# Exam 1 Review

15-110 – Monday 09/29

# Announcements

- Check3 was due today
- Check2/Hw2 revision deadline **tomorrow (Tuesday) at noon!**
- No Gradescope exercise today (no new material)
- **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 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 by Monday EOD.
  - **When:** this weekend (Saturday-Sunday, a few slots on Monday)
  - **Where:** TA's choice

- How to sign up for a code review slot
  - **Link:** https://www.cs.cmu.edu/~110/hw/hw2-code-review.html
  - **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

- Looping over Strings
- Nesting
- Addition in Circuits

# Looping Over Strings

# Strings are Made of Characters

Strings are naturally composed of many individual parts – the individual characters that compose the string. This means we can write algorithms that work with the individual parts, using a **loop**.

How do we access a single character out of a string? Use indexing!

```
s = "testing"
i = 3
s[i] # "t"
```

# Loop Over Indexes

How can we access **all** the characters in the string? Use a loop to visit each character, one at a time.

For now it's easiest to loop over the indexes of the string. Start at `0`, increment by `1`, and end at `len(s)`; we can use `range(len(s))` to visit all the positions.

```
for i in range(len(s)):
    print("Character:", s[i])
```

# Problem Solving with Strings

What do we do inside a loop? It depends on the problem we're solving.

Let's look at a few common examples…

# countUpper

**Problem:** given a string, count the number of uppercase characters in the string.

```
def countUpper(s):
    result = 0
    for i in range(len(s)):
        if s[i].isupper():
            result = result + 1
    return result
```

# makeMatch

**Problem:** given two strings of equal length, make a new 'match' string. To make this string go through each position in the strings; if the characters at the same position are the same, include it; otherwise put an `"X"` at that position. For example, `makeMatch("cat", "car") -> "caX"`

```
def makeMatch(s1, s2):
    result = ""
    for i in range(len(s1)):
        if s1[i] == s2[i]:
            result = result + s1[i]
        else:
            result = result + "X"
    return result
```

# You do: usesLetters

**Problem:** the function `usesLetters` is given two strings, `word` and `letters`, and returns `True` if `word` only contains characters in the string `letters`, and `False` otherwise. For example:

```
usesLetters("happy", "ahpy") -> True
usesLetters("happy", "ahnry") -> False
usesLetters("aaaaaaa", "par") -> True
```

# 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

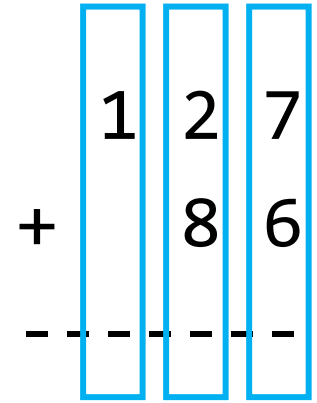
Let's consider this problem a new way by starting from the goal and working backwards. **How can we teach a computer to add two numbers?**

(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.




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 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

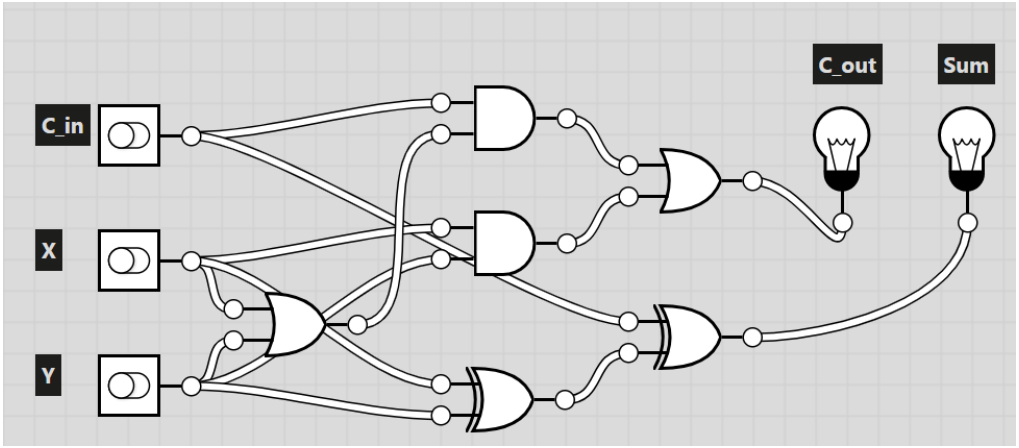
There are only three inputs (two digits and a carried digit), so treat this like learning the multiplication table. **Memorize** all the possible inputs and their outputs.

```
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

Using some problem solving beyond the scope of this class, we can figure out which gates to use to correctly generate the sum and carry bits from the three inputs.

| $C_{in}$ | 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.