Unit 1 Review

15-110 – Monday 02/22

Announcements

- Check2 was due today
- Week1-2 revision deadline is **tomorrow at noon EST**
- **Quiz1 on Wednesday**
  - Make sure you practice using LockDown Browser before then!

Agenda

- Unit Overview
- Half-Adders / Full-Adders / N-bit Adders
- Function Call Stack
- For loops
- While loops

Unit 1 Overview

Unit 1 Goals

Our first unit had two major themes: developing key **programming skills** and understanding the basics of **computer organization**.

How do the topics we discussed fit into these themes?

Programming Skills

We started with **programming basics**. A program is an implementation of an **algorithm**. **Data** and **variables** are the core part of any program. Variables also have **scope** based on where they are defined.

While programming, we'll sometimes run into **errors**. We learned the basic **error types**, discovered **what causes them**, and discussed **debugging**.

Programming Skills (continued)

We use **control structures** to change how we move through the steps of a program. **Nesting** control structures lets us create more complex algorithms.

Conditionals let us choose whether or not to run a series of steps.

Loops (either **while loops** or **for loops**) let us repeat actions, as long as we define the **loop control variable**.

Functions let us **define** an algorithm under a name and **call** that function later on. A function has **argument(s)**, a **returned value**, and **side effect(s)**.

Computer Organization

We also discussed the basics of how **computers organize data**. This is done partially through the process of **abstraction** to change levels of detail in systems.

We discussed how the computer **tokenizes, parses, and translates** Python into a language the computer understands.

We also discussed how the **call stack** helps the computer keep track of information.

Computer Organization (continued)

We explored how computers represent data in **binary** and implement algorithms using **circuits**. We explored how circuits can also be represented as **Boolean expressions** and **truth tables**.

We discussed how these concepts can be abstracted by implementing **addition via circuits** and implementing **text and colors via binary**.

Upcoming Topics

In the next unit, we'll dive deeper into programming by focusing on algorithm design. We'll discuss **data structures** (new ways to organize data) and **efficiency** (how to determine how 'fast' our algorithms are).

We'll get back to computer organization in Unit 3, where we'll discuss how computers **scale up** to work on much larger inputs.

Addition in Circuits

Addition Using Circuits

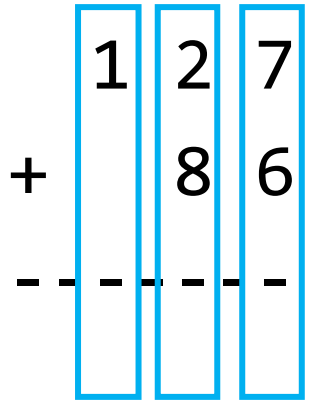
Let's consider this problem a new way by starting from the goal and working backwards. **How can we teach a computer to add two numbers?**

(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.

Function Call Stack

Control Structures and Execution

When Python runs a program, it runs through the statements sequentially until a **control structure** comes along. Control structures change how functions decide which line of code to run next. We call this **control flow**.

A **function definition** is a control structure, because when the code reaches a definition, it doesn't execute the code in the definition normally. Instead, it **loads** the definition into memory, to be revisited if needed.

When Python reaches a **function call**, it **redirects** the control flow by jumping to the associated definition. The code will run through the definition until it hits a **return** statement, then go back to where it was originally called. It's like clicking on a URL on a website to look something up, then clicking the 'Back' button to go back to where you were before.

The Call Stack manages memory

How does Python know where to go back when it's done with a function? It uses the **call stack**, both to keep track of where it needs to go *and* to keep track of the local **state** at each point in the program.

Understanding the function call stack helps us with *code reading*. Being able to read and trace code is an important skill in programming.

Let's do an example together of tracing a function call stack.

Example code

```
def a(x, y):
    z = x + y
    if b(z):
        x = x + 1
    return x - y

def b(y):
    y = y * 10
    return y == 110

x = 6
result = a(x, 5)
print("Result:", result)
```

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 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?

Example: Code Writing

Now let's write code with a for loop. Let's write the isPrime function from before. It should take an integer and return `True` if the int is prime, and `False` otherwise.

While Loops

While Loops

A **while loop** implements a loop, just like a for loop. However, while loops work a little differently. Instead of pre-deciding what to loop over, while loops keep running the code in the body **while the condition is True**.

This means while loops take a little longer to write, since you have to set up and update the loop control variable yourself. But it also means we can write programs where we don't know how many times we need to loop.

Example: Code Reading

What does the following function do?

```
def mystery(num):
    x = num + 1
    while not isPrime(x):
        x = x + 1
    return x
```

Example: Code Writing

Could we write `isPrime` using a while loop instead of a for loop?

Sure! **Anything that a for loop using a range can do, a while loop can do too**.

Agenda

- Unit Overview
- Half-Adders / Full-Adders / N-bit Adders
- Function Call Stack
- For loops
- While loops

- Feedback: http://bit.ly/110-s21-feedback