
WELCOME!

(download slides and .py files and follow along!)

6.0001 LECTURE 1

TODAY

- course info
- what is computation
- python basics
- mathematical operations
- python variables and types
- NOTE: **slides and code files up before each lecture**
 - highly encourage you to download them before lecture
 - take notes and run code files when I do
 - bring computers to answer **in-class practice exercises!**

COURSE INFO

- Grading

- approx. 20% Quiz
- approx. 40% Final
- approx. 30% Problem Sets
- approx. 10% MITx Finger Exercises

COURSE POLICIES

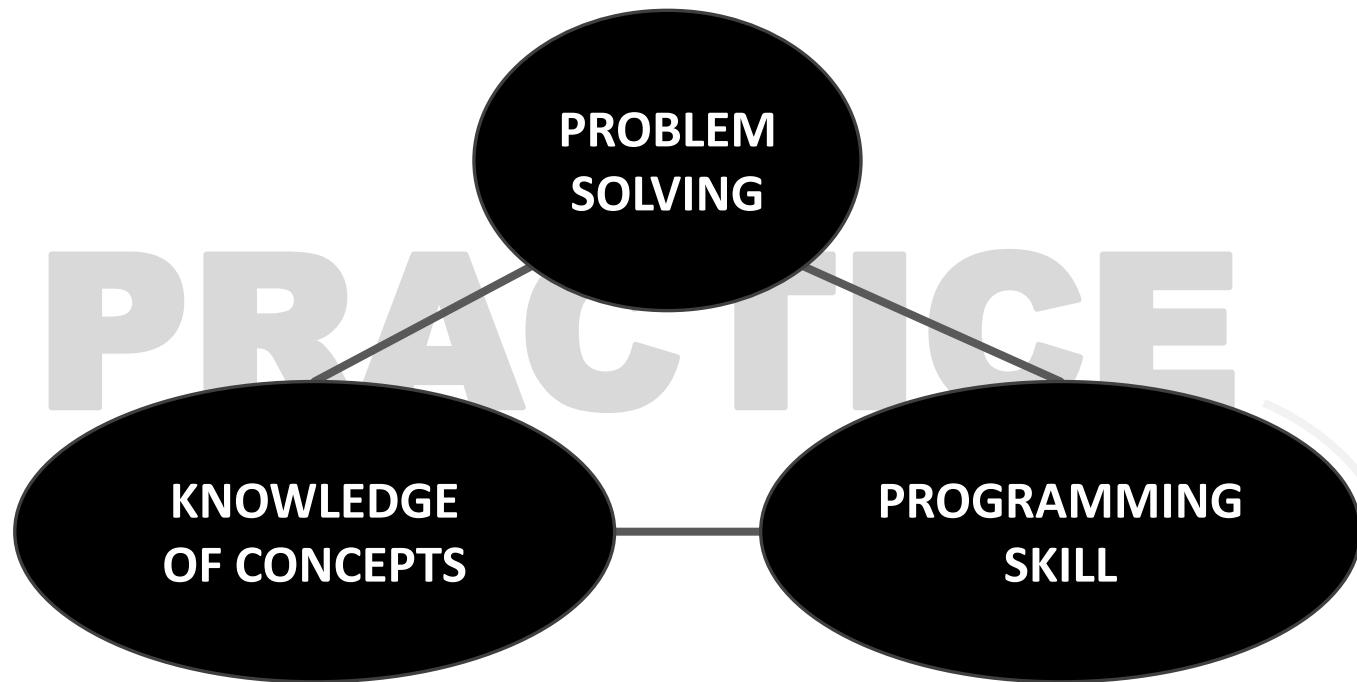
- Collaboration
 - may collaborate with anyone
 - required to write code independently and write names of all collaborators on submission
 - we will be running a code similarity program on all psets
- Extensions
 - **no extensions**
 - **late days**, see course website for details
 - **drop and roll** weight of max two psets in final exam grade
 - should be EMERGENCY use only

RECITATIONS

- not mandatory
- two flavors
 - 1) Lecture review: **review** lecture material
 - if you missed lecture
 - if you need a different take on the same concepts
 - 2) Problem solving: teach you **how to solve** programming problems
 - useful if you don't know how to set up pseudocode from pset words
 - we show a couple of harder questions
 - walk you through how to approach solving the problem
 - brainstorm code solution along with the recitation instructor
 - will post solutions after

FAST PACED COURSE

- Position yourself to succeed!
 - **read psets when they come out** and come back to them later
 - use late days in emergency situations
- New to programming? **PRACTICE. PRACTICE? PRACTICE!**
 - can't passively absorb programming as a skill
 - download code before lecture and follow along
 - do MITx finger exercises
 - don't be afraid to try out Python commands!



TOPICS

- represent knowledge with **data structures**
- **iteration and recursion** as computational metaphors
- **abstraction** of procedures and data types
- **organize and modularize** systems using object classes and methods
- different classes of **algorithms**, searching and sorting
- **complexity** of algorithms

WHAT DOES A COMPUTER DO

- Fundamentally:

- performs **calculations**
a billion calculations per second!
 - **remembers** results
100s of gigabytes of storage!

- What kinds of calculations?

- **built-in** to the language
 - ones that **you define** as the programmer

- computers only know what you tell them

TYPES OF KNOWLEDGE

- **declarative knowledge** is **statements of fact**.
 - someone will win a Google Cardboard before class ends
- **imperative knowledge** is a **recipe** or “how-to”.
 - 1) Students sign up for raffle
 - 2) Ana opens her IDE
 - 3) Ana chooses a random number between 1st and nth responder
 - 4) Ana finds the number in the responders sheet. Winner!

A NUMERICAL EXAMPLE

- square root of a number x is y such that $y^*y = x$
- recipe for deducing square root of a number x (16)
 - 1) Start with a **guess**, g
 - 2) If g^*g is **close enough** to x , stop and say g is the answer
 - 3) Otherwise make a **new guess** by averaging g and x/g
 - 4) Using the new guess, **repeat** process until close enough

g	g^*g	x/g	$(g+x/g)/2$
3	9	16/3	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

WHAT IS A RECIPE

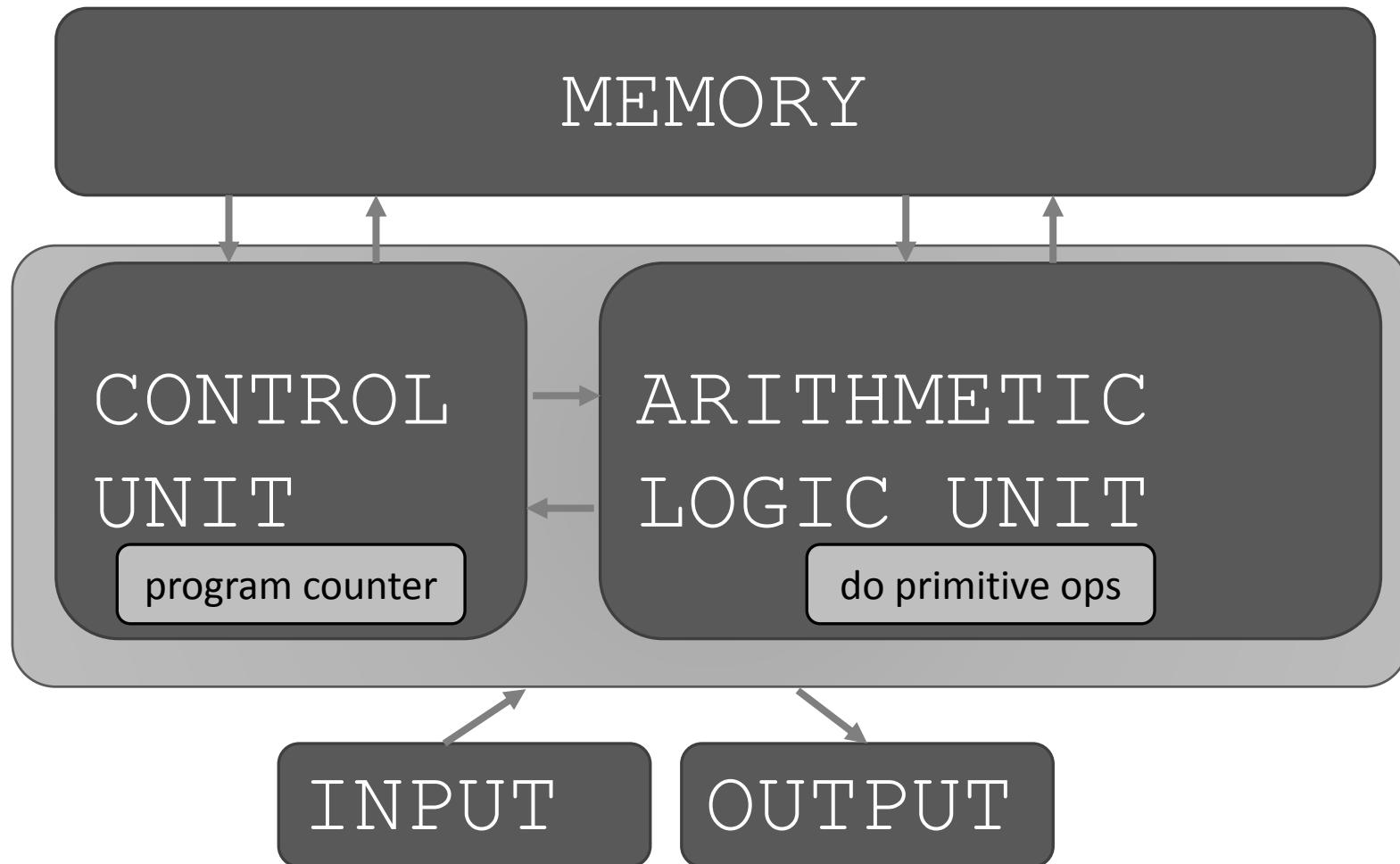
- 1) sequence of simple **steps**
- 2) **flow of control** process that specifies when each step is executed
- 3) a means of determining **when to stop**

$1+2+3 =$ an **algorithm!**

COMPUTERS ARE MACHINES

- how to capture a recipe in a mechanical process
- **fixed program** computer
 - calculator
- **stored program** computer
 - machine stores and executes instructions

BASIC MACHINE ARCHITECTURE



STORED PROGRAM COMPUTER

- sequence of **instructions stored** inside computer
 - built from predefined set of primitive instructions
 - 1) arithmetic and logic
 - 2) simple tests
 - 3) moving data
- special program (interpreter) **executes each instruction in order**
 - use tests to change flow of control through sequence
 - stop when done

BASIC PRIMITIVES

- Turing showed that you can **compute anything** using 6 primitives
- modern programming languages have more convenient set of primitives
- can abstract methods to **create new primitives**
- anything computable in one language is computable in any other programming language

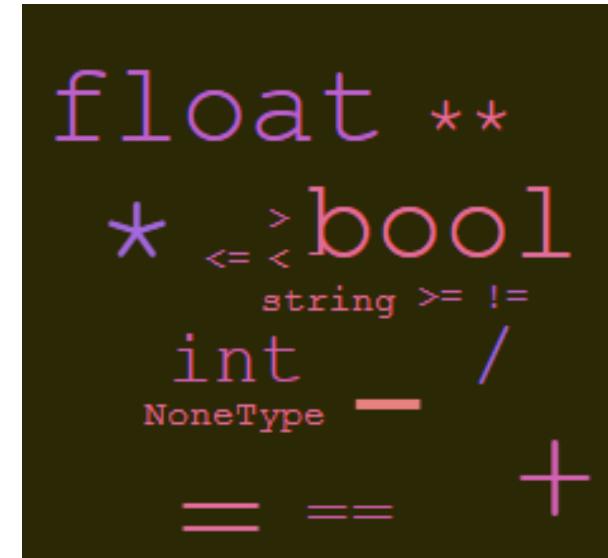
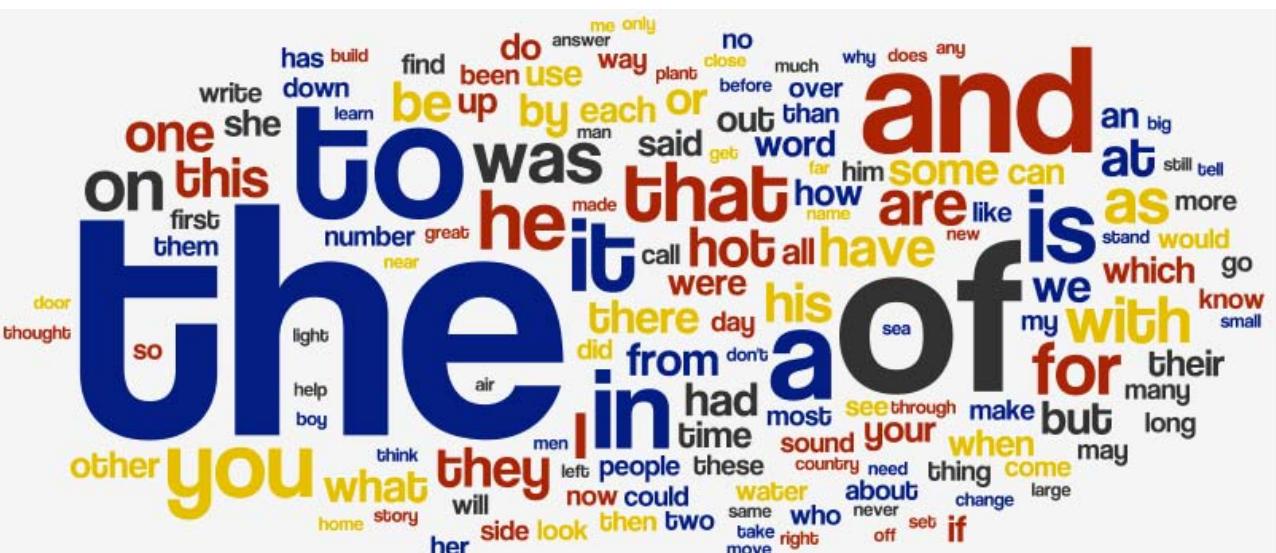
CREATING RECIPES

- a programming language provides a set of primitive **operations**
- **expressions** are complex but legal combinations of primitives in a programming language
- expressions and computations have **values** and meanings in a programming language

ASPECTS OF LANGUAGES

■ primitive constructs

- English: words
- programming language: numbers, strings, simple operators



Word Cloud copyright Michael Twardos, All Right Reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>.

Word Cloud copyright unknown, All Right Reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>.

ASPECTS OF LANGUAGES

■ **syntax**

- English: "cat dog boy" → not syntactically valid
"cat hugs boy" → syntactically valid
- programming language: "hi"5 → not syntactically valid
3.2*5 → syntactically valid

ASPECTS OF LANGUAGES

- **static semantics** is which syntactically valid strings have meaning
 - English: "I are hungry" → syntactically valid
but static semantic error
 - programming language: $3.2 * 5$ → syntactically valid
 $3 + "hi"$ → static semantic error

ASPECTS OF LANGUAGES

- **semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors
 - English: can have many meanings "Flying planes can be dangerous"
 - programming languages: have only one meaning but may not be what programmer intended

WHERE THINGS GO WRONG

- **syntactic errors**

- common and easily caught

- **static semantic errors**

- some languages check for these before running program
 - can cause unpredictable behavior

- no semantic errors but **different meaning than what programmer intended**

- program crashes, stops running
 - program runs forever
 - program gives an answer but different than expected

PYTHON PROGRAMS

- a **program** is a sequence of definitions and commands
 - definitions **evaluated**
 - commands **executed** by Python interpreter in a shell
- **commands** (statements) instruct interpreter to do something
- can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated
 - Problem Set 0 will introduce you to these in Anaconda

OBJECTS

- programs manipulate **data objects**
- objects have a **type** that defines the kinds of things programs can do to them
 - Ana is a human so she can walk, speak English, etc.
 - Chewbacca is a wookie so he can walk, “mwaaarhrhh”, etc.
- objects are
 - scalar (cannot be subdivided)
 - non-scalar (have internal structure that can be accessed)

SCALAR OBJECTS

- int – represent **integers**, ex. 5
- float – represent **real numbers**, ex. 3.27
- bool – represent **Boolean** values True and False
- NoneType – **special** and has one value, None
- can use type () to see the type of an object

```
>>> type(5)
```

```
int
```

```
>>> type(3.0)
```

```
float
```

*what you write into
the Python shell*

*what shows after
hitting enter*

TYPE CONVERSIONS (CAST)

- can **convert object of one type to another**
- `float(3)` converts integer 3 to float 3.0
- `int(3.9)` truncates float 3.9 to integer 3

PRINTING TO CONSOLE

- to show output from code to a user, use `print` command

```
In [11]: 3+2  
Out [11]: 5
```

*"Out" tells you it's an
interaction within the
shell only*

```
In [12]: print(3+2)  
5
```

*No "Out" means it is
actually shown to a user,
apparent when you
edit/run files*

EXPRESSIONS

- **combine objects and operators** to form expressions
- an expression has a **value**, which has a type
- syntax for a simple expression
 $<\text{object}> \text{ } <\text{operator}> \text{ } <\text{object}>$

OPERATORS ON ints and floats

- $i + j$ → the **sum** if both are ints, result is int
if either or both are floats, result is float
 - $i - j$ → the **difference**
 - $i * j$ → the **product**
 - i / j → **division** result is float
-
- $i \% j$ → the **remainder** when i is divided by j
 - $i ** j$ → i to the **power** of j

SIMPLE OPERATIONS

- parentheses used to tell Python to do these operations first
- **operator precedence** without parentheses
 - `**`
 - `*`
 - `/`
 - `+` and `-` executed left to right, as appear in expression

BINDING VARIABLES AND VALUES

- equal sign is an **assignment** of a value to a variable name

variable *value*
`pi = 3.14159`

`pi_approx = 22/7`

- value stored in computer memory
- an assignment binds name to value
- retrieve value associated with name or variable by invoking the name, by typing `pi`

ABSTRACTING EXPRESSIONS

- why **give names** to values of expressions?
- to **reuse names** instead of values
- easier to change code later

```
pi = 3.14159  
radius = 2.2  
area = pi*(radius**2)
```

PROGRAMMING vs MATH

- in programming, you do not “solve for x”

```
pi = 3.14159
```

```
radius = 2.2
```

```
# area of circle
```

```
area = pi*(radius**2)
```

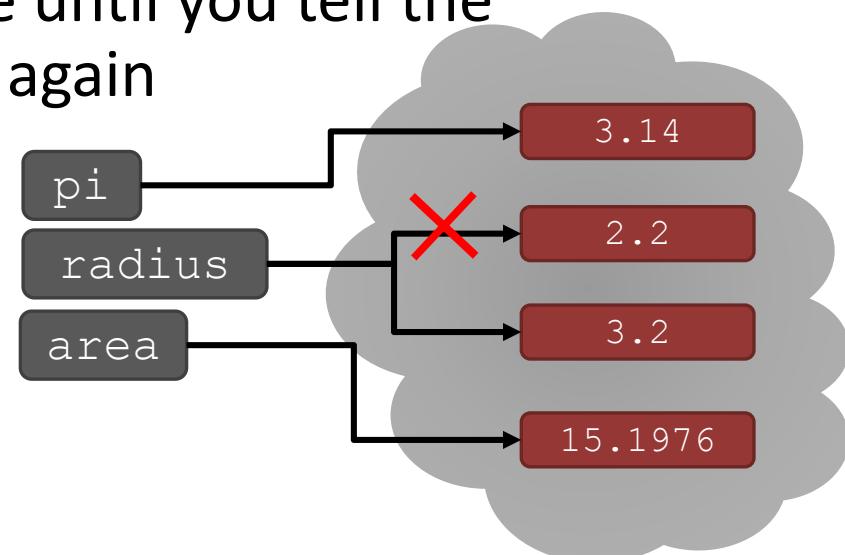
```
radius = radius+1
```

an assignment
* expression on the right, evaluated to a value
* variable name on the left
* equivalent expression to `radius = radius + 1`
is `radius += 1`

CHANGING BINDINGS

- can **re-bind** variable names using new assignment statements
- previous value may still stored in memory but lost the handle for it
- value for area does not change until you tell the computer to do the calculation again

```
pi = 3.14  
radius = 2.2  
area = pi*(radius**2)  
radius = radius+1
```



MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

BRANCHING, ITERATION

(download slides and .py files 'follow along!')

6.0001 LECTURE 2

LAST TIME

- syntax and semantics
- scalar objects
- simple operations
- expressions, variables and values

TODAY

- string object type
- branching and conditionals
- indentation
- iteration and loops

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **concatenate strings**

```
name = "ana"
```

```
greet = hi + name
```

```
greeting = hi + " " + name
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3
```

INPUT/OUTPUT: print

- used to **output** stuff to console
- keyword is `print`

```
x = 1  
print(x)  
  
x_str = str(x)  
print("my fav num is", x, ".", "x =", x)  
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: `input ("")`

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")  
print(5*text)
```

- `input` **gives you a string** so must cast if working with numbers

```
num = int(input("Type a number... "))  
print(5*num)
```

COMPARISON OPERATORS ON int, float, string

- i and j are variable names
- comparisons below evaluate to a Boolean

$i > j$

$i \geq j$

$i < j$

$i \leq j$

$i == j \rightarrow \text{equality}$ test, True if i is the same as j

$i != j \rightarrow \text{inequality}$ test, True if i not the same as j

LOGIC OPERATORS ON bools

- **a** and **b** are variable names (with Boolean values)

not a → True if a is False
False if a is True

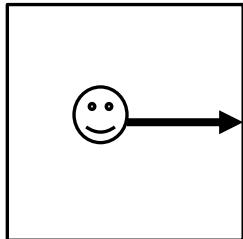
a and b → True if both are True

a or b → True if either or both are True

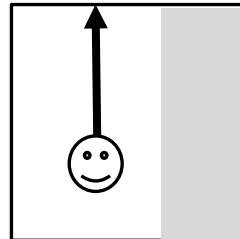
A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

COMPARISON EXAMPLE

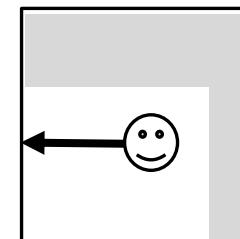
```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
derive = True
drink = False
both = drink and derive
print(both)
```



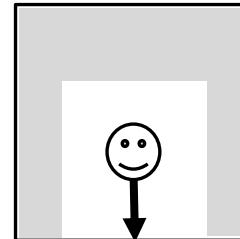
If right clear,
go right



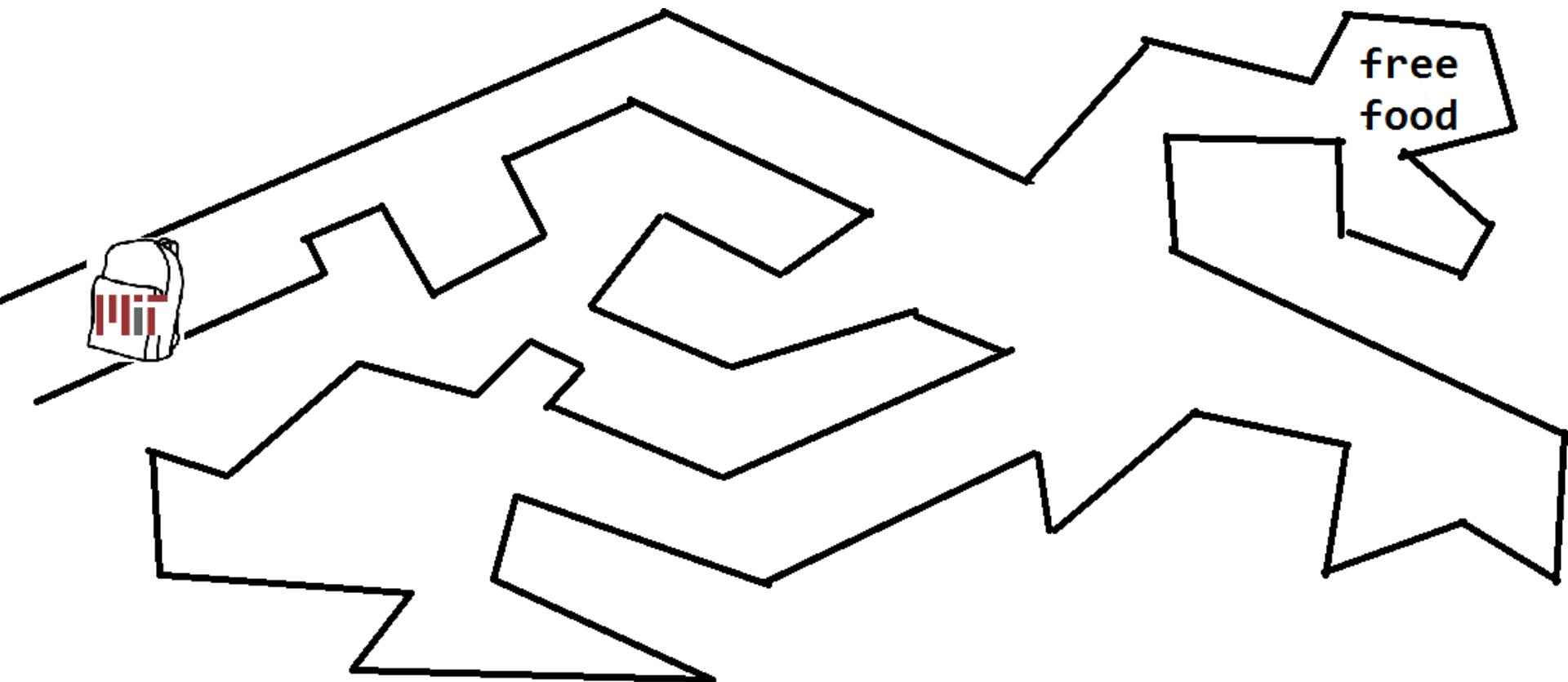
If right blocked,
go forward



If right and
front blocked,
go left



If right , front,
left blocked,
go back



CONTROL FLOW - BRANCHING

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- <condition> has a value True or False
- evaluate expressions in that block if <condition> is True

INDENTATION

- matters in Python
- how you denote blocks of code

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

= VS ==

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

What if x=y here?
get a SyntaxError



- Legend of Zelda – Lost Woods

- keep going right, takes you back to this same screen, stuck in a loop

Image Courtesy Nintendo, All Rights Reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.

```
if <exit right>:  
    <set background to woods_background>  
    if <exit right>:  
        <set background to woods_background>  
        if <exit right>:  
            <set background to woods_background>  
            and so on and on and on...  
        else:  
            <set background to exit_background>  
    else:  
        <set background to exit_background>  
else:  
    <set background to exit_background>
```



- Legend of Zelda – Lost Woods

- keep going right, takes you back to this same screen, stuck in a loop

Word Cloud copyright unknown, All Right Reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.

```
while <exit right>:  
    <set background to woods_background>  
<set background to exit_background>
```

CONTROL FLOW: while LOOPS

```
while <condition>:  
    <expression>  
    <expression>  
    ...
```

- <condition> evaluates to a Boolean
- if <condition> is True, do all the steps inside the while code block
- check <condition> again
- repeat until <condition> is False

while LOOP EXAMPLE

You are in the Lost Forest.



