```
TODO:
        - do an entry survey
                - current year (Fresh, Soph, Junior, Senior, Grad)
                - major (or planned major)
                - why taking the course?
                - likelihood of switching to a computing major?Ã\237
                - what else?

Week 1:

        start with firstday.pptx
        - overview of the course, computer science, programming

        first Python example:  magic 8 ball

        NOTE:  LAST YEAR I AVOIDED I/O COMPLETELY UNTIL GETTING TO LOOPS
        BUT PEOPLE RESPOND TO INTERACTIVITY!
        SO MAYBE WORTH DOING SOMETHING INTERACTIVE WITH I/O EARLY
        - NO LOOPS BUT CAN STILL HAVE A SEQUENCE OF Qs and As

        I WAS WORRIED THAT I/O BREAKS FUNCTIONAL PROGRAMMING
        BUT IT'S ORTHOGONAL TO RECURSION VS. LOOPS
        - AND IT PROVIDES A WAY TO GIVE THEM A USEFUL DEBUGGING METHODOLOGY WITH PRINT ST
ATEMENTS

        to work up to that, let's back up and talk about how to interact with Python
        - see doc on how to get python

        a common usage of Python is *interactive*
                - like a calculator
                - several ways to run Python
                - we'll use the IDLE *integrated development environment* that comes with
 Python
         - >>> waits for input *expression*; returns the *value*
         - example: 3*5 - 2
         - for historical reasons, these functions use *infix* notation
         - most functions use a standard math *prefix* notation f(x,y)
                - pow(3, 4)
         - of course, can have function calls within function calls
                - pow(pow(2,3), 4)

         - Python has other features that go beyond what calculators do, and allow for
                arbitrary computation
                - fun fact: every programming language is equally powerful
                        - can compute the same transformations
                - Python = C = C++ = Java = ...

         - we'll see a few of the key features today, which we'll need for our magic 8 ba
ll
           #1: give a name to a value
           x = 3*5
           - evaluates 3*5 to a value, just like it does when you run it directly
           - but instead of returning the value, it just gives the value the name x
           - a kind of *statement*:  performs some action and returns nothing
                - we'll see others as we go
           x
           x * 2
           y = x * x
           y
           - a simple form of *abstraction*
             - someone can use the value without having to know how it was computed
           pi = 3.14159
           radius = 50
           area = pi * radius * radius

           #2: define new functions
           - what if I want to compute the areas of many different circles?
           - of course, I can just do it
                rad2 = 8
```

```
                area2 = pi * rad2 * rad2

                in class there was confusion about the name area2
                        - should mention clearly that a name is some alphanumeric sequenc
e
                                - starting with alphabetic character
                                - no spaces allowed!

                ...
           - but what I want is to define the area computation once and reuse it

           def circleArea(r):

           let's unpack it:
                - "def" is a *keyword*
                - then a name for your function
                - then a comma-separated list of *formal parameter/argument* names
                - then a colon

                what follows is the *body* (definition) of the function
                - a list of statements
                - must be indented

                  pi = 3.14159
                  return pi * r * r

                "return" statement ends the function and returns the value of the given e
xpression

                show how to run it

           functions
           - a powerful kind of abstraction
                 - extend the vocabulary of the language
                 - treat complex computations as a single unit
           - critical for large-scale programming
           - analogy: our bodies
                - atoms --> molecules --> cellular structures --> cells --> organs --> sy
stems
                - each level of abstraction lets you treat a complex process as a unit
                        - understand its functionality as a whole, without needing to und
erstand the pieces

           general:

           "def" <name>(<parameter/argument names>):
                        statements
                        ** must be indented **

                now introduce files in IDLE
                  - want to be able to save functions
                        - so can use tomorrow
                  - annoying to type complex things into the interpreter directly
                        - want a text editor that makes it easy to edit

                  - show how to open a file, type into it, save it, run it

                ** ACTIVE **
                define a function that produces the average (arithmetic mean) of two numb
ers
                - average(2, 6) should return 4
                - ask for solutions
                KEEP IN MIND HERE AND LATER:
                        THE PERSON WHO RESPONDED KNOWS HOW TO DO IT
                        BUT THAT DOESN'T MEAN THAT OTHERS DO!
                        GO *CAREFULLY* OVER THE SOLUTION
                                - HOW IT WORKS LINE BY LINE!

                break?
```

```
let's do some logistics now:
        - syllabus
        - academic integrity

back to the 8 ball!
we've seen numbers as basic kinds of data
another basic kind of data is characters and strings (sequence of charact
ers):
"hello"
'hello'
'we are in CS97 right now'
- evaluate to themselves
- various operations -- ideas?

magic8ball function:
  - parameters?
  - what is the computation?

first stab:
  - just return a specific answer like "Definitely!"

how to improve?
  - need to choose among multiple possible answers
  - "if" statements let you do exactly that
        - only execute a set of statements if some condition is true
  - write the skeleton of it
        if (???):
                return "Definitely!"
    if (???):
                return "Not a chance.  :("
        if (???):
                return "Ask again tomorrow."
        if (???):
                return "I believe so."
  - explain the semantics
        - we'll see later that if is more general

  - so now we just have to figure out what *guards* to put in each *branc
h*
        - guard is an expression that evaluates to either True or False
                e.g. 4 > 3, 5 == 5, etc.
                - show these
                - operations on ints just like +
                        - but they return a *Boolean* value
    - start with a version that takes n and returns one of two values if
n is positive
        - so you can show the semantics of "if"
    - now, how to decide which branch to take?
    - choose randomly!
    - lucky for us, someone else has written a *library* (collection of f
unctions) for making random choices
        - abstraction at work!
        - in fact, there's a lot of fascinating theory and practice invol
ved in generating random numbers
          - turns out that getting a true source of randomness is difficu
lt
          - these are *pseudorandom* numbers
                - roughly, deterministic but would take a lot of work for
 someone to figure out the pattern
                        - "a lot of work" meaning would take computation
time longer than the life of the universe
                        - so random in practice
    - show how to import randint from random
        - run randint a few times at the command line
    - update code
        - give a name to the random int
        - test it on a few examples

    Week 1, 2nd lecture
```

```
            - piazza, academic integrity forms
                - review magic8Ball
                        - clarify semantics of if
                                - make clear: we have three different if statements
                                - if is a *compound* statement
                                        - consists of one or more other statements
                        - do a version with just one return at the end
                        - weird: we want the choices to be *exclusive*
                                - it's true, but not obvious
                                - and could make a mistake
                                - and this version unnecessarily tests n many times
                        - introduce elif and else
                                - allow for a bunch of exclusive options
                                - describe semantics
                                - make clear: we now have one big if statement
                        - show python tutor!
            ACTIVE:
                        function to take a number and return "pos" "neg" or "zero"

                picobot
                - why doing it?
                  - getting an intro to Python
                  - but the focus is on syntax and features
                  - algorithms are boring (so far)
                  - will get to more interesting algorithms
                  - for now, picobot lets us do that with minimal syntactic overhead
                - do picobot slides
                        - and do live on the fly
                NOTE: LAST YEAR PEOPLE OVERWHELMINGLY LIKED THAT WE DID THIS
                - SURVEYED IN CLASS IN WEEK 5

Week 2:

        homework strategies:
        - create a set of subtasks for each problem
                - solve and test one by one
                - example: just make it work when the player chooses 'rock'
                - example: draw a row, draw a row and then move to the next row
                - later today we'll see an easy way to define and run tests for Python

        - debugging
                - step through the code
                        - Picobot stepper, PythonTutor.com
                        - allows you to see exactly what's happening at any given point i
n execution


        - academic integrity
                - cannot work together with someone else
                        - even if you list their name on your assignment
                - can try other problems together
                - can discuss general concepts from the class
                - can get high-level ideas when you are stuck

                ***GIVE A SURVEY***

        languages are very simple!
        a few built-in **data types** (e.g., we've seen ints, strings, and booleans) and
operations on them

PEOPLE UNDERSTAND INTS BUT NOT BOOLEANS
NEED TO EXPLAIN
- only two values:  True, False
- only three operations:  and, or, not
        - explain each, starting with not
useful because tests of other data returns booleans

now explain exactly how an if statement works operationally;
before this it seems like magic to them
```

```
example:

def bothPos(x, y):
        return x > 0 and y > 0

ACTIVE
def exactlyOnePos(x,y)

        in fact, the language of the underlying machine is even simpler!
                - just numbers as data
                - a very primitive form of "if" statement
                - that's it

                - so ultimately all of the other stuff in Python gets translated to that
simple form
                  - the power of programming is the fact that we can define new *abstract
ions* and then
                        program in terms of them, even though the hardware does not under
stand them!
                        - languages have built in abstractions
                        - they also have mechanisms for programmers to define their own a
bstractions

                THIS STUFF ADDS EXTRA COGNITIVE OVERHEAD
                FORGET CHARACTERS
                INSTEAD DO MORE EXAMPLES WITH INTEGERS
                - example: characters do not exist!
                        ord('h') shows the encoding of a character
                        - various different encodings -- ASCII, Unicode, etc. -- by vario
us standards bodies
                                https://en.wikipedia.org/wiki/ASCII
                        - so in the machine 'h' is just represented as the number 104
