Exam 1 Review

15-110 – Monday 10/02

Announcements

- Check3 was due today
- Check2/Hw2 revision deadline **tomorrow (Tuesday) at noon!**
- No Gradescope exercise today (no new material)
- **Exam1 on Wednesday in McConomy!**
  - Bring your paper notes (<= 5 pages), something to write with, and your andrewID card
  - Arrive **early if possible** – we're checking IDs at the door

Announcements – Code Reviews

- **Code reviews!**
  - **What:** meet with a TA for 10-15 minutes to get qualitative feedback on your code from your Hw2 submission. Attending the meeting and actively participating gets you 5 points on Hw3.
  - **Why:** code style and structure are important, but not assessed by the autograder. The TA will point out different ways to solve the problems and areas where you can code more clearly or more robustly
    - Some students may be exempted from this meeting if they already have good style. We'll let you know if you're in that group before sign-ups are released.
  - **When:** this weekend (Saturday-Sunday, a few slots on Monday)
  - **Where:** TA's choice

- How to sign up for a code review slot
  - **Link:** TBA on Piazza
  - **Important:** sign-ups for each TA slot close 5pm Friday
  - **Also important:** don't be late! If you are more than 3 minutes late to your meeting, you will not get credit on Hw3.
    - If something comes up and you need to cancel, notify the TA at least an hour before your timeslot. Do not do this multiple times.

Review Topics

- Quick Strings Refresher
- For Loops
- Nesting
- Circuit Addition

Strings

Indexing

In a string (or list), each character (item) has a specific **position**. Positions start at `0` and go to `len(value) - 1`.

You can access an individual character from a string (item from a list) with **indexing,** using square brackets with something that evaluates to an integer.

```
s = "studying"
s[3] # "d"
x = 5
s[x] # "i"
s[len(s) – 1] # "g"
```

Slicing

You can also extract a substring from a string (or sublist from a list) using **slicing**. Slicing uses square brackets with colons in between to specify the **start, end,** and **step** of a slice.

If any of these three components are left blank, they evaluate to a default value – `0` for start, `len(value)` for end, `1` for step. If the step is left to a default, the second colon can also be removed.

```
ready = "Lots of practice!"
ready[8:len(ready)]
    # "practice!"
ready[0:8:2]
    # "Lt f"
```

If you need to calculate slices or indices in a string, make it easy on yourself by writing the index values beneath the characters!

L	o	t	s		o	f		p	r	a	c	t	i	c	e	!
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

For Loops

For Loops

A **loop** is a control structure that lets you repeat a number of statements (the **body** of the loop) a certain number of times.

A **for-range loop** implements this looping by setting the loop control variable to a **pre-determined set of numbers**. The numbers are generated by the **range** expression.

We usually use for loops when we know **exactly how many times we need to loop**.

Example: Code Reading

Consider the following code snippet:

```
count = 0
for x in range(1, 101):
    if isPrime(x):
        print(x)
        count = count + 1
print("Total:", count)
```

If we assume that `isPrime` has been written and works correctly, what does this do?

For Loops over Strings

A common pattern is to loop over each character of a string, and check a property or use the character in a computation.

Here's an example:

```
def allA(s):
    result = ""
    for i in range(len(s)):
        if s[i] == "a":
            result = result + s[i]
    return result
```

What will this function return if called on the string "banana"?

For Loops over Strings

You do: Write `findMatches(s1, s2)`, which takes two same-length strings and returns `True` if they ever have the same character at the same index, and `False` otherwise.

Examples:

`findMatches("apple", "guava")` returns `False`.

`findMatches("apple", "grape")` returns `True`, because the `e`s match.

Nesting

Nesting Changes a Program's Control Flow

Nesting is the process of indenting control structures so that they occur inside other control structures. It is used to manipulate the control flow of a program to produce certain intended effects.

So far, we've learned about several control structures: **function definitions, conditionals, while loops,** and **for loops**. All of these structures have **bodies**, and each can be indented so it occurs inside the body of another structure.

Common Nested Structures - Functions

Though any nesting configuration you can think of is possible, some arrangements are more common than others.

Functions – we usually write function definitions at the top level of a program, and nest conditionals/loops inside them when they're needed. When we **return** in a nested conditional/loop, we exit that structure and the whole function immediately.

```
def hasVowels(s):
    for i in range(len(s)):
        if s[i] in "aeiou":
            return True
    return False
```

Note how the loop is indented inside the function, and its body is indented again.

If the line '`return True`' is reached, the function will exit immediately without finishing the loop.

Common Nested Structures - Functions

It’s also common to include a function call inside the definition of another function.