Go left or right?

PROGRAM:

```
n = input("You're in the Lost Forest. Go left or right? ")  
while n == "right":  
    n = input("You're in the Lost Forest. Go left or right? ")  
print("You got out of the Lost Forest!")
```

CONTROL FLOW: while and for LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1
```

```
# shortcut with for loop
for n in range(5):
    print(n)
```

CONTROL FLOW: for LOOPS

```
for <variable> in range(<some_num>):  
    <expression>  
    <expression>  
    ...
```

- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the prev value + 1
- etc.

range(start, stop, step)

- default values are start = 0 and step = 1 and optional
- loop until value is stop - 1

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

break STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:  
    while <condition_2>:  
        <expression_a>  
        break  
        <expression_b>  
    <expression_c>
```

break STATEMENT

```
mysum = 0  
for i in range(5, 11, 2):  
    mysum += i  
    if mysum == 5:  
        break  
    mysum += 1  
print(mysum)
```

- what happens in this program?

for

VS while LOOPS

for loops

- **know** number of iterations
- can **end early** via break
- uses a **counter**
- **can rewrite** a for loop using a while loop

while loops

- **unbounded** number of iterations
- can **end early** via break
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a while loop using a for loop

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

STRING MANIPULATION, GUESS-and-CHECK, APPROXIMATIONS, BISECTION

(download slides and .py files 'follow along!')

6.0001 LECTURE 3

LAST TIME

- strings
- branching – if/elif/else
- while loops
- for loops

TODAY

- string manipulation
- guess and check algorithms
- approximate solutions
- bisection method

STRINGS

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc.
- len () is a function used to retrieve the **length** of the string in the parentheses

```
s = "abc"
```

```
len(s) → evaluates to 3
```

STRINGS

- square brackets used to perform **indexing** into a string to get the value at a certain index/position

s = "abc"

index: 0 1 2 ← indexing always starts at 0

index: -3 -2 -1 ← last element always at index -1

s [0] → evaluates to "a"

s [1] → evaluates to "b"

s [2] → evaluates to "c"

s [3] → trying to index out of bounds, error

s [-1] → evaluates to "c"

s [-2] → evaluates to "b"

s [-3] → evaluates to "a"

STRINGS

- can **slice** strings using `[start:stop:step]`
- if give two numbers, `[start:stop]`, `step=1` by default
- you can also omit numbers and leave just colons

`s = "abcdefg"`

`s[3:6]` → evaluates to "def", same as `s[3:6:1]`

`s[3:6:2]` → evaluates to "df"

`s[::]` → evaluates to "abcdefg", same as `s[0:len(s):1]`

`s[::-1]` → evaluates to "hgfedcba", same as `s[-1:-(len(s)+1):-1]`

`s[4:1:-2]` → evaluates to "ec"

If unsure what some command does, try it out in your console!

STRINGS

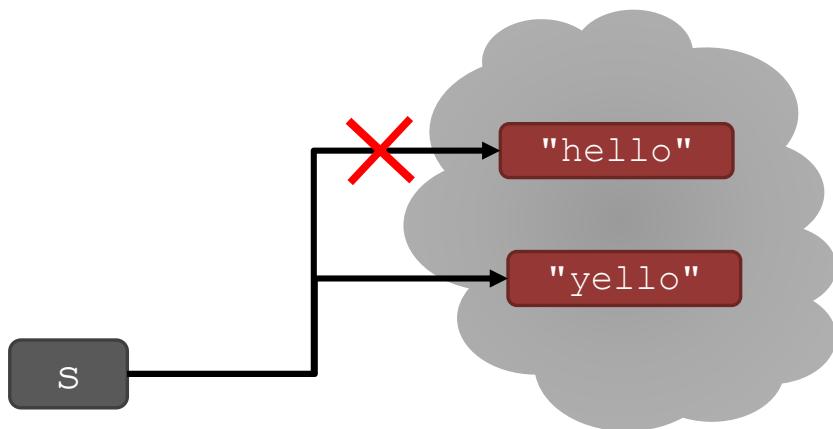
- strings are “**immutable**” – cannot be modified

```
s = "hello"
```

```
s[0] = 'y'
```

```
s = 'y'+s[1:len(s)]
```

→ gives an error
→ is allowed,
s bound to new object



for LOOPS RECAP

- for loops have a **loop variable** that iterates over a set of values

```
for var in range(4) :      → var iterates over values 0,1,2,3  
    <expressions>          → expressions inside loop executed  
                            with each value for var
```

```
for var in range(4, 6) : → var iterates over values 4,5  
    <expressions>
```

- range is a way to iterate over numbers, but a for loop variable can **iterate over any set of values**, not just numbers!

STRINGS AND LOOPS

- these two code snippets do the same thing
- bottom one is more “pythonic”

```
s = "abcdefg"  
  
for index in range(len(s)):  
    if s[index] == 'i' or s[index] == 'u':  
        print("There is an i or u")  
  
  
for char in s:  
    if char == 'i' or char == 'u':  
        print("There is an i or u")
```

CODE EXAMPLE: ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"  
  
word = input("I will cheer for you! Enter a word: ")  
times = int(input("Enthusiasm level (1-10): "))
```

```
i = 0  
while i < len(word):  
    char = word[i]  
    if char in an_letters:  
        print("Give me an " + char + "! " + char)  
    else:  
        print("Give me a " + char + "! " + char)  
i += 1  
print("What does that spell?")  
for i in range(times):  
    print(word, "!!!!")
```

for char in word: ✓

EXERCISE

```
s1 = "mit u rock"  
s2 = "i rule mit"  
if len(s1) == len(s2):  
    for char1 in s1:  
        for char2 in s2:  
            if char1 == char2:  
                print("common letter")  
                break
```

GUESS-AND-CHECK

- the process below also called **exhaustive enumeration**
- given a problem...
- you are able to **guess a value** for solution
- you are able to **check if the solution is correct**
- keep guessing until find solution or guessed all values

GUESS-AND-CHECK

– cube root

```
cube = 8  
  
for guess in range(cube+1):  
    if guess**3 == cube:  
        print("Cube root of", cube, "is", guess)
```

GUESS-AND-CHECK

– cube root

```
cube = 8

for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break

if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')
else:
    if cube < 0:
        guess = -guess

    print('Cube root of '+str(cube)+' is '+str(guess))
```

APPROXIMATE SOLUTIONS

- **good enough** solution
- start with a guess and increment by some **small value**
- keep guessing if $| \text{guess}^3 - \text{cube} | \geq \text{epsilon}$ for some **small epsilon**
- decreasing increment size → slower program
- increasing epsilon → less accurate answer

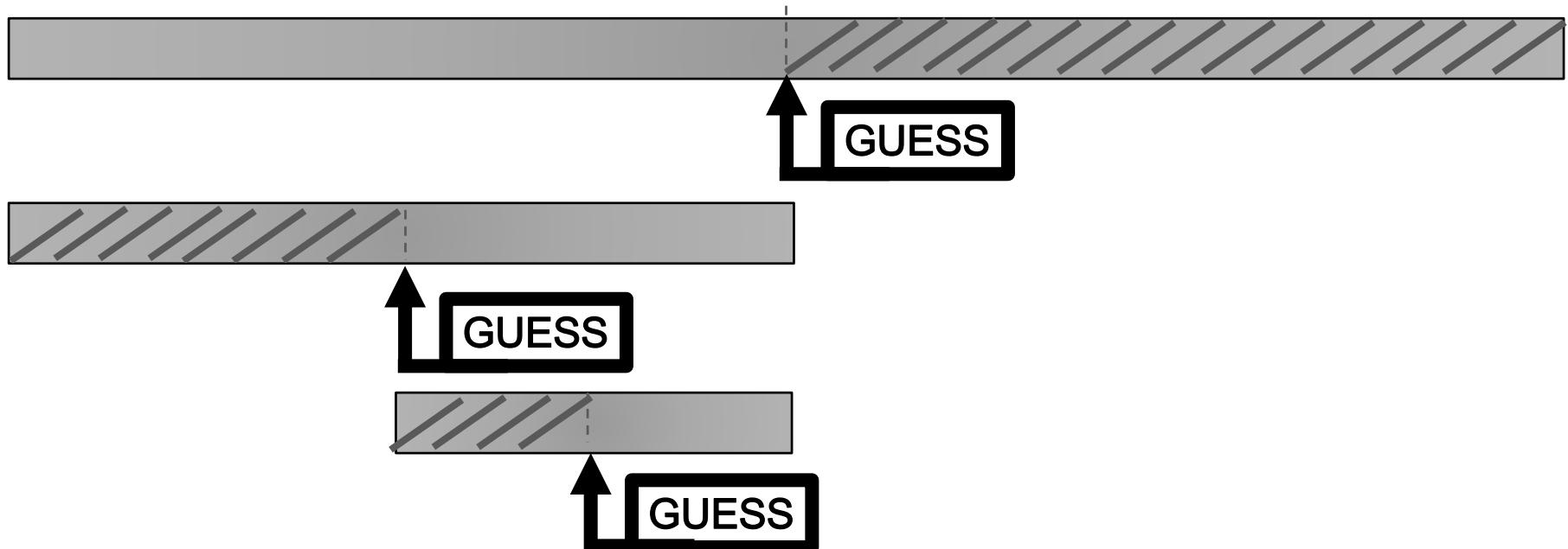
APPROXIMATE SOLUTION

– cube root

```
cube = 27
epsilon = 0.01
guess = 0.0
increment = 0.0001
num_guesses = 0
while abs(guess**3 - cube) >= epsilon and guess <= cube :
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube)
else:
    print(guess, 'is close to the cube root of', cube)
```

BISECTION SEARCH

- half interval each iteration
- new guess is halfway in between
- to illustrate, let's play a game!



BISECTION SEARCH

– cube root

```
cube = 27
epsilon = 0.01
num_guesses = 0
low = 0
high = cube
guess = (high + low)/2.0
while abs(guess**3 - cube) >= epsilon:
    if guess**3 < cube :
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0
    num_guesses += 1
print 'num_guesses =', num_guesses
print guess, 'is close to the cube root of', cube
```

BISECTION SEARCH CONVERGENCE

- search space
 - first guess: $N/2$
 - second guess: $N/4$
 - k th guess: $N/2^k$
- guess converges on the order of $\log_2 N$ steps
- bisection search works when value of function varies monotonically with input
- code as shown only works for positive cubes > 1 – why?
- challenges → modify to work with negative cubes!
→ modify to work with $x < 1$!

$x < 1$

- if $x < 1$, search space is 0 to x but cube root is greater than x and less than 1
- modify the code to choose the search space depending on value of x

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

DECOMPOSITION, ABSTRACTION, FUNCTIONS

(download slides and .py files 'follow along!')

6.0001 LECTURE 4

LAST TIME

- while loops vs for loops
- should know how to write both kinds
- should know when to use them
- guess-and-check and approximation methods
- bisection method to speed up programs

TODAY

- structuring programs and hiding details
- functions
- specifications
- keywords: `return` vs `print`
- scope

HOW DO WE WRITE CODE?

- so far...
 - covered language mechanisms
 - know how to write different files for each computation
 - each file is some piece of code
 - each code is a sequence of instructions

- problems with this approach
 - easy for small-scale problems
 - messy for larger problems
 - hard to keep track of details
 - how do you know the right info is supplied to the right part of code

GOOD PROGRAMMING

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- introduce **functions**
- mechanism to achieve **decomposition** and **abstraction**

EXAMPLE – PROJECTOR

- a projector is a black box
- don't know how it works
- know the interface: input/output
- connect any electronic to it that can communicate with that input
- black box somehow converts image from input source to a wall, magnifying it
- **ABSTRACTION IDEA:** do not need to know how projector works to use it

EXAMPLE – PROJECTOR

- projecting large image for Olympics decomposed into separate tasks for separate projectors
- each projector takes input and produces separate output
- all projectors work together to produce larger image
- **DECOMPOSITION IDEA:** different devices work together to achieve an end goal

APPLY THESE CONCEPTS

TO PROGRAMMING!

CREATE STRUCTURE with DECOMPOSITION

- in projector example, separate devices
- in programming, divide code into **modules**
 - are **self-contained**
 - used to **break up** code
 - intended to be **reusable**
 - keep code **organized**
 - **keep code coherent**
- this lecture, achieve decomposition with **functions**
- in a few weeks, achieve decomposition with **classes**

SUPPRESS DETAILS with ABSTRACTION

- in projector example, instructions for how to use it are sufficient, no need to know how to build one
- in programming, think of a piece of code as a **black box**
 - cannot see details
 - do not need to see details
 - do not want to see details
 - hide tedious coding details
- achieve abstraction with **function specifications** or **docstrings**

FUNCTIONS

- write reusable pieces/chunks of code, called **functions**
- functions are not run in a program until they are “**called**” or “**invoked**” in a program
- function characteristics:
 - has a **name**
 - has **parameters** (0 or more)
 - has a **docstring** (optional but recommended)
 - has a **body**
 - **returns** something

HOW TO WRITE and CALL/INVOKE A FUNCTION

```
keyword      name      parameters  
def is_even( i ): or arguments  
    """  
        Input: i, a positive int  
        Returns True if i is even, otherwise False  
    """  
body  
    print("inside is_even")  
    return i%2 == 0
```

specification,
docstring

later in the code, you call the
function using its name and
values for parameters

```
is_even(3)
```

IN THE FUNCTION BODY

```
def is_even( i ):
```

```
    """
```

Input: i , a positive int

Returns True if i is even, otherwise False

```
    """
```

```
    print("inside is_even")
```

```
    return i%2 == 0
```

keyword

expression to
evaluate and return

run some
commands

VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

```
def f( x ):    formal parameter
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )    actual parameter
```

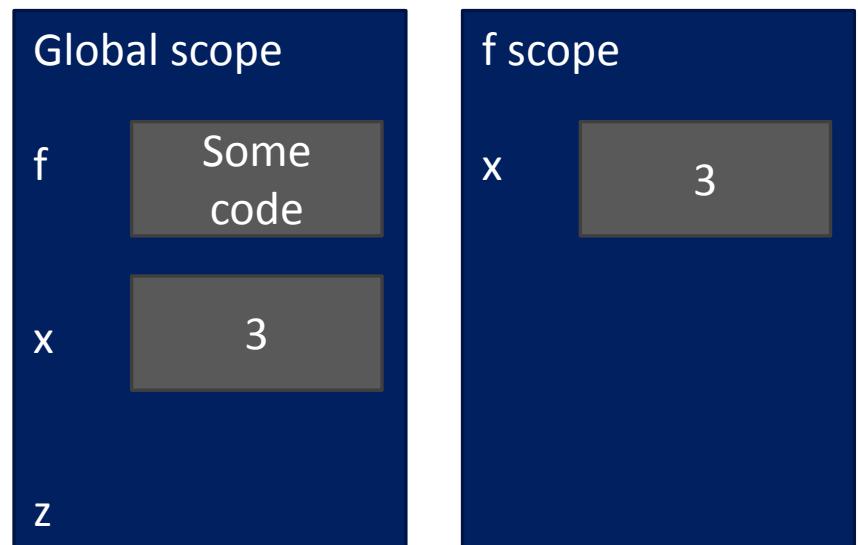
Function definition

Main program code

- * initializes a variable x
- * makes a function call f(x)
- * assigns return of function to variable z

VARIABLE SCOPE

```
def f( x ):  
    x = x + 1  
    print('in f(x): x =', x)  
    return x  
  
x = 3  
z = f( x )
```



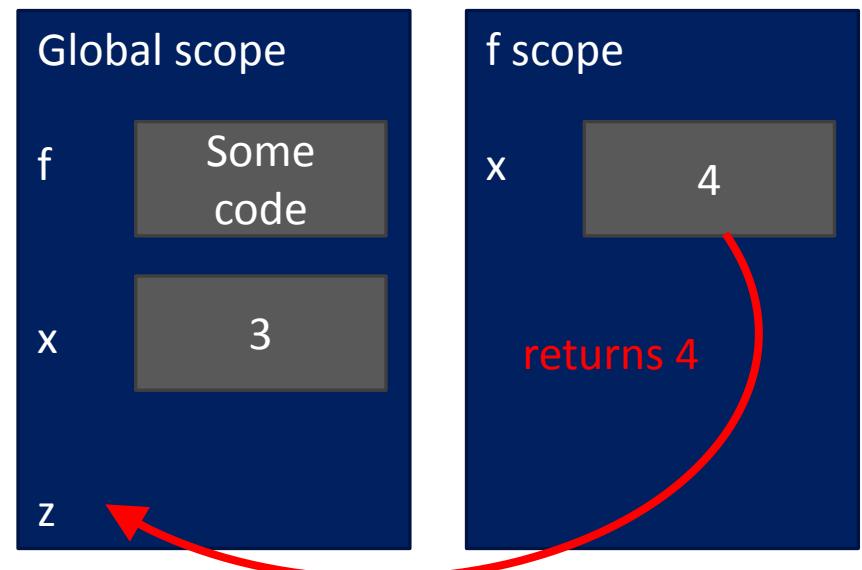
VARIABLE SCOPE

```
def f( x ):  
    x = x + 1  
    print('in f(x): x =', x)  
    return x  
  
x = 3  
z = f( x )
```



VARIABLE SCOPE

```
def f( x ):  
    x = x + 1  
    print('in f(x): x =', x)  
    return x  
  
x = 3  
z = f( x )
```



VARIABLE SCOPE

```
def f( x ):  
    x = x + 1  
    print('in f(x): x =', x)  
    return x  
  
x = 3  
z = f( x )
```



ONE WARNING IF NO return STATEMENT

```
def is_even( i ):  
    """  
        Input: i, a positive int  
        Does not return anything  
    """
```

```
i % 2 == 0
```

without a return
statement

- Python returns the value **None, if no return given**
- represents the absence of a value

return

VS.

print

- return only has meaning **inside** a function
- only **one** return executed inside a function
- code inside function but after return statement not executed
- has a value associated with it, **given to function caller**

- print can be used **outside** functions
- can execute **many** print statements inside a function
- code inside function can be executed after a print statement
- has a value associated with it, **outputted** to the console

FUNCTIONS AS ARGUMENTS

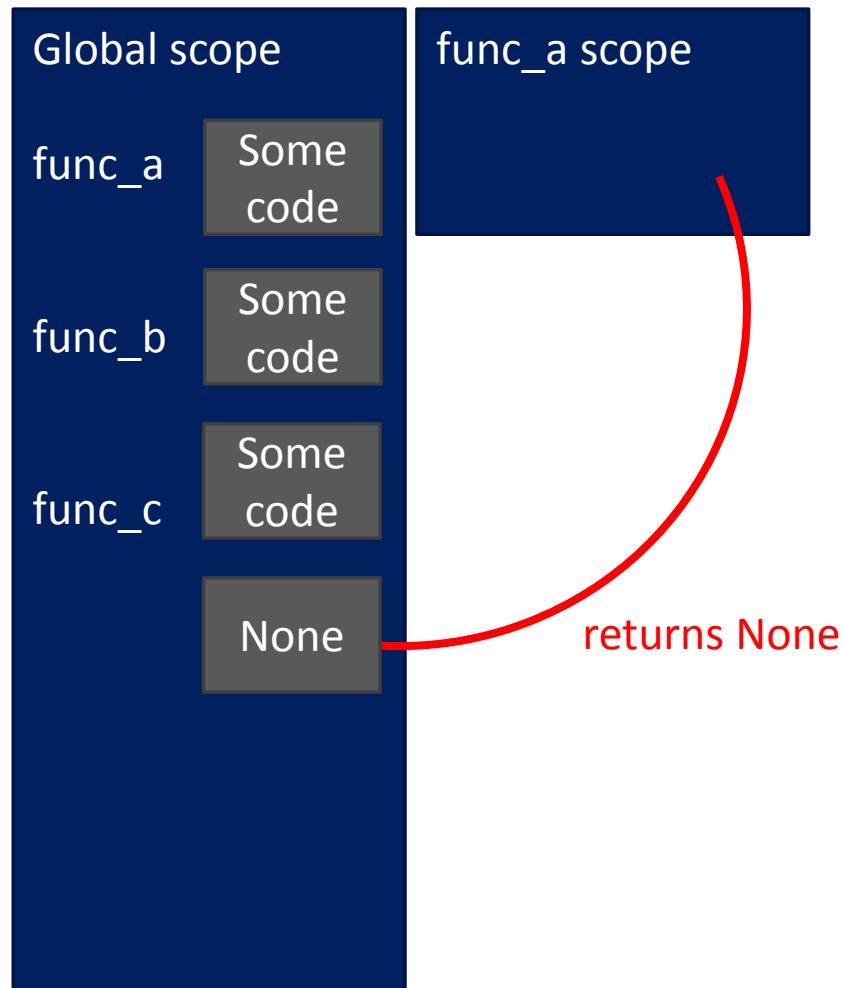
- arguments can take on any type, even functions

```
def func_a():  
    print 'inside func_a'  
  
def func_b(y):  
    print 'inside func_b'  
    return y  
  
def func_c(z):  
    print 'inside func_c'  
    return z()  
  
print func_a()  
print 5 + func_b(2)  
print func_c(func_a)
```

call func_a, takes no parameters
call func_b, takes one parameter
call func_c, takes one parameter, another function

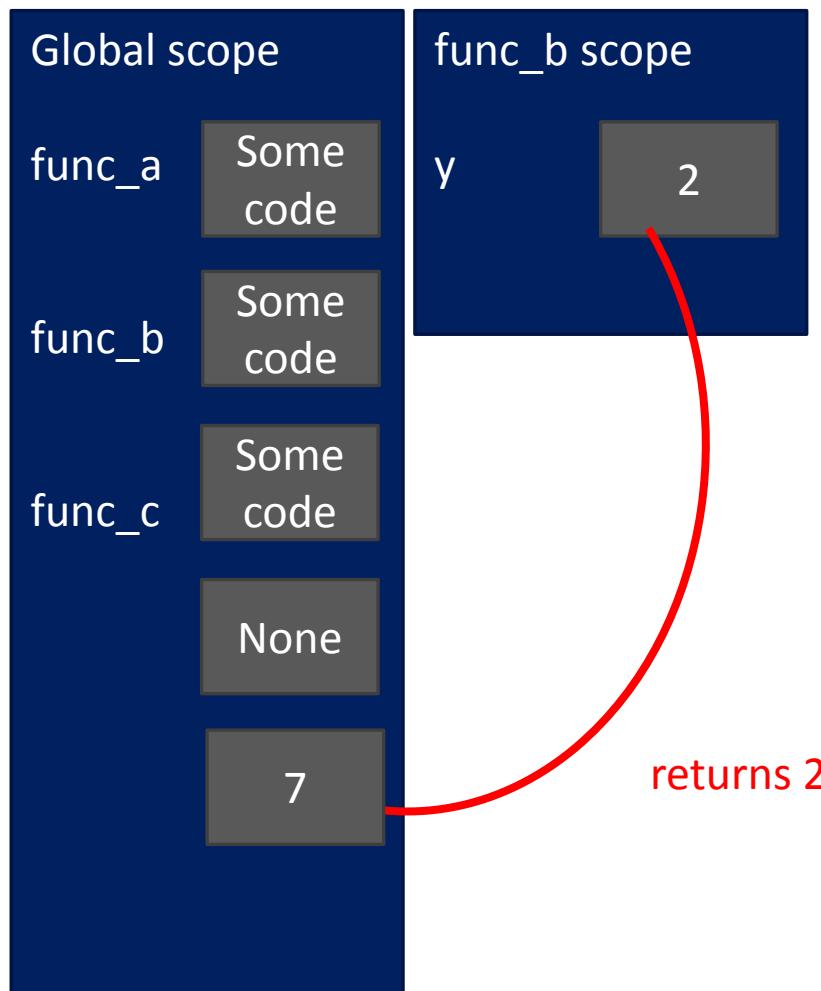
FUNCTIONS AS ARGUMENTS

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



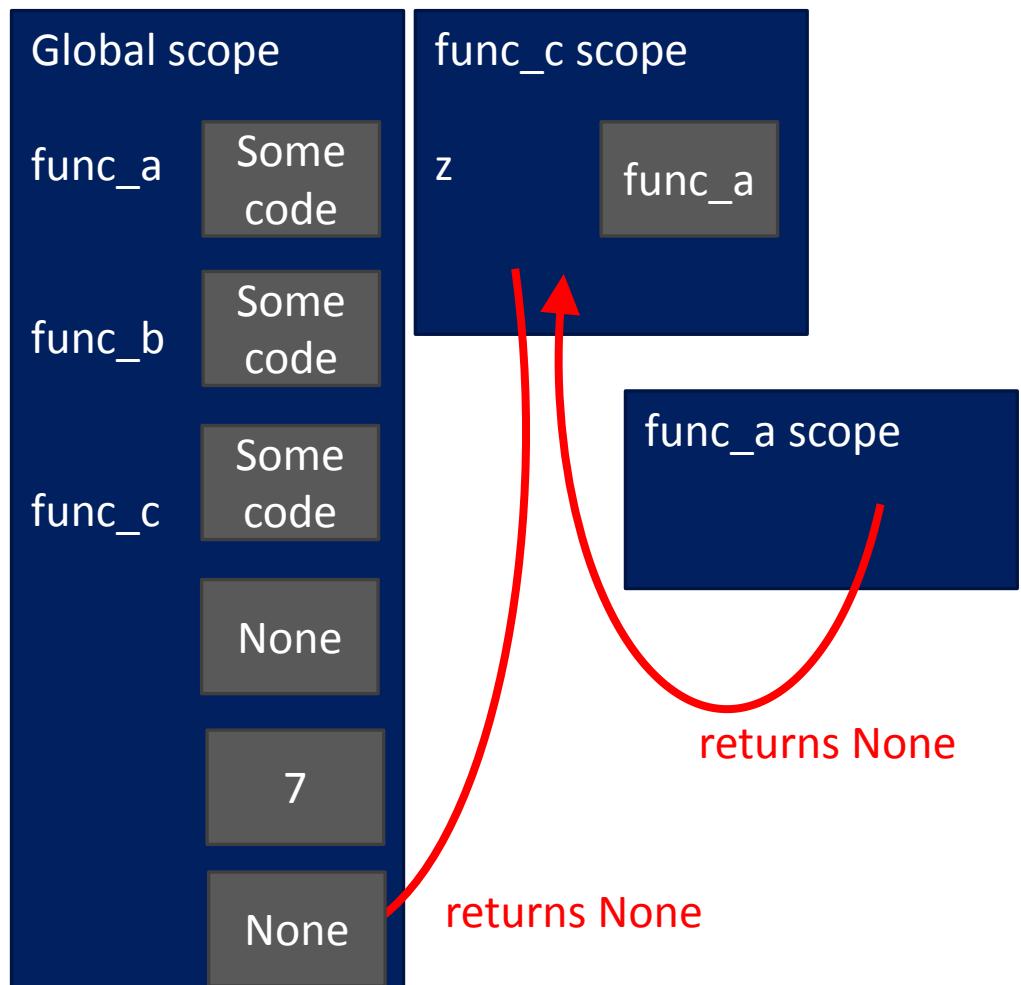
FUNCTIONS AS ARGUMENTS

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



FUNCTIONS AS ARGUMENTS

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can use **global variables**, but frowned upon

