CSCI 150: Foundations of Computer Science
Fall 2015 Lecture Notes

January 6, 2016

1 Introduction (Wednesday, 26 August)

Setup

- Music: Copland, Fanfare for the Common Man
- Meet students.
- Introduce myself.
- Have students meet each other. (Name, where from, why taking this class, one of your favorite things.) Prize for first student to know all names.

What is Computer Science?

This class is Foundations of Computer Science. What is Computer Science? Look at each component separately.

- **What is a computer?** Get some responses.
- **What do computers do?** Lots of things. Commonality: information. Communicating, transforming, analyzing, storing.
- **What is science?** Get some responses. Scientists use rational investigation & analysis, mathematics, etc. to study the natural world.

So what do computer scientists study? **NOT computers!** Rather, **information** = anything that can be expressed digitally, i.e. with numbers/symbols. "Computer science" is actually a bad name, *cf.* "telescope science". Information structure of the universe.

Why study it? (Maybe get some student responses)

- Beautiful ideas, new ways of thinking.
- Many applications! Can contribute directly to human flourishing.

- Computers are everywhere. Understanding principles of CS = being an informed, engaged citizen.

Could do this without a computer, but computers are excellent enabling tools.

Watch code.org video.

Administrivia

- Syllabus review online.
- Academic integrity.
- BYOL.
- Moodle.
- Office hours.
- Remember to come to lab!

For next time:

- Read chapter 1 of textbook.
- Do HW 0, "who are you", if not already done.

Scratch intro

Basic intro to Scratch for first lab.

2 Algorithms and language (Friday, 28 August)

Setup

- Music: Attaboy from Goat Rodeo Sessions. 5:42. Start at 8:04.
- Bring origami paper.

Administrivia

- I will be gone next week. ICFP. Functional Programming — super cool. Really get to see intersection of math & CS. Take CSCI 490 in the spring (with MATH 240)!
- In my place, Connor Bell will be lecturing & will introduce you to Python. Please show him the same respect you would show me!
- Please download Python for Monday. (Show python website, linked from our webpage. Use version 2.7.) **Bring your laptop!**

IRC

Show IRC channel. Explain what it is for. Explain ground rules. Have everyone log in & try it out.

Reading review

- Review 5 aspects of algorithms (input, output, math, conditionals, repetition). Where did we see them in Scratch? Didn't use conditionals but saw the others.
- Review 3 kinds of errors (syntax, semantic, runtime). Talk about each in context of Scratch (doesn't have syntax errors !!!; lots of semantic errors; runtime = running into wall)
- Fact that Scratch doesn't have syntax errors is a Really Big Deal. Imagine if when learning a foreign language, every time you made even a small grammatical mistake the other person just cut you off and said "I don't understand." That's what it will feel like learning Python. So don't be discouraged—remember what you could do with Scratch. You'll get there with Python too.
- Talk about formal vs natural languages, tokens, parsing

Quiz Monday on Chapter 1.

Collatz Conjecture

- Write out hailstone function.
- Note all 5 aspects of algorithm.
- Try it on some inputs: 4, 8, 6, 11. Draw a tree etc. Try 27, realize we need a computer.
- Simple algorithms can have very surprising results! Visit Wikipedia page.

Origami

- Explain what we are going to do: make a dinosaur.
  - Everyone will be able to do it by the end of class.
  - Then write instructions IN ENGLISH and find a confederate to follow them. Can't show them image or video, or your dinosaur. Can't help interpret your instructions. Just tell them to keep going as well as they can and get to the end.
  - Monday: turn in both dinosaurs in class. Turn in your instructions & writeup on Moodle.
- Hand out origami paper.
- Show video & instruction image on the screen at the same time.
- Wander around and make sure everyone can do it.

3 XXX

4 XXX

5 XXX

6 Conditionals (Wednesday, 9 September)

Setup

- Music: Whitacre "i thank You God". 6:56. Start at 8:03.
- Talk about Fabienne Serriere. Show website.
- Collect puzzle HW.

Quiz

Quiz 2

We saw Boolean values `True` and `False`, in the puzzles, with operators `and`, `or`, `not` last time. How else can we get Booleans?

Comparison operators: `>`, `<`, `>=`, `<=`, `!=`, `==`. Demonstrate on numbers, strings, all generate Booleans.

Now, the computer has a basis to make decisions. Sets up branches in the code, execute this or that.

```
passwd = input("What is the password? ")
if (passwd == "lemur"):
    print "Here be secrets."
```

Tabbing is important! Colon is important!

Introduce random module. (`from random import *`)

- `random()` is number in $[0, 1)$. Uniform.
- `randint(n)` is random integer between 0 and n (inclusive). (Aside: how can we write this in terms of `random()`?)

Do coin flip. Conditionally do... something. Needs to include

- `else`
- `elif`
- nested `if` (flip two coins in a row)

7 Information encoding I (Friday, 11 September)

Setup/administrivia

- Music: Dave Brubeck Quartet, Kathy's Waltz
- NB: exam 1 a week from today. In class, closed notes, closed computer. Covers material through today and Monday. Wednesday will be exam review.
- Collect any remaining logic puzzle HW!

Ada Lovelace

How many of you have heard of Ada Lovelace? How about Charles Babbage?

