Exam 1 Review

15-110 – Monday 10/03

Announcements

- Check3 was due today

- **Exam1 on Wednesday!**
  - 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 Check3.
  - **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. Prof. Kelly will let you know if you're in that group before sign-ups are released.
  - **When:** this weekend (Friday-Sunday)
  - **Where:** TA's choice

- **Tutorial:** 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

- Half Adder, Full Adder, N-bit Adder
- Error Messages
- Indexing and Slicing
- Nesting

Addition in Circuits

Addition Using Circuits

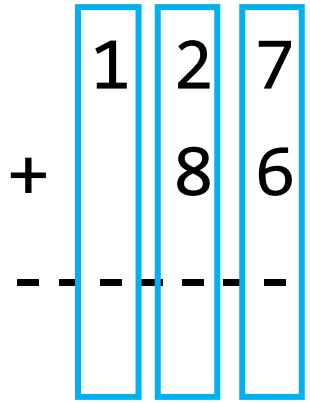
Let's consider this problem a new way by starting from the goal and working backwards. **How can we teach a computer to add two numbers?**

(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.



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
```

Finding the Algorithm

Once you've made a truth table, you can look for **patterns** in the truth table to derive an **algorithm**. That algorithm can then be made into a circuit.

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

C_{out} is 1 when at least two of C_{in}, X, and Y are 1. Combine pairs with **and**, then combine all three possibilities with **or**.

Sum is 1 if an odd number of C_{in}, X, and Y are 1. Use **xor** on all three to get the same result.

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.

Error Messages

Three Error Types

Syntax Errors – Python can't parse the code you've written into a meaningful structure

- Illegal tokens, jumbled token order, missing tokens

Runtime Errors – Python can parse the code you've written, but something goes wrong when it tries to run the code

- Type mismatch, out-of-bounds error, unknown variable

Logical Errors – Python can parse and run the code you've written, but the result isn't what you wanted

- Caught using test cases with `assert`

Practice with Kahoot!

Let's do a Kahoot to practice recognizing errors in code.

Link: https://kahoot.it/

String Indexing and Slicing

Indexing Strings

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"
```

Slicing Strings

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.

```
lst = ["Ready", "for", "the", "exam"]
lst[0:len(lst):2] # [ "Ready", "the" ]
lst[2:] # [ "the", "exam"]
```

You Do: Practice Coding with Indexing/Slicing

You do: write several short lines of code using indexing and slicing to solve problems.

1. What slice would remove the first and last characters from a string `s`?
2. What index would access the middle character from a string `s`? (You can assume it has odd length).
3. What expression would produce a 'doubled' version of a string `s`; for example, the string `"coding"` would become `"codcodinging"`?

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 – 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!

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

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

You Do: Parsons Puzzle

A **Parsons Puzzle** is an activity where you're given all the lines of code in a program, but you need to rearrange them and indent them to make the program work.

Arrange the program for the function `sameChars`, which takes two strings (`s1` and `s2`) and returns `True` if `s1` contains only characters that also occur in `s2` (and `False` otherwise).

Hint: consider how you would solve this problem yourself. What order do the steps need to take place in? Also, beware the distractor line!

Link: http://parsons.problemsolving.io/puzzle/2222cffb733d4361b93e73cd2f41879c