```
def f(y):  
    x = 1  
    x += 1  
    print(x)  
  
x = 5  
f(x)  
print(x)
```

*x is re-defined
in scope of f*

*different x
objects*

```
def g(y):  
    print(x)  
    print(x + 1)  
  
x = 5  
g(x)  
print(x)
```

*x from
outside g*

*x inside g is picked up
from scope that called
function g*

```
def h(y):  
    x += 1  
  
x = 5  
h(x)  
print(x)
```

*UnboundLocalError: local variable
'x' referenced before assignment*

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can use **global variables**, but frowned upon

```
def f(y):  
    x = 1  
    x += 1  
    print(x)  
  
x = 5  
f(x)  
print(x)
```

```
def g(y):  
    print(x)  
  
x = 5  
g(x)  
print(x)
```

```
def h(y):  
    x += 1  
  
x = 5  
h(x)  
print(x)
```

* from
global/main
program scope

HARDER SCOPE EXAMPLE



IMPORTANT
and
TRICKY!

Python Tutor is your best friend to help sort this out!

<http://www.pythontutor.com/>

SCOPE DETAILS

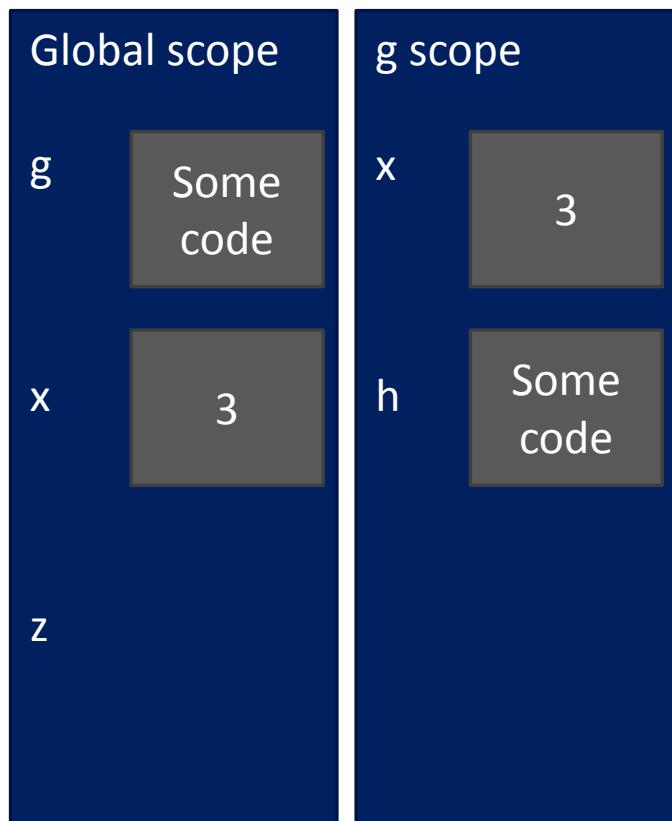
```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```

Some code



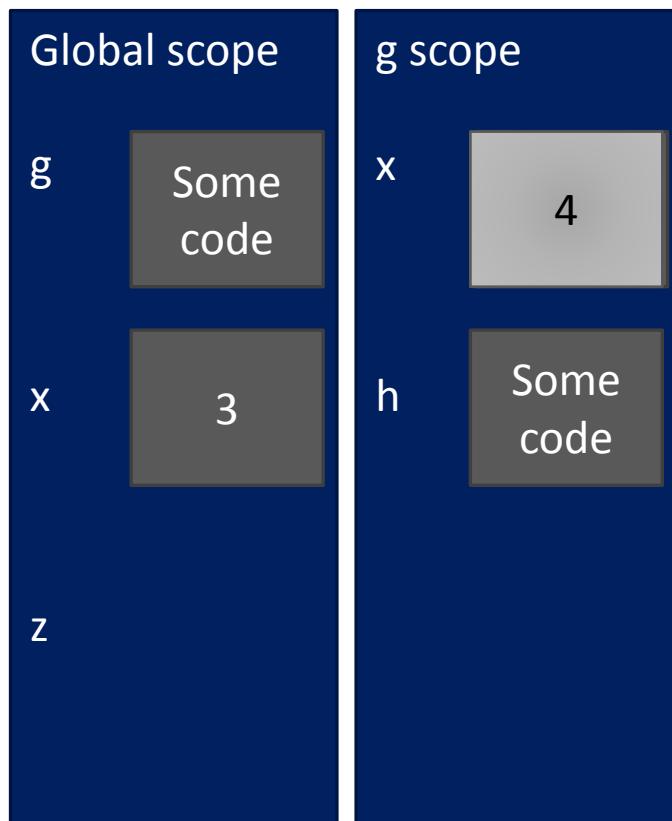
SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



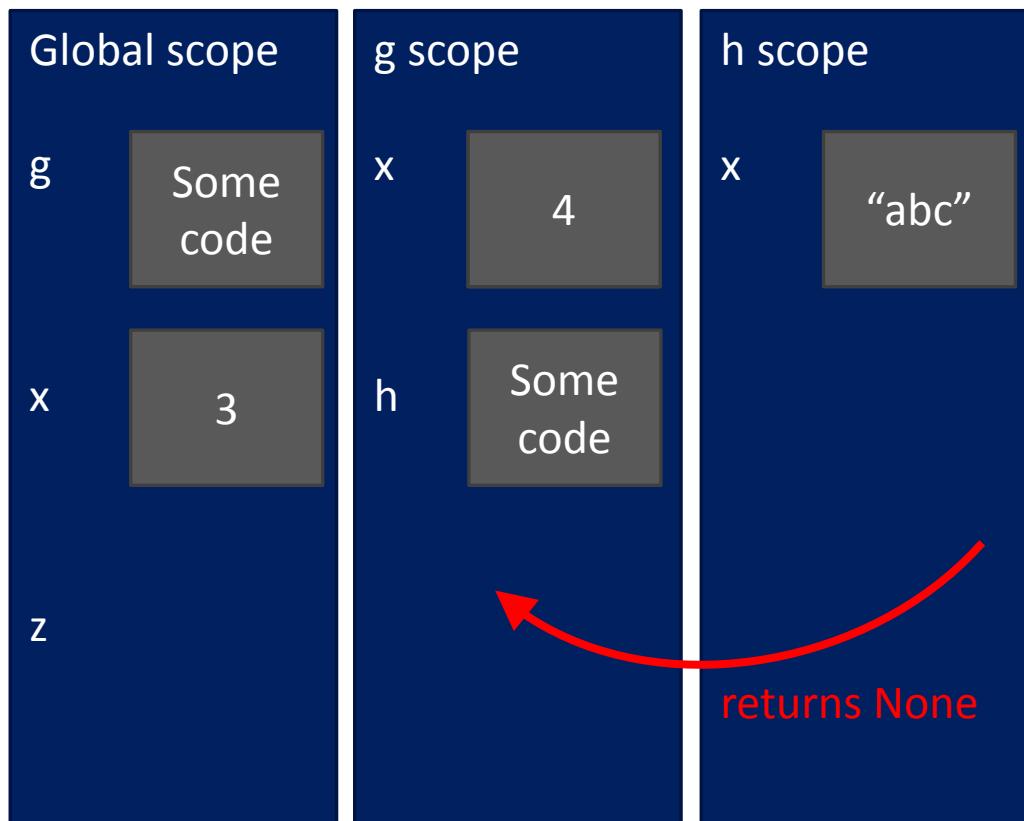
SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



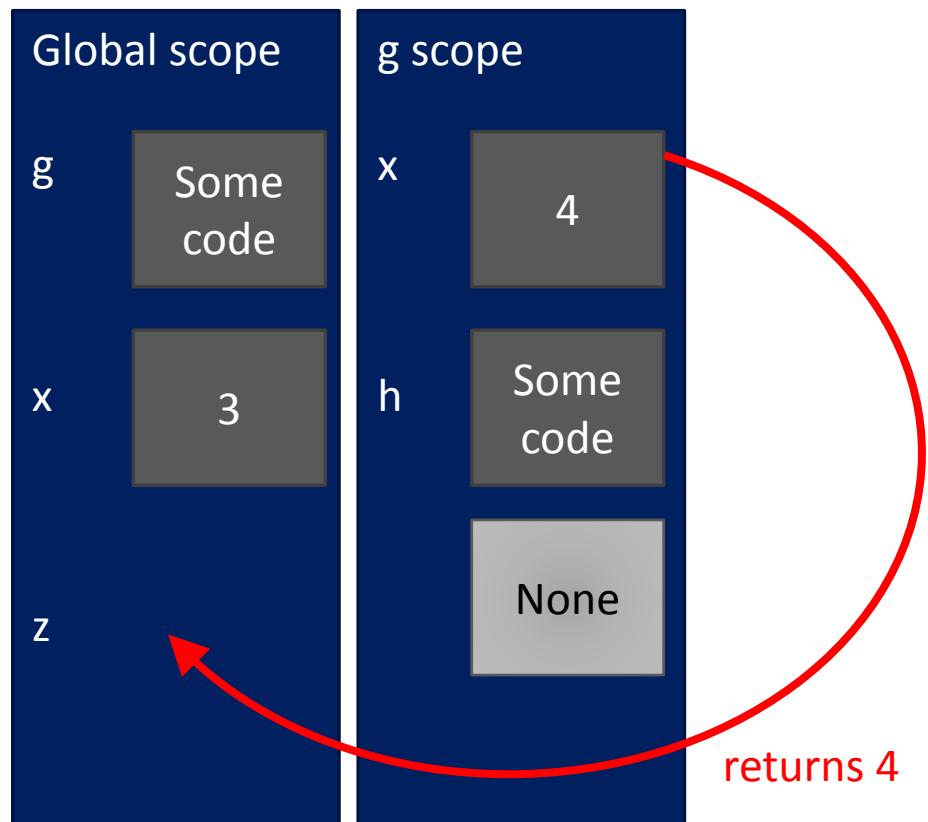
SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
    x = x + 1  
    print('g: x =', x)  
    h()  
    return x  
  
x = 3  
z = g(x)
```



SCOPE DETAILS

```
def g(x):  
    def h():  
        x = 'abc'  
        x = x + 1  
        print('g: x =', x)  
        h()  
    return x  
  
x = 3  
z = g(x)
```



DECOMPOSITION & ABSTRACTION

- powerful together
- code can be used many times but only has to be debugged once!

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

TUPLES, LISTS, ALIASING, MUTABILITY, CLONING

(download slides and .py files and follow along!)

6.0001 LECTURE 5

LAST TIME

- functions
- decomposition – create structure
- abstraction – suppress details
- from now on will be using functions a lot

TODAY

- have seen variable types: int, float, bool, string
- introduce new **compound data types**
 - tuples
 - lists
- idea of aliasing
- idea of mutability
- idea of cloning

TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

te = () *empty tuple*

t = (2, "mit", 3)

t[0] → evaluates to 2

(2, "mit", 3) + (5, 6) → evaluates to (2, "mit", 3, 5, 6)

t[1:2] → slice tuple, evaluates to ("mit", ,)

t[1:3] → slice tuple, evaluates to ("mit", 3)

len(t) → evaluates to 3

t[1] = 4 → gives error, can't modify object

*remember
strings?*

*extra comma
means a tuple
with one element*

TUPLES

- conveniently used to **swap** variable values

x = y

y = x



temp = x

x = y

y = temp



(x, y) = (y, x)



- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):  
    q = x // y  
    r = x % y  
    return (q, r)
```

integer
division

```
(quot, rem) = quotient_and_remainder(4, 5)
```

MANIPULATING TUPLES

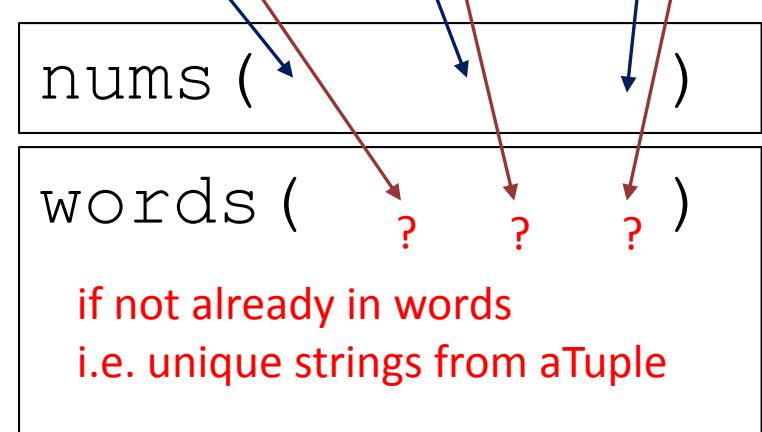
aTuple: ((ints), (strings), (ints))

- can **iterate** over tuples

```
def get_data(aTuple):  
    nums = ()  
    words = ()  
    for t in aTuple:  
        nums = nums + (t[0],)  
        if t[1] not in words:  
            words = words + (t[1],)  
  
    min_n = min(nums)  
    max_n = max(nums)  
    unique_words = len(words)  
    return (min_n, max_n, unique_words)
```

empty tuple

singleton tuple



LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, []
- a list contains **elements**
 - usually homogeneous (ie, all integers)
 - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

INDICES AND ORDERING

a_list = [] empty list

L = [2, 'a', 4, [1, 2]]

len(L) → evaluates to 4

L[0] → evaluates to 2

L[2]+1 → evaluates to 5

L[3] → evaluates to [1, 2], another list!

L[4] → gives an error

i = 2

L[i-1] → evaluates to 'a' since L[1] = 'a' above

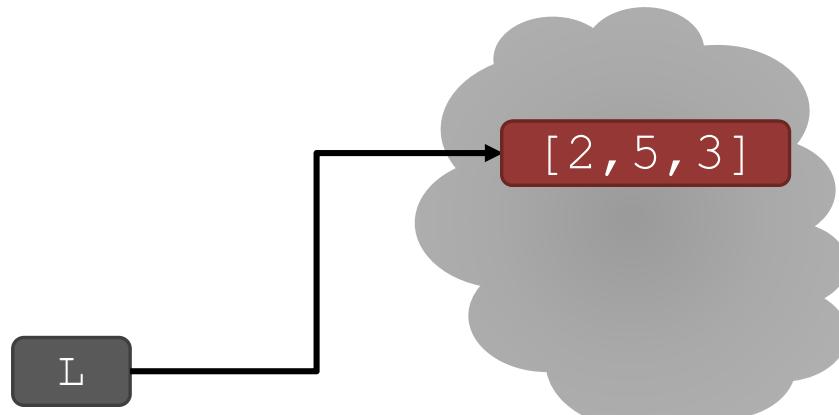
CHANGING ELEMENTS

- lists are **mutable!**
- assigning to an element at an index changes the value

```
L = [2, 1, 3]
```

```
L[1] = 5
```

- L is now [2, 5, 3], note this is the **same object L**



ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0  
  
for i in range(len(L)):  
    total += L[i]  
  
print total
```

```
total = 0  
  
for i in L:  
    total += i  
  
print total
```

like strings,
can iterate
over list
elements
directly

- notice
 - list elements are indexed 0 to `len(L) - 1`
 - `range(n)` goes from 0 to `n-1`

OPERATIONS ON LISTS - ADD

- **add** elements to end of list with `L.append(element)`
- **mutates** the list!

`L = [2, 1, 3]`

`L.append(5)` → L is now `[2, 1, 3, 5]`



- what is the dot?
 - lists are Python objects, everything in Python is an object
 - objects have data
 - objects have methods and functions
 - access this information by `object_name.do_something()`
 - will learn more about these later

OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator, to give you a new list
- **mutate** list with `L.extend(some_list)`

`L1 = [2, 1, 3]`

`L2 = [4, 5, 6]`

`L3 = L1 + L2` → `L3` is `[2, 1, 3, 4, 5, 6]`
`L1, L2` unchanged

`L1.extend([0, 6])` → mutated `L1` to `[2, 1, 3, 0, 6]`

OPERATIONS ON LISTS - REMOVE

- delete element at a **specific index** with `del (L[index])`
- remove element at **end of list** with `L.pop ()`, returns the removed element
- remove a **specific element** with `L.remove (element)`
 - looks for the element and removes it
 - if element occurs multiple times, removes first occurrence
 - if element not in list, gives an error

all these operations mutate the list

<code>L = [2, 1, 3, 6, 3, 7, 0] # do below in order</code>
<code>L.remove (2) → mutates L = [1, 3, 6, 3, 7, 0]</code>
<code>L.remove (3) → mutates L = [1, 6, 3, 7, 0]</code>
<code>del (L[1]) → mutates L = [1, 3, 7, 0]</code>
<code>L.pop () → returns 0 and mutates L = [1, 3, 7]</code>

CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- use `' '.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

<code>s = "I<3 cs"</code>	→ <code>s</code> is a string
<code>list(s)</code>	→ returns <code>['I', '<', '3', ' ', 'c', 's']</code>
<code>s.split('<')</code>	→ returns <code>['I', '3 cs']</code>
<code>L = ['a', 'b', 'c']</code>	→ <code>L</code> is a list
<code>' '.join(L)</code>	→ returns <code>"abc"</code>
<code>'_'.join(L)</code>	→ returns <code>"a_b_c"</code>

OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`
- and many more!

<https://docs.python.org/3/tutorial/datastructures.html>

`L=[9 , 6 , 0 , 3]`

`sorted(L)` → returns sorted list, does **not mutate** L

`L.sort()` → **mutates** L= [0 , 3 , 6 , 9]

`L.reverse()` → **mutates** L= [9 , 6 , 3 , 0]

MUTATION, ALIASING, CLONING



IMPORTANT
and
TRICKY!

*Again, Python Tutor is your best friend
to help sort this out!*

<http://www.pythontutor.com/>

LISTS IN MEMORY

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

AN ANALOGY

- attributes of a person
 - singer, rich
- he is known by many names
- all nicknames point to the **same person**
 - add new attribute to **one nickname** ...

Justin Bieber singer rich troublemaker

- ... **all his nicknames** refer to old attributes AND all new ones

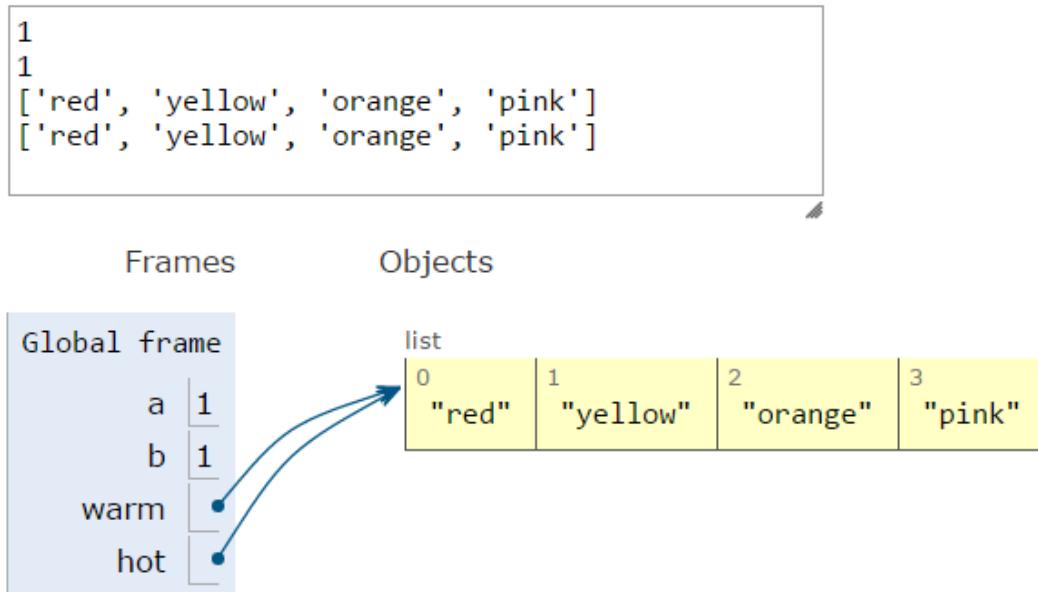
The Bieb singer rich troublemaker

JBeebs singer rich troublemaker

ALIASES

- hot is an **alias** for warm – changing one changes the other!
- append() has a side effect

```
1 a = 1
2 b = a
3 print(a)
4 print(b)
5
6 warm = ['red', 'yellow', 'orange']
7 hot = warm
8 hot.append('pink')
9 print(hot)
10 print(warm)
```

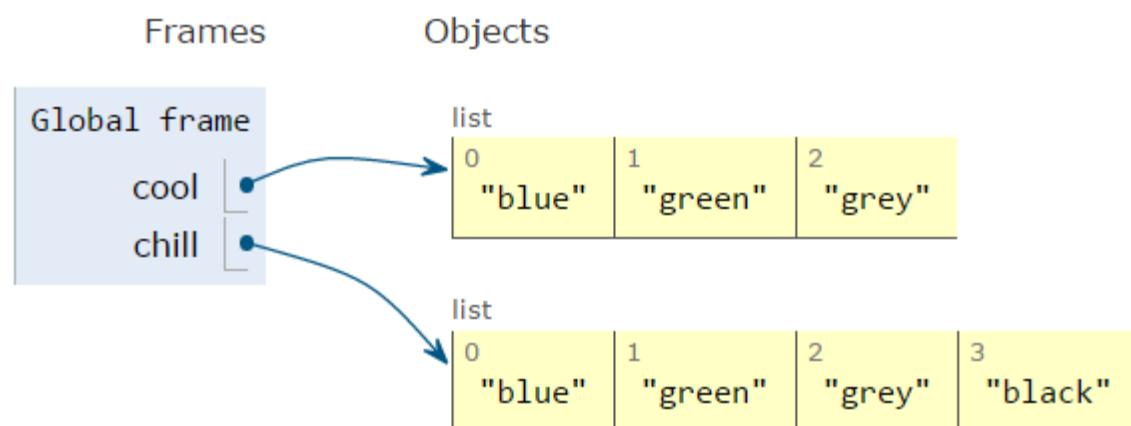


CLONING A LIST

- create a new list and **copy every element** using
chill = cool[:]

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

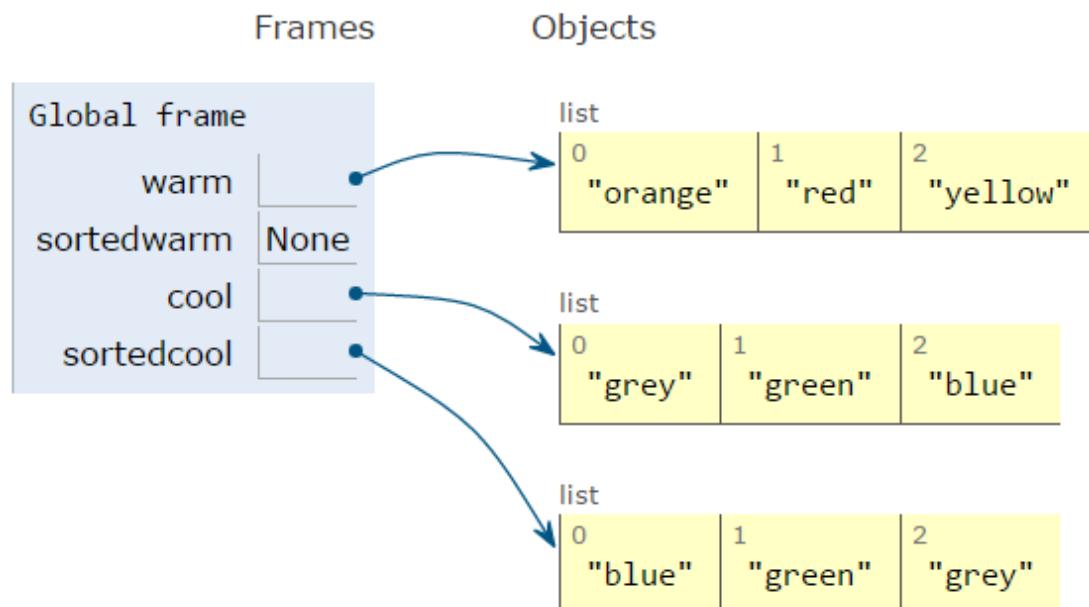


SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

```
1 warm = ['red', 'yellow', 'orange']
2 sortedwarm = warm.sort()
3 print(warm)
4 print(sortedwarm)
5
6 cool = ['grey', 'green', 'blue']
7 sortedcool = sorted(cool)
8 print(cool)
9 print(sortedcool)
```

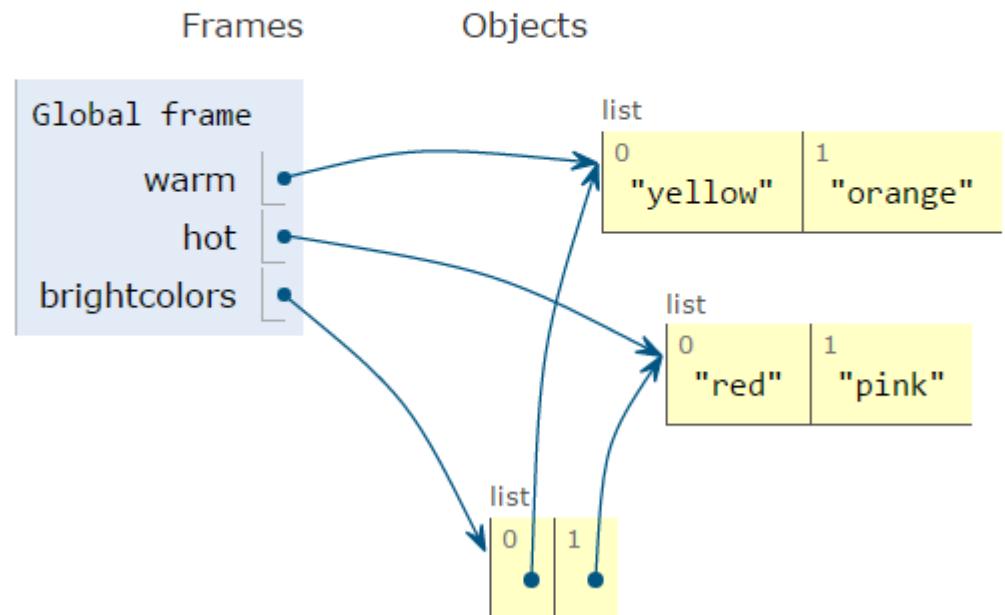
```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```



LISTS OF LISTS OF LISTS OF....

