Lab 6: Booleans

Feb 13, 2018

Main Event

1. Bank values

Generate two lists containing 12 random integer values. These two lists represent the bank values of a man and a woman. Use a loop to find which person had more prosperous months than the other.

Specifically, compare the monthly balances of each person. If the man had more months in which his balance was greater than the hers, print “He had more prosperous months”; if she had more months of higher balances, print “She had more prosperous months”; otherwise, print “They both had great years”.

Solution:

```
import random

months = 12
account_max = 100
his_account = []
her_account = []
for i in range(months):
    his_account.append(random.randrange(account_max))
    her_account.append(random.randrange(account_max))

print('His: ', his_account)
print('Hers:', her_account)

i = 0
richer = [0, 0]
while i < months:
    index = his_account[i] > her_account[i]
    richer[index] += 1
    i += 1

if richer[0] > richer[1]:
    print('He had more prosperous months')
elif richer[1] > richer[0]:
    print('She had more prosperous months')
else:
    print('They both had great years')
```

2. Binary addition

Write a program that adds two binary numbers (represented as strings) using a loop.

Note: you do not have to worry about carrying for these exercises. They are strictly about basic addition. If you are curious about carrying, check out the additional practice.

1. First, focus on the logic: create two strings:

   ```
   x = '01'
   y = '10'
   ```

   Each character of the respective strings can be thought of as a “bit”. You will have to consider each individually when calculating the overall sum; in this case `'10'` (as a string). While it may be tempting to reverse the string prior to looping, challenge yourself by not doing so: *use the loop, Luke*.

2. Update the code to handle situations in which `x` and `y` do not have the same number of digits:

   ```
   x = '10101'
   y = '1010'
   ```

3. Convert the final value back to a decimal. Again, this should be done using only a loop and simple logic. Remember, the digit’s place dictates its contribution to the final value:

$$\begin{aligned}111010 &= 2^5 \times 1 + 2^4 \times 1 + 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0 \\ &= 2^5 + 2^4 + 2^3 + 2^1 \\ &= 58\end{aligned}$$

Python has the ability to convert numbers to various bases. Recall int, a function that converts a value to an integer. By default, it assumes the value you are passing it is a value of base 10. By passing a second argument, however, you can specify that you would like a different base:

```
>>> x = '10101'
>>> int(x)
10101
>>> int(x, 2)
21
```

Python has another function, bin that will convert a number to a binary string. Continuing the previous interaction:

```
>>> bin(int(x, 2))
'0b10101'
```

As you play with different values to ensure your algorithms are correct, use these Python built-in's to check your work.

Solution:

```
# 1. logic
x = '01'
y = '10'

result = ''
for i in range(-1, -(len(x) + 1), -1): # can also use a while-loop
    single = int(x[i]) ^ int(y[i])
    result = str(single) + result
print(x, '+', y, '=', result)

answer = bin(int(x, 2) + int(y, 2)) # check our work!
print('Correct:', result == answer[2:])

# 2. different lengths
x = '10101'
y = '1010'

x = x.zfill(len(y))
y = y.zfill(len(x))
```

```python
result = ''
for i in range(-1, -(len(x) + 1), -1):
    single = int(x[i]) ^ int(y[i])
    result = str(single) + result
x = x.lstrip('0')
y = y.lstrip('0')
print(x, '+', y, '=', result) # remove leading zeros

answer = bin(int(x, 2) + int(y, 2)) # check our work!
print('Correct:', result == answer[2:])

# 3. convert to decimal
decimal = 0
for i in range(len(result)):
    decimal += (2 ** i) * int(result[-i - 1])
print(result, '=', decimal)

print('Correct:', decimal == int(result, 2)) # check our work!
```

Additional Practice

1. Full adder

You may have noticed that the adder you previously developed did not always work. In particular, if the addition required a carry, things probably went awry. This exercise will have you correct that.

First, set `x` and `y` to values in which a carry is required. A good start is

```python
x = 11
y = 01
```

You should definitely try more complicated values at some point, but in just getting started, using the simplest example possible will help you focus on getting things right.

Hairy carry

When no carry was involved, it was convenient to view binary addition as simply xor. However, now you must understand the complete extent of the story. Addition is actually two components: the addition itself, and calculation of the carry. These two steps comprise of what is known as a “half adder.” Remember, however, that the addition of any given component may have a carry value coming in. Taking into account this value is what comprises the other half of the adder. Together the two halves make up a "full adder."

So that programs are general, it is assumed that during the first addition (the right-most components) a carry of zero came in. Further, it is assumed that there is an implicit final addition (after the left-most component) of two zeros so that a potential final carry can be taken into account.

Being able to add binary numbers lies at the foundation of calculators in particular, and computers in general. Understanding how it is done also gives you an appreciation for the type of problems hardware designers must solve.