                        - chr goes the other way
                        - so manipulating characters ultimately comes down to math operat
ions
                                - e.g. ord('H')
                                - so charToUpperCase involves subtracting 32 from an uppe
r-case character
                                - do it
                                - does weird things to other characters
                                        - how to update to just leave them unchanged?

                        ACTIVE: caesarCipher(c, n):
                                assume c is a lowercase character
                                - return the character n higher than it in the alphabet
                                - wrap around when reaching the end of the alphabet

        we've already seen how to define functions.
        - let's be a bit more precise on how function calls work
        simple one: double
                - define it
                - double(3)
                - can pass any *expression* as an argument
                        double(3+5)
                        abs(-3)
                        double(abs(-3))
                        - what happens?
                                - argument is first evaluated to a value
                                - that value is passed to double
                        double(double(3))
                - if mulitple args, they get evaluated from left to right
                - key point: a function is always passed *values*, never *expressions*
                pow(double(3), abs(double(-2)))
                - what's first call evaluated?
                - what's second?
                - what's third?

                same situation for operators:

                        5 * 7 * 3 is actually two nested invocations:  (5 * 7) * 3
```

```
                        - so it is evaluated as if it were a function call multiply(multi
ply(5, 7), 3)
                        - confusing because it allows you to leave off the parentheses
                        - and even more confusing because of the conventions of math
                        5 + 7 * 3
                        - so you have to understand these issues of *associativity* and *
precedence*
                                to know the behavior of an expressions
                                - explicitly use parens to clarify and prevent accidental
 errors!

        - the power of functions is that they can be easily *composed*
                - functions can call other functions, which call other functions, ...
                - e.g. sumOfDoubles calls double twice
                        - multiple parameters
                - what happens when we call sumOfDoubles?
                  - use PythonTutor to show it
                        - when executing a function call:
                        - pause the execution of the current function; ignore it complete
ly
                        - execute the function call and get result
                        - continue with the current function
                  - key principle: modularity/isolation
                        - function call should behave the same regardless of who called i
t
                                - same value gets same result
                        - critical for large-scale programming
                        - achieved by having each function have its own *scope* for varia
bles
                                - variables/args in sumOfDoubles not accessible by double
                                - and vice versa
                                - so the names can even be the same!

        - IMPORTANT: NEED MORE EXAMPLES OF FUNCTIONS CALLING OTHER FUNCTIONS
                - they need to get that concept well before moving on to
                  recursion

        - write documentation for our existing functions
                - docstring in triple double-quotes at the top of the function body:
                      description of the function for users; info (at least type info)
on args and result
                - help(double)
                - allows me to also bring up the notion of types
                        - we've seen several *types* of data:  ints, strings, booleans
                                - each with their own operations that make sense for them
                        - functions / variables naturally are intended to hold a single t
ype of data often
                        - e.g. sumOfDoubles('hello')
                                - explain where it fails
                        - Python is a *dynamic* language
                                - allows all syntactically correct programs to execute
                                - but they can fail in many ways during execution
                                        - if a built-in operation is called with argument
s of the wrong types
                        - annoying that there are all of these ways that things can crash
!
                        - many other languages have automated "typechecking" to ensure be
fore execution that
                                these kinds of errors can't occur
                                - only programs that pass this checking phase will be exe
cuted
                        - *extremely* important for large-scale programming
                - comments are for the implementation
                        - to clarify for implementers
                        # ...

        - testing
                - testing is critical!
                        - run your code on a bunch of examples to ensure it's doing what
you intend
```

```
                        - identify "corner cases" that test unusual behaviors
                - the doctest library makes it easy to write and run tests!
                  - write doctests for our existing stuff and run them
                        >>> sumOfDoubles(3,4)
                14
                >>> sumOfDoubles(0,5)
                10
                  - talk about the usage of test-driven development
                   - though it's a bit of magic, the easiest way to do things is to add
                        from doctest import testmod
                  to the top of the file and
                        if __name__ == "__main__":
                        testmod()
          to the bottom
          - now the tests are run automatically each type we Run Module!
            - silence = success
            - show failure

    - what makes a good function?
                - a well-defined task
                - reusable (and so somewhat general)
                - key principle: avoid code duplication!
                        - if you find yourself writing the same code more than once,
                                make a function for it
                        - so the code is written once and called multiple times
                                - easier to understand, maintain, update

        # By the way, what is the difference between print() and return?
        # THEY ARE COMPLETELY DIFFERENT!!
        # print(e) evaluates e to a value and prints that value
        #   - execution then continues on the next line of code
        # return e evaluates e to a value and halts execution of the current function
        #  - that value is returned to the caller of the function
        #  - nothing is printed anywhere

        # Your functions should almost never be printing anything.
        # Rather, they should be just like mathematical functions:
        #  - they take arguments, do some computation, and return a result
        # Such functions are easily *composable*
        #  - other functions can call them, get the result, and do more computation
                 with that result
        # The only printing happens at the top-level at the very end,
        # to print the final value of the whole program
        #  - done implicitly at the Python command prompt

        # Explicit printing should only be used when you need user interaction in the
        # middle of a computation.
        # - example: implementing a game of tic-tac-toe

        - review of the statements we've seen so far:
                - assignment
                        - x = e
                        - evaluates e to a value and assigns the variable x to have that
value
                        - e is never re-executed again
                        - a simple form of modularity:
                                - define a value once
                                - others can use it many times
                                        - no need to know how it was computed
                                - functions are a more sophisticated form of modularity
                                        - define a *computation* once and reuse it with d
ifferent arguments
                - return e
                        - evaluates e to a value
                        - halts the current function and returns that value as the result

                - if statement

        if expression:
                statements
```

```
        elif expression:
                statements
        ...
        else:
                statements

           - execute the first block of statements whose associated *guard* evaluates to
True
                - else implicitly has guard True - always suceeds
             - elif and else clauses are optional
           - confusing: Python allows guards that evaluate to things other than True/Fals
e
                - a bad decision; can slightly simplify some code, but is confusing and c
an
                        have surprising behavior
                - especially bad due to Python's dynamic nature
                        - can silently do the wrong thing
                        - e.g., x == 5 or 7
            - in our class the guards must *always* evaluate to booleans


    key operations on strings supported by Python:
                - length ('len')
                - indexing
                        - show in bounds and out of bounds accesses
                        - can also index negatively to grab back to front!
                                - -1 equivalent to len(s)-1
                - membership ('in')
                - slicing
                        - show full form
                        - show how can leave out first or second and get defaults
                        - note: returns a new string; the old one is unchanged
                - append ('+')
                        - again it's functional

        ACTIVE: makeExcitedSentence(word1, word2)
                        - makeExcitedSentence('hello', 'there') returns 'Hello there!'
                        - use our charToUpperCase from before

        PROBABLY A FEW MORE SIMPLE FUNCTIONS WITH STRINGS THAT DO SLICING ETC.

        Do an interlude on how to deal with syntax errors
                - see the discussion below that should have been done here instead

        - [recursion]
                DO toUpperCase INSTEAD
                EASIER TO UNDERSTAND
                THEN CAN DO cipher
                let's generalize caesarCipher
                def encrypt(s, n):
                        l = len(s)
                        if (l == 0):
                                return s
                        elif (l == 1):
                                return caesarCipher(s, n)
                        elif (l == 2):
                                return caesarCipher(s[0], n) + caesarCipher(s[1], n)
                        ...

                Problem: an infinite number of cases to write!

                Can break this infinite regress by recognizing that encrypting a string s
 can be decomposed into two pieces:
                        - convert the first character of the string
                                - can be done by calling caesarCipher
                        - convert the rest of the string
                                - can be done by having encrypt call itself!

                def encrypt(s, n):
                        if (len(s) == 0):
```

```
                        return s
                else:
                        encFirst = caesarCipher(s[0], n)
                        encRest = encrypt(s[1:], n)
                        return encFirst + encRest
```