- can have **nested** lists
- side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]  
['red', 'pink']  
[['yellow', 'orange'], ['red', 'pink']]
```



```
1 warm = ['yellow', 'orange']  
2 hot = ['red']  
3 brightcolors = [warm]  
4 brightcolors.append(hot)  
5 print(brightcolors)  
6 hot.append('pink')  
7 print(hot)  
8 print(brightcolors)
```

MUTATION AND ITERATION

Try this in Python Tutor!

- **avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):  
    for e in L1:  
        if e in L2:  
            L1.remove(e)
```



```
def remove_dups(L1, L2):  
    L1_copy = L1[:]  
    for e in L1_copy:  
        if e in L2:  
            L1.remove(e)
```



```
L1 = [1, 2, 3, 4]  
L2 = [1, 2, 5, 6]  
remove_dups(L1, L2)
```

- L1 is [2, 3, 4] not [3, 4] Why?
 - Python uses an internal counter to keep track of index it is in the loop
 - mutating changes the list length but Python doesn't update the counter
 - loop never sees element 2

clone list first, note
that L1_copy = L1
does NOT clone

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

RECURSION, DICTIONARIES

(download slides and .py files and follow along!)

6.0001 LECTURE 6

QUIZ PREP

- a paper and an online component
- open book/notes
- not open Internet, not open computer
- start printing out whatever you may want to bring

LAST TIME

- tuples - immutable
- lists - mutable
- aliasing, cloning
- mutability side effects

TODAY

- recursion – divide/decrease and conquer
- dictionaries – another mutable object type

RECURSION

Recursion is the process of repeating items in a self-similar way.

WHAT IS RECURSION?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
 - reduce a problem to simpler versions of the same problem
- Semantically: a programming technique where a **function calls itself**
 - in programming, goal is to NOT have infinite recursion
 - must have **1 or more base cases** that are easy to solve
 - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

ITERATIVE ALGORITHMS SO FAR

- looping constructs (while and for loops) lead to **iterative** algorithms
- can capture computation in a set of **state variables** that update on each iteration through loop

MULTIPLICATION – ITERATIVE SOLUTION

- “multiply $a * b$ ” is equivalent to “add a to itself b times”

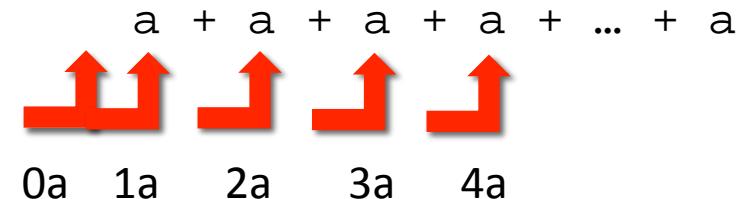
- capture **state** by

- an **iteration** number (i) starts at b

- $i \leftarrow i-1$ and stop when 0

- a current **value of computation** (result)

- $\text{result} \leftarrow \text{result} + a$



```
def mult_iter(a, b):  
    result = 0  
    while b > 0:  
        result += a  
        b -= 1  
    return result
```

iteration
current value of computation,
a running sum
current value of iteration variable

MULTIPLICATION – RECURSIVE SOLUTION

■ recursive step

- think how to reduce problem to a **simpler/ smaller version** of same problem

■ base case

- keep reducing problem until reach a simple case that can be **solved directly**
- when $b = 1$, $a^*b = a$

$$\begin{aligned} a^*b &= \underbrace{a + a + a + a + \dots + a}_{b \text{ times}} \\ &= a + \underbrace{a + a + a + a + \dots + a}_{b-1 \text{ times}} \\ &= a + \boxed{a * (b-1)} \end{aligned}$$

recursive reduction

```
def mult(a, b):
```

```
    if b == 1:  
        return a
```

base case

```
    else:
```

```
        return a + mult(a, b-1)
```

FACTORIAL

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$$

- for what n do we know the factorial?

$n = 1 \rightarrow$ if $n == 1:$
 return 1

base case

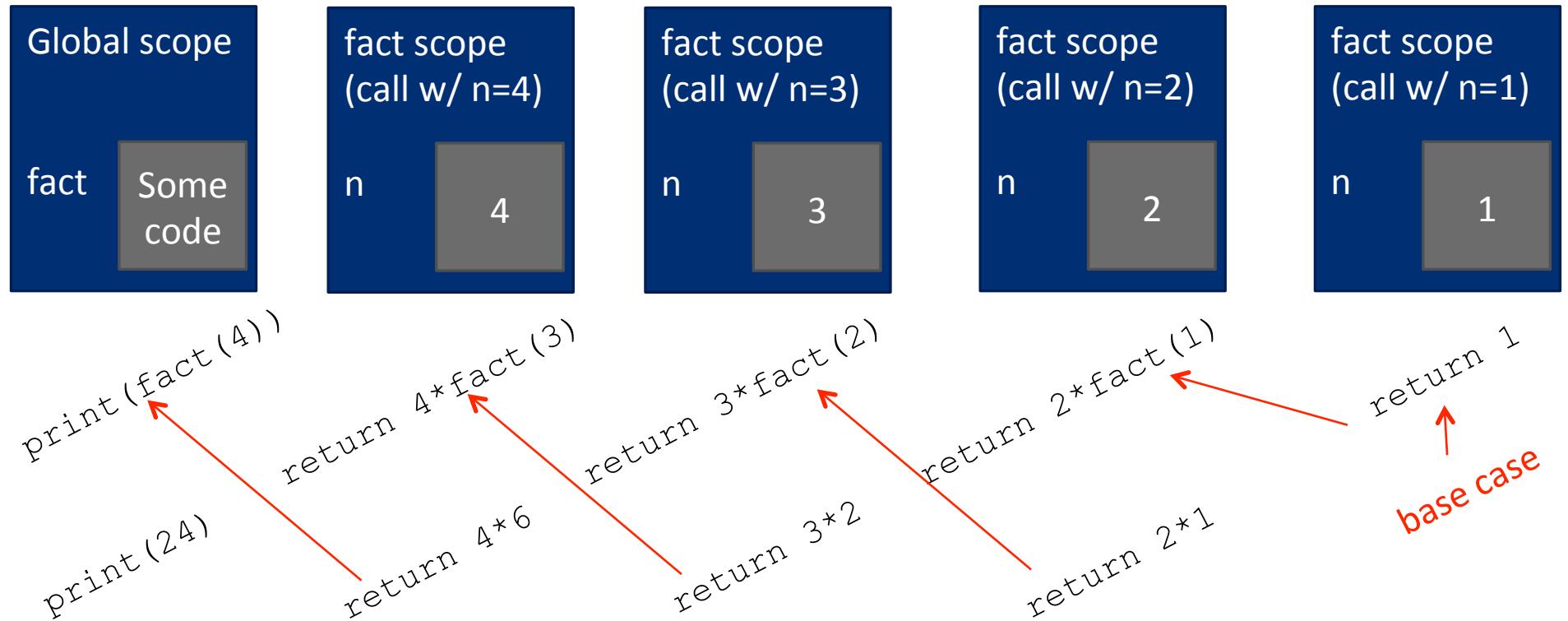
- how to reduce problem? Rewrite in terms of something simpler to reach base case

$n * (n-1)! \rightarrow$ else:
 return $n * \text{factorial}(n-1)$

recursive step

RECURSIVE FUNCTION SCOPE EXAMPLE

```
def fact(n):  
    if n == 1:  
        return 1  
    else:  
        return n*fact(n-1)  
  
print(fact(4))
```



SOME OBSERVATIONS

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

using the same variable
names but they are different
objects in separate scopes

ITERATION vs. RECURSION

```
def factorial_iter(n):      def factorial(n):  
    prod = 1                  if n == 1:  
    for i in range(1,n+1):     return 1  
        prod *= i              else:  
    return prod                  return n*factorial(n-1)
```

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer POV
- recursion may not be efficient from computer POV

INDUCTIVE REASONING

- How do we know that our recursive code will work?
- `mult_iter` terminates because `b` is initially positive, and decreases by 1 each time around loop; thus must eventually become less than 1
- `mult` called with `b = 1` has no recursive call and stops
- `mult` called with `b > 1` makes a recursive call with a smaller version of `b`; must eventually reach call with `b = 1`

```
def mult_iter(a, b):  
    result = 0  
  
    while b > 0:  
        result += a  
        b -= 1  
  
    return result
```

```
def mult(a, b):  
  
    if b == 1:  
        return a  
  
    else:  
  
        return a + mult(a, b-1)
```

MATHEMATICAL INDUCTION

- To prove a statement indexed on integers is true for all values of n :
 - Prove it is true when n is smallest value (e.g. $n = 0$ or $n = 1$)
 - Then prove that if it is true for an arbitrary value of n , one can show that it must be true for $n+1$

EXAMPLE OF INDUCTION

- $0 + 1 + 2 + 3 + \dots + n = (n(n+1))/2$
- Proof:
 - If $n = 0$, then LHS is 0 and RHS is $0*1/2 = 0$, so true
 - Assume true for some k , then need to show that
$$0 + 1 + 2 + \dots + k + (k+1) = ((k+1)(k+2))/2$$
 - LHS is $k(k+1)/2 + (k+1)$ by assumption that property holds for problem of size k
 - This becomes, by algebra, $((k+1)(k+2))/2$
 - Hence expression holds for all $n \geq 0$

RELEVANCE TO CODE?

- Same logic applies

```
def mult(a, b):  
    if b == 1:  
        return a  
  
    else:  
        return a + mult(a, b-1)
```

- Base case, we can show that `mult` must return correct answer
- For recursive case, we can assume that `mult` correctly returns an answer for problems of size smaller than b , then by the addition step, it must also return a correct answer for problem of size b
- Thus by induction, code correctly returns answer

TOWERS OF HANOI

- The story:
 - 3 tall spikes
 - Stack of 64 different sized discs – start on one spike
 - Need to move stack to second spike (at which point universe ends)
 - Can only move one disc at a time, and a larger disc can never cover up a small disc

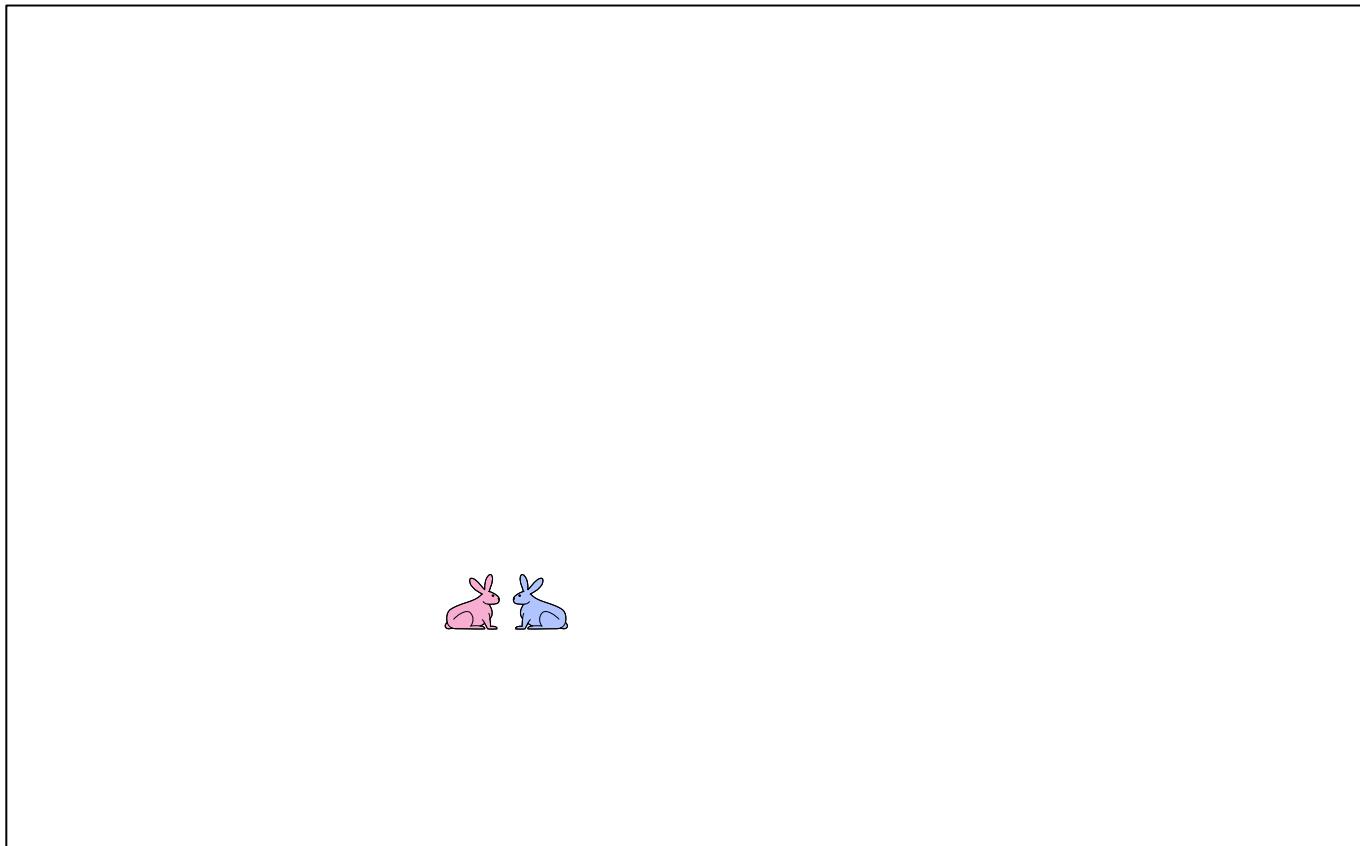
TOWERS OF HANOI

- Having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves?
- **Think recursively!**
 - Solve a smaller problem
 - Solve a basic problem
 - Solve a smaller problem

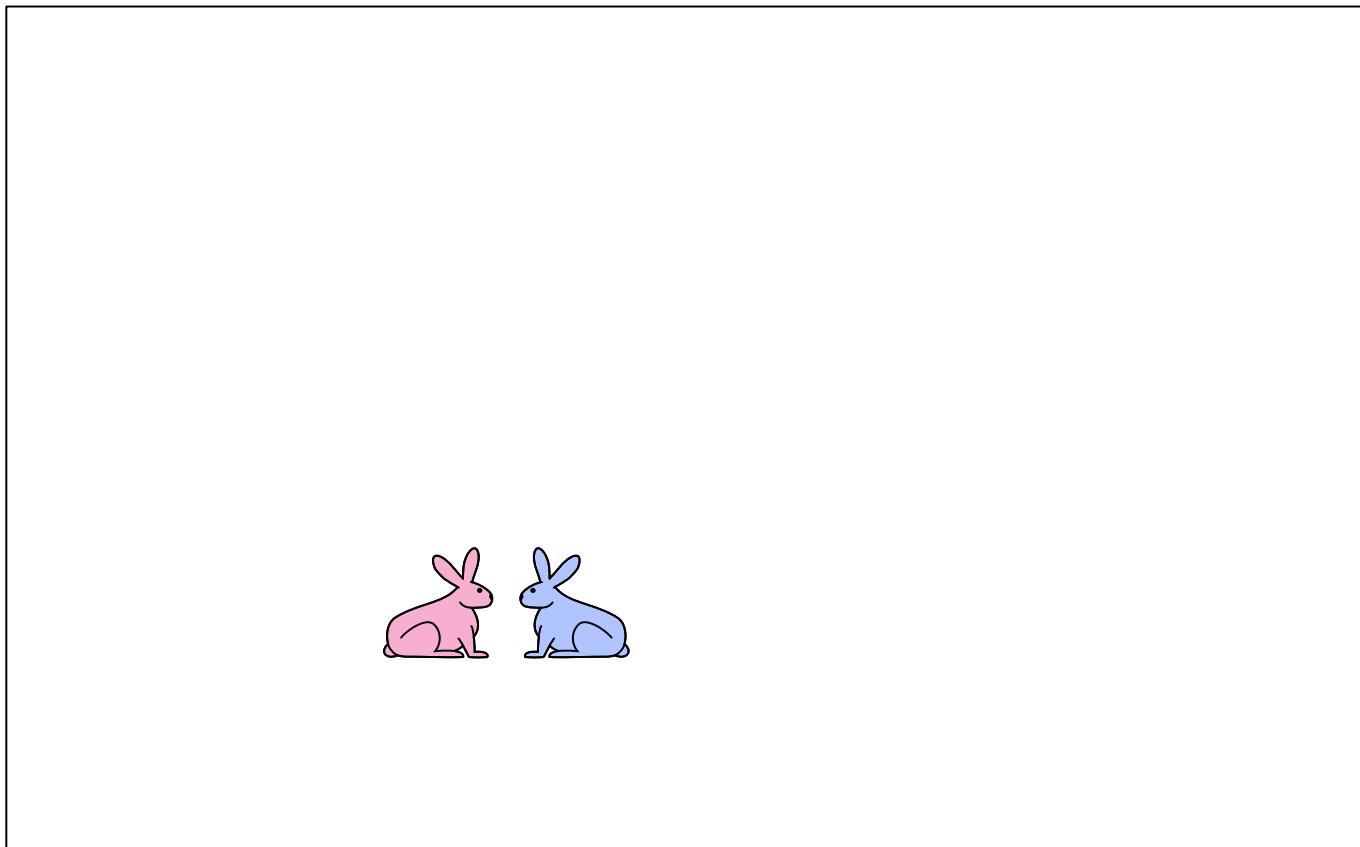
```
def printMove(fr, to):  
    print('move from ' + str(fr) + ' to ' + str(to))  
  
def Towers(n, fr, to, spare):  
    if n == 1:  
        printMove(fr, to)  
    else:  
        Towers(n-1, fr, spare, to)  
        Towers(1, fr, to, spare)  
        Towers(n-1, spare, to, fr)
```

RECURSION WITH MULTIPLE BASE CASES

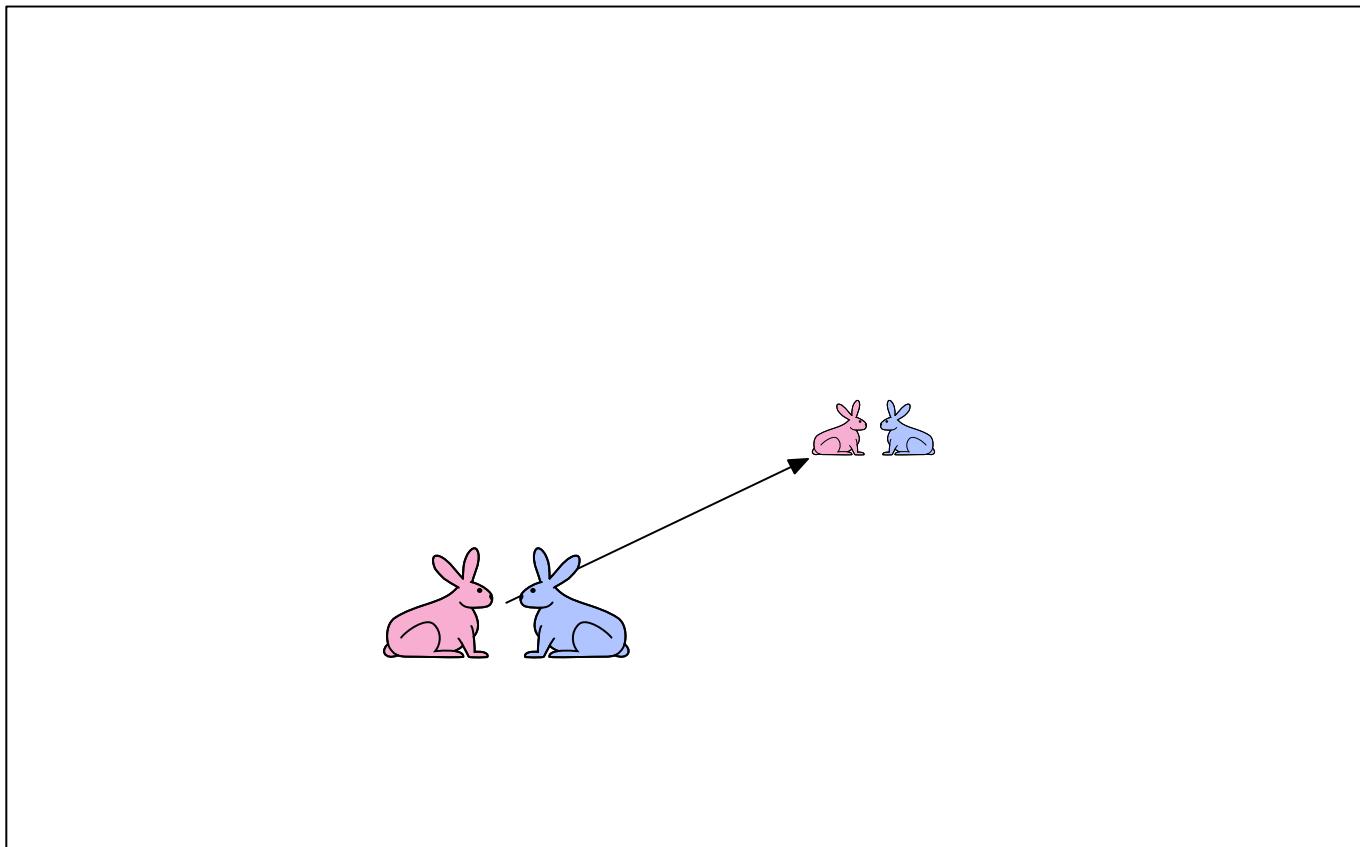
- Fibonacci numbers
 - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
 - Newborn pair of rabbits (one female, one male) are put in a pen
 - Rabbits mate at age of one month
 - Rabbits have a one month gestation period
 - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
 - How many female rabbits are there at the end of one year?