Ada Lovelace: 1815 (December 10!)–1852, first programmer. Wrote programs for Charles Babbage's Analytical Engine (never built, but definitely works in principle). Babbage's purpose for the AE was limited to making tables of numbers, but Lovelace had a much more expansive and far-seeing vision:

> "[The Analytical Engine] might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine...
>
> Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent."

She saw potential for computers to operate on many kinds of information instead of just making tables of numbers. Over the next 2 classes we will consider some of the fundamentals that make this possible.

Binary

Computers represent numbers using *binary* = base 2 instead of base 10. Recall how base 10 works: ones ($= 10^0$) place, tens ($= 10^1$) place, hundreds ($= 10^2$) place, ... And ten different digits. For example

$$4397_{10} = 4 \times 10^3 + \ldots$$

Base 2 is the same but

- We use 2 instead of 10 (ones place, 2's place, 4's place, 8's place...)

- We use 2 digits $(0, 1)$ instead of 10. Called *bits* = *bi*nary dig*its*.

Computers do everything in binary since (1) it is no less expressive than e.g. decimal and (2) from a physical point of view, it is easier to design hardware that can distinguish two different states (e.g. high and low voltage) than 10.

Converting base 2 to base 10: have them do some examples. e.g. $10110100_2 =$? Then do one example together on the board,

$$10110100_2 = 1 \times 2^7 + 1 \times 2^5 + \ldots$$

Note 1 *byte* = 8 bits. Incidentally, one *kilobyte* is not 1000 bytes, but actually $2^{10} = 1024$ bytes! Megabyte is 2^{20} bytes, and so on.

Questions for students (pair & share etc):

1. How many different binary numbers using n bits are there? (Write out 16 binary numbers using 4 bits.)
2. What is the biggest number that can be represented using n bits?
3. How many bits are required to count up to n?

Algorithm for converting from n in base 10 to binary:

- Find largest power of two $\leq n$, say, 2^k.
- As long as $n > 0$:
  - If $2^k \leq n$, subtract 2^k from n and write a 1 (in the 2^k place). Else, write a 0.
  - Decrease k by 1.

(This has all five usual aspects of algorithms: input, output, math, conditionals, repetition. We can't quite write this algorithm in Python yet because we don't know how to do repetition. Soon!) Do an example, e.g. 103_{10}.

Representing integers

Integers are represented like this inside the computer. Show entering binary numbers directly into Python using `0b10110` notation.

Most modern computers use 64 bits to represent an integer (*how big is that?*); some use 32. Show Python switching from `int` to `long`:

- Some small and very big examples
- `2**62`
- `2**63`
- Apply `type` to each of the above.

What about negative numbers? One possibility: use one bit for sign (plus or minus). (Turns out there is a better way, "2's complement"; learn about it in CSO or ask if you are curious.)

8 Information encoding II (Monday, 14 September)

Setup

- Music: ?
- Quiz 3

Representing floating-point numbers

Back to base 10: what does 123.45 mean?

$$\ldots + 4 \times 10^{-1} + 5 \times 10^{-2}$$

We can do binary "decimals" ("binarals"?) the same way:

$$1101.011 = 1 \times 2^3 + \ldots + 1 \times 2^{-1} + 1 \times 2^{-2} = 13\frac{3}{8}$$

Also recall "scientific notation", e.g. 1.23×10^{17}. This is how "floating point" numbers are represented: scientific notation, but in base 2. For example, with 64 bits:

- 1 bit for sign ($\pm$)
- 11 bits for exponent (base 2 integer)
- 52 bits for value

e.g. -1011011×2^{-3}.

How would you represent 0.1_{10}? Can't represent exactly using base 2 (infinite)! Show $0.1 + 0.1 + 0.1$ at python prompt.

Representing text

Basic idea: use a different number to represent each letter. ASCII (American Std. Code for Info. Exch.) — early 1960's. Specified 128 different characters, each using 7 bits. (In many cases 1 bit left over for parity checking or just set to 0.)

- Show ASCII chart.
- Illustrate chr and ord functions in python.

128 characters may have been enough for the white, American, English-speaking men who made it up. But it sure isn't any more. Unicode—currently over 120,000 characters. (How many bits needed? Often uses a more complex scheme to allow different numbers of bits, & extending indefinitely without changing existing. Ask if you're curious.)

Representing images

Image = grid of colored points ("pixels" = *pic*ture *el*ements). Each pixel = mix of red, green, blue (additive primary colors). Each primary color has 256 possible intensities, from 0 (off) to 255 (as bright as possible). So each primary = 8 bits (1 byte), each pixel = 24 bits. How many bits/bytes for a 500×500 image?

Hexadecimal

Base 16. Need 16 symbols: 0–9, a–f. So $a_{16} = 10_{10}$ and so on. $\text{ace}_{16} = ?$

Note, we can group the binary digits into 4s. $16 = 2^4$. So each 4 bits corresponds to 1 hexadecimal digit. Conversion back and forth is super easy. We often use hexadecimal as a more convenient way to read and write binary. Easier for humans to read and remember.

Show entering hexadecimal directly into Python using `0xace` notation.

Show example of RGB colors expressed in hexadecimal: show `style.css` from course website.