use PythonTutor to show what happens
- with strings of size 0, 1, 2, 3, 5

a function that calls itself is *recursive* -- a powerful technique for building functions that must execute some functionality an unknown number of times

"recipe" for recursion:
- 1 or more *base cases*
  - handle the "smallest" arguments
  - implemented directly
  - why needed?
- 1 or more *recursive cases*
  - rely on the results of one or more recursive calls with "smaller" arguments
  - get to *assume* these recursive calls do the right thing!

ACTIVE: factorial(n) = n * (n-1) * ... * 1
- the number of ways that n things can be ordered

STRESS -- this is unintuitive
note: the multiplication in fact(5) happens from 1 up to 5
- opposite how you might expect
- but typical and natural when using recursion
- nothing interesting happens on the way down
  - just recursing until hitting the base case
  - all the work happens as we unwind the recursion
  - because we need the recursive result before we can do the multiplication with n

Lists:
- a built-in *data structure*
  SAY WHAT IT IS USEFUL FOR
  E.g., list of sales per day over the last month
  list of songs on a CD
  - a *container* for other data
  [1,2,3]
  [1, "hello", 3.14]
  [[1,2,3], [4,5,6]]

- convenient! same operations as strings
  indexing, slicing, membership, concatenation

  l[0] is the *head* of the list
  l[1:] is the *tail* of the list
  - note that it's always itself a list

- example: sum the elements of a list

  BE CLEAR ON ORDER OF EVALUATION!
  GIVE NAMES TO HEAD/TAIL/SUMREST
  AND THEN SHOW THAT THE VERSION BELOW IS
  EQUIVALENT!!

```
def sumList(l):
        if (l == []):
                return 0
        else:
                return l[0] + sumList(l[1:])
```

Last year I also posted a bunch of exercises. See recursion_exercises.txt. Also see Lana's partial solutions/hints in recursion_exercises_partial_solutions.py -- very useful (though the hint for the last one seems different from how I solved it).

RECAP:

```
                        You ARE DOING REAL PROGRAMMING
                        - we have learned the constructs that are involved in essentially
 all programming
                                - functions, if statements, basic kinds of data
                                - professional programmers use them continually
                                EVERY LANGUAGE HAS THESE OR ANALOGOUS,
                                BUT WITH DIFFERENT NAMES OF OPERATIONS.

                                - why are functions useful?
                                 - code reuse
                                        - write a computation once, use it many times
                                 - abstraction
                                        - build complex behaviors by composing simpler on
es
                                        - cells – organs – systems – bodies
                                - why are lists useful?
                                        - often need to collect up many pieces of data to
gether
                                        - one of the simplest, and most common collection
s is a
                                                sequence
                                        - lists are an implementation of a sequence
                                        - you'll see others in other languages, e.g. arra
ys
                                        - example: the menus in IDLE are implemented in a
 sequence
                                                - and each menu has a list of the command
s underneath it
                                        - example: your FB / Instagram feed
                                        - example: list of your courses, your grades, you
r music
                                - data and their operations
                                        - numbers:  +, *, /, ==, >, <, //, %
                                        - strings: concat, indexing, slicing, len, in
                                        - lists: same
                                        - booleans:
                                                - True and False
                                                - why needed?
                                                - the results of *tests* like ==, >,etc.
                                                - operations: not, and, or

                        Good programming is like painting
                                - iterative process
                                - write a bit of code
                                - run it
                                - change it
                                - run it
                                - add more
                                - fix errors at every step
                                - use the power of your machine!

                         Errors from IDLE:
                                - these are *syntax* errors
                                        - the text is not in the grammar of Python
                                        - like a grammatically incorrect English sentence
                                        - unfortunately computers are not nearly as forgi
ving as
                                                humans in these situations!
                                - IDLE is not very helpful!
                                  - just points out the location where it got stuck
                                - how to address?
                                        - don't panic
                                        - look for obvious, common issues:
                                                (show examples of each)
                                                - bad indentation
                                                        - to fix: move text to previous l
ine and press enter
                                                - missing colon
                                                        - at the end of def, if, elif, el
```

```
se lines
                                                - keyword typo (e.g., elf instead of elif
)
                                                        - colors are helpful
                                        - otherwise:
                                                - look at examples of the same kinds of s
tatements
                                                - search "syntax of if statements Python"
                                                - remove lines and re-add incrementally
                                        - this part will get much easier with more practi
ce!
                                                - getting past it allows you to focus on
the more
                                                        interesting stuff -- the actually
 algorithm

                        Other errors
                                - *type* errors
                                        - passing an argument of one type to an operation
 for
                                                another type
                                        - show some examples
                                        - says TypeError
                                                - read the scary red error message
                                                        - lots of useful info
                                                        - start at the bottom
                                - name errors
                                        - referring to a variable that has not been decla
red
                                        - again, read the error message
                                - other errors
                                        - somewhere a wrong value is being computed
                                - PythonTutor is your friend!
                                - can see all values of interest
                                  - if necessary, give names to subexpressions
                                        e.g., z = f(g(3)) becomes x = g(3) and z = f(x)
                                - simplify!
                                        - smaller inputs
                                        - remove parts of the code


                        turtle graphics
                        - show a triangle function

                        - BE VERY CLEAR: A PROCEDURAL WORLD NOW!!
                                - functions take an *external action*
                                        - write to a display
                                - they return nothing
                                - breaks our functional model

                        First do zigzag where you do a full zigzag in each recursive call
, so you end up drawing 2n segments.  Then get to this version, which needs to be done wi
th care so people understand that the thing being recursed on changes at each recursive c
all, but other args can also change at each recursive call.

                        def zigzag(n, angle):
                            if n == 0:
                                return
                            else:
                                forward(100)
                                right(angle)
                                zigzag(n-1, -angle)

                        ACTIVE:

                                positives, inRange(low, high, l) -- keep only elements in
 range,

                                some cases with interesting base cases or recursion:
                                everyOther, pairify, convert an int to a list of digits,
```

zip

So far when you've been asked to write a function F that performs a certa in computation, that only required you to define a single function named F. In general, however, getting F to do the right thing may require that you also define functions G and H that do part of the work and that are called by F. This may be done just to make the code easier to read, or it may actually be necessary for various reasons (we'll see one b elow). In turn you may even define other functions I and J that are called by G and H, a nd so on.

Functions like G, H, I, and J are called helper functions; they are not i nteresting on their own but rather are only useful because they help you to implement F.

In part 1 of HW2, you are not allowed to use any helper functions.

In the first problem of part 2, on the other hand, the hint tells you tha t a helper function will be necessary. Here's an example where a helper function is nece ssary -- the situation on the first problem of part 2 is similar. Suppose you want to im plement a function find(x,l) that returns the smallest index i such that l[i] == x, or -1 if no such index exists. For example, find(1, [0,20,1,4]) returns 2.