Make it pretty

What would beautifully working full adder be without beautifully looking output? Format your output such that it is visually clear. If `x = '100101'` and `y = '10101'`, your code would produce:

```
   100101
+   10101
--------
   111010 = 58
```

A slightly longer example:

```
     11110100111001110001
+   10101001111011011111
-----------------------
   100100100011000011000 = 1197592
```

Solution:

```
x = '100101'
y = '10101'

# x = '11110100111001110001'
# y = '10101001111011011111'

x = x.zfill(len(y) + 1) # an additional zero in case there's a final c
y = y.zfill(len(x) + 1)
```

```python
result = ''
carry = 0

for i in range(-1, -(len(x) + 1), -1):
    x_int = int(x[i])
    y_int = int(y[i])

    # take into account the values
    a = x_int ^ y_int
    b = x_int and y_int

    # take into account the incoming carry
    c = a ^ carry
    d = a and carry

    result = str(c) + result
    carry = b or d

# convert to decimal (same as before)
decimal = 0
for i in range(len(result)):
    decimal += (2 ** i) * int(result[-i - 1])

# make the result pretty
addition = '+ '

x = x.lstrip('0')
print(' ' * (len(addition) + len(result) - len(x)) + x)

y = y.lstrip('0')
print(addition + ' ' * (len(result) - len(y)) + y)

print('-' * (len(result) + len(addition)))
print(result.rjust(len(addition) + len(result)), '=', decimal)
print()

# show validity
answer = bin(int(x, 2) + int(y, 2))
print('Addition valid:', result == answer[2:])
print('Conversion valid:', decimal == int(result, 2))
```

2. Tear drops

This exercise will help you get comfortable using the command line. In doing so, you'll build "tear drops," a program that only works in the terminal.

Terminal

The "terminal" is a command line interface that allows you to interact with your computer. It exposes both your system, and raw programs that allow you to do different things. Many window applications call these programs on your behalf—from the terminal you can call them directly.

There are several programs available to you. The most important, however, are:

- **cd** Change to a specified directory.
- **ls** List the contents of a directory. (In Windows, the equivalent command is dir)
- **python3** Pass a given file to the Python interpreter (version 3, which we've been using thus far).

If you've never interacted with your system via the command line, then you probably think of files and folders as icons in a windowing system. Double clicking a folder opens that folder, exposing other files; double clicking a file opens that file in a predetermined program.

Files and folders on your computer actually live in a hierarchy. The top level of that hierarchy is known as the "root." The root contains various folders and files, for example folders A and B, and file c. Once inside folder A, you cannot see folder B; you have to go back to the root to get access.

Try it: open a terminal and type `ls`. If you're using a Mac, one of the folders you might see in this listing is "Downloads". Change to the Downloads directory (`cd Downloads`). Run `ls` again—notice a difference? To get back to your previous location, type (`cd ..`, which means go "up" the file hierarchy, which is happens to be the previous location in this example).

When you write programs in Spyder (or whatever editor you are running), you save those files somewhere on your file system. Identify where that is in Spyder (*Save As* should make it clear) and navigate to that place in terminal. For example, if you saved the file to "Desktop", you would type `cd Desktop` (the first command after opening terminal).

Assuming your file were called “best.py”, you would type `python3 best.py` to run it.

Clear screen

What’s nice about running things in the terminal is that you can clear the screen at anytime, and print things on what appears to be a blank canvas. Try it:

```
import os
os.system('clear')
print("Hello")
```

(For Window’s users, you should pass the string “cls” to `os.system` instead of “clear”.)

You can use this to give the illusion of animation:

```
import os

for i in range(10):
    os.system('clear')
    print(i)
```

You can slow that down if it’s too fast by using the sleep function in the time module:

```
import os
import time

for i in range(10):
    os.system('clear')
    print(i)
    time.sleep(2 ** -3)
```

Where `sleep` takes the amount of time you want the program to pause.

Tear drops

You can use this idea to make tear drops fall down your screen. In reality, these are actually `*`’s that “move” up and down. In the first frame:

```
***********
*** ** ****
```

```
**  ** * **
**  *  * **
```

In the second:

```
***********
*** *******
*** ** ****
 *        *
```

It’s more impressive when the first is cleared, and the second follows. Ask for a demonstration if you still don’t get the gist.

Aside from clearing the screen and sleep’ing, you’ll need to figure out how to control these stars. Essentially, it’s a list-of-lists in which the inner lists contain the stars for either a row or a column. At each iteration, you need to figure out whether a given set of stars should have a value removed, added, or remain constant (that’s essentially how they “move”).

While it might seem straightforward to make inner lists be the rows on the screen, it is probably much easier if they represent the columns. When it comes time to print, you can then transpose that matrix so that things appear to move top-to-bottom.

Finally, helpers.py is a module that provides variables you can use for your board dimensions:

```
import helpers.py
print(helpers.terminal.rows, helpers.terminal.columns)
```

These variables are initialized based on the dimensions of your terminal. As always, make sure helpers.py is in the same directory as your teardrops implementation.

Solution:

```
import os
import time
import random
import helpers

drop = '*'
pause = 2 ** -3
clear = 'clear' # use 'cls' on Windows
```

```python
# Build the initial screen. Internally, columns are represented as
# rows.
previous = []
for i in range(helpers.terminal.columns):
    string = drop * random.randrange(1, helpers.terminal.rows)
    previous.append(string)

while True:
    os.system(clear)

    # Create the next screen: each column changes by at most one drop
    current = []
    for i in previous:
        if len(i) == helpers.terminal.rows: # "column" is full
            c = i[:-1]
        elif len(i) == 0: # "column" is empty
            c = '*'
        else:
            # Decide whether to remove, add, or hold
            # (respectively). Considering using random.choices to
            # control how random this decision is.
            action = random.randrange(3)
            if action == 0:
                c = i[:-1]
            elif action == 1:
                c = i + drop
            else:
                c = i

        current.append(c)

    # Because columns are kept as rows, printing requires the matrix
    # be transposed.
    for i in range(helpers.terminal.rows):
        row = []
        for j in range(helpers.terminal.columns):
            column = current[j]
            if len(column) > i:
                c = column[i]
            else:
                c = ' '
            row.append(c)
        print(''.join(row))

    time.sleep(pause)
    previous = current
```