsigned_value('1000000000000000')
32768
>>> unsigned_value('0000000000001111')
15
>>> unsigned_value('0000000011111111')
255
>>> unsigned_value('0000111111111111')
4095
>>> unsigned_value('1111111111111111')
65535
```

NOTE: You should test your function on many more values to make sure it works correctly. Look at Piazza soon for some testing hints!

2. [2 points] Recall that a signed binary integer uses the leftmost bit as its sign. If the sign bit is 0, the number is considered positive; if the sign bit is 1, the number is considered negative. In this problem, we'd like to write a Python function that converts an 8-bit signed negative integer to its 8-bit positive equivalent. For example, if we start with the 8-bit value 11001100 representing -52, its positive equivalent +52 is 00110100 (as discussed in lecture).

We can represent 8-bit values as strings of 1s and 0s, but Python does not allow us to change individual characters of a string. So we will need to use two new features. One converts a string to a list of characters using the **`list`** function:

```
>>> my_string
'11001100'
>>> my_list = list(my_string)
>>> my_list
['1', '1', '0', '0', '1', '1', '0', '0']
```

To convert a list of characters back to a string, we use the **`join`** function on an empty string:

```
>>> my_list
['1', '1', '0', '0', '1', '1', '0', '0']
>>> my_string = "".join(my_list)
>>> my_string
'11001100'
```

Write a function `to_positive(bytestring)` (in `to_positive.py`) that takes a string named `bytestring` of some combination of eight 1's and 0's and returns a string representing the positive value of the given string. If the given string represents a positive integer, you should just return that string as your answer. Otherwise, follow this algorithm:

1. Convert *bytestring* to a list named *bytelist*.
2. For each character in *bytelist*, if it is a '1', change it to a '0'; otherwise, change it to a '1'.
3. Set *i* equal to 7.
4. While the i^{th} character in *bytelist* is '1', set the i^{th} character of *bytelist* to '0' and decrease *i* by 1.
5. If *i* is still greater than 0 after step 4 is completed, set the i^{th} character of *bytelist* to '1'.
6. Convert *bytelist* back into a string and return this string as your answer.

Example usage:

```
> python3 -i to_positive.py
>>> to_positive('11001100')
'00110100'
>>> to_positive('11110000')
'00010000'
>>> to_positive('11111111')
'00000001'
>>> to_positive('01010101')
'01010101'
```

3. [2 points] This problem will explore hexadecimal notation.

Write a function `bin_to_hex(bitstring)` (in `bin_to_hex.py`) that takes a string named `bitstring` containing a combination of sixteen 1s and 0s. It returns the hexadecimal equivalent of the binary string.

To solve this problem, you will need two lists in your function. One list will have all of the 4-bit binary values in numerical order:

```
binlist = ['0000', '0001', '0010', … , '1110', '1111']
```

(You can't use '…'; you have to write them all out!)

The other list will have all of the single digit hex values in numerical order:

```
hexlist = ['0', '1', '2', … , 'E', 'F']
```

(Again you can't use '…'; you have to write them all out!)

General algorithm:
1. Set *result* equal to the empty string.
2. For *i* equal to 0, 4, 8 and 12, do the following:
    a. Set *pattern* equal to the 4-bit substring of *bitstring* starting at index *i*.
    b. Set *j* equal to the index of *pattern* in the *binlist*. (What list function do you use here?)
    c. Set *result* equal to *result* plus the j^{th} string in *hexlist*.
3. Return *result* as your final answer.

String reminder: If `s` is a string, then `s[a:b]` is the substring of `s` starting at index `a` up to but not including index `b`.

Example usage:

```
> python3 -i bin_to_hex.py
>>> bin_to_hex('0000000100100011')
'0123'
>>> bin_to_hex('1111111011011100')
'FEDC'
>>> bin_to_hex('1000100110101011')
'89AB'
>>> bin_to_hex('0111010001010110')
'7456'
>>> bin_to_hex('1011111011101111')
'BEEF'
```

4. [4 points] Both functions for this part deal with ASCII values for characters.

For this problem you will take a letter of the alphabet and change it to the letter *i* positions afterwards, wrapping around back to A if necessary. For example, if the *letter* is 'A' and *i* is 4, then we would return 'E'. If the *letter* is 'X' and *i* is 5, then we would return 'C'.

Recall that characters are stored in the computer as codes (integers) using the ASCII standard. Luckily the letters have codes that are in the same sequence as the letters. For example, 'A' is 65 in ASCII, 'B' is 66 in ASCII, 'C' is 67 in ASCII, etc. We can't do arithmetic with characters in Python. Instead, we must get the character's code using the `ord` function, do the arithmetic, and convert the result back to a character using the `chr` function.

Examples:

```
>>> ord('A')
65
>>> ord('Z')
90
>>> chr(65)
'A'
>>> chr(90)
'Z'
```

For this problem, you are not allowed to use any built-in character functions other than `ord` and `chr`. Instead, use the character codes and numeric operations.

(a) (2 points) Write a function `forward(c, i)` in `forward.py` that takes an upper-case character stored in the parameter `c` and a positive integer `i` and returns the upper-case character that is `i` positions forward, wrapping around to the start of the alphabet if necessary.

Example usage:

```
> python3 -i forward.py
>>> forward('A',1)
'B'
>>> forward('A',2)
'C'
>>> forward('C',5)
'H'
>>> forward('X',1)
'Y'
>>> forward('X',5)
'C'
>>> forward('M',26)
'M'
>>> forward('Q',51)
'P'
```

HINT:
As you can see, `i` could be larger than 25. To deal with this, once you get the ASCII value of the character, shift it back so it lies between 0 and 25, inclusive. Then add `i` and deal with the wraparound. (Modulo might be helpful here!) Then shift back so the value lies between 65 and 90, inclusive. Then you can convert back to a character.

(b) (2 points) Write a function called `capitalize(s)` in `capitalize.py`. The parameter is a string `s` containing a sequence of 1 or more words that are all in lowercase with one space between words. The function returns another string that is the same as *s* except that all words are capitalized. To capitalize a word means to convert its first letter to upper case. A word is any sequence of letters (not including spaces).

General algorithm: Loop over each character of the string s. If the character is the start of a word, add its uppercase equivalent to the new string. Otherwise, add the character to the new string as is. In general, a character is at the start of a word if the previous character was a space. (There is a special case here you should figure out.)

ASCII HINT: The difference between the ASCII code for an uppercase letter and its corresponding lowercase letters is 32.

Example usage:

```
> python3 -i capitalize.py
>>> capitalize('how now brown cow')
'How Now Brown Cow'
>>> capitalize('python is a snake')
'Python Is A Snake'
>>> capitalize('pittsburgh')
'Pittsburgh'
```

Submission

You should now have the `pa7` folder that contains the following Python files:

a. `unsigned_value.py`
b. `to_positive.py`
c. `bin_to_hex.py`
d. `forward.py`
e. `capitalize.py`

Zip up the folder and submit the zipped file named as `pa7.zip` on Autolab.
Be sure to check your submission to see that you submitted the correct code.