Writing find recursively you will get stuck quickly! The problem is that as you recurse down the list you also need to keep track of what index you are at in the original list, so that the right index can be returned as the answer. The easiest way t o solve that problem is to write a helper function that takes that index as an extra argu ment and updates it on each recursive call. Now the helper function can be written recur sively, and the original function simply calls the helper function.

```
def find(x, myList):
  return findHelper(x, myList, 0)

def findHelper(x, myList, i):
  # i is the index of the head of myList in terms of the *original* list
  if myList == []:
    return -1
  elif x == myList[0]:
    # found a matching element at the head so return its index
    return i
  else:
    # increment the index and search the rest of the list
    return findHelper(x, myList[1:], i+1)
```

Q: how to modify to get the last index instead of the first?

more examples that need helper functions:

```
def toDigit(l):
    return toDigitHelper(l, len(l) - 1)

def toDigitHelper(l, exp):
    if l == []:
        return 0
    else:
        return (l[0] * (10**exp)) + toDigitHelper(l[1:], exp-1)

def toDigit2(l):
    if l == []:
        return 0
    else:
        lastPos = len(l) - 1
        return l[lastPos] + 10 * toDigit2(l[:lastPos])
```

final example (from book): edit distance
- goal: define a function to compute the 'edit distance' between two strings
- formalizes our goal from the first lecture to compare two gene sequence s

```
                - also useful for many other things, e.g. suggesting spelling error fixes
                - how many insert/delete/substitutions do we need to go from one word to
the other?
                        - want the minimal number
                - e.g., editDistance('python', 'pithy') = 3
                        - note: symmetric

                - first version: let's assume the strings have the same length
                        let's also assume we can only do substitutions
                        - so the edit distance is simply the number of positions where th
e strings have a different character
                        - note: not the same as edit distance:
                                - simpleDistance('hello', 'elloh') = 5
                                - editDistance('hello', 'elloh') = 2

                ACTIVE
                def simpleDistance(s1, s2):
                            if len(s1) == 0:     # len(s2) is also 0 since strings
                                                 # have the same length
                                return 0         # base case
                            elif s1[0] == s2[0]: # recursive step, case 1
                                return simpleDistance(s1[1:], s2[1:])
                            else:                # recursive step, case 2:
                                                 #   s1[0] != s2[0]
                                return 1 + simpleDistance(s1[1:], s2[1:])

                Finally, we need to generalize this to the case when the lengths are diff
erent.
                - base cases:
                        - since the lengths are not the same, len(s1) == 0 does not imply
 that s2 is
                                - if not, then what is the edit distance?
                                - e.g., editDistance("", "todd")
                        - need a second base case
                                - what?
                                - symmetric case

                - recursive cases:
                        - if first characters match, nothing to do but move on
                                - not completely obvious but true after a bit of thought
                        - what if they don't match?
                                need to try all possibilities!
                                        - insert, delete, substitute
                                        - then take their minimum
                                                - built-in function

                        def distance(first, second):
                                    if first == '':
                                        return len(second)
                                    elif second == '':
                                        return len(first)
                                    elif first[0] == second[0]:
                                        return distance(first[1:], second[1:])
                                    else:
                                        substitution = 1 + distance(first[1:], second[1:]
)
                                        deletion = 1 + distance(first[1:], second)
                                        insertion = 1 + distance(first, second[1:])
                                        return min(substitution, deletion, insertion)

                        another example: longest common subsequence or subset sum
                                - LCS worked well
                                        - motivated by auto-correct suggestions when text
ing,
                                                track changes in Word, diff

Week 3 (well actually it will probably be later than that):

MIDTERM DEBRIEF
```

good programming is about building reusable code
- allows you and others to easily build on it

functions are the key to reuse, as we've seen
- write a computation once
- reuse it in many different way (based on arguments)

but functions as we've seen so far still have their limits. if you write a lot of func
tions, you'll start to notice code that you are writing over and over again.

consider:

- double each element of a list (from hw2)

```
def doubleAll(l):
    if l == []:
            return []
    else:
            return [l[0] * 2] + incList(l[1:])
```

- make each string more excited

```
def exclaimAll(l):
    if l == []:
            return []
    else:
            return [l[0] + '!'] + exclaimList(l[1:])
```

Q: what do these two functions have in common?
- both take a list and transform each element in some way

define a function called double and rewrite doubleAll
define a function called exclaim and rewrite exclaimAll

so the only difference is what function they call on each element

a *very* common pattern
- how can we define the pattern once and reuse it for
each of the above, and more?

simple, powerful idea: functions can take other functions
as arguments!

```
def transformAll(f, l):
            if l == []:
                    return []
            else:
                    return [f(l[0])] + transformAll(f, l[1:])
```

- then show how to call transformAll to do doubleAll and exclaimAll

transformAll is traditionally called *map* -- you map a function over a list.
- show in Python
  - unfortunately map doesn't return a list but rather a "map object", which is kind o
f a *lazy list* -- avoids actually computing the elements until they are needed
- use the list() function to convert the result to a list

now can define doubleAll in terms of map

What's cool about this?
  - write a common pattern once; reuse it for various purposes
  - can program at a higher level of abstraction
          - working over a list at once, rather than element by element

  - can think at a higher level of abstraction
    "map the double function over this list"

  - users only have to say what's special about their traversal
    - the boilerplate code for traversing a list is done *once*

This idea has been around since the 1950s in "functional programming languages". It

took a while but all other languages have caught up.
- can now pass functions to other functions in Python, JS, Java, C++, etc.
- they all have libraries for doing maps and related traversals on data structures

Good style to use map where possible, instead of implementing the recursion manually.

THIS CONFUSED PEOPLE -- THEY DON'T KNOW WHAT PARALLELISM IS
SKIP THIS!!!!
What really pushed languages to adopt this technology?
- surprisingly, the need for "big data" analysis
- need to process huge amounts of data
- the only way to do that is to divide the data into pieces and separately process them on multiple machines at once
  - *parallelization*
- observation: map is trivially parallelizable
  - can break the list into pieces, map each piece, then recombine
  - other common traversals have the same property
- so if you can express your analysis as a map (and some other kinds of traversals that we'll see later), then it can be automatically parallelized on many machines
- Google FB, Twitter, etc. all use parallel implementations of map and other traversals on a daily basis
  - see Google MapReduce, Hadoop, Apache Spark, etc.
  - users just provide the function on an individual element
  - the system provides an efficient parallel map
  - e.g., what's the average number of friends per person on Facebook?
  - can map the "countFriends" function across each person
  - then sum
    - turns out this is also highly parallelizable
  - finally divide by total number

ACTIVE:
convert a list of strings to a list of their lengths
convert a list of ints to a list of their absolute values

The cool thing is that map is not some special operator
- it comes "for free" from the ability to pass functions to other functions

This ability allows for lots of useful generalizations of common functions.

* Example:
- last class we saw the min function; let's define it

```
def min(l):
    if len(l) == 1:
        return l[0]
    else:
        minRest = min(l[1:])
        if min < l[0]:
            return min
        else:
            return l[0]
```

This works but it hard-codes a particular way of comparing elements.
- for strings this is alphabetical order
- what if I want another order, e.g. by string length
Show how to generalize min to take a comparison function

Useful because now I can use min for my own kinds of data.
E.g., student midterm grades
[['john', 'doe', 95], ['jane', 'smith', 100], ['todd', 'millstein', 80]]
find the person with the minimum score
Similar example: your music library
- find the most recently played, find the first song/album/artist alphabetically, etc.
Another example: representing fractions as pairs of integers
[[1,4], [2,9], [3,5]]
how to do < for fractions?
f1[0] * f2[1] < f2[0] * f1[1]

```
Map is one common pattern for traversing a list
- easy to define more common patterns for traversing lists
- and can do for other kinds of data too

Review of map:
  incAll

REVIEW OF THE BOOLEAN DATA TYPE AND OPERATIONS
- integer values:  0, 1, 2, 3....
- integer operations:  +, -, etc.
booleans are much simpler!
- only two values:  True, False
- only three operations:  and, or, not
        - explain each, starting with not
useful because tests of other data returns booleans

def bothPos(x, y):
        return x > 0 and y > 0

ACTIVE
def exactlyOnePos(x,y)

* Example:

  positives([1,0,3,-3,5]) = [1,3,5]

  - define it

  largeStrings(["hello", "there", "how", "are", "you", "today"])
   = ["hello", "there", "today"]

   - define it

  define filter

  MAKE CLEAR THAT THE ARG FUNCTION RETURNS A BOOLEAN!
  show two special cases:
  - always True
  - always False

  ACTIVE:
        def onlyEvens
        retain only even elements from a list

        more examples if useful:
                - remove all pairs whose second element is less than the first element

Nested and anonymous functions:
   can define double local to doubleAll
   - avoids *polluting* the top-level namespace with double
   - show it

   can go even further and define the double function without giving it a name
   - just as you can create and use numbers, lists, etc. without naming them
   - show it
        - weird syntax
                - lambda
                - body is just an expression
                        - so limited in what you can do
                        - implicitly returned

   Good style to use anonymous functions if they are just helpers for the main computatio
n that are used once.

  ACTIVE:
        rewrite onlyEvens to use a nested, anonymous function

        remove all negatives and double all positives

Show the "e if e else e" syntax, which is useful for lambdas
```

```
        example: absAll (absolute value)

   Easy to think of other traversals that would be useful
        ACTIVE
        write a function forall(p, l) that checks if every element of l satisfies p

        show an example usage, e.g. all positives

   One powerful feature of nested functions is that they can refer to any variables in th
e surrounding scope.

    Simple example:  generalize doubleAll to multBy
    - define multBy
    - show why you can't pass in multBy directly to map
    - can declare a nested function instead


    Another example:  simple primality testing using forall
    - idea: n is prime if nothing below it divides it
    - introduce the range function
    ACTIVE
    def isPrime(n):
        return forall(lambda x: n % x != 0, range(2,n))

    Now simple primality testing for a list of numbers

    def primes(l):
        ACTIVE

    Can do prime sieves method now too (see Chap 3)
      - describe the overall approach

    def sieve(l):
        if l == []:
                return l
        else:
                p = l[0]
                noPMultiples = ACTIVE
                return [p] + sieve(noPMultiples)


* A final common pattern.

Consider sumList (define it)
        - assume the list is non-empty
Consider prodList (define it)

Let's generalize it:

def combineInts(f,l):
        if len(l) == 1:
                return l[0]
        else:
                return f(l[0], combineInts(f, l[1:]))

combineInts f [x1,...,xn] is f(x1, f(x2, ... f(xn-1,xn) ... ))

show how to do sumList and prodList

called reduce typically

not by default in Python 3 (was in Python 2) but in the functools library
but it actually associates things the other way

reduce f [x1,...,xn] is f(f(...f(x1, x2), ... ), xn)

can define it by walking through the list backward

DONT SHOW THIS!  TOO CONFUSING
def reduce(f,l):
```

```
        if len(l) == 1:
                return l[0]
        else:
                last = len(l) - 1
                return f(reduce(f, l[:last]), l[last])
```