9 Exam 1 review (Wednesday, 16 Sep)

Have them do sample exam. Go over it. Things to emphasize:

- "Semantic errors": looking for explanation of how/why the program works.
- Don't just print out a number, print it with a nice message!
- ...

10 Exam 1 (Friday, 18 Sep)

Functions I (Monday, 21 Sep)

Setup

- Music: Darlingside "Good Man"
- Lovelace & Babbage: prize for whoever can say names of all classmates on two separate occasions

Exam 1 review

- Hand back exams.
- Go over some solutions:
  - Write exemplar solution for programming question
  - Go over semantic errors question
  - Come ask me or Dr. Goadrich with questions about other problems.

Project overview

Show them the project. Calendar strangeness due to tracking floating point value (ratio of revolution to rotation) using integers!

Functions

Read Chapter 5! Note quiz on Wednesday!

We have used functions in Python already, like `raw_input`, `cos`?, other math things?

We can define our own functions! Same rules for names as variables. Header line: parens mean no inputs, colon, indent. Stuff inside the indent (body) says what happens.

```
def say_hi():
    print "Hello there, CSCI 150!"
    print "This is a function."
```

Function definition makes a variable which is a function object. (Note hex value in output! =) To call it, use parentheses.

Functions can be used inside other functions.

```
def say_hi_twice():
    say_hi()
    say_hi()
```

Put things inside a script. Note the definition itself does not cause body to be executed. Note functions must be defined before they are used.

Why functions?

Get them to discuss why this is a worthwhile feature. Why define functions? Pair & share.

From Downey:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Parameters and arguments

Make C to F conversion into function, takes C as argument:

```
def C_to_F(temp_C):
    F = C * 9.0 / 5.0 + 32
    print "The temperature in degrees Fahrenheit is ", F
```

`temp_C` is a variable called a *parameter*.

The values given to functions are called *arguments*. To call a function, give one argument for each parameter. The parameter will stand for the argument. Do some examples. Note you can put arbitrary expressions for arguments.

Variables created inside functions are local! Parameters are also local. Note that `F` and `temp_C` are not defined afterwards.

Functions II (Wednesday, 23 Sep)

Setup/administrivia

- Music: Holst Jupiter
- Send me your exam grades!
- Does anyone know all their classmates' names?

Quiz: arguments & parameters

Fruitful functions

Recall function from last time:

```
def C_to_F(temp_C):
    F = C * 9.0 / 5.0 + 32
    print "The temperature in degrees Fahrenheit is ", F
```

This function is actually really awkward. Takes `temp_C` as a parameter, but then *prints* the result. What if we want to compute this but not print anything? Make each function do ONE job. Better:

```
def C_to_F(temp_C):
    F = C * 9.0 / 5.0 + 32
    return F
```

Book calls it a *fruitful function*. (As a mathematician I would just call it a *function*...) Note we could also get rid of the `F` and just return the whole expression; matter of taste/experience.

`return` means "stop execution of the function right now and return this value". What will this do?

```
def C_to_F(temp_C):
    F = C * 9.0 / 5.0 + 32
    return F
    print "The temperature is", F
```

OK to have multiple return statements. Let's try a function to tell us about the temperature (*do something like this, students can come up with details*):

```
def F_to_qualitative(temp_F):
    if F < 32:
        return "freezing"
    elif F < 50:
        return "cold"
    elif F < 75:
        return "warm"
    elif F < 90:
        return "hot"
    else:
        return "scorching"
```

Functions can take multiple parameters.

```
def is_divisible(x, y):
    if (x % y == 0):
        return True
    else:
        return False
```

Try some examples. Note this is actually written in a redundant way. We can just return `x % y == 0` directly.

Flow of execution & stack diagrams

Execution starts at the first line of a program and continues. But when a function is called, execution jumps to that function, and when done picks back up where it left off. This can get complicated, since functions can call other functions! In general Python has to keep track of a "stack" of places to resume. Imagine you were reading a book but it told you to go read another book and then come back. Then THAT book told you to go read another book, etc. You would keep the old books, with bookmarks, in a pile. When done, pick up the next book from the top of the pile and continue.

Do an example. Write a function that calls `C_to_F` on several temperatures. Then write another function which does that twice, and so on. Then trace execution. Introduce an error and see the "traceback". Draw a stack diagram with local variables.

While loops (Friday, 25 Sep)

Setup

- Music: Carmina Burana
- Announce TAs!
- Reminder: Project 1 due today
- Announce office hours after class—projects
- Anyone know classmates' names?

Guess my number game

- First, play the game with class.
- Try writing a program to play it (human guesses). Have to replicate code. . . ?

While loops

Introduce while loops. Explain basics of how they work. Using them, three important things:

- Declare and initialize variable(s) you need
- Write the condition with those variables
- Change those variables somewhere in the loop

Do a few examples.

- Count from 1 to 10.

- Keep printing "hi" until user wants to stop — have students write this code together on their computers. Then have two or three share code?

Back to Guess My Number. Introduce a loop. Make hi/lo checking into a function. Promise: after we learn a bit more about strings we can make another function to get valid int input from user in a loop.

Homework (due Monday): write version where computer guesses!