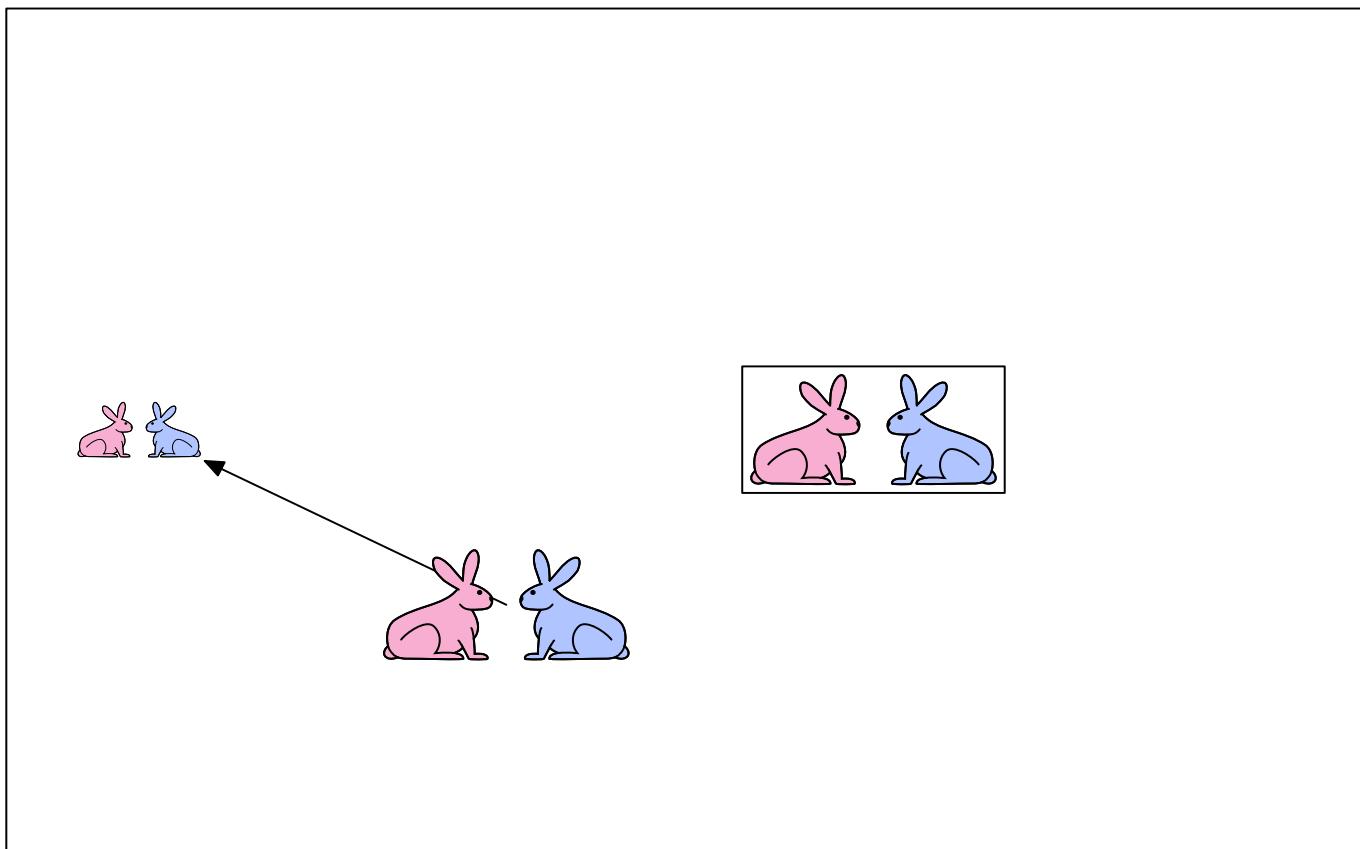
Demo courtesy of Prof. Denny Freeman and Adam Hartz



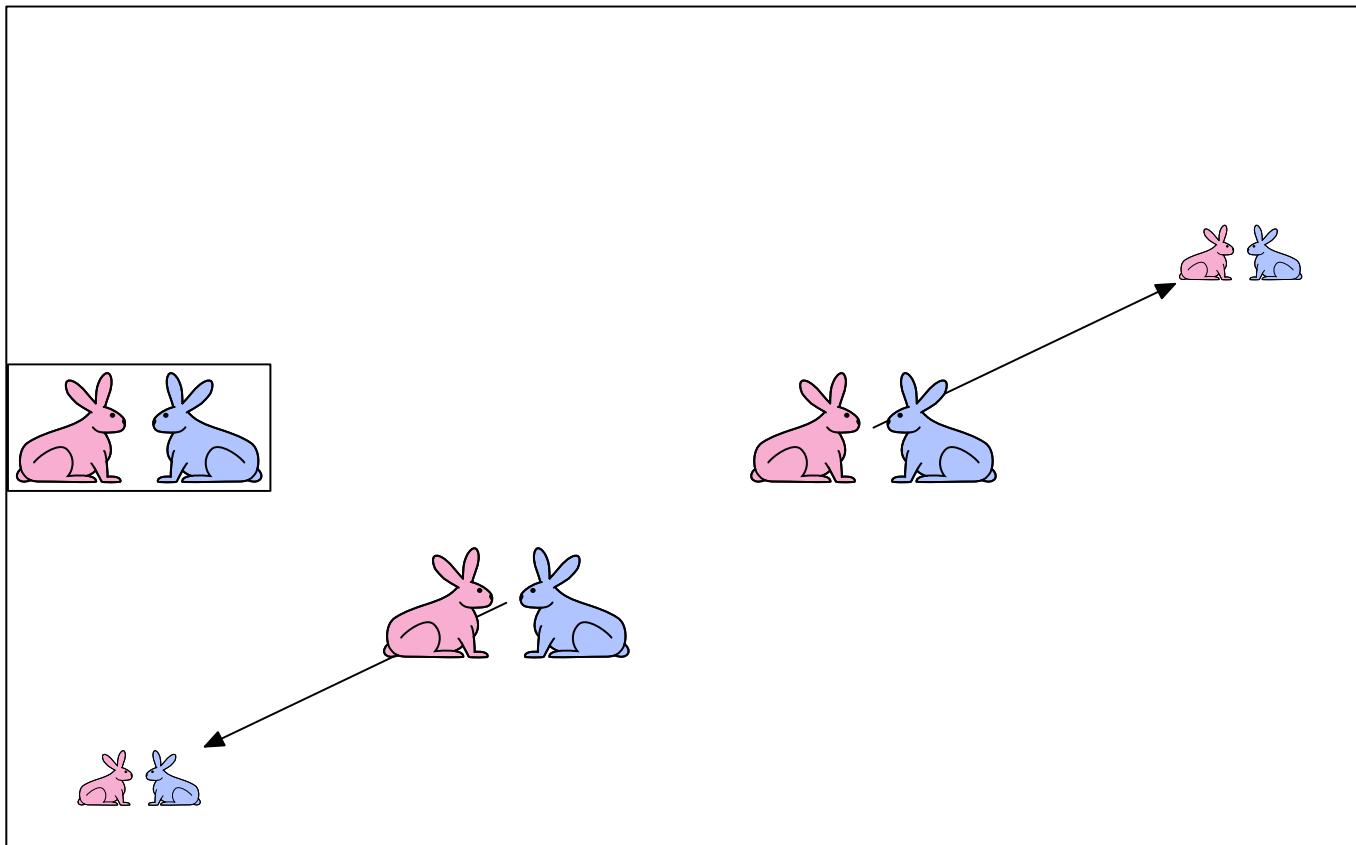
Demo courtesy of Prof. Denny Freeman and Adam Hartz



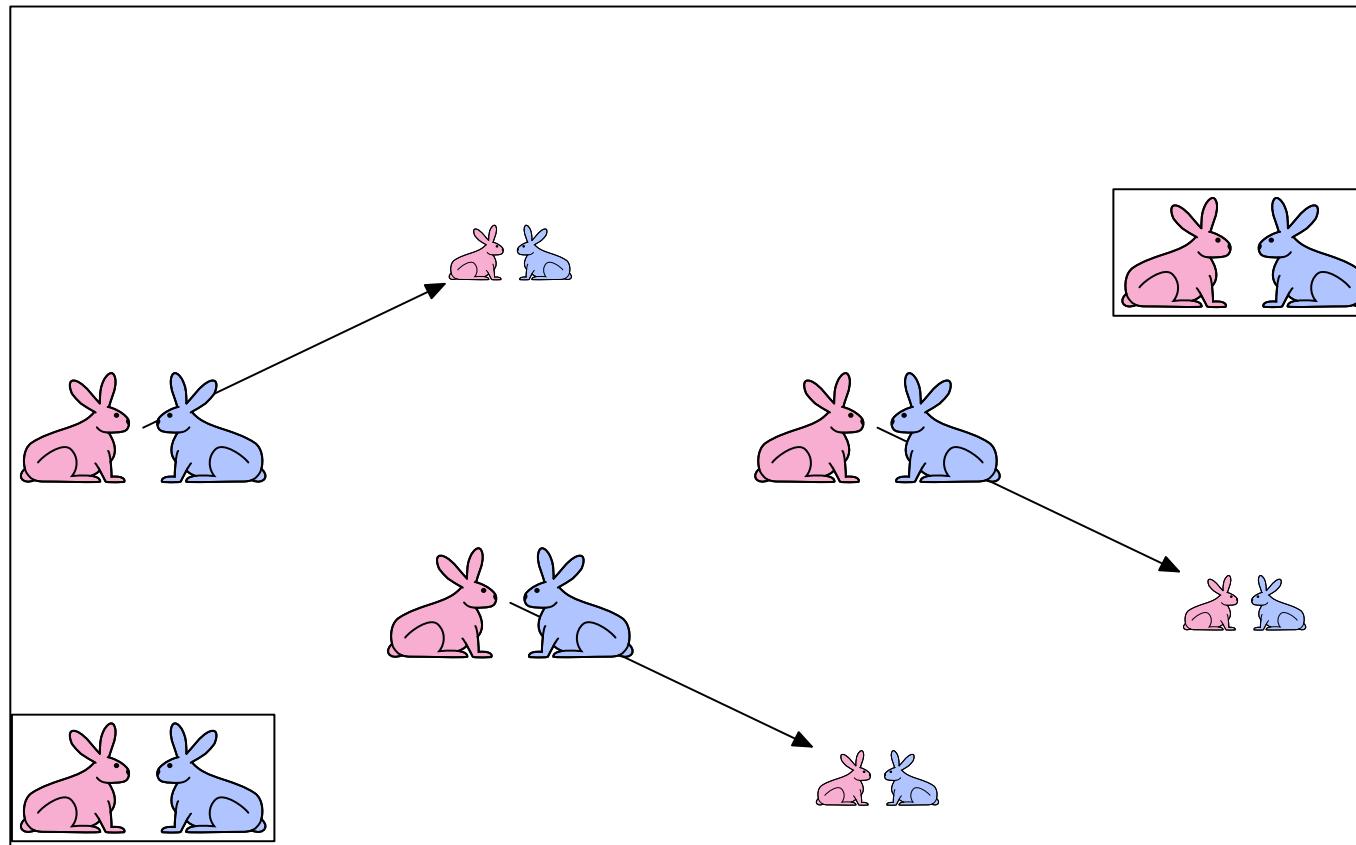
Demo courtesy of Prof. Denny Freeman and Adam Hartz



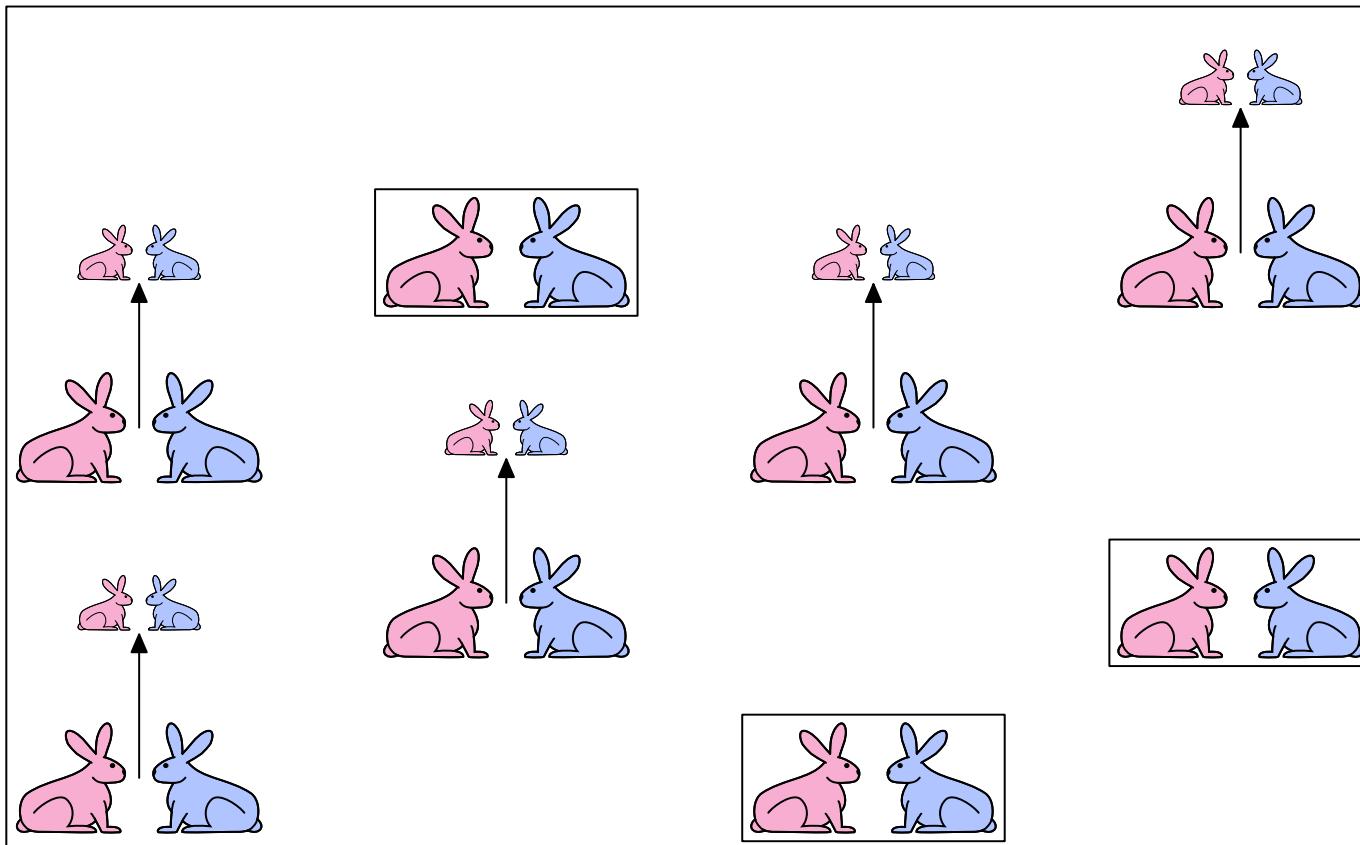
Demo courtesy of Prof. Denny Freeman and Adam Hartz



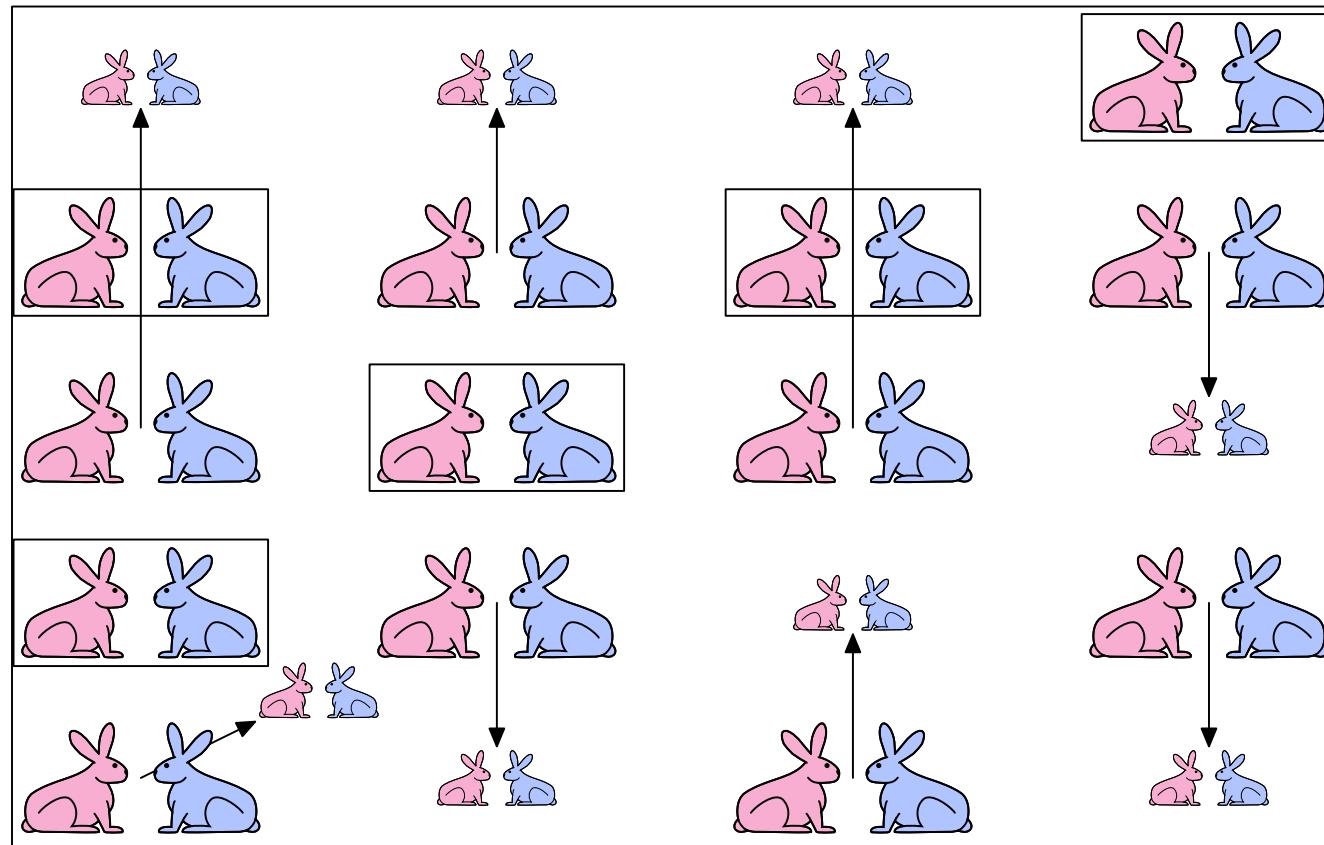
Demo courtesy of Prof. Denny Freeman and Adam Hartz

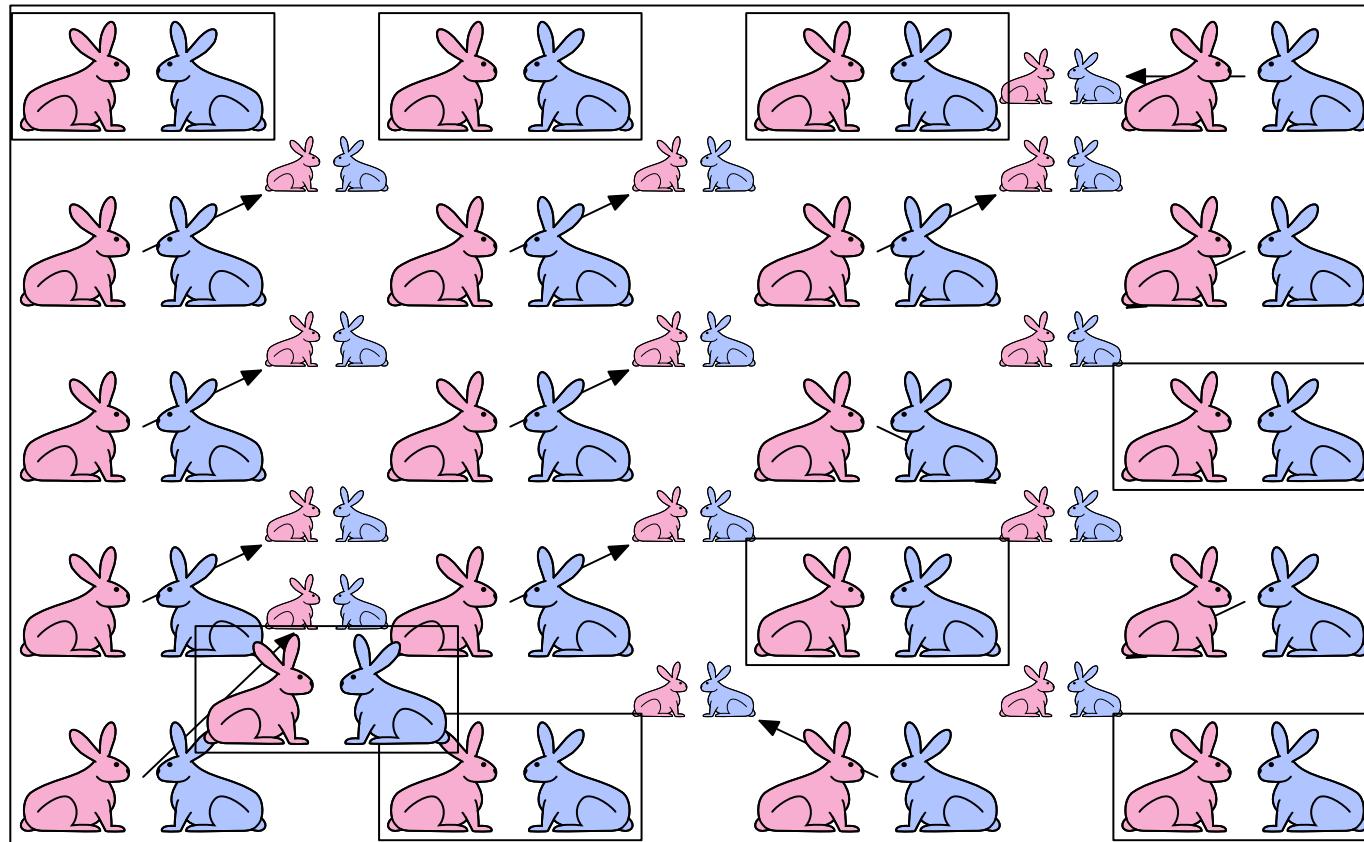


Demo courtesy of Prof. Denny Freeman and Adam Hartz

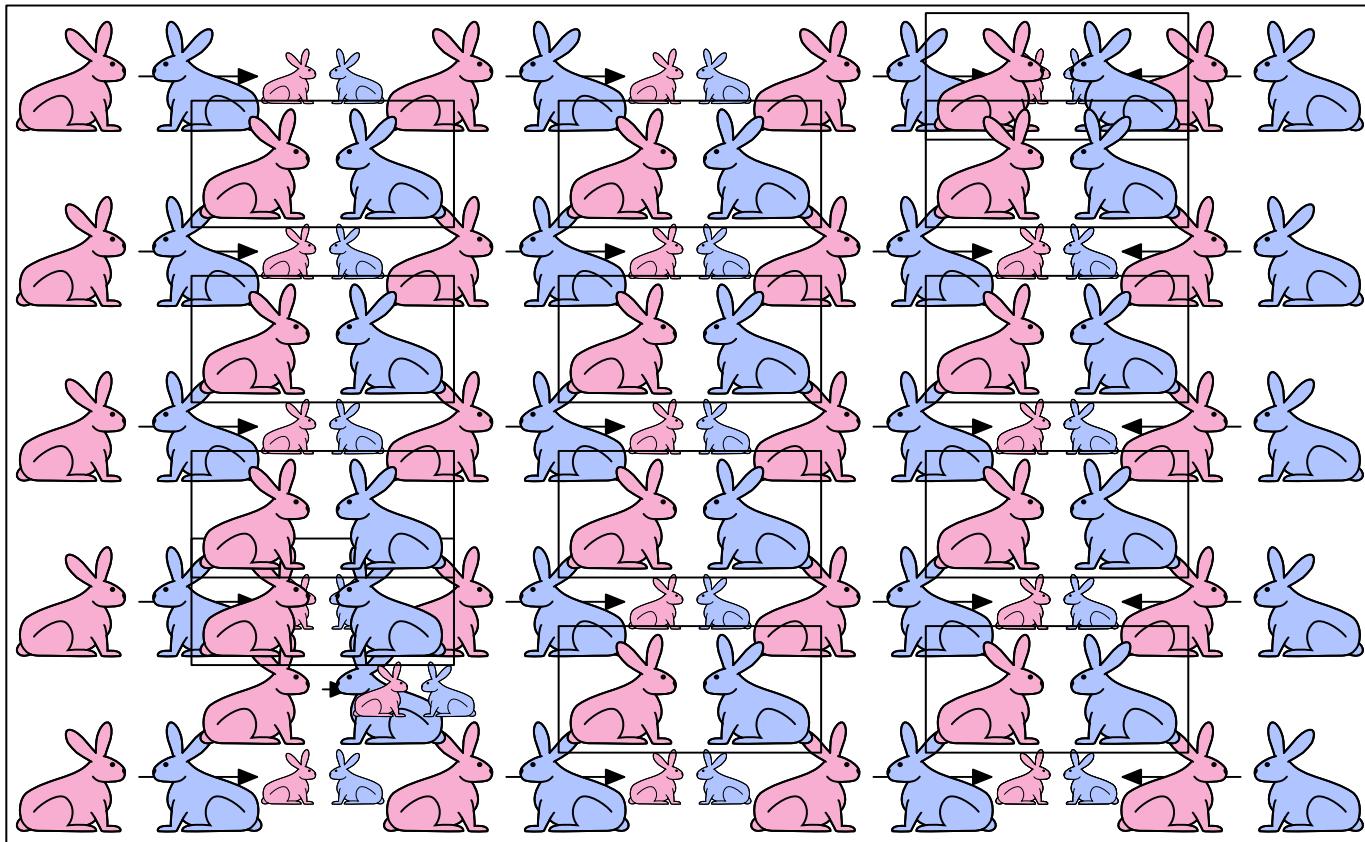


Demo courtesy of Prof. Denny Freeman and Adam Hartz





Demo courtesy of Prof. Denny Freeman and Adam Hartz



Demo courtesy of Prof. Denny Freeman and Adam Hartz

FIBONACCI

After one month (call it 0) – 1 female

After second month – still 1 female (now pregnant)

After third month – two females, one pregnant, one not

In general, females(n) = females($n-1$) + females($n-2$)

- Every female alive at month $n-2$ will produce one female in month n ;
- These can be added those alive in month $n-1$ to get total alive in month n

Month	Females
0	1

FIBONACCI

- Base cases:
 - Females(0) = 1
 - Females(1) = 1
- Recursive case
 - $\text{Females}(n) = \text{Females}(n-1) + \text{Females}(n-2)$

FIBONACCI

```
def fib(x):  
  
    """assumes x an int >= 0  
        returns Fibonacci of x"""  
  
    if x == 0 or x == 1:  
  
        return 1  
  
    else:  
  
        return fib(x-1) + fib(x-2)
```

RECURSION ON NON-NUMERICS

- how to check if a string of characters is a palindrome, i.e., reads the same forwards and backwards
 - “Able was I, ere I saw Elba” – attributed to Napoleon
 - “Are we not drawn onward, we few, drawn onward to new era?” – attributed to Anne Michaels



Image courtesy of [wikipedia](#), in the public domain.



By Larth_Rasnal (Own work) [GFDL (<https://www.gnu.org/licenses/fdl-1.3.en.html>) or CC BY 3.0 (<https://creativecommons.org/licenses/by/3.0/>)], via Wikimedia Commons.

SOLVING RECURSIVELY?