can see the difference if f is not associative
        e.g., do subtraction

ACTIVE:
        concat all strings
        multiply a list of fractions (represented as pairs of numbers)
        - produce a new fraction
        get the minimum value in a list of numbers

First-class functions are widely used today.
- big data analysis
- another major use: web programming

THIS NEEDS MORE THOUGHT
NEED TO MOTIVATE CLEARLY WHY FIRST CLASS FUNCTIONS
ARE NEEDED.
ALSO THE SYNTAX IS CONFUSING
 - OO dispatch
 - JQuery $(...) syntax
How do you program a web application?

Show button.html

Uses HTML to create a button.

But the button doesn't do anything!

How to make it do something?
Problem: different buttons need to do different things
How does the graphics library know what to do when the button is pressed?
Solution: let the programmer provide a function that gets called whenever
the button is pressed!

Uncomment the code, explain, and show it running.
Then add a second function.

GIVE OUT THE SURVEY!!!

SKIP - NOT WELL MOTIVATED
Functions can also return other functions.
Why would you want to do this??
Allows you to programmatically create new functions.

Example:

Python includes pow function.
How to create the square function?

```
def square(x):
        return pow(x, 2)
```

How to create the cube function?
        similar

Why useful?
        - give the exponent once
        - get back a dedicated function that does that

With functions returning functions, we can make a function to *generate* square, cube, et
c.

```
def makeNthPower(n):
```

```
        return lambda x: pow(x, n)
```

- Note: relies critically on nested functions

```
square = makeNthPower(2)
cube = makeNthPower(3)
```

Another example: function composition

```
def compose(f, g):
        return lambda x: f(g(x))
```

Similar example as in book:

```
        def caesarCipher(n):
                <return encrypt/decrypt functions>

        if you want to actually do this, start here:

                def rotate(c, n):
                        a = ord('a')
                        return chr((ord(c) - a + n) % 26 + a)
```

MIDTERM 2 REVIEW

Recap why recursion goes "backwards"
- only because can't invoke a function or operator
  until all of its arguments have been evaluated
- use sumList as an example
  - write as a one-line else and also by giving
    names to the pieces

Should be able to do/understand how they work for the problems on Homework 3
- partition without helper functions
- the functions with multiple recursive calls

Can you program them now on paper?

Recap that the function given to filter must return True or False; nothing else.
show two special case examples:
- lambda x: True
- lambda x: False

Take questions.

Have people do the practice problems.
Also practice writing your own functions that take functions:
map2, forall, map, filter, reduce

Imperative programming:

Several main styles of programming. The style of programming we've been doing so far is called *functional programming*. No clear definition, but some hallmarks of the functional style:

1. Functions are first class. So can define things like map, filter, etc. that take functions as arguments.

2. Nothing changes! The value of a variable is never re-assigned / updated. Instead, we just create new things from old things, but the old things stay the same.

E.g., show doubleAll
- give names to head, tail, recursiveResult
- they never change
- each recursive call has its own variables
- x + y creates a *new* list

This simplifies your understanding of, debugging of code. Don't need to worry about some

thing changing after it's been defined.

This also implies a very important property: functions are strongly *isolated* from one another. In particular, each functions are *pure* -- just takes some inputs and produce a result, like a mathematical function. Functions don't depend on anything other than their arguments, and they don't have any visible effect other than providing a result.

Sounds obvious, but in fact that property is easy to violate in most languages, as we'll see later.

Example:
- l2 = doubleAll(l)
- what is l2?
- what is l?
- same for using map instead

Why is purity so important? Consider what the alternative means -- calling a function has some effect that *changes* the way that other code behaves. How? By updating some existing variable's value that other code uses.

So now it matters what *order* functions are called in, how many times each is called, what state they each update, etc. Sounds crazy, but most languages make it very easy to do that, as we'll see. Purity implies that functions are *isolated* from one another -- they only interact through arguments and results.

As you want to write larger and larger programs, the ability to isolate parts from one another becomes *hugely* important. Easier to understand, easier to modify/extend, less chance of unintended consequences, both accidental and malicious.

Back to our discussion of functional language hallmarks...

These days all languages have #1 and good programmers rely on them heavily (Javascript, Python, Java, C#, C++, etc.).

Streams in Java:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

C++ Standard library:
http://en.cppreference.com/w/cpp/algorithm
- e.g., for_each is like map; also has reduce

Some languages, known as functional languages, enforce or encourage #2, by making it the default way of doing things: Lisp, OCaml, Haskell, Facebook Reason (based on OCaml).

But most don't. Including Python! By default, it's instead common and easy to update the values of variables. Can simply re-assign a variable's value. These languages are called *imperative* languages.

CRITICAL:
Show a simple update example (no functions) in the interpreter.
ALSO UPDATE IT IN TERMS OF ITSELF, e.g., x = x+1
THE OLD VALUE IS GONE AFTER THE UPDATE!

But function purity is still important! In addition to the reasons I said above, it's required for using libraries that support certain common activities: large-scale data analysis (map/reduce), some web programming frameworks (Facebook React).

As we'll see it's still possible to ensure functions are pure. But it's also easy to violate that. Critical for you to understand when you're violating it, and to have a good reason for it. This requires understanding more about what the computer is actually doing to represent your variables and data.

Given that variables can be re-assigned, *iteration* is more natural and more common in most languages than recursion. Iteration means executing the same code multiple times. Done through constructs called *loops*.

Most languages, including Python, have two kinds of loops: for loops and while loops. For loops are also called *bounded* loops -- they execute for a fixed number of times.

NEED TO CAREFULLY INTRODUCE PRINT IF HAVEN'T ALREADY.
PEOPLE DON'T UNDERSTAND THE DIFFERENCE WITH RETURN.

```
for x in [1,2,3]:
        print('hello!')
```

the variable named x is being declared there
- can choose any name

can also use x:

```
for x in [1,2,3]:
        print(x)
```

can also use list-like things:
- do the above with a string
- do the above with a range object

Example: sumList

Show the recursive version of sumList.

Now show the version with for loop:

```
def sumList(l):
        count = 0               # an *accumulator* for the result
        for x in l:
                count = count + x
        return count
```

Walk through carefully how it works on [1,2,3], showing count's value being updated at ea
ch step.
        - rhs is evaluated on the old state, then updates count

So the additions are forward through the list. Opposite what recursion naturally does.