Strings (Monday, September 28)

Setup

- Music: Giant Steps
- Classmates' names? New rules: MOST by end of semester
- Show dinosaur comic: `http://www.qwantz.com/index.php?comic=2883`

Guess My Number review

Open code from last time. Talk about modifying it for other version.

Remember:

- Variables are local!
- No recursion, just loops. Talk about recursion later.

Strings

We sort of know about strings, but the only thing we can do with them so far is concatenate them with `+`.

- Strings are *sequences of characters*.
- We can reference individual characters in a string by their *index*: `mystring[3]`. Note that *indices start at 0*.
- We can calculate the length of a string with `len()`. (Relationship of `len` to valid indices?)
- String slices: `mystr[a:b]`. Have students play around to figure out details?
  - Characters starting at index a, up to *but not including* index b.
  - Can leave off first, or second, or both.
  - Negative indices count backwards from the end.
- We've seen `+`: concatenation.

- `*` does repetition.
- Strings have *methods*, i.e. functions you can call on strings. Do `help(str)`, `help(str.upper)` etc.
- Show some useful meethods: `upper`, `count`, `replace`, `find`, `isdigit`

10.1 More while loop & function practice

First, function to "explode" a string. Write it collaboratively.

```
def explode(s):
    i = 0
    while i < len(s):
        print s[i]
        i += 1
```

Even better if they make some bugs in the above, run it and see what is wrong.

While loop & string practice (Wednesday, Sep 30)

Setup

- Music:
- Show next dinosaur comic
- Classmate names record to beat: 7 (Mason)
- Lab today/tomorrow. Not easy! Please take the pre-coding design component seriously.

More while loop practice

Start using docstrings. From now on they should always write them for their functions. Show them that we can type `help(int_input)` at a prompt.

```
def int_input(prompt):
    """Repeatedly prompt the user for a number using the
    given prompt, until they enter a valid integer.
    """
    valid = False
    while (not valid):
        uinput = raw_input(prompt)
        if uinput.isdigit():
            valid = True
        else:
            print "That's not a valid integer"
    return int(uinput)
```

Do version of `int_input` that accepts negative numbers as well.

```
import time
def countdown(n):
    """Count down from n to zero."""
    while n > 0:
        print str(n) + "..."
        time.sleep(1)
        n = n - 1
    print "BLASTOFF!"
```

Talk about `n -= 1` syntax ("decrement") and `n += 1` ("increment"). Can also use `*=`, `/=` etc.

Collatz Conjecture. Idea: write a program to test the conjecture. What functions will we need? Bottom-up approach. Write each function and then test it independently. (Use docstrings.)

```
def hailstone(n):
    if n % 2 == 0:
        return (n/2)
    else:
        return (3*n + 1)

def collatz(n):
    count = 0
    while n != 1:
        n = hailstone(n)
        count += 1
    return count

def check_collatz_up_to(max):
    n = 1
    while n <= max:
        print n, collatz(n)
        n += 1
```

(What happens if we take out `max` check and just say `while true`? A legitimate use of an infinite loop!)

Lists I (Friday, 2 Oct)

Setup

- Music: Melody of Rhythm
- Open Quiz 5 form
- Hand back HW & quiz
- Distribute binary cards. Learn the name of 0-bit pairing.

Quiz 5 (Strings)

Lab (Doublets) review/hints

1-bit pairing with binary cards and discuss lab. Questions?

Lists

Lists are a lot like strings, but hold arbitrary data instead of characters.

- `[]` is the empty list, like `""` is the empty string
- Can hold numbers, strings, anything you want. Show `[1,2,3]` syntax. Show `type` on lists.
- Elements are ordered, in a sequence. Indexed from zero.
- Show `range` function.
- `+`, `*`, `len()`, slices all work the same

Mutating lists

Strings are *immutable*, lists are *mutable*.

```
animals = ['cat', 'dog', 'bird']
animals[1] = 'lemur'
print animals
```

Draw a stack diagram of the above situation.

Other list mutation methods:

- `t.append(element)` for adding to end
- Removal:
  - `t.pop(index)` by index, returns deleted elt
  - `t.remove(element)` by element
  - `del t[index]` by index, doesn't return anything BUT `del t[slice]` works too

Lists II (Monday, 5 Oct)

Setup

- Music: Roomful of Teeth
- Project 1 is graded. Let us know of any questions. Let me know if you want more feedback.
- Hand out bit cards.

Review

Find bit-2 partner. `animals` is a list, how to append 3 copies of `"gerbil"` onto the end? Go over possibilities.

Lists and strings

Lists of characters are different than strings!

- Converting strings to list:
  - `list` function.
  - `split()` method.
- List to string: `join()` method on delimiter.

```
s = "banana"
t = s.split("n")
```

Yields `["ba", "a", "a"]`.

```
t = ['fun', 'sun', 'sub', 'rub']
delimiter = " -> "
delimiter.join(t)
```

Mutability, objects, and aliasing

Warm-up: what happens here? Find bit-3 partner & decide.

```
nums  = [1,2,3]
nums2 = nums
del nums[1]
print nums2
```

Try it at IDLE prompt. This is really weird.