You do: what will this print?

```
def foo(a, b):
    y = a + b
    print("y in foo:", y)
    return y + 3
def bar(x):
    y = x + 1
    print("y in bar:", y)
    return foo(x, y)
print(bar(4))
```

Common Nested Structures – Loop-Conditionals

Loop-Conditional – very often we nest a conditional inside a loop to check a certain property for every element that is iterated over.

While it's possible to pair an else with the nested if, it's only used if there's a clear alternative action. It's okay to do nothing on iterations that don't meet the requirement!

```
def countVowels(s):
    result = 0
    for i in range(len(s)):
        if s[i] in "aeiou":
            result = result + 1
    return result
```

We don't need to update result if the letter isn't a vowel, so do nothing instead.

Common Nested Structures – Nested Loop

Nested Loop – if you need to iterate over multiple dimensions, a nested loop (one loop nested inside another) will manage the complex iteration. Each loop control variable manages one dimension.

It's important that the two loop control variables have different names, so that they can be referred to separately!

The outer loop moves more 'slowly', as it only iterates once for each complete working of the inner loop.

```
def coordinates(x, y):
    for xNum in range(x):
        for yNum in range(y):
            print("(" + str(xNum) + ", " +
                        str(yNum) + ")")
```

Addition in Circuits

Addition Using Circuits

Let's consider this problem a new way by starting from the goal and working backwards. **How can we teach a computer to add two numbers?**

(Why do we care about this? Computers can only take actions that are built into their hardware. We need to implement the core algorithmic actions – including addition! – if we want to build programs that do interesting things.)

We can't just provide the computer numbers like 127 and 86- we have to translate them to **binary** first. That way, the computer can store them as high/low levels of electricity.

Adding Large Numbers

How do you as a human approach the task of adding two really large numbers? You break it up into parts and solve each part independently.

```
     1 2 7
+      8 6
- - - - - - -
```

An **n-bit** adder will work the same way, by adding one column of numbers at a time. But it will add **binary** digits, not decimal digits.

Adding a column of digits

Now we just need to teach the computer how to add a column of digits.

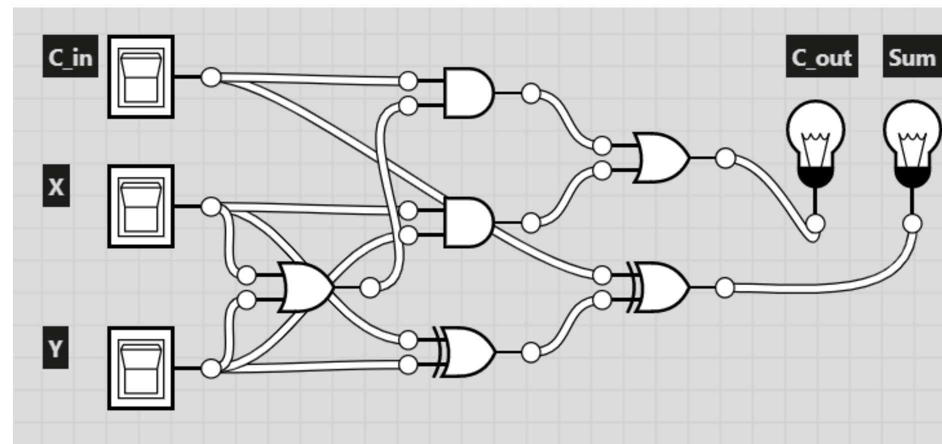
There are only three inputs (two digits and a carried digit), so treat this like learning the multiplication table. **Memorize** all the possible inputs and their outputs.

```
0 + 0 + 0 = 00
0 + 0 + 1 = 01
0 + 1 + 0 = 01
0 + 1 + 1 = 10
1 + 0 + 0 = 01
1 + 0 + 1 = 10
1 + 1 + 0 = 10
1 + 1 + 1 = 11
```

Adding Large Numbers

- In decimal addition, you sometimes have to carry a digit to the next column. The same happens in binary addition.
  - That means we need two output bits (the sum for this column, and the carry to the next)
  - We also need an extra input bit to hold the carry from the previous column

C_{in}	X	Y	C_{in} + X + Y	C_{out}	Sum
1	1	1	11	1	1
1	1	0	10	1	0
1	0	1	10	1	0
1	0	0	01	0	1
0	1	1	10	1	0
0	1	0	01	0	1
0	0	1	01	0	1
0	0	0	00	0	0



Put it all together

Once we have a circuit that can add a whole column of digits (a **full adder**), just chain it together with other full adders to add as many digits as you need.

We 'carry' digits by passing the C_{out} result from one column to the C_{in} input of the next.

Half Adders

Why did we learn about half adders if they aren't used in the final n-bit adders?

Half adders provide a **simplified approach** to adding a single column of numbers. They only work when a number hasn't been carried over, but it's easier to see how the table maps to the circuit.