Despite these differences, from the outside you can't tell the difference -- they both in
particular are *pure*. Show how calls to each don't change anything, like the list.
- key point: the only updates are to local variables
        - count
        - not visible after the call ends
- we'll see other kinds of updates later

How would real programmers implement sumList?
1st choice: If it exists in a library, just call it.
        - Python in fact has a built-in sum function
        - be lazy when possible!
                - why? that code is already well tested, efficient
                - your code stays simpler to read and understand
2nd choice: If can build it with map, filter, reduce, etc. do it.
        - Again those functions are well tested and efficient
        - Again your code stays simpler to read and understand
3rd choice: Write your own loop.

ACTIVE: factorial, contains

So far our loops have just accumulated numbers or booleans. Can also build up lists.

Example (again, you'd really use map!):

```
def doubleAll(l):
        result = []
        for x in l:
                result = result + [2*x]
        return result
```

Explain what it's doing.

ACTIVE: positives, reverse

At some point introduce the += notation!

GOT TO HERE ON MONDAY

Another example: max
- first do it normally
- now what if we want to also return the index of the max?
  - add another variable that gets updated!
  - no need to just have one updating variable
  - show it

- alternative: loop directly on the indexes
  - show it

GO SLOWLY HERE - VERY DIFFERENT THAN BEFORE
- with recursion you only access the head and tail
  - list becomes "smaller" with each recursive call
- with iteration you simply walk over the list,
  using an index to keep track of where you are
- this is the standard way to do things in most
  languages!

ACTIVE:
find, indexValuePairs

How to manipulate nested lists? With nested loops!

Example: sum a list of lists

ACTIVE: doubleAllInner
max of a list of lists;
then, can you also record the index of the
max item's inner list and element number?

In all of these examples, we know how many times the list needs to execute: once per elem
ent of the list (either one we're given or one we create).

Sometimes you can't know in advance how many times a loop should run.

Example: computing the gcd with Euclid's algorithm

gcd(m, n) = gcd(n, m mod n) [assume m >= n; otherwise reverse]

so iteratively update m and n according to this formula
until n is 0; then m is the gcd of the original numbers

example:
gcd(20, 12)

20 12
12 8
8 4
4 0

20 13
13 7
7 6
6 1
1 0

20 15
15 5
5 0

```
def gcd(m, n):
        while(n != 0):
                oldM = m  # note: need the temporary var!!!
                m = n
                n = oldM % n
```

```
        return m


Note: not obvious that this loop terminates!
        - while loops may run forever
        - need to convince yourself that this will not happen

while can do anything that for can do
e.g., let's rewrite sumList
        - do it

But you have to manually modify n, while the for loop
        does that implicitly
Better to use the for loop when you can.

ACTIVE:
                THIS ONE IS TRICKY TO DO WELL WITH A WHILE LOOP
                FIND A BETTER EXAMPLE
                findOutOfOrder(l)
                        - find the first item in the list that is less
                                than the previous element
                        - can't use a for loop -- why not?

                THIS IS A GREAT EXAMPLE
                collatz(n)
                        - produce the sequence of numbers it takes
                                to get to 1
                        - if n is even, halve it, else 3n+1
                        - e.g., 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

def collatz(n):
        result = [n]
        while(n != 1):
                if n % 2 == 0:
                        n = n // 2
                else:
                        n = 3 * n + 1
                result += [n]
        return result

RECAP:

"for" loop lets you iterate over the elements of a list one at a time
- a bounded loop -- executes a fixed number of times

"while" loop lets you iteratively execute some statements while a particular condition (a
 Boolean-valued expression) is True
- an unbounded loop -- executes an unknown number of times

use a "for" loop when possible, because it's simpler and more reliable
- when you know how many times to iterate

examples: intersection(l1, l2) - a list containing the elements
                  that l1 and l2 have in common
                  commonPrefix(s1, s2) - a string of the maximal prefix that s1 and s2 ha
ve in common
                  commonPrefix('heck', 'hello') = 'he'

def intersection(l1, l2):
        result = []
        for x in l1:
                if x in l2:
                        result += [x]
        return result

def commonPrefix(s1, s2):
        result = ""
        i = 0
        while(i < len(s1) and i < len(s2) and s1[i] == s2[i]):
                result += s1[i]
```

```
                i += 1
        return result
```

One situation when you often don't know how many times you need to iterate, and hence nee
d a while loop, is when you are interacting with a user.

In our programs so far, the only *input* from the user is provided at the very beginning:
- the arguments to the function being called
And the only *output* to the user is provided at the very end:
- after that function completes, its return value is printed
- note: that function could call many other functions
(and itself recursively)
- so functions don't inherently print anything
- rather, the Python interpreter prints the final result

But sometimes a program needs to take input / provide output *during* the program's execu
tion.
- called an interactive application
- example: a game
- player needs to make moves (input)
- program needs to show the results of moves (output)

We've seen how to do output in the middle of a program
- the print() function
- print(e) evaluates e to a value, prints it, moves on
- totally different than return!
Need a way to input
- Python has a simple way to do that
- input(msg) function
- prints msg (a string); returns a string (user input)
- show an example at the command line

Example: guessing game
- I choose a number between 1 and 100
- you guess and I say higher/lower until you get it

```
def guessingGame():
    secret = randint(1, 100)

    sGuess = input("Guess a number between 1 and 100: ")
    guess = int(sGuess)

    while(guess != secret):
        if guess > secret:
            guess = int(input("Lower! "))
        else:
            guess = int(input("Higher! "))

    print("You got it!")
```

ACTIVE: extend this to ask the user if he/she wants to play again, and if so, restart the
game
- could do it recursively
- could use nested loops

References and Mutation:

We've seen that imperative programming requires *mutation* -- updating the values of exis
ting variables.

Why?
- that's how a loop accumulates the results of each iteration
- if a loop body did not update any existing variables, then
it would have no effect
- and if it's a while loop it would run forever

The kind of mutation we've seen so far is to *update the value of a local variable*
- because it's a local variable, it goes away after the function call ends
- therefore, *callers of the function never see, and cannot be affected by, these updates

*
- they just see the final result

But it also possible to actually mutate components of the data that is stored in a variable, for some kinds of data!

```
l = [1,2,3]
l[1]
l[1] = 44
l
```

There is a *huge* difference between this kind of update and the updates we've been doing.

```
l = [1,2,3]
l = [1, 44, 3]
```

In the above case, the re-assignment to l ignores the old list and creates a new one to store in l.

In the earlier case, the assignment to l[1] updates the list [1,2,3] *in place* - it updates the existing data rather than creating new data.

Example: doubleAll

```
def doubleAll(l):
        result = []
        for x in l:
                result += [2*x]
        return result

myL = [1,2,3]
doubleAll(myL)
myL
```

- the variable result exists only during the call to doubleAll
  - it is created when first mentioned
  - it is implicitly removed after the function returns
- the function is *pure* -- its only effect is to return a value

Here's a different implementation of doubleAll:

```
def doubleAll2(l):
        result = l
        for i in range(0, len(result)):
                result[i] = result[i] * 2
        return result

myL = [1,2,3]
doubleAll2(myL)
myL
```

Walk through carefully what each iteration above does.

So calling doubleAll2 returns the same result as doubleAll, but it also changes the value of the original list!
- the original value is gone and cannot be recovered!
Further, this may have had the effect of changing the values of other variables in the program that originally had the list [1,2,3]!

When doing imperative programming, you have to *carefully* understand how visible your updates are -- in general you want to isolate your updates to as small a region of code as possible, or else it is very easy to cause errors.

To understand what's going on, you need to know a bit about how data is stored when a program executes.