As hinted last time, the fact that lists are mutable opens a huge can of worms. Let's start by pulling back the curtain a little bit. Up until now we have talked about variables having values, but we need to be more precise. Draw stack diagram with variables referencing two different lists. Write these definitions on the board:

- Variables reference *objects* which have a *value*.
- Variables which reference objects that have the same value are *equal*. You can test equality with `==`.
- Variables which reference the *same object* are *identical*. You can test identity with `is`.

Do this at the REPL, with accompanying pictures:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a == b
True
>>> a is b
False
>>> del a[0]
>>> a
[2, 3]
>>> b
[1, 2, 3]
>>> a == b
False
>>> a = b
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
>>> a == b
True
>>> a is b
True
>>> del a[0]
>>> a
[2, 3]
>>> b
[2, 3]
```

Note in particular that the assignment `a = b` does not copy the value of `b` into `a`, but just makes `a` *reference the same thing as* `b`.

This has implications for function parameters as well. Consider:

```
def animal_release(animals):
  while len(animals) > 2:
  animals.pop(0)

print animals
animal_release(animals)
print animals
```

Ask: what gets printed? Draw stack diagram to understand.
Some related things:

- What if you really *do* want to make a *copy* of a list, so modifying it doesn't modify the original? You can write `some_list[:]`. This is a dirty, terrible, disgusting hack. Perhaps someday you will understand why I feel this way.
- Be careful about methods that modify vs functions that make a new thing. e.g. `list.sort()` modifies the list. But `sorted(list)` makes a new list! And of course `string.upper()` makes a new string because strings are immutable. Confusing, don't memorize: the point is you should be careful and look up help to see what things do. Show `help(list.sort)`, `help(sorted)`, `help(str.upper)`.

[INSERT OPTIONAL RANT ABOUT MUTATION HERE]

For Loops (Wednesday, 7 Oct)

Setup

- Music:
- Announce: exam 2 next Wednesday

For Loops

Recall our `explode` function from a week or two ago:

```
def explode(s):
  index = 0
  while index < len(s):
    print s[index]
```

Recall how easy it was to make errors like starting index at the wrong place, using `<=` instead of `<`, and so on. Well, it turns out there is a better way!

```
def explode(s):
  for c in s:
    print c
```

Simply loops through each character in the string one at a time. Each time through the loop the next character is assigned to `c`. This also works for lists:

```
animals = ["cow", "chicken", "pig", "rabbit"]
for animal in animals:
  print "And on this farm he had a " + animal + "!"
```

What do we lose, as compared to a `while` loop? The index. But actually, we don't! Just have to cleverly put together a few things we have already seen:

```
for i in range(len(numbers)):
  numbers[i] *= 2
```

This is way better than using a `while` loop since it rules out errors putting the starting or ending the indices at the wrong number.

For loop practice

```
def print_each(s):
    index = 0
    while index < len(s):
        c = s[index]
        print(c)
        index += 1

def print_each_for(s):
    for c in s:
        print(c)

def count(s, c):
    num = 0
    for c2 in s:
        if c2 == c:
            num += 1
    return num

def find(s, c):
    for i in range(len(s)):
        if s[i] == c:
            return i
    return -1

def find_all(s, c):
    all_the_things = []
    for i in range(len(s)):
        if s[i] == c:
            all_the_things.append(i)
    return all_the_things

def nos(s):
    news = ""
    for c in s:
        if c != "s":
            news += c
    return news
```

```
def sum(t):
    total = 0
    for n in t:
        total += n
    return total

def mean(t):
    total = 0
    for n in t:
        total += n
    return float(total) / len(t)

def mean2(t):
    return float(sum(t)) / len(t)

def vacation(s):
    for i in range(len(s) - 1):
        if s[i] == s[i + 1]:
            return True
    return False

def classify(strs):
    for s in strs:
        print ('*' * len(s))

def powersof2(n):
    the_powers = []
    for i in range(n):
        the_powers.append(2 ** n)
    return the_powers

def triangle(n):
    total = 0
    for k in range(1,n+1):
        total += k
    return total
```

For loop practice (Friday, 9 Oct)

Setup

- Music: Waltz for Debby
- Pass out binary cards

For loops: reading exercises

CLOSE COMPUTERS. Find bit-partners. Different one for each exercise. Go over each and try it.

```
def oogie(m):
    p = m[0]
    for g in m:
        if g > p:
            p = g
    return p

def yayaya(q):
    for y in range(len(q) - 1):
        if q[y] == q[y + 1]:
            return True
    return False

def pulu(r):
    b = []
    for k in range(r):
        b.append(k ** 3)
    return b
```

Exam 2 review (Monday, 12 Oct)

Exam 2 (Wednesday, 14 Oct)

(Fall Break)

Exam wrap-up, demo project (Monday, 19 Oct)

10.2 Exam wrap-up

Hand back exams; quickly go over exam solutions.

10.3 Demo project

Introduce Project 2 (word games).

Livecode the beginning of an exemplary Doublets lab. Things to focus on:

- Start with `main`. Hierarchical decomposition; write stub functions.
- Think about what information each function needs to do its job, and what information it returns.

- Write `get_starting_word` function.
- Start writing `get_ending_word` function, realize that they are going to have a lot of similar code.
- Abstract common part out into a new function, `get_word`, that takes a parameter describing the desired word.
- Stub out a single round of the game, then talk about adding a while loop.

Recursion I: intro (Wednesday, 21 Oct)

Setup

- Write stone-counting rules somewhere (see below)?

Stone counting

Rules:

1. You may not look inside the bowl.
2. You may only touch one stone.
3. If someone gives you the bowl and asks you how many stones it contains, you must eventually give the bowl back and answer truthfully.
4. You may give the bowl to someone else and ask them how many stones it contains.

Hand the bowl to a student and ask how many stones it contains.

Recursion! Functions that call themselves. In a sense you were all "the same function" because you were running the same program (rules).

Factorial function: $n! = 1 \times 2 \times \ldots \times (n-1) \times n$. Let's implement in Python.