- First, convert the string to just characters, by stripping out punctuation, and converting upper case to lower case
- Then
 - Base case: a string of length 0 or 1 is a palindrome
 - Recursive case:
 - If first character matches last character, then is a palindrome if middle section is a palindrome

EXAMPLE

- ‘Able was I, ere I saw Elba’ → ‘ablewasiereisawleba’
- `isPalindrome('ablewasiereisawleba')`
is same as
 - ‘a’ == ‘a’ and
`isPalindrome('blewasiereisawleb')`

```
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))
```

DIVIDE AND CONQUER

- an example of a “divide and conquer” algorithm
- solve a hard problem by breaking it into a set of sub-problems such that:
 - sub-problems are easier to solve than the original
 - solutions of the sub-problems can be combined to solve the original

DICTIONARIES

HOW TO STORE STUDENT INFO

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']  
grade = ['B', 'A+', 'A', 'A']  
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):  
    i = name_list.index(student)  
    grade = grade_list[i]  
    course = course_list[i]  
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

A BETTER AND CLEANER WAY – A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
...	...

index
element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom
index by
label
element

A PYTHON DICTIONARY

- store pairs of data
 - key
 - value

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

my_dict = `{}` *empty dictionary*

grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}

custom index by label

element

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up the key**
- **returns the value** associated with the key
- if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}  
grades['John']      → evaluates to 'A+'  
grades['Sylvan']    → gives a KeyError
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

'John' in grades	→ returns True
'Daniel' in grades	→ returns False

- **delete** entry

```
del(grades['Ana'])
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}
```

- get an **iterable that acts like a tuple of all keys** *no guaranteed order*
`grades.keys()` → returns ['Denise', 'Katy', 'John', 'Ana']
- get an **iterable that acts like a tuple of all values**
`grades.values()` → returns ['A', 'A', 'A+', 'B']

DICTIONARY KEYS and VALUES

- values
 - any type (**immutable and mutable**)
 - can be **duplicates**
 - dictionary values can be lists, even other dictionaries!
- keys
 - must be **unique**
 - **immutable** type (int, float, string, tuple, bool)
 - actually need an object that is **hashable**, but think of as immutable as all immutable types are hashable
 - careful with float type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

list vs dict

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**
- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

EXAMPLE: 3 FUNCTIONS TO ANALYZE SONG LYRICS

- 1) create a **frequency dictionary** mapping str : int
- 2) find **word that occurs the most** and how many times
 - use a list, in case there is more than one word
 - return a tuple (list, int) for (words_list, highest_freq)
- 3) find the **words that occur at least X times**
 - let user choose “at least X times”, so allow as parameter
 - return a list of tuples, each tuple is a (list, int) containing the list of words ordered by their frequency
 - IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY

```
def lyrics_to_frequencies(lyrics):  
    myDict = {}  
    for word in lyrics:  
        if word in myDict:  
            myDict[word] += 1  
        else:  
            myDict[word] = 1  
    return myDict
```

can iterate over list
in dictionary
update value
associated with key

USING THE DICTIONARY

```
def most_common_words(freqs):
    values = freqs.values()
    best = max(values)
    words = []
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

Annotations:

- values = freqs.values():
this is an iterable, so can apply built-in function
- for k in freqs:
can iterate over keys in dictionary

LEVERAGING DICTIONARY PROPERTIES

```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])
        else:
            done = True
    return result

print(words_often(beatles, 5))
```

can directly mutate
dictionary; makes it
easier to iterate

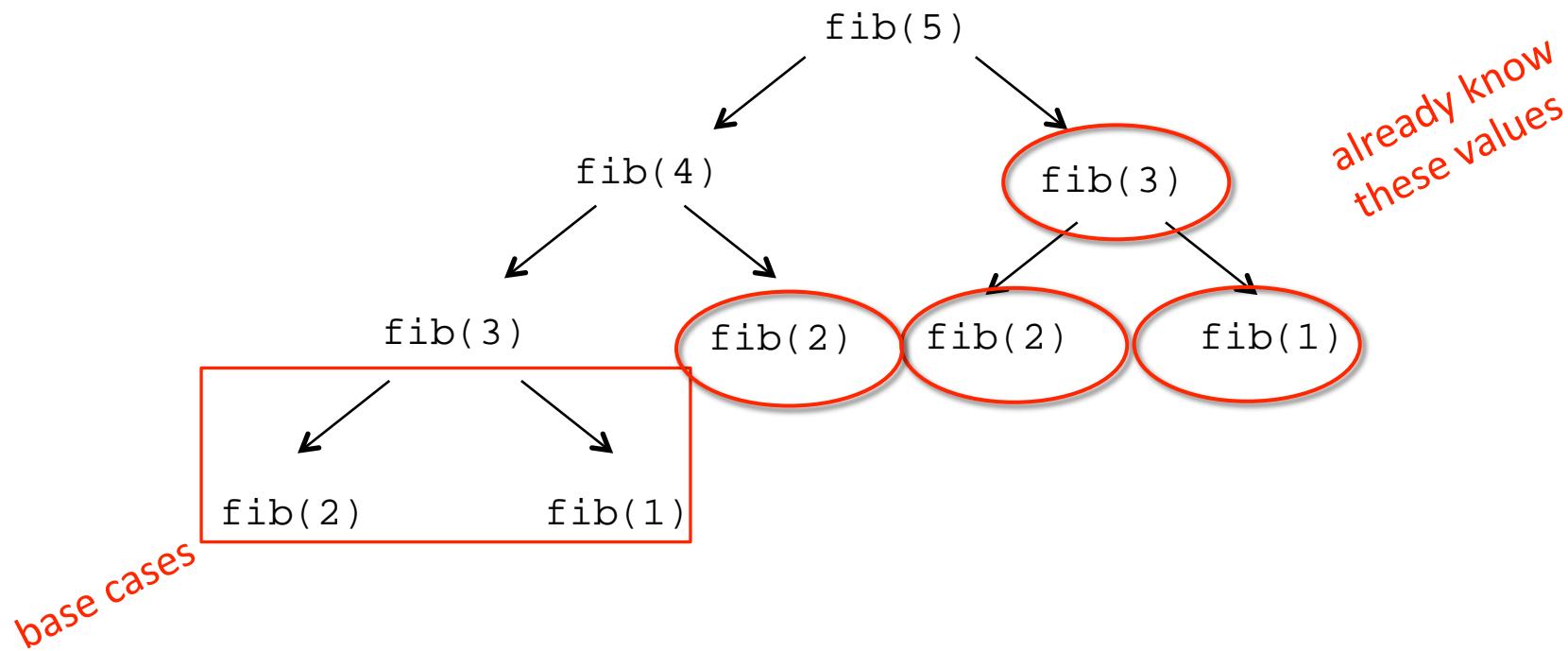
FIBONACCI RECURSIVE CODE

```
def fib(n):  
    if n == 1:  
        return 1  
    elif n == 2:  
        return 2  
    else:  
        return fib(n-1) + fib(n-2)
```

- two base cases
- calls itself twice
- this code is inefficient

INEFFICIENT FIBONACCI

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$



- **recalculating** the same values many times!
- could keep **track** of already calculated values

FIBONACCI WITH A DICTIONARY

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
    return ans

d = {1:1, 2:2}
print(fib_efficient(6, d))
```

Method sometimes
called "memoization"

Initialize dictionary
with base cases

- do a **lookup first** in case already calculated the value
- **modify dictionary** as progress through function calls

EFFICIENCY GAINS

- Calling `fib(34)` results in 11,405,773 recursive calls to the procedure
- Calling `fib_efficient(34)` results in 65 recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works for procedures without side effects (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.



TESTING, DEBUGGING, EXCEPTIONS, ASSERTIONS

(download slides and .py files and follow along!)

6.0001 LECTURE 7



WE AIM FOR HIGH QUALITY – AN ANALOGY WITH SOUP

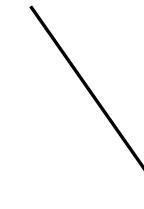
You are making soup but bugs keep falling in from the ceiling. What do you do?

- check soup for bugs
 - testing
- keep lid closed
 - defensive programming
- clean kitchen
 - eliminate source of bugs



DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)



TESTING/VALIDATION

- **Compare** input/output pairs to specification
- “It’s not working!”
- “How can I break my program?”

DEBUGGING

- **Study events** leading up to an error
 - “Why is it not working?”
 - “How can I fix my program?”

SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

- from the **start**, design code to ease this part
- break program up into **modules** that can be tested and debugged individually
- **document constraints** on modules
 - what do you expect the input to be?
 - what do you expect the output to be?
- **document assumptions** behind code design

WHEN ARE YOU READY TO TEST?

- ensure **code runs**
 - remove syntax errors
 - remove static semantic errors
 - Python interpreter can usually find these for you
- have a **set of expected results**
 - an input set
 - for each input, the expected output

CLASSES OF TESTS

- **Unit testing**

- validate each piece of program
- **testing each function** separately

- **Regression testing**

- add test for bugs as you find them
- **catch reintroduced** errors that were previously fixed

- **Integration testing**

- does **overall program** work?
- tend to rush to do this

TESTING APPROACHES

- **intuition** about natural boundaries to the problem

```
def is_bigger(x, y):  
    """ Assumes x and y are ints  
    Returns True if y is less than x, else False """
```

- can you come up with some natural partitions?

- if no natural partitions, might do **random testing**

- probability that code is correct increases with more tests
- better options below

- **black box testing**

- explore paths through specification

- **glass box testing**

- explore paths through code

BLACK BOX TESTING

```
def sqrt(x, eps):  
    """ Assumes x, eps floats, x >= 0, eps > 0  
    Returns res such that x-eps <= res*res <= x+eps """
```

- designed **without looking** at the code
- can be done by someone other than the implementer to avoid some implementer **biases**
- testing can be **reused** if implementation changes
- **paths** through specification
 - build test cases in different natural space partitions
 - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

BLACK BOX TESTING

```
def sqrt(x, eps):  
    """ Assumes x, eps floats, x >= 0, eps > 0  
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

GLASS BOX TESTING

- **use code** directly to guide design of test cases
 - called **path-complete** if every potential path through code is tested at least once
 - what are some **drawbacks** of this type of testing?
 - can go through loops arbitrarily many times
 - missing paths
 - guidelines
 - branches
 - for loops
 - while loops
-
- The diagram shows red annotations for loop guidelines. A large red arrow points from the word 'guidelines' to the list items. Red arrows also point from each list item to its corresponding annotation:
 - 'branches' points to 'exercise all parts of a conditional'
 - 'for loops' points to 'loop not entered body of loop executed exactly once'
 - 'while loops' points to 'body of loop executed more than once same as for loops, cases that catch all ways to exit loop'

GLASS BOX TESTING

```
def abs(x):  
    """ Assumes x is an int  
    Returns x if x>=0 and -x otherwise """  
    if x < -1:  
        return -x  
    else:  
        return x
```

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but $\text{abs}(-1)$ incorrectly returns -1
- should still test boundary cases

DEBUGGING

- steep learning curve
- goal is to have a bug-free program
- tools
 - **built in** to IDLE and Anaconda
 - **Python Tutor**
 - **print** statement
 - use your brain, be **systematic** in your hunt

PRINT STATEMENTS

- good way to **test hypothesis**
- when to print
 - enter function
 - parameters
 - function results
- use **bisection method**
 - put print halfway in code
 - decide where bug may be depending on values

DEBUGGING STEPS

- **study** program code
 - don't ask what is wrong
 - ask how did I get the unexpected result
 - is it part of a family?

- **scientific method**
 - study available data
 - form hypothesis
 - repeatable experiments
 - pick simplest input to test with

ERROR MESSAGES – EASY

- trying to access beyond the limits of a list

test = [1, 2, 3] then test[4] → IndexError

- trying to convert an inappropriate type

int(test) → TypeError

- referencing a non-existent variable

a → NameError

- mixing data types without appropriate coercion

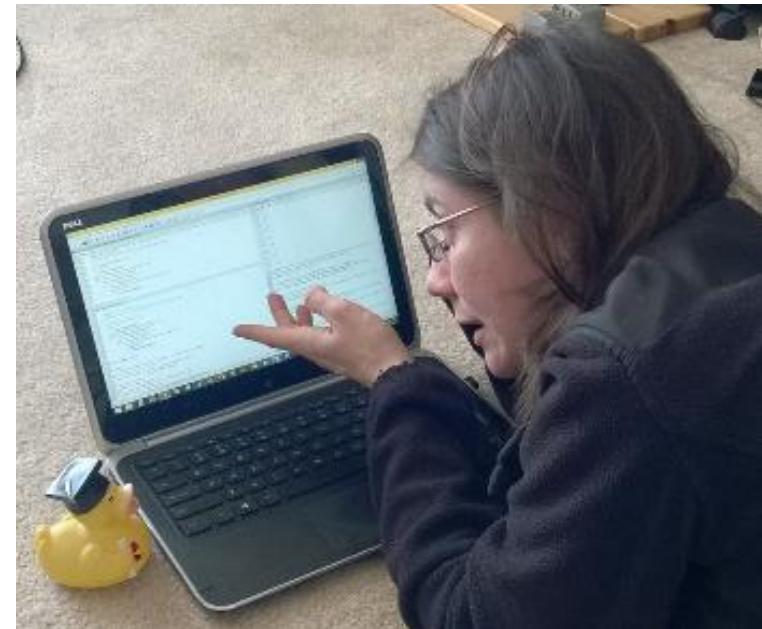
'3' / 4 → TypeError

- forgetting to close parenthesis, quotation, etc.

a = len([1, 2, 3]
print(a) → SyntaxError

LOGIC ERRORS - HARD

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
 - someone else
 - a rubber ducky



DON'T

- Write entire program
- Test entire program
- Debug entire program



DO

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration testing ***

- Change code
- Remember where bug was
- Test code
- Forget where bug was or what change you made
- Panic



- Backup code
- Change code
- Write down potential bug in a comment
- Test code
- Compare new version with old version

EXCEPTIONS AND ASSERTIONS

- what happens when procedure execution hits an **unexpected condition?**
- get an **exception**... to what was expected
 - trying to access beyond list limits

test = [1, 7, 4]

test[4]

→ IndexError

- trying to convert an inappropriate type

int(test)

→ TypeError

- referencing a non-existing variable

a

→ NameError

- mixing data types without coercion

'a' / 4

→ TypeError

OTHER TYPES OF EXCEPTIONS

- already seen common error types:
 - `SyntaxError`: Python can't parse program
 - `NameError`: local or global name not found
 - `AttributeError`: attribute reference fails
 - `TypeError`: operand doesn't have correct type
 - `ValueError`: operand type okay, but value is illegal
 - `IOError`: IO system reports malfunction (e.g. file not found)

DEALING WITH EXCEPTIONS

- Python code can provide **handlers** for exceptions

```
try:  
    a = int(input("Tell me one number:"))  
    b = int(input("Tell me another number:"))  
    print(a/b)  
except:  
    print("Bug in user input.")
```

- exceptions **raised** by any statement in body of **try** are **handled** by the **except** statement and execution continues with the body of the except statement

HANDLING SPECIFIC EXCEPTIONS

- have **separate except clauses** to deal with a particular type of exception

try:

```
a = int(input("Tell me one number: "))
b = int(input("Tell me another number: "))
print("a/b = ", a/b)
print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute
if these errors
come up

for all
other
errors

OTHER EXCEPTIONS

- `else`:
 - body of this is executed when execution of associated `try` body **completes with no exceptions**
- `finally`:
 - body of this is **always executed** after `try`, `else` and `except` clauses, even if they raised another error or executed a `break`, `continue` or `return`
 - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

WHAT TO DO WITH EXCEPTIONS?

- what to do when encounter an error?
- **fail silently:**
 - substitute default values or just continue
 - bad idea! user gets no warning
- return an “**error**” value
 - what value to choose?
 - complicates code having to check for a special value
- stop execution, **signal error** condition
 - in Python: **raise an exception**
`raise Exception ("descriptive string")`

EXCEPTIONS AS CONTROL FLOW

- don't return special values when an error occurred and then check whether 'error value' was returned
- instead, **raise an exception** when unable to produce a result consistent with function's specification

```
raise <exceptionName>(<arguments>)
```

```
raise ValueError("something is wrong")
```

keyword

name of error
you want to raise

optional, but typically a
string with a message

EXAMPLE: RAISING AN EXCEPTION

```
def get_ratios(L1, L2):  
    """ Assumes: L1 and L2 are lists of equal length of numbers  
        Returns: a list containing L1[i]/L2[i] """  
    ratios = []  
    for index in range(len(L1)):  
        try:  
            ratios.append(L1[index]/L2[index])  
        except ZeroDivisionError:  
            ratios.append(float('nan')) #nan = not a number  
        except:  
            raise ValueError('get_ratios called with bad arg')  
    return ratios
```

manage flow of
program by raising
own error

EXAMPLE OF EXCEPTIONS

- assume we are **given a class list** for a subject: each entry is a list of two parts
 - a list of first and last name for a student
 - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],  
                [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],  
[['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.66667]]
```

EXAMPLE CODE

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],  
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```
def get_stats(class_list):  
    new_stats = []  
    for elt in class_list:  
        new_stats.append([elt[0], elt[1], avg(elt[1])])  
    return new_stats  
  
def avg(grades):  
    return sum(grades)/len(grades)
```

ERROR IF NO GRADE FOR A STUDENT

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],  
                [['bruce', 'wayne'], [10.0, 8.0, 74.0]],  
                [['captain', 'america'], [8.0, 10.0, 96.0]],  
                [['deadpool'], []]]
```

- **get** ZeroDivisionError: float division by zero because try to

```
return sum(grades) / len(grades)
```

length is 0

OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- decide to **notify** that something went wrong with a msg

```
def avg(grades):  
    try:  
        return sum(grades)/len(grades)  
    except ZeroDivisionError:  
        print('warning: no grades data')
```

- running on some test data gives

```
warning: no grades data
```

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],  
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],  
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],  
[['deadpool'], [], None]]
```

flagged the error
because avg did
not return anything
in the except

OPTION 2: CHANGE THE POLICY

- decide that a student with no grades gets a **zero**

```
def avg(grades):  
    try:  
        return sum(grades)/len(grades)  
    except ZeroDivisionError:  
        print('warning: no grades data')  
        return 0.0
```

- running on some test data gives

```
warning: no grades data
```

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],  
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],  
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],  
[['deadpool'], [], 0.0]]
```

still flag the error

now avg returns 0

ASSERTIONS

- want to be sure that **assumptions** on state of computation are as expected
- use an **assert statement** to raise an `AssertionError` exception if assumptions not met
- an example of good **defensive programming**

EXAMPLE

```
def avg(grades) :  
    assert len(grades) != 0, 'no grades data'  
    return sum(grades)/len(grades)
```

function ends
immediately if
assertion not met

- raises an `AssertionError` if it is given an empty list for `grades`
- otherwise runs ok

ASSERTIONS AS DEFENSIVE PROGRAMMING

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that **execution halts** whenever an expected condition is not met
- typically used to **check inputs** to functions, but can be used anywhere
- can be used to **check outputs** of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

WHERE TO USE ASSERTIONS?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a **supplement** to testing
- raise **exceptions** if users supplies **bad data input**
- use **assertions** to
 - check **types** of arguments or values
 - check that **invariants** on data structures are met
 - check **constraints** on return values
 - check for **violations** of constraints on procedure (e.g. no duplicates in a list)

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

OBJECT ORIENTED PROGRAMMING

(download slides and .py files · follow along!)

6.0001 LECTURE 8

OBJECTS

- Python supports many different kinds of data

1234 3.14159 "Hello" [1, 5, 7, 11, 13]

{ "CA": "California", "MA": "Massachusetts" }

- each is an **object**, and every object has:
 - a **type**
 - an internal **data representation** (primitive or composite)
 - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
 - 1234 is an instance of an `int`
 - "hello" is an instance of a `string`

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
 - explicitly using `del` or just “forget” about them
 - python system will reclaim destroyed or inaccessible objects – called “garbage collection”

WHAT ARE OBJECTS?

- objects are **a data abstraction** that captures...

(1) an **internal representation**

- through data attributes

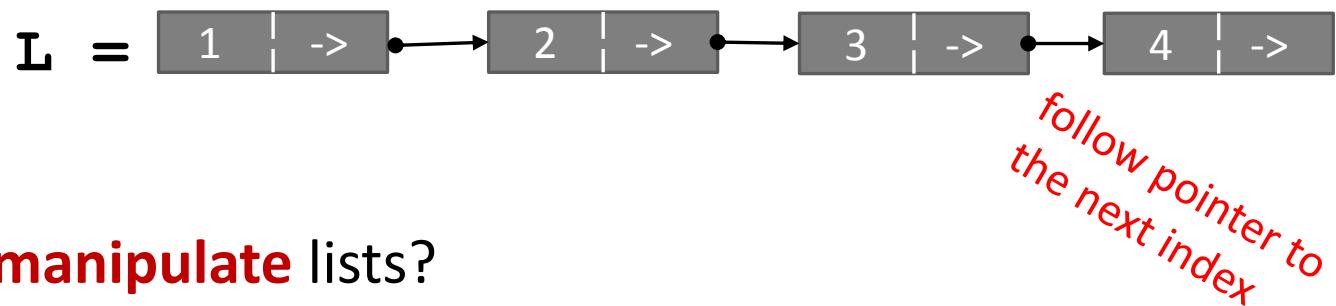
(2) an **interface** for interacting with object

- through methods
(aka procedures/functions)
- defines behaviors but hides implementation

EXAMPLE:

[1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?

- L[i], L[i:j], +
- len(), min(), max(), del(L[i])
- L.append(), L.extend(), L.count(), L.index(),
L.insert(), L.pop(), L.remove(), L.reverse(), L.sort()

- internal representation should be private

- correct behavior may be compromised if you manipulate internal representation directly

ADVANTAGES OF OOP

- **bundle data into packages** together with procedures that work on them through well-defined interfaces
- **divide-and-conquer** development
 - implement and test behavior of each class separately
 - increased modularity reduces complexity
- classes make it easy to **reuse** code
 - many Python modules define new classes
 - each class has a separate environment (no collision on function names)
 - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- make a distinction between **creating a class** and **using an instance** of the class
- **creating** the class involves
 - defining the class name
 - defining class attributes
 - *for example, someone wrote code to implement a list class*
- **using** the class involves
 - creating new **instances** of objects
 - doing operations on the instances
 - *for example, `L=[1, 2]` and `len(L)`*

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type

```
class Coordinate(object) :
```

#define attributes here

- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance next lecture)
 - `Coordinate` is a subclass of `object`
 - `object` is a superclass of `Coordinate`

WHAT ARE ATTRIBUTES?

- data and procedures that “**belong**” to the class
- **data attributes**
 - think of data as other objects that make up the class
 - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
 - think of methods as functions that only work with this class
 - how to interact with the object
 - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

```
class Coordinate(object):
```

```
    def __init__(self, x, y):
```

special method to
create an instance
`__` is double
underscore

```
        self.x = x
```

```
        self.y = y
```

two data attributes for
every Coordinate object

what data initializes a
Coordinate object

parameter to
refer to an
instance of the
class

ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3, 4)  
origin = Coordinate(0, 0)  
print(c.x)  
print(origin.x)
```

use the dot to
access an attribute
of instance c

create a new object
of type
Coordinate and
pass in 3 and 4 to
the __init__

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
 - convention is to use **self** as the name of the first argument of all methods
- the “.” **operator** is used to access any attribute
 - a data attribute of an object
 - a method of an object

DEFINE A METHOD FOR THE Coordinate CLASS

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def distance(self, other):  
        x_diff_sq = (self.x - other.x) ** 2  
        y_diff_sq = (self.y - other.y) ** 2  
        return (x_diff_sq + y_diff_sq) ** 0.5
```

use it to refer to any instance
another parameter to method
dot notation to access data

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):  
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3, 4)  
zero = Coordinate(0, 0)  
print(c.distance(zero))
```

object to call
method on name of
 method parameters not
 including self
 (self is
 implied to be c)

- equivalent to

```
c = Coordinate(3, 4)  
zero = Coordinate(0, 0)  
print(Coordinate.distance(c, zero))
```

name of
class name of
 method
 parameters, including an
 object to call the method
 on, representing self

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3, 4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **__str__ method** for a class
- Python calls the **__str__** method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3, 4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def distance(self, other):  
        x_diff_sq = (self.x-other.x)**2  
        y_diff_sq = (self.y-other.y)**2  
        return (x_diff_sq + y_diff_sq)**0.5  
    def __str__(self):  
        return "<" + str(self.x) + ", " + str(self.y) + ">"
```

name of
special
method

must return
a string

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- can ask for the type of an object instance

```
>>> c = Coordinate(3, 4)
>>> print(c)
<3, 4>
>>> print(type(c))
<class '__main__.Coordinate'>
```

return of the `str` method
the type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class '__main__.Coordinate'>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class
a Coordinate class is a type of object

- use `isinstance()` to check if an object is a Coordinate

```
>>> print(isinstance(c, Coordinate))
True
```

SPECIAL OPERATORS

- +, -, ==, <, >, len(), print, and many others

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

- like print, can override these to work with your class
- define them with double underscores before/after

<code>__add__(self, other)</code>	→	<code>self + other</code>
<code>__sub__(self, other)</code>	→	<code>self - other</code>
<code>__eq__(self, other)</code>	→	<code>self == other</code>
<code>__lt__(self, other)</code>	→	<code>self < other</code>
<code>__len__(self)</code>	→	<code>len(self)</code>
<code>__str__(self)</code>	→	<code>print self</code>
... and others		

EXAMPLE: FRACTIONS

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers
 - numerator
 - denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with Fraction objects
 - add, subtract
 - print representation, convert to a float
 - invert the fraction
- the code for this is in the handout, check it out!

THE POWER OF OOP

- **bundle together objects** that share
 - common attributes and
 - procedures that operate on those attributes
- use **abstraction** to make a distinction between how to implement an object vs how to use the object
- build **layers** of object abstractions that inherit behaviors from other classes of objects
- create our **own classes of objects** on top of Python's basic classes

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

UNDERSTANDING PROGRAM EFFICIENCY: 1

(download slides and .py files and follow along!)

6.0001 LECTURE 10

Today

- Measuring orders of growth of algorithms
- Big “Oh” notation
- Complexity classes

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

- computers are fast and getting faster – so maybe efficient programs don't matter?
 - but data sets can be very large (e.g., in 2014, Google served 30,000,000,000,000 pages, covering 100,000,000 GB – how long to search brute force?)
 - thus, simple solutions may simply not scale with size in acceptable manner
- how can we decide which option for program is most efficient?

- separate **time and space efficiency** of a program
- tradeoff between them:
 - can sometimes pre-compute results are stored; then use “lookup” to retrieve (e.g., memoization for Fibonacci)
 - will focus on time efficiency

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

Challenges in understanding efficiency of solution to a computational problem:

- a program can be **implemented in many different ways**
- you can solve a problem using only a handful of different **algorithms**
- would like to separate choices of implementation from choices of more abstract algorithm

HOW TO EVALUATE EFFICIENCY OF PROGRAMS

- measure with a **timer**
- **count** the operations
- abstract notion of **order of growth**

will argue that this is the most appropriate way of assessing the impact of choices of algorithm in solving a problem; and in measuring the inherent difficulty in solving a problem

TIMING A PROGRAM

- use time module

- recall that
importing means to
bring in that class
into your own file

```
import time  
  
def c_to_f(c):  
    return c*9/5 + 32  
  
■ start clock → t0 = time.clock()  
■ call function → c_to_f(100000)  
■ stop clock → t1 = time.clock() - t0  
Print("t =", t, ":", t1, "s,")
```

TIMING PROGRAMS IS INCONSISTENT

- GOAL: to evaluate different algorithms
- running time **varies between algorithms** 
- running time **varies between implementations** 
- running time **varies between computers** 
- running time is **not predictable** based on small inputs 
- time varies for different inputs but cannot really express a relationship between inputs and time 

COUNTING OPERATIONS

- assume these steps take **constant time**:
 - mathematical operations
 - comparisons
 - assignments
 - accessing objects in memory
- then count the number of operations executed as function of size of input

```
def c_to_f(c):  
    return c*9.0/5 + 32  
  
def mysum(x):  
    total = 0  
    for i in range(x+1):  
        total += i  
    return total
```

1 op loop x times 3 ops 1 op
 2 ops

mysum → 1+3x ops

COUNTING OPERATIONS IS BETTER, BUT STILL...

- GOAL: to evaluate different algorithms
 - count **depends on algorithm** 
 - count **depends on implementations** 
 - count **independent of computers** 
 - no clear definition of **which operations** to count 
-
- count varies for different inputs and can come up with a relationship between inputs and the count 

STILL NEED A BETTER WAY

- timing and counting **evaluate implementations**
- timing **evaluates machines**

- want to **evaluate algorithm**
- want to **evaluate scalability**
- want to **evaluate in terms of input size**

STILL NEED A BETTER WAY

- Going to focus on idea of counting operations in an algorithm, but not worry about small variations in implementation (e.g., whether we take 3 or 4 primitive operations to execute the steps of a loop)
- Going to focus on how algorithm performs when size of problem gets arbitrarily large
- Want to relate time needed to complete a computation, measured this way, against the size of the input to the problem
- Need to decide what to measure, given that actual number of steps may depend on specifics of trial

NEED TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

- want to express **efficiency in terms of size of input**, so need to decide what your input is
- could be an **integer**
 - mysum (x)
- could be **length of list**
 - list_sum (L)
- **you decide** when multiple parameters to a function
 - search_for_elmt (L, e)

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- a function that searches for an element in a list

```
def search_for_elmt(L, e):  
    for i in L:  
        if i == e:  
            return True  
    return False
```

- when e is **first element** in the list → BEST CASE
- when e is **not in list** → WORST CASE
- when **look through about half** of the elements in list → AVERAGE CASE
- want to measure this behavior in a general way

BEST, AVERAGE, WORST CASES

- suppose you are given a list L of some length $\text{len}(L)$
 - **best case:** minimum running time over all possible inputs of a given size, $\text{len}(L)$
 - constant for `search_for_elmt`
 - first element in any list
 - **average case:** average running time over all possible inputs of a given size, $\text{len}(L)$
 - practical measure
 - **worst case:** maximum running time over all possible inputs of a given size, $\text{len}(L)$
 - linear in length of list for `search_for_elmt`
 - must search entire list and not find it
- generally will
focus on this case

ORDERS OF GROWTH

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth – as tight as possible
- do not need to be precise: “**order of**” not “**exact**” growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

MEASURING ORDER OF GROWTH: BIG OH NOTATION

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth
- **Big Oh or $O()$** is used to describe worst case
 - worst case occurs often and is the bottleneck when a program runs
 - express rate of growth of program relative to the input size
 - evaluate algorithm **NOT** machine or implementation

EXACT STEPS vs O()

```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

- computes factorial
- number of steps: $1 + 5n + 1$
- worst case asymptotic complexity: $O(n)$
 - ignore additive constants
 - ignore multiplicative constants

WHAT DOES $O(N)$ MEASURE?