The computer has access to some *memory* while it executes.
- Apple -> About this mac
        16 GB of RAM

```
        16 billion bytes
        - a character fits if 1 byte
        - an integer fits in 4 bytes
- basically just a big list
- Python *allocates* memory from this list to store data as the program runs

>>> x = 42
# Python allocates a chunk of memory for 42
# Python allocates a chunk of memory for x [if first assignment]
# x holds the *address* of 42
#   - called a *reference* to 42

draw a picture

can actually find out what that address is

>>> id(x)

key point #1: variables *always* hold references to data; never the data itself

>>> x = 43
**Important to say clearly**
Assignment ignores whatever was there before, doesn't touch it.
Instead, just stores a reference to a different piece of data.

Key issue is what happens when a variable is "copied":

>>> y = x

draw a picture

we say that x and y are *aliases* -- they are references to the same data

>>> id(y)

key point #2:  "copying" a variable simply copies the reference, not the data

Same goes for parameter passing / result returning.

def inc(z):
   return z + 1

inc(x)

go carefully through what happens

Weird!  Why?  Doesn't make much sense for integers, but consider lists:

>>> l1 = list(range(2000))

FIRST: HOW ARE LISTS REPRESENTED IN MEMORY?

>>> l2 = l1

Lists can get big!  Don't want to pay the cost of copying the entire list every time we a
ssign it to a new variable, pass it as a parameter to a function, return it as the result
 of a function.

But here's the issue:  unlike integers, lists are *mutable* -- we can change their compon
ents.

Show this carefully:

x = 42
id(x)

can't change 42, can only reassign x

x = 43
```

creates a new memory location (the old one is unchanged)

```
id(x)
```

But lists are mutable:

```
l = [1,2,3]
id(l)

l[2] = 55
```

ASSIGNMENT HERE IS THE SAME AS WHAT HAPPENS FOR VARS
IGNORE THE CONTENTS; STORE A DIFFERENT REFERENCE

```
l
id(l)
```

Mutation combined with aliasing means that functions can have many effects on the program that are *implicit* -- happening through assignments within the function rather than thr ough an explicit return value.

```
l2 = l
l[0] = 33
l
l2
```

This is a key reason why functional programming is simpler -- if there is no mutation, th en the programmer never has to worry about whether something is a copy of the data or an alias to it.
- behavior can't depend on that
- just an issue of efficiency

What if you want to copy the list, to avoid aliasing?
- have to do it explicitly

```
l3 = l[:]
l3
id(l3)
```

Let's go back to doubleAll2 and understand what's going on. Go through it carefully.

another example:

```
def incPair(p):
        first = p[0]
        first = first + 1
        p[1] = p[1] + 1
        return [first, p[1]]

myPair = [10, 20]
newPair = incPair(p)
```

ACTIVE:
- what is newPair after the call?
- what is myPair after the call?

go through it carefully

you have to be very aware when you call a function what its effects are!
- example: x += v is not a shorthand for x = x + v !!

```
        l1 = [1,2,3]
        l2 = l1
        l1 = l1 + [4]
```

updates to variables can't affect other variables

```
        l2
```

but:

```
        l1 = [1,2,3]
        l2 = l1
        l1 += [4]
        l2
        ACK!!
```

  and as a function writer you have to be very aware of the effects your function has, an
d why
  - writes to local variables are always invisible to callers
  - writes to the *contents* of local variables affect all
        aliases and so should be done with care

  all languages do the key points above...
  - stress them again
  - gives you a simple, uniform way to understand how all data is treated
        - some might optimize things under the covers,
                but you can always understand code this way

  except C/C++
  - lets you choose whether to copy value or reference
  - you explicitly create references to data when desired
        - called *pointers*
  - more explicit control over memory
        - but also more confusing / error-prone

LOOPS REDUX

iteration with loops encourages a different way of thinking than recursion!

example: summing a list
        - recursion:  split the list into head and tail
                recursively sum the tail and add it to head

        - iteration:  simply enumerate the elements of the list
                one by one, keeping a running sum as you go
          - for loops are the means of doing this enumeration

simplest kind of loop in Python:

```
for x in l:
        <statements>
```

enumerates each element of the list l one by one

example: greaterThan10
        - could use filter, but we'll do it with a for loop

problem:  it doesn't tell you what index each element is at
        - sometimes you need that information
        - e.g., greaterThan10 but now we want the indices, not the values

        first approach:  add a variable to keep track of the index
        - do it

        so the for loop is "automatically" giving us the next element of the list each ti
me, and we are "manually" keeping track of the index

        idea:  have the for loop iterate over the indexes instead of the values
        - how?  with range
        - show what list(range(0, l)) is for some list l
        - ok, but then how to get access to the values?
                - just index into the list l

        show this version

        in fact, the kind of for loop you'll learn next quarter is most commonly used in
exactly this way
        - iterate over indexes, not over elements

```
        ACTIVE:  incByIndex(l):
                        increment each element of the list l by its index
                        incByIndex([3,5,7,9]) = [3, 6, 9, 12]


        finally, a while loop is used when you can't create a list to iterate over
                - or can't do it easily enough

        e.g., compute the highest power of 2 lower than n

                powerOf2Below(50) = 32

                how to solve?
                - idea:  start at 1, keep multiplying by 2
                        for how long?
                        - hard to know in advance
                        1 2 4 8 16 32 64 --> 32

                # assume n > 1
                def powerOf2Below(n):
                        result = 1
                        while (2 * result < n):
                                result = 2 * result
                        return result


WHY IMPERATIVE PROGRAMMING?
- nothing wrong with loops that only update local variables
        - I use them all the time
        - from the outside can't tell the difference vs. functional
        - and often more efficient
        - in fact, map/filter/reduce actually implemented with loops
                - ACTIVE: implement map with a loop
- the key difference is what happens when you update the *data* itself
        - called updating "in place"
        l = [1,2,3]
        l = [1, 44, 3]
        vs.
        l = [1,2,3]
        l[1] = 44

        DO THAT CAREFULLY
        Note: Re-assignment ignores the current contents.
        Note: Lists are *data structures* - they contain references     to other data

        key distinction:  creating new data versus updating an existing piece of data

 - why would you want to do this?

  - efficiency
        - e.g., suppose Amazon keeps a count of each item in
                its warehouse
                [['Hamlet', 2045], ['Python for dummies', 444], ...]
          - update it whenever someone buys a book
          - copying the entire list would be very slow
  - related:  models the real world
        - when a book leaves the warehouse, it's gone
                - can't sell it to two different people
        - similarly, ticketmaster selling tickets
        - so you actually do want to get rid of the old one
  - but need to do so with care
        - any part of the code still using the old version?


MEMORY SAFETY

While we're on the subject of memory, one of the most important guarantees that a languag
e provides is called *memory safety*.  Memory safety just means that a program will never
 read from or write to memory that it did not allocate.
```

The language does various things during program execution to ensure this property.

Example: bounds checks on list access
- show both read and write

Python has to check that the index of each list access is between 0 (inclusive) and lengt
h of the list (exclusive).

What would happen if it simply allowed the access?
- would be reading/writing some other memory
- has arbitrary behavior
  - read garbage data
  - worse: overwrite someone else's data!

Every language except C and C++ provides memory safety
- Java, C#, Javascript, ...
- they all do bounds checks on lists/arrays, just like Python
- and other things
- ensures that arbitrary behavior is never possible
  - crashes the program with an error before that happens

C and C++ were designed in the 1970s and 1980s, for a different set of assumptions:
- computers are slow
  - bounds checks take time
- programs run in a trusted environment
  - on your trusted computer
  - before the WWW and the cloud existed

So they made the reasonable decision at the time to simply allow reading/writing past the
bounds of a list/array. Called a *buffer overflow*.
- obviously a program error, but one that the programmer would eventually find/fix

Unfortunately, this decision is now the single biggest cause of security vulnerabilities
in the world today.
- attackers can take advantage of a buffer overflow to take control of a computer!
- intuition: some memory is used by Python to keep track of what instruction a program sh
ould execute next
  - example: keeps track during a function call of where to return back to after th
e call completes
  - called the *return address*
  - draw a picture of a memory with some program data,
    the return address
- if you can overwrite that memory, you can control what gets executed!
- further, the program's code is itself in memory!
  - show it
  - can use a buffer overflow to inject your own code and cause the program to execute
it!

Recent example: WannaCry ransomware attack in May 2017
- exploits a buffer overflow that exists in Microsoft Windows
  - implemented in C/C++
- takes control of the operating system
  - which is itself just a program that's always running on your machine!
- encrypts your data and demands a ransom payment
- infected 230,000 computers in 150 countries
- shut down parts of the UK hospital system

Can use the same technique for many other things:
- stealing secret information
- installing spyware
- deleting data

We've seen that memory is allocated when we create data in our program.
When is the memory deallocated?
- one option: only after a program finishes
  - but some programs may use a *lot* of memory
    - leaves little for other programs
    - or for themselves to reuse
  - and some programs may run for a long time

- so we want to deallocate memory once it's no longer being used by the program
- memory-safe languages do this for you automatically
    - called *garbage collection*
    - periodically finds data that is no longer used and reclaims that memory
    - show a simple example

```
l = [1,2,3]
l = [4,5,6]
```

- C/C++ require the programmer to explicitly deallocate memory
    - more efficient
    - more onerous
    - can cause other memory errors, including security holes
        - deallocate something before it is actually "dead"
            - when later accessed you can get garbage

Why are C and C++ still used today?
- lots of existing code that is unreasonable to completely rewrite in another language
- performance-critical code may be willing to trade off some security/reliability for efficiency (e.g., high-frequency trading)
- low-level code that requires bit-level access to memory (e.g., an operating system)
- small devices (e.g., phones, watches) that have limited memory and where energy efficiency is critical

C and C++ are not really required for any of these situations; it's a question of cost-benefit analysis. Unfortunately in our world, performance trumps security. Will take a major security incident to motivate a change in attitude.