```
def fact(n):
    product = 1
    for k in range(1,n+1):
        product *= k
    return product
```

Now consider $n! = (1 \times 2 \times \ldots \times (n-1)) \times n = (n-1)! \times n$. Also need to define $0! = 1$. Can we translate this directly into Python?

```
def fact_rec(n):
    if n == 0:
        return 1
    else:
        result = fact_rec(n-1)
        return result * n
```

It works!

Show call stack. Every recursive call has its own copy of local variables!

Big idea: transform a problem into a slightly simpler version, solve that simpler problem, do a bit of work to compute an answer to the original problem.

Typically:

- Base case(s): simple case where we know the answer without doing any work. Always do this first!
- Recursive case(s): make a recursive call to solve a slightly simpler (i.e. closer to the base case) problem.

Another example: Fibonacci function. Show mathematical definition, implement in Python. Show how slow it is, show call tree. In fact recursion is not always the best way to do things.

If time, brief intro to Python turtle library and Kock snowflake (in preparation for fractal lab).

Recursion II: recursive functions (Fri, 23 Oct)

Start by reinforcing "base case + recursive case". Talk about "leap of faith": trust that recursive calls will work without thinking about how. Trust your future self.

Re-code factorial function live, adding a bunch of comments and narrating thought process.

Up until now all the examples we've done have been math functions. Let's do a few more "computer-sciency" examples processing strings and lists. Note, in Python this is not the best way to implement these functions (though in some languages it is). Just for illustration purposes. If you take 151 you will start to see real, legitimate uses of recursion, e.g. processing trees.

- Do example together: list sum.
- Have them work in pairs to implement string `reverse`.
- Have them work in (different) pairs to implement `is_palindrome`.

File I/O, dictionaries (intro) (Mon, 26 Oct)

Our programs have no state between each time they are called. Everything is stored in memory during the program, but afterwords, its all gone. This is RAM. We need hard drive storage to keep it around. Save and read things in files.

To write to a file:

```
fout = open("myfile.txt", "w")
```

then to save data in the file, write out as strings

```
fout.write("hello\n")
fout.write("world\n")
fout.close()
```

Note the newline characters.

To read from a file:

```
fin = open("myfile.txt", "r")
```

Two ways to get stuff out of the file.

```
s = fin.read()
```

which returns a string that includes all the carriage returns, or

```
s = ""
for line in fin.readlines():
    s += line.strip()
```

which strips them out of each line as it is added. `readlines()` returns a list of strings.

After both, say `fin.close()` to finish accessing the file.

Then do basic intro to dictionaries? Next time, need to figure out some kind of activity to get them used to the idea. Covering technical details does not take 1.5 lectures.

Dictionaries (Wed, 28 Oct)

Setup

- Music: Trio Medieval
- HW: read Section 1 from Zen for Friday

Recall lists. Draw an alternate picture of lists as a mapping from indices to values. For example `['tapir', 'ferret', 'alligator']` can be thought of as

```
0 -> 'tapir'
1 -> 'ferret'
2 -> 'alligator'
```

0, 1, 2 are *keys*, animal names are *values*. A *mapping* or *association* between keys and values. A *dictionary* is the obvious generalization of this to arbitrary keys instead of just 0, 1, 2 .... For example, suppose we are keeping these animals in a zoo and we want to keep track of them by their habitat ID numbers.

```
'A23' -> 'tapir'
'B19' -> 'ferret'
'B12' -> 'alligator'
```

Or we could use numbers as keys. Important point: keys have to be immutable. So we can't use lists or dictionaries.

Show how to make this dictionary.

- Write using curly braces and colons.
- Alternatively, start with empty one and add them.

Other things we can do with a dictionary:

- Get all keys with `keys()`
- Get all values with `values()`
- Check if a key is in the dictionary using `in`
- Ask for the number of keys with `len`
- Delete a mapping with `del`
- Iterate over keys with `for`

Show `https://docs.python.org/2.7/library/stdtypes.html#mapping-types-dict`.

Rewrite `frequency_counts(s)` to return a dictionary. Why this is better: directly look up by letter value. No need to use `ord` or alphabet string etc.