- Interested in describing how amount of time needed grows as size of (input to) problem grows
- Thus, given an expression for the number of operations needed to compute an algorithm, want to know asymptotic behavior as size of problem gets large
- Hence, will focus on term that grows most rapidly in a sum of terms
- And will ignore multiplicative constants, since want to know how rapidly time required increases as increase size of input

SIMPLIFICATION EXAMPLES

- drop constants and multiplicative factors
- focus on **dominant terms**

$O(n^2)$: $n^2 + 2n + 2$

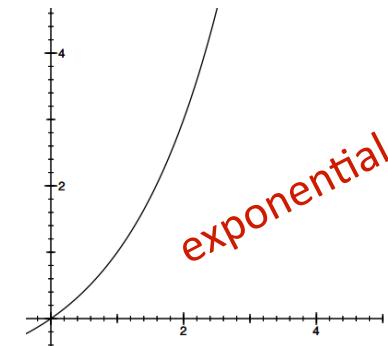
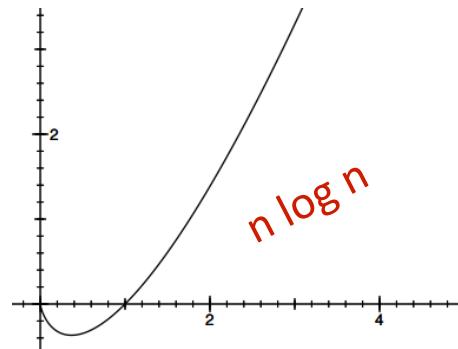
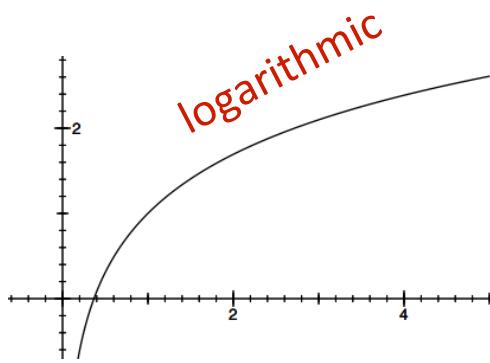
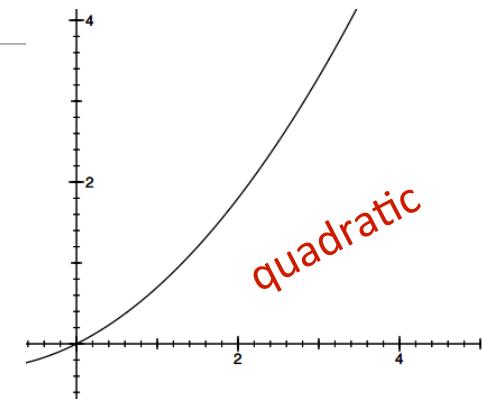
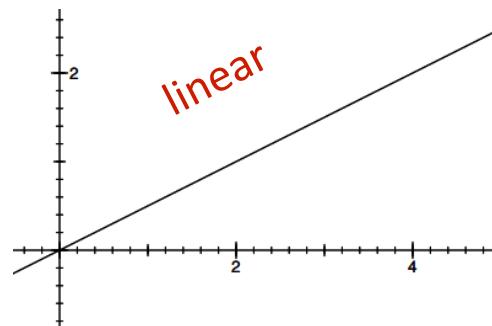
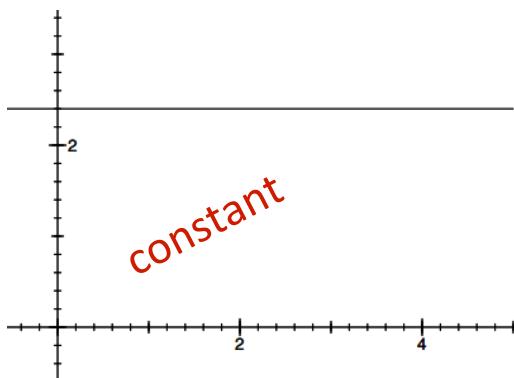
$O(n^2)$: $n^2 + 100000n + 3^{1000}$

$O(n)$: $\log(n) + n + 4$

$O(n \log n)$: $0.0001 * n * \log(n) + 300n$

$O(3^n)$: $2n^{30} + 3^n$

TYPES OF ORDERS OF GROWTH



ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **combine** complexity classes
 - analyze statements inside functions
 - apply some rules, focus on dominant term

Law of Addition for $O()$:

- used with **sequential** statements
- $O(f(n)) + O(g(n))$ is $O(f(n) + g(n))$
- for example,

```
for i in range(n):  
    print('a')  
for j in range(n*n):  
    print('b')
```

$O(n)$

$O(n^2)$

$O(n) + O(n^2)$



is $O(n) + O(n^2) = O(n+n^2) = O(n^2)$ because of dominant term

ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **combine** complexity classes
 - analyze statements inside functions
 - apply some rules, focus on dominant term

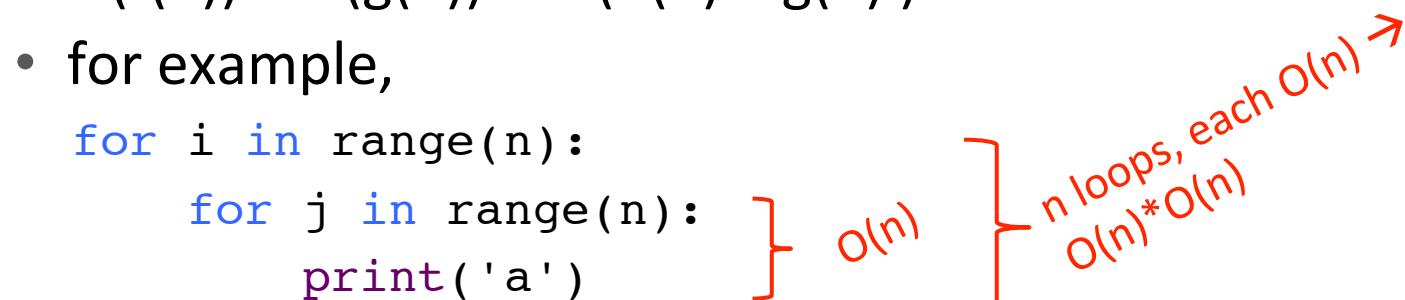
Law of Multiplication for $O()$:

- used with **nested** statements/loops
- $O(f(n)) * O(g(n))$ is $O(f(n) * g(n))$
- for example,

```
for i in range(n):
```

```
    for j in range(n):
```

```
        print('a')
```

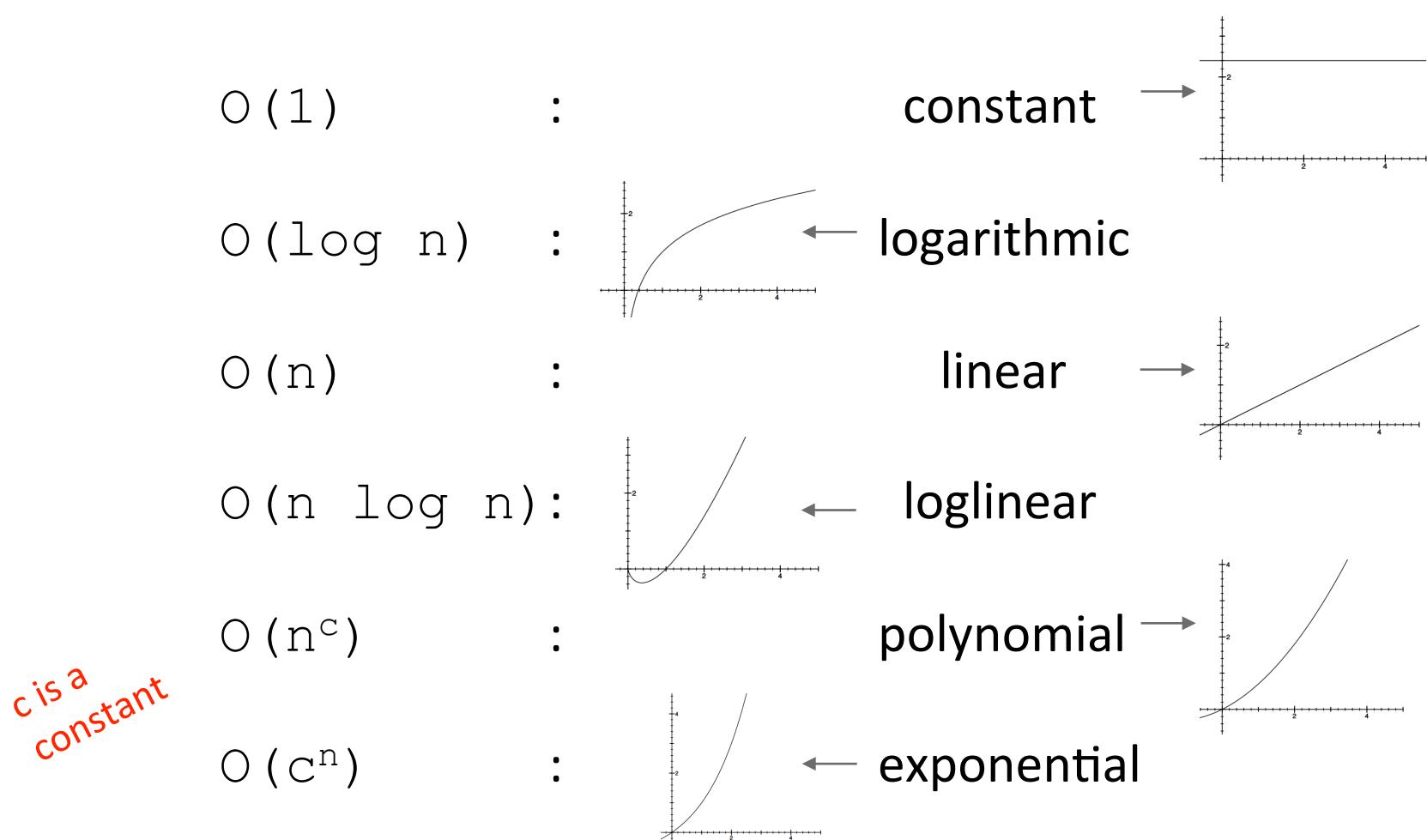


is $O(n)*O(n) = O(n*n) = O(n^2)$ because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

COMPLEXITY CLASSES

- $O(1)$ denotes constant running time
- $O(\log n)$ denotes logarithmic running time
- $O(n)$ denotes linear running time
- $O(n \log n)$ denotes log-linear running time
- $O(n^c)$ denotes polynomial running time (c is a constant)
- $O(c^n)$ denotes exponential running time (c is a constant being raised to a power based on size of input)

COMPLEXITY CLASSES ORDERED LOW TO HIGH



COMPLEXITY GROWTH

CLASS	n=10	= 100	= 1000	= 1000000
O(1)	1	1		1
O(log n)	1	2		3
O(n)	10	100		1000
O(n log n)	10	200		3000
O(n^2)	100	10000		100000000000
O(2^n)	1024	12676506 00228229 40149670 3205376	1071508607186267320948425049060 0018105614048117055336074437503 8837035105112493612249319837881 5695858127594672917553146825187 1452856923140435984577574698574 8039345677748242309854210746050 6237114187795418215304647498358 1941267398767559165543946077062 9145711964776865421676604298316 52624386837205668069376	Good luck!!



LINEAR COMPLEXITY

- Simple iterative loop algorithms are typically linear in complexity

LINEAR SEARCH ON UNSORTED LIST

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

speed up a little by
returning True here,
but speed up doesn't
impact worst case

- must look through all elements to decide it's not there
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
 - $O(1 + 4n + 1) = O(4n + 2) = O(n)$
- overall complexity is **$O(n)$ – where n is $\text{len}(L)$**

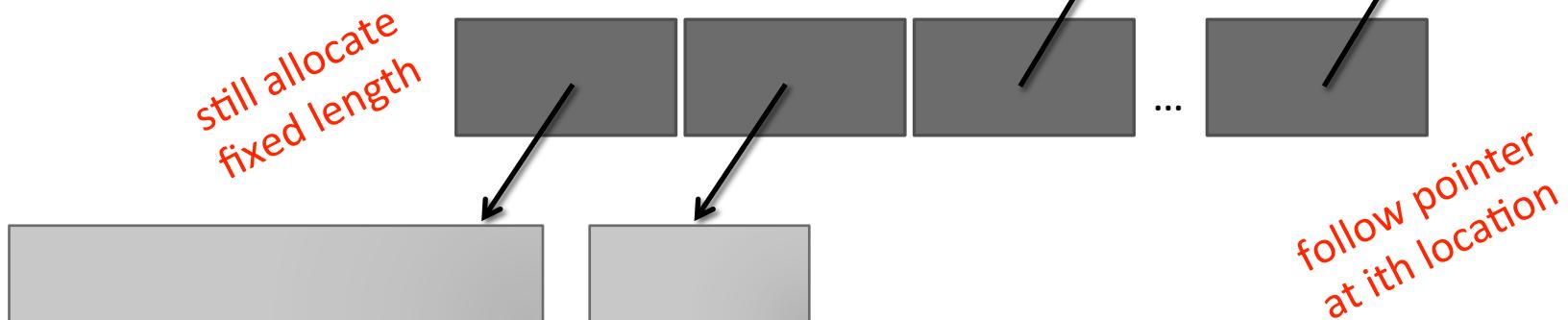
Assumes we can
retrieve element
of list in constant
time

CONSTANT TIME LIST ACCESS

- if list is all ints
 - i^{th} element at
 - $\text{base} + 4*i$



- if list is heterogeneous
 - indirection
 - references to other objects



LINEAR SEARCH ON SORTED LIST

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- must only look until reach a number greater than e
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
- overall complexity is **$O(n)$ – where n is $\text{len}(L)$**
- **NOTE:** order of growth is same, though run time may differ for two search methods

worst case will need
to look at whole list

LINEAR COMPLEXITY

- searching a list in sequence to see if an element is present
- add characters of a string, assumed to be composed of decimal digits

```
def addDigits(s):  
    val = 0  
    for c in s:  
        val += int(c)  
    return val
```

- $O(\text{len}(s))$

LINEAR COMPLEXITY

- complexity often depends on number of iterations

```
def fact_iter(n):  
    prod = 1  
    for i in range(1, n+1):  
        prod *= i  
    return prod
```

- number of times around loop is n
- number of operations inside loop is a constant (in this case, 3 – set i, multiply, set prod)
 - $O(1 + 3n + 1) = O(3n + 2) = O(n)$
- overall just $O(n)$

NESTED LOOPS

- simple loops are linear in complexity
- what about loops that have loops within them?

QUADRATIC COMPLEXITY

determine if one list is subset of second, i.e., every element of first, appears in second (assume no duplicates)

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

QUADRATIC COMPLEXITY

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

outer loop executed $\text{len}(L1)$ times

each iteration will execute inner loop up to $\text{len}(L2)$ times, with constant number of operations

$$O(\text{len}(L1) * \text{len}(L2))$$

worst case when L1 and L2 same length, none of elements of L1 in L2

$$O(\text{len}(L1)^2)$$

QUADRATIC COMPLEXITY

find intersection of two lists, return a list with each element appearing only once

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

QUADRATIC COMPLEXITY

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

first nested loop takes $\text{len}(L1) * \text{len}(L2)$ steps
second loop takes at most $\text{len}(L1)$ steps
determining if element in list might take $\text{len}(L1)$ steps
if we assume lists are of roughly same length, then
 $O(\text{len}(L1)^2)$

O() FOR NESTED LOOPS

```
def g(n):
    """ assume n >= 0 """
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

- computes n^2 very inefficiently
- when dealing with nested loops, **look at the ranges**
- nested loops, **each iterating n times**
- **$O(n^2)$**

THIS TIME AND NEXT TIME

- have seen examples of loops, and nested loops
- give rise to linear and quadratic complexity algorithms
- next time, will more carefully examine examples from each of the different complexity classes

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

UNDERSTANDING PROGRAM EFFICIENCY: 2

(download slides and .py files and follow along!)

6.0001 LECTURE 11

TODAY

- Classes of complexity
- Examples characteristic of each class

WHY WE WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

- how can we reason about an algorithm in order to predict the amount of time it will need to solve a problem of a particular size?
- how can we relate choices in algorithm design to the time efficiency of the resulting algorithm?
 - are there fundamental limits on the amount of time we will need to solve a particular problem?

ORDERS OF GROWTH: RECAP

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth – as tight as possible
- do not need to be precise: “**order of**” not “**exact**” growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

COMPLEXITY CLASSES: RECAP

- 
- $O(1)$ denotes constant running time
 - $O(\log n)$ denotes logarithmic running time
 - $O(n)$ denotes linear running time
 - $O(n \log n)$ denotes log-linear running time
 - $O(n^c)$ denotes polynomial running time (c is a constant)
 - $O(c^n)$ denotes exponential running time (c is a constant being raised to a power based on size of input)

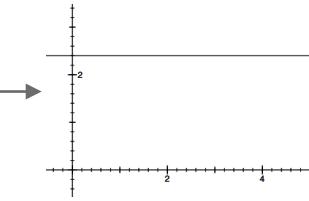
COMPLEXITY CLASSES ORDERED LOW TO HIGH



$O(1)$

:

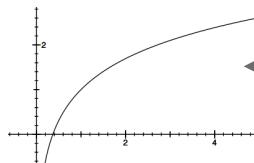
constant



$O(\log n)$

:

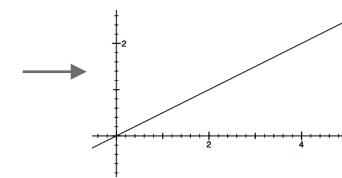
logarithmic



$O(n)$

:

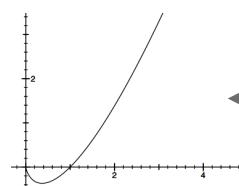
linear



$O(n \log n)$

:

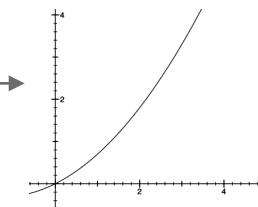
loglinear



$O(n^c)$

:

polynomial

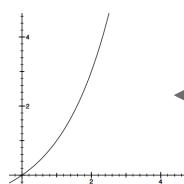


c is a
constant

$O(c^n)$

:

exponential



COMPLEXITY GROWTH

CLASS	n=10	= 100	= 1000	= 1000000
O(1)	1	1		1
O(log n)	1	2		3
O(n)	10	100		1000
O(n log n)	10	200		3000
O(n^2)	100	10000		100000000000
O(2^n)	1024	12676506 00228229 40149670 3205376	1071508607186267320948425049060 0018105614048117055336074437503 8837035105112493612249319837881 5695858127594672917553146825187 1452856923140435984577574698574 8039345677748242309854210746050 6237114187795418215304647498358 1941267398767559165543946077062 9145711964776865421676604298316 52624386837205668069376	Good luck!!



CONSTANT COMPLEXITY

- complexity independent of inputs
- very few interesting algorithms in this class, but can often have pieces that fit this class
- can have loops or recursive calls, but ONLY IF number of iterations or calls independent of size of input

LOGARITHMIC COMPLEXITY

- complexity grows as log of size of one of its inputs
- example:
 - bisection search
 - binary search of a list

BISECTION SEARCH

- suppose we want to know if a particular element is present in a list
- saw last time that we could just “walk down” the list, checking each element
- complexity was linear in length of the list
- suppose we know that the list is ordered from smallest to largest
 - saw that sequential search was still linear in complexity
 - can we do better?

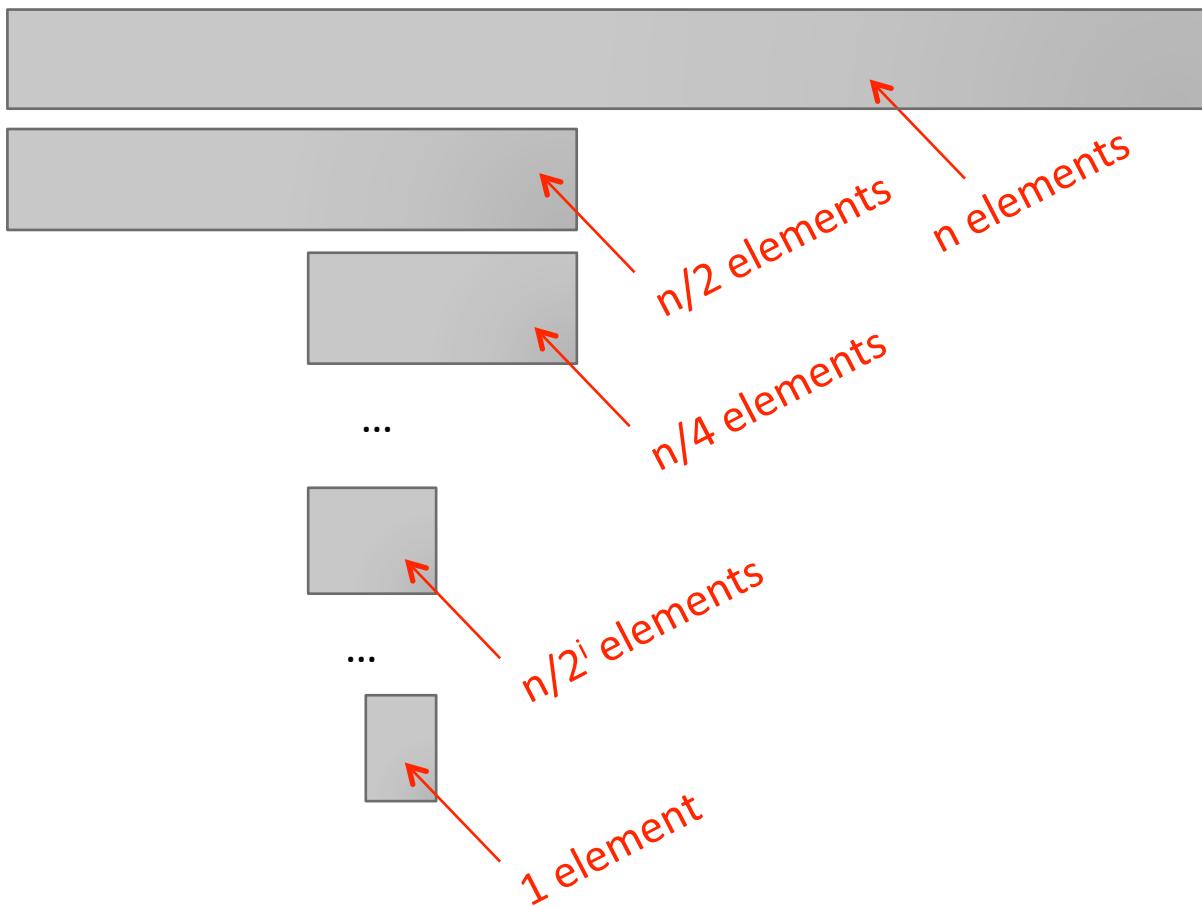
BISECTION SEARCH

1. pick an index, i , that divides list in half
2. ask if $L[i] == e$
3. if not, ask if $L[i]$ is larger or smaller than e
4. depending on answer, search left or right half of L for e

A new version of a divide-and-conquer algorithm

- break into smaller version of problem (smaller list), plus some simple operations
- answer to smaller version is answer to original problem

BISECTION SEARCH COMPLEXITY ANALYSIS



- finish looking through list when

$$1 = n/2^i$$

$$\text{so } i = \log n$$

- complexity of recursion is **O(log n)** – where n is len(L)

BISECTION SEARCH IMPLEMENTATION 1

```
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1(L[:half], e)
        else:
            return bisect_search1(L[half:], e)
```

constant
 $O(1)$

constant
 $O(1)$

constant
 $O(1)$

NOT constant,
copies list

NOT constant

NOT constant

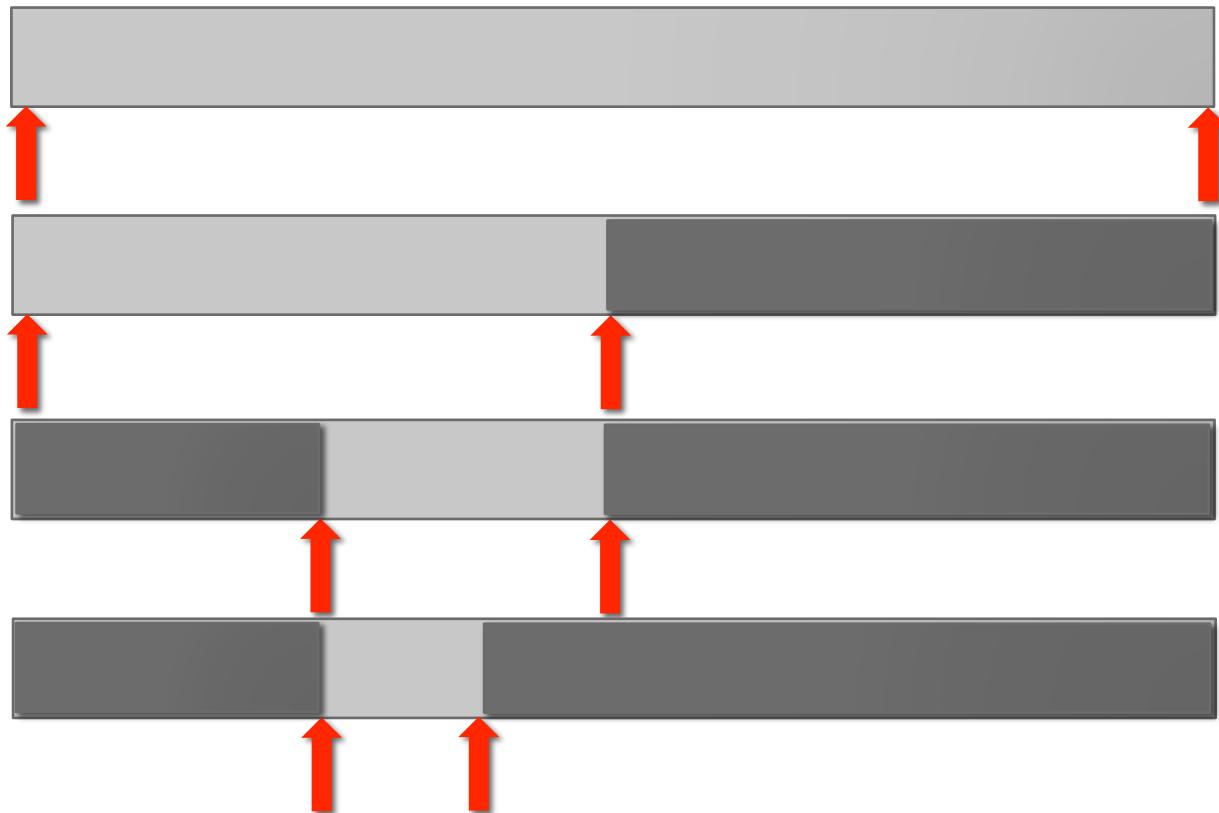
NOT constant

COMPLEXITY OF FIRST BISECTION SEARCH METHOD

■ **implementation 1 – bisect_search1**

- $O(\log n)$ bisection search calls
 - On each recursive call, size of range to be searched is cut in half
 - If original range is of size n , in worst case down to range of size 1 when $n/(2^k) = 1$; or when $k = \log n$
- $O(n)$ for each bisection search call to copy list
 - This is the cost to set up each call, so do this for each level of recursion
- $O(\log n) * O(n) \rightarrow O(n \log n)$
 - if we are really careful, note that length of list to be copied is also halved on each recursive call
 - turns out that total cost to copy is **$O(n)$** and this dominates the $\log n$ cost due to the recursive calls

BISECTION SEARCH ALTERNATIVE



- still reduce size of problem by factor of two on each step
- but just keep track of low and high portion of list to be searched
- avoid copying the list
- complexity of recursion is again **$O(\log n)$ – where n is $\text{len}(L)$**

BISECTION SEARCH IMPLEMENTATION 2

```
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

constant other
than recursive call

constant other
than recursive call

COMPLEXITY OF SECOND BISECTION SEARCH METHOD

- **implementation 2 – bisect_search2** and its helper
 - $O(\log n)$ bisection search calls
 - On each recursive call, size of range to be searched is cut in half
 - If original range is of size n , in worst case down to range of size 1 when $n/(2^k) = 1$; or when $k = \log n$
 - pass list and indices as parameters
 - list never copied, just re-passed as a pointer
 - thus $O(1)$ work on each recursive call
 - $O(\log n) * O(1) \rightarrow O(\log n)$

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10
    return result
```

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return result
```

only have to look at loop as no function calls

within while loop, constant number of steps

how many times through loop?

- how many times can one divide i by 10?
- $O(\log(i))$

LINEAR COMPLEXITY

- saw this last time
 - searching a list in sequence to see if an element is present
 - iterative loops

$O()$ FOR ITERATIVE FACTORIAL

- complexity can depend on number of iterative calls

```
def fact_iter(n):  
    prod = 1  
    for i in range(1, n+1):  
        prod *= i  
    return prod
```

- overall $O(n)$ – n times round loop, constant cost each time

O() FOR RECURSIVE FACTORIAL

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

- computes factorial recursively
- if you time it, may notice that it runs a bit slower than iterative version due to function calls
- still **$O(n)$** because the number of function calls is linear in n , and constant effort to set up call
- **iterative and recursive factorial** implementations are the **same order of growth**

LOG-LINEAR COMPLEXITY

- many practical algorithms are log-linear
- very commonly used log-linear algorithm is merge sort
- will return to this next lecture

POLYNOMIAL COMPLEXITY

- most common polynomial algorithms are quadratic, i.e., complexity grows with square of size of input
- commonly occurs when we have nested loops or recursive function calls
- saw this last time

EXPONENTIAL COMPLEXITY

- recursive functions where more than one recursive call for each size of problem
 - Towers of Hanoi
- many important problems are inherently exponential
 - unfortunate, as cost can be high
 - will lead us to consider approximate solutions as may provide reasonable answer more quickly

COMPLEXITY OF TOWERS OF HANOI

- Let t_n denote time to solve tower of size n
- $t_n = 2t_{n-1} + 1$
- $= 2(2t_{n-2} + 1) + 1$
- $= 4t_{n-2} + 2 + 1$
- $= 4(2t_{n-3} + 1) + 2 + 1$
- $= 8t_{n-3} + 4 + 2 + 1$
- $= 2^k t_{n-k} + 2^{k-1} + \dots + 4 + 2 + 1$
- $= 2^{n-1} + 2^{n-2} + \dots + 4 + 2 + 1$
- $= 2^n - 1$
- so order of growth is $O(2^n)$

Geometric growth

$$\begin{aligned} a &= 2^{n-1} + \dots + 2 + 1 \\ 2a &= 2^n + 2^{n-1} + \dots + 2 \\ a &= 2^n - 1 \end{aligned}$$

EXPONENTIAL COMPLEXITY

- given a set of integers (with no repeats), want to generate the collection of all possible subsets – called the power set
- $\{1, 2, 3, 4\}$ would generate
 - $\{\}, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$
- order doesn't matter
 - $\{\}, \{1\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \{4\}, \{1, 4\}, \{2, 4\}, \{1, 2, 4\}, \{3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$

POWER SET – CONCEPT

- we want to generate the power set of integers from 1 to n
- assume we can generate power set of integers from 1 to n-1
- then all of those subsets belong to bigger power set (choosing not include n); and all of those subsets with n added to each of them also belong to the bigger power set (choosing to include n)
- ~~{}, {1} {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}, {4}, {1, 4}, {2, 4}, {1, 2, 4}, {3, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}~~
- nice recursive description!

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]] #list of empty list
    smaller = genSubsets(L[:-1]) # all subsets without
last element
    extra = L[-1:] # create a list of just last element
    new = []
    for small in smaller:
        new.append(small+extra) # for all smaller
solutions, add one with last element
    return smaller+new # combine those with last
element and those without
```

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

assuming append is
constant time

time includes time to solve
smaller problem, plus time
needed to make a copy of
all elements in smaller
problem

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

but important to think
about size of smaller

know that for a set of size
 k there are 2^k cases

how can we deduce
overall complexity?

EXPONENTIAL COMPLEXITY

- let t_n denote time to solve problem of size n
- let s_n denote size of solution for problem of size n
- $t_n = t_{n-1} + s_{n-1} + c$ (where c is some constant number of operations)
- $t_n = t_{n-1} + 2^{n-1} + c$
- $= t_{n-2} + 2^{n-2} + c + 2^{n-1} + c$
- $= t_{n-k} + 2^{n-k} + \dots + 2^{n-1} + kc$
- $= t_0 + 2^0 + \dots + 2^{n-1} + nc$
- $= 1 + 2^n + nc$

Thus
computing
power set is
 $O(2^n)$

COMPLEXITY CLASSES

- $O(1)$ – code does not depend on size of problem
- $O(\log n)$ – reduce problem in half each time through process
- $O(n)$ – simple iterative or recursive programs
- $O(n \log n)$ – will see next time
- $O(n^c)$ – nested loops or recursive calls
- $O(c^n)$ – multiple recursive calls at each level

SOME MORE EXAMPLES OF ANALYZING COMPLEXITY

COMPLEXITY OF ITERATIVE FIBONACCI

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

constant O(1)

constant O(1)

linear O(n)

constant O(1)

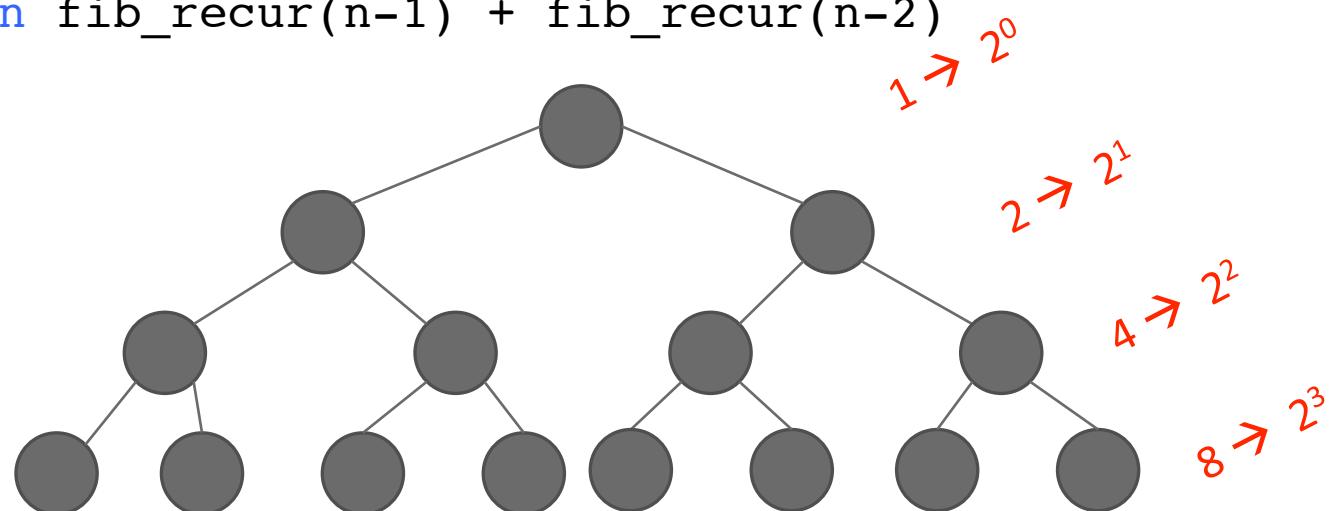
- Best case: $O(1)$
- Worst case: $O(1) + O(n) + O(1) \rightarrow O(n)$

COMPLEXITY OF RECURSIVE FIBONACCI

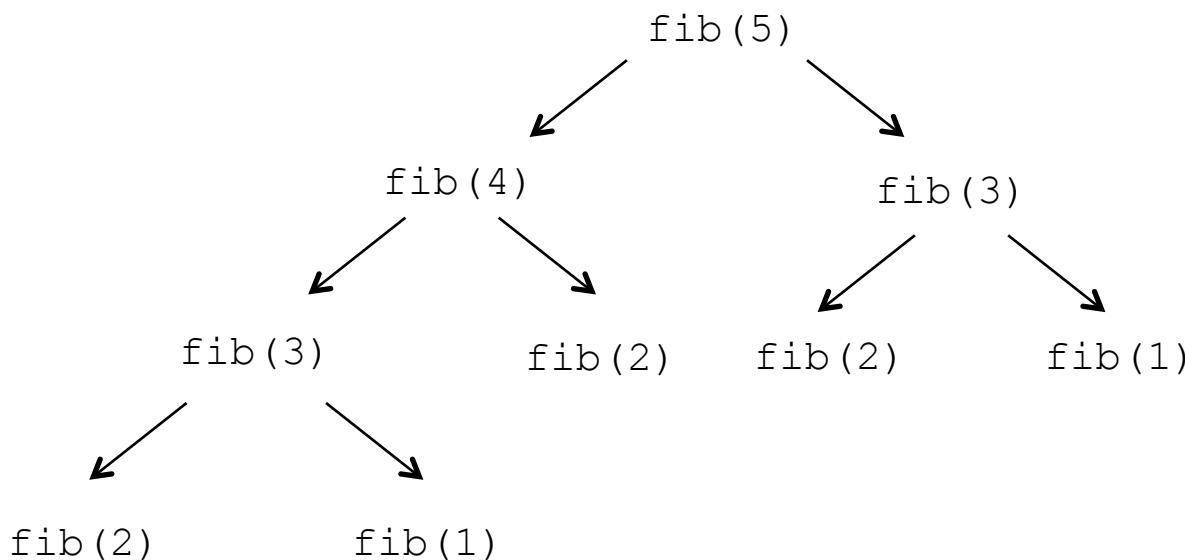
```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

- Worst case:

$O(2^n)$



COMPLEXITY OF RECURSIVE FIBONACCI



- actually can do a bit better than 2^n since tree of cases thins out to right
- but complexity is still exponential

BIG OH SUMMARY

- compare **efficiency of algorithms**
 - notation that describes growth
 - **lower order of growth** is better
 - independent of machine or specific implementation

- use Big Oh
 - describe order of growth
 - **asymptotic notation**
 - **upper bound**
 - **worst case** analysis

COMPLEXITY OF COMMON PYTHON FUNCTIONS

- Lists: n is `len(L)`
 - `index` $O(1)$
 - `store` $O(1)$
 - `length` $O(1)$
 - `append` $O(1)$
 - `==` $O(n)$
 - `remove` $O(n)$
 - `copy` $O(n)$
 - `reverse` $O(n)$
 - `iteration` $O(n)$
 - `in list` $O(n)$
- Dictionaries: n is `len(d)`
 - worst case
 - `index` $O(n)$
 - `store` $O(n)$
 - `length` $O(n)$
 - `delete` $O(n)$
 - `iteration` $O(n)$
 - average case
 - `index` $O(1)$
 - `store` $O(1)$
 - `delete` $O(1)$
 - `iteration` $O(n)$

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.

SEARCHING AND SORTING ALGORITHMS

(download slides and .py files and follow along!)

6.0001 LECTURE 12

SEARCH ALGORITHMS

- search algorithm – method for finding an item or group of items with specific properties within a collection of items
- collection could be implicit
 - example – find square root as a search problem
 - exhaustive enumeration
 - bisection search
 - Newton-Raphson
- collection could be explicit
 - example – is a student record in a stored collection of data?

SEARCHING ALGORITHMS

- linear search
 - **brute force** search (aka British Museum algorithm)
 - list does not have to be sorted
- bisection search
 - list **MUST be sorted** to give correct answer
 - saw two different implementations of the algorithm

LINEAR SEARCH ON UNSORTED LIST: RECAP

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

speed up a little by
returning True here,
but speed up doesn't
impact worst case

- must look through all elements to decide it's not there
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
- overall complexity is **$O(n)$ – where n is $\text{len}(L)$**

Assumes we can
retrieve element
of list in constant
time

LINEAR SEARCH ON SORTED LIST: RECAP

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- must only look until reach a number greater than e
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
- overall complexity is **$O(n) - \text{where } n \text{ is } \text{len}(L)$**

USE BISECTION SEARCH: RECAP

1. Pick an index, i , that divides list in half
2. Ask if $L[i] == e$
3. If not, ask if $L[i]$ is larger or smaller than e
4. Depending on answer, search left or right half of L for e

A new version of a divide-and-conquer algorithm

- Break into smaller version of problem (smaller list), plus some simple operations
- Answer to smaller version is answer to original problem

BISECTION SEARCH IMPLEMENTATION: RECAP

```
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

COMPLEXITY OF BISECTION SEARCH: RECAP

- **bisect_search2** and its helper
 - $O(\log n)$ bisection search calls
 - reduce size of problem by factor of 2 on each step
 - pass list and indices as parameters
 - list never copied, just re-passed as pointer
 - constant work inside function
 - → **$O(\log n)$**

SEARCHING A SORTED LIST

-- n is $\text{len}(L)$

- using **linear search**, search for an element is **O(n)**
- using **binary search**, can search for an element in **O(log n)**
 - assumes the **list is sorted!**
- when does it make sense to **sort first then search?**
 - $\text{SORT} + O(\log n) < O(n) \rightarrow \text{SORT} < O(n) - O(\log n)$
 - when sorting is less than $O(n)$
- **NEVER TRUE!**
 - **to sort a collection of n elements must look at each one at least once!**

AMORTIZED COST

-- n is $\text{len}(L)$

- why bother sorting first?
- in some cases, may **sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches
- $\text{SORT} + K * O(\log n) < K * O(n)$
 - for large K, **SORT time becomes irrelevant**, if cost of sorting is small enough

SORT ALGORITHMS

- Want to efficiently sort a list of entries (typically numbers)
- Will see a range of methods, including one that is quite efficient

MONKEY SORT

- aka bogosort, stupid sort, slowsort, permutation sort, shotgun sort
- to sort a deck of cards
 - throw them in the air
 - pick them up
 - are they sorted?
 - repeat if not sorted



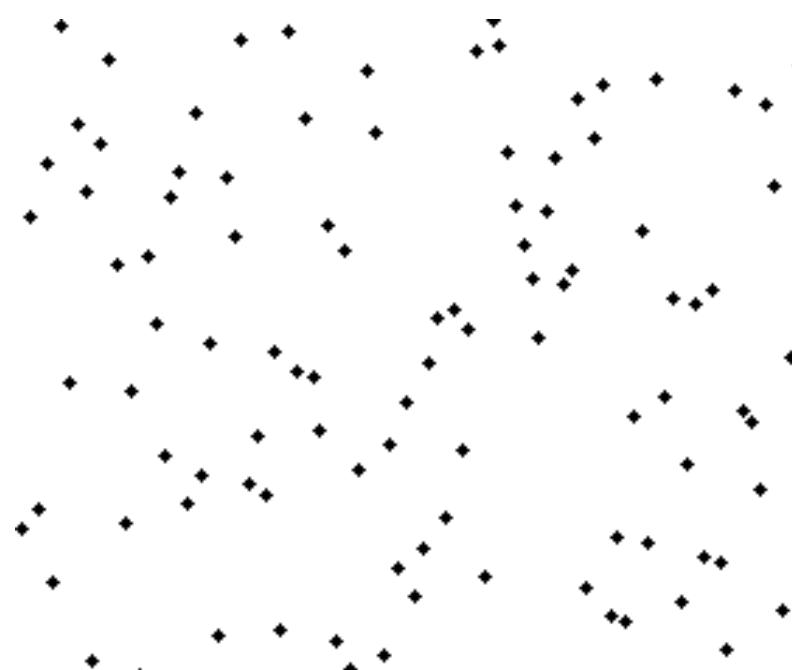
COMPLEXITY OF BOGO SORT

```
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

- best case: **O(n) where n is $\text{len}(L)$** to check if sorted
- worst case: **O(?)** it is **unbounded** if really unlucky

BUBBLE SORT

- **compare consecutive pairs** of elements
 - **swap elements** in pair such that smaller is first
 - when reach end of list, **start over** again
 - stop when **no more swaps** have been made
 - largest unsorted element always at end after pass, so at most n passes
- CC-BY Hydrargyrum
https://commons.wikimedia.org/wiki/File:Bubble_sort_animation.gif



COMPLEXITY OF BUBBLE SORT

```
def bubble_sort(L):
    swap = False
    while not swap: O(len(L))
        swap = True
        for j in range(1, len(L)): O(len(L))
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **O(n^2) where n is $\text{len}(L)$**
to do $\text{len}(L)-1$ comparisons and $\text{len}(L)-1$ passes

SELECTION SORT

- first step
 - extract **minimum element**
 - **swap it** with element at **index 0**
- subsequent step
 - in remaining sublist, extract **minimum element**
 - **swap it** with the element at **index 1**
- keep the left portion of the list sorted
 - at i'th step, **first i elements in list are sorted**
 - all other elements are bigger than first i elements

ANALYZING SELECTION SORT

- loop invariant
 - given prefix of list $L[0:i]$ and suffix $L[i+1:len(L)]$, then prefix is sorted and no element in prefix is larger than smallest element in suffix
 1. base case: prefix empty, suffix whole list – invariant true
 2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
 3. when exit, prefix is entire list, suffix empty, so sorted

COMPLEXITY OF SELECTION SORT

```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1
```

len(L) times
→ $O(\text{len}(L))$

len(L) – suffixSt times
→ $O(\text{len}(L))$

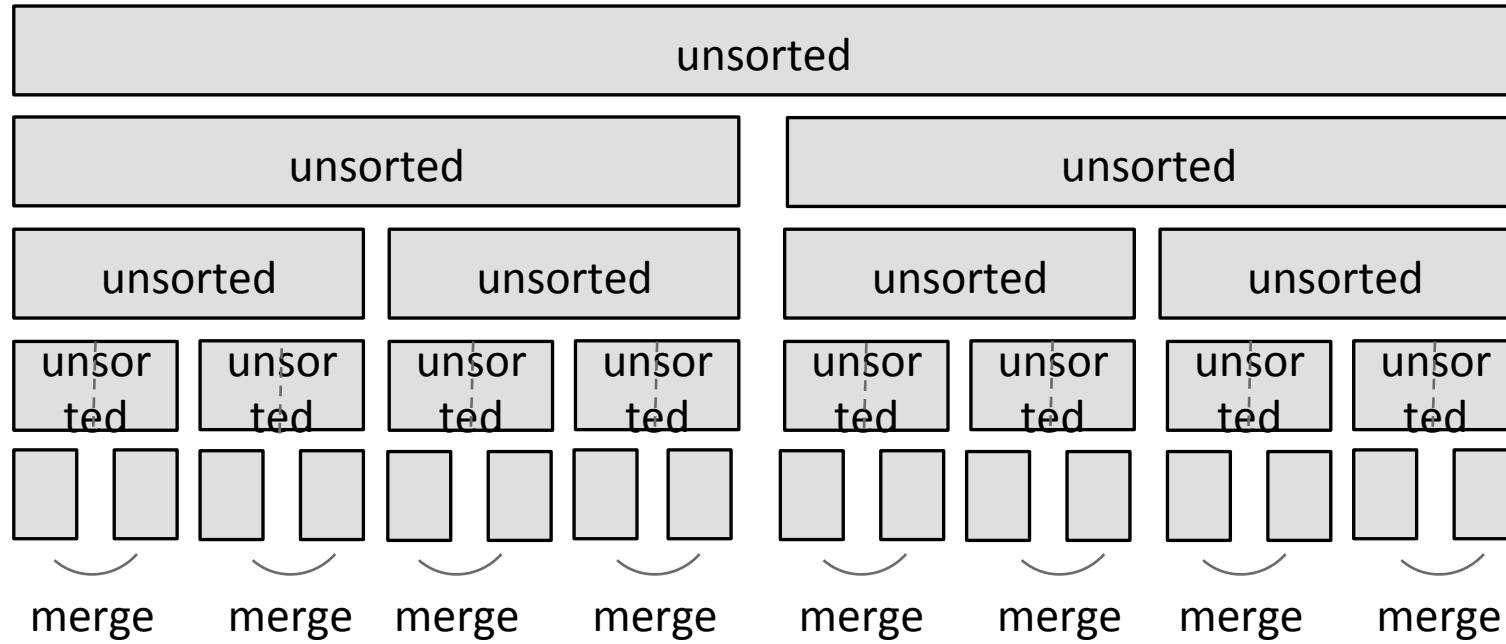
- outer loop executes $\text{len}(L)$ times
- inner loop executes $\text{len}(L) - i$ times
- complexity of selection sort is **$O(n^2)$ where n is $\text{len}(L)$**

MERGE SORT

- use a divide-and-conquer approach:
 1. if list is of length 0 or 1, already sorted
 2. if list has more than one element, split into two lists, and sort each
 3. merge sorted sublists
 1. look at first element of each, move smaller to end of the result