Moral: Don't use C or C++ unless you have a *really* good reason to do so. They are no longer general-purpose languages -- they are needed for specific settings.

---

Believe it or not, you have learned almost everything there is to know about programming! The concepts and language features you've seen -- functions, loops, conditionals, assignments, map/filter/reduce, recursion -- are the main things used by professional programmers every day.

What's missing?

Language features that support safe and reliable programming. Key idea: help programmers isolate different parts of the code from one another!

Especially critical for large software programs, which are too big to fit in your head at one time.

Allows you to understand your code in pieces rather than having to have the whole thing in your head at once.
Allows different team members to implement, test, debug different parts separately.
Allows you to extend/modify them later without affecting other parts.

In this class we've seen one of the most common ways to decompose a program into parts:
*functional decomposition*
- the program is a bunch of functions, each of which solves some subtask

One way to help isolate functions from one another is to make them *pure*:
- only input to a function are arguments
- only output is the result
- no other *side effects*

But even pure functions in Python have a big problem: it is easy to crash them!
- simply provide the wrong kind of arguments

simple example:

```
def inc(x):
    return x+1
```

can happen accidentally (unclear on exactly what is expected)
or maliciously (an attacker trying to crash your code or worse - cause unintended behavior)

```
problem: it is not safe for the function to trust its callers

further, the requirements of the function implicitly impose requirements on callers:

def incAll(l):
        return list(map(inc, l))

Here l must be a list of integers, or else we crash
        - but you have to read through the code to figure that out!

how to address?
- functions in Python need to be *very* defensive
        - check their arguments at the top and reject bad ones
                - show pseudocode: if not an integer, crash("Message")
                - prevents unexpected behavior
                - but still crashes the program
        - relies on the programmer to remember to do this
                - and to do this properly

- and the same issue happens for the return value!

  if inc has a bug and sometimes returns a string, incAll will
        crash
        - show it
                if x == 34
                        return '35'

  so incAll has to be defensive against inc
  - and as things evolve these checks need to be updated
        correctly
  - a LOT of overhead
  - so in practice Python programs are very brittle
        - easy to crash them

to address this problem, many languages have *static typechecking*
        - static means before the program runs
        - typechecking means we check the types of data
        - Java, C, C++, ...

idea:  the programmer explicitly specifies the types of arguments and results of each fun
ction
        - the typechecker checks that these types are respected
                by the code
        - before the code runs
        - early error detection
        - and a strong guarantee for all possible inputs

I'll show how this works in OCaml, whose syntax is not that different from Python.

# let inc(x : int) : int =
    x + 1
  ;;
val inc : int -> int = <fun>

Callers are now checked to only pass ints

# inc(3)
  ;;
- : int = 4
# inc("hello")
  ;;
Error: This expression has type string but an expression was expected of type
         int

The above error is happening before the inc function is executed.

Can see this clearly by putting it in a function in Python and then in OCaml:

def callsInc():
        return inc('hello')
```

The requirements of inc propagate up explicitly to callers:

```
# let incAll(l : int list) : int list =
    (List.map inc l);;
val incAll : int list -> int list = <fun>
```

If we instead give l the type string list, we get an error
- show it

```
# incAll([1;2;3]);;
- : int list = [2; 3; 4]
# incAll([1;2;"hi"])
  ;;
Error: This expression has type string but an expression was expected of type
         int
#
```

But also, the typechecker has typechecked inc itself, to make sure it really provides the return value that it says it does

```
# let inc(x : int) : int =
    if x = 34
    then "35"
    else x + 1
  ;;
Error: This expression has type string but an expression was expected of type
         int
```

So we have a strong guarantee: for all possible executions,
if inc is given an int, it will return an int

This in turn ensures that incAll won't crash when calling inc

Python can *never* give you this guarantee
- best you can do is testing

```
# incAll([1;2;3]);;
- : int list = [2; 3; 4]
```

Static typechecking is critical for real-world software development.
- as I mentioned, C, C++, Java already have it
- other languages don't
  - Python, Javascript, PHP

But companies that need to use these other languages for various reasons have recognized the need:
SHOW THE WEBSITES
- mypy: typed Python (Dropbox)
- Typescript: typed Javascript (Microsoft)
- Reason: typed Javascript (Facebook)
- Hack: typed PHP (Facebook)

---
user-defined types

allow programmers to write software as a set of interacting *components*.

For this to be useful, it must be possible to implement, test, debug components separatel y. This allows team members to individually work separately on different components and later reliably put them together, allows you to easily reuse code written by others, etc.

Requirement: separation of *interface* from *implementation*
- each component has a well-defined interface to the rest of the program
  - typically a set of functions that can be called
- each component's implementation is *hidden*
  - ensures that other components cannot depend on implementation details
    - so they can be changed later without affecting the rest of the program
  - ensures that other components can't modify the component's state in a potential ly unsafe way

```
                - but instead only through the interface
                        - aka *API*

TOO ABSTRACT
JUST JUMP TO WANTING CHANGEABILITY / CONTROL OVER INVARIANTS
- can change rep
- can ensure ints < 256, lists of same length, etc.
THEN ADD STATIC TYPES FOR THAT
example from Python:  integers
        - you have no idea how they are implemented
                - how many bits?  in what format?
        - all you can do is pass them around and compute with
                them according to their interface
                - the operations they support:  +, -, //, <, %, etc.

        - enforces the *abstraction* of an integer

        why is this a good thing?

        - means you don't have to know how integers are implemented

        - means that if tomorrow the new version of Python completely changes the underly
ing representation of integers, your code still behaves the same
                - because your code cannot depend on the implementation details

        in other words, we've reduced *dependencies*:
                - code can't depend on the implementation of integers
                - allows for separation of concerns
                        - Python's job is to implement the interface
                        - my code gets to assume that interface

        this same exact property, separation of interface from implementation, is necessa
ry between code components as well

        example:  Suppose you and some friends are making a competitor to Instagram.  You
 are responsible for implementing the various filters on images.

        Fortunately you did hw5, so you will just use that.
        Provides an interface for reading/writing/manipulating image files.
        - show it

        Problem:  the homework exposes the underlying implementation of an image as a lis
t of list of lists.
        - show how readPPM produces a list

        Why is this a problem?
        - means that the other code in the system can depend on that representation
                - so you can't later change it easily

        - means that others can break your code!
                - e.g., provide lists with RGB values > 255
                - e.g., provide lists with pixels of size != 3
                - e.g., provide lists with unequal-sized inner lists
          - then what happens is implementation-dependent
                - can be a security risk

        - this could happen accidentally or maliciously
                - the Internet is a hostile place
        - makes your job much harder
                - have to program defensively

        Note that these problems will never occur if users only use your API:
                - call readPPM to get an image, use your transformations, call writePPM t
o write the image

        Need a way to *enforce* that users can only use this interface.

        A common way to do that is with object-oriented programming.
                - you'll learn about this is CS31/32 with C++
```

Actually does a bunch of things, but the most important is that it gives you a wa
y to define your own type of data and *hide* its implementation.
- just like ints are a built-in type of data with a hidden implementation

So I can create a new type of data (called a *class* in OO parlance), say ppmImag
e, along with a set of associated operations: readPPM returns a ppmImage, negate takes a
ppmImage and returns another one, writePPM takes a ppmImage and writes it to a file.

Critical: a ppmImage is implemented as a list of lists of pixels but *is not* a
list of lists of pixels. Can't pass a list of lists of pixels to negate any longer. The
only thing you can pass is a ppmImage and the only way to get one is to call readPPM!

When you learn about OO programming next quarter, keep in mind this key point: i
ts main benefit is the separation of interface from implementation. Critical for buildin
g reusable components, building large software projects in a reliable way.