```
def frequency_counts(s):
    freqs = {}
    for c in s:
        if c not in freqs:
            freqs[c] = 0
        freqs[c] += 1

    normalize(freqs)
    return freqs

def normalize(d):
    total = 0
    for k in d:
        total += d[k]

    for k in d:
        d[k] /= float(total)
```

Go over idea for `proportional_choice` function on lab.

Zen Reading discussion (Friday, 30 Oct)

Setup

- HW: system analysis (see below)

Discussion

somewhat confusing? yes. you picked up in the middle of a journey the author is taking with his son and friends across the country on a motorcycle along the way he talks about the philosophy of quality, goodness, etc, this excerpt is groundwork for that discussion classical vs romantic understanding underlying form vs appearance negative views of each side (dull, uninteresting for classical, frivolous, erratic for romantic) masculine vs feminine association? Sexist? generous interpretations, because of culture? 70s mentality? in any case, this stereotype should not persist, obviously wrong women can be engineers, men can be artists and have feelings how is CSCI related to these understandings? Why did we read this? classical, definitely but elements of romantic too, creativity of solutions? choices in problem solving, uniqueness of approaches to projects, labs application of classical analysis to motorcycle talk through some of the details of the diagrams impossible to understand unless you already know how one works knife diving up the motorcycle intellectual scalpel no two motorcycle manufacturers divide same way romantic part of how this choice is made this skill of using the knife has been what we have been practicing all semester back to CSCI connection when we have talked about data types, (int, bool, string, list, dict) they have been ways we use the knife to separate information they are

all what we will now call OBJECTS we want to make our own new datatypes for many different situations two main pieces to make an object, components and functions, just like motorcycle above hierarchies component hierarchy has-a relationship eventually get to atomic elements (int, bool, string, list etc) function hierarchy is-a relationship motorcycle is a vehicle so is a bicycle, boat, truck, train, spaceship all related by function Your task as a team of 3 or four students apply this analysis to a system need more experience with using the knife in this way first, list off the components then think of some of their functions what do they do motorcycle throttle up, brake, park, inflate tires, etc how do the components communicate and pass information between themselves how do they make the system work not their existential purpose, not asking why right now, just how then group into two different hierarchies has-a is-a possible systems airport, hospital, us government, zoo, farm, soccer game see what they do discussion on Monday of their analysis which ones could we simulate with a computer, what choices and divisions and hierarchies make this easier?

System analysis HW

Split into groups of 3 or 4 (use binary cards), have each group choose a system:

- Hospital
- Soccer game
- College class
- US Government
- Zoo
- Airport

For Monday, analyze your system.

- Components?
- Functions of components?
- Information flow bewteen components?
- Is-a hierarchy
- Has-a hierarchy

Prepare a presentation, *use projector* (don't waste time drawing on board) and address all 5 of the above things.

System/hierarchy analysis presentations (Monday, 2 Nov)

Intro to classes & objects (Wednesday, 4 Nov)

Birthday cake problem:

```
# If n lit candles, blow out a random # between
#   1 and n
# How many blows does it take (on average)
#   before they are all out?
```

- Introduce idea of objects:

```
# We can package up data into "objects"
# Objects have functions, i.e. things they
#   can do. aka "Methods".

# Python "classes" allow us to design our
#   own new kinds of objects.
```

- Introduce idea of classes:

```
# class = template for objects
# an object is an "instance of" a class
# e.g. class = Car, object = my car
```

- Make empty `Cake` class.
- Show how we can make a `Cake` object.
- Show how we can set a variable, `num_candles`, inside it. Draw stack/memory diagram.
- Create an `__init__` method with a parameter. (Explain `self`; explain `__init__`.)
- Create `blowout` and `allout` methods.
- Fill out the rest of the simulation:

```
class Cake:

    # __init__ is a special method
    #   that gets called when a
    #   new object is created.
    #   i.e. when Cake() is called.
    def __init__(self, num_candles):
```

```
        # self is a special first parameter
        # that gets filled in by Python

        # self.candles is a variable in the
        #   object being created
        #   (candles = ... would make a local var)
        self.candles = num_candles

    # Methods = things that all Cakes can do

    def blowout(self):
        n = random.randint(1, self.candles)
        self.candles -= n

    def allout(self):
        return (self.candles == 0)


def happy_birthday(c):
    count = 0
    while not c.allout():
        c.blowout()
        count += 1

    return count

def average_blows(num_candles, trials):
    total = 0
    for i in range(trials):
        c = Cake(num_candles)
        total += happy_birthday(c)
    return total / float(trials)

def main():
    for num_candles in range(100):
        print num_candles, average_blows(num_candles, 10000)

main()
```

- Run it and paste output into Excel, do a regression to show that it is close to natural log.

More classes & objects (Friday, 6 Nov)

Traffic light example. Main point: encapsulation, abstraction.

```
# Traffic light
#
# How would you model it in Python?
# What functions should it have?
# How would it work?

# Possible functions:
#   - separate red, green, yellow light functions
#   - interval function
#   - sensor
#   - change to next color
#   - say what the current color is
#   - run
# Variables / state:
#   - current color
#   - timer (time left until next change?)


# Version 1:
#   - easy to write
#   - tedious (imagine cycling through 20 colors)
#   - difficult to change
class TrafficLight:

    # Variables:
    #   - current_color (string)

    # Create a new red traffic light
    def __init__(self):
        self.current_color = "RED"

    # Change to next color
    def change(self):
        if self.current_color == "RED":
            self.current_color = "GREEN"
        elif self.current_color == "GREEN":
            self.current_color = "YELLOW"
        elif self.current_color == "YELLOW":
            self.current_color = "ORANGE"
        elif self.current_color == "ORANGE":
            self.current_color = "RED"
        else:
            print "The sky is falling!!"  # This should never happen

    # Return the current color
    def color(self):
```

```
        return self.current_color

# Version 2:
# Use a dictionary of colors
class TrafficLight2:

    # Variables:
    #   - current_color
    #   - color_dict

    def __init__(self):
        self.current_color = "RED"
        self.color_dict = \
          { "RED" : "GREEN",
            "GREEN" : "YELLOW",
            "YELLOW" : "ORANGE",
            "ORANGE" : "RED" }

    def change(self):
        self.current_color = self.color_dict[self.current_color]

    def color(self):
        return self.current_color

# Version 3: list of colors
class TrafficLight3:

    # Variables:
    #   - color_list
    #   - current_index

    def __init__(self):
        self.color_list = ["RED", "GREEN", "YELLOW", "ORANGE"]
        self.current_index = 0

    def change(self):
        self.current_index += 1
        self.current_index %= len(self.color_list)

    def color(self):
        return self.color_list[self.current_index]

def main():
    t = TrafficLight3()
    print t.color()
    for i in range(10):
```

```
        t.change()
        print t.color()

main()
```

Yet More classes & objects (Monday, 9 Nov)

Mechanical pencil exercise. Have them work in teams to code a class representing a mechanical pencil. Paste code into Google doc, look through it together.

Yet Even More classes & objects (Wednesday, 11 Nov)

Card and Deck examples.

- Make `Card` class with `rank`, `suit` (strings), and `face_up` (boolean). Though can leave `face_up` until later. Just has getter methods. Write `__repr__`. (If usung `face_up`, `__repr__` should return `**` or something like that if face down.)
- Start making `Deck` class with list of cards. Make `__init__` (create 52-card deck), `shuffle` (use `random.shuffle`), `draw` (draw one card), `deal(n)` (draw n cards)

Still Yet Even More classes & objects (Friday, 13 Nov)

Finish Deck, make Hand, Player, HumanPlayer, ComputerPlayer.

Fill in stuff!

Queues

I usually start with a discussion of the queue data structure itself and why it would be useful. We might want to model lines of people in the supermarket or at a bank, and determine why each has a different way of organizing people (multiple lines with one checkout person each, vs one line with multiple tellers).

I then walk through the methods of a queue (`add, remove, size, is_empty`) and some important words (front, back) and write up the super class Queue with all the method bodies being pass.

Easiest way to implement a queue is to use a list, in terms of lines of code. List already do everything we need, and were just constraining the interactions to be with the front [0] and back [-1]. We code up the ListQueue class together.

Then we test the ListQueue with the tester function and the main function. As we put stress on the queue with larger numbers, we see its not a linear relationship, but quadratic. 10 ** 7 is a good number to stop, since it takes about 100 seconds to complete. I havent in the past, but I think I will make a log-log plot of the size vs the time.

So, maybe a list, because it is very flexible and can access any element, is too much for this Queue task, and we could do better with another way of implementing a Queue where we only really care about the front and back.

Then, I discuss the Node abstraction. Break up a list into individual pieces that fit together like Legos. Its a recursive data type, something that stores an element and another Node as components. This requires a discussion of the word None. Weve seen it as the return value of a void function. Python has a word for nothing, and we can use that word when we need a variable but it has no value.

Conveniently, we can calculate the size of a Node with a very simple recursive function.

How do we use this to make a Queue? Make the NodeQueue class, and have two components in init, the front and back, which start off as None.

Write size, very simple, and `is_empty`, base it on self.front being empty.

First, I try to complete add and remove the natural, but wrong, way. A node is then someone standing in line. When people stand in lines, they look at the person in front of them. So we start with the `is_empty` case, and set set.front = Node(e) and self.back = self.front. This is ok.

We then write the case when there are already people in line. We have a new element to add to the back, so we write

temp = Node(e) temp.next = self.back self.back = temp

to reset the back pointer. All looks good so far.

But we cant remove people from the queue easily. We can get the element we need from self.front, but who is the next self.front? self.front.next == None, and this is a problem. We could walk from the back to get the element right behind self.front, with complicated two-finger algorithm, and it takes a long time. Booo.

So instead, we reverse the orientation of everyone in line. You stand in line backwards and stare at the person behind you. They are your next. If you do this in real life, people will get a little uncomfortable. Just go with it, computer scientists are weird sometimes because its the right thing to do, like starting to count at 0

Then we rewrite add to match the code in the queuestuff.py file, and finally we can write a remove method, to set self.front = self.front.next. Both are now constant time.

How does this fare with our tests? For small numbers, a little more expensive. The overhead of making new Node objects all the time is seen.

But as it gets larger, the time is linear with the size. for 10 ** 6 and beyond, NodeQueue will beat ListQueue hands down. You can go back to plot them in excel. Ive included an image in the code directory of these two lines on the log-log plot.

I finish with discussing how this analysis and construction of data structures is at the core of the science part of computer science, and will be heavily discussed in CSCI 151 with more formal tools and for more complicated structures, within the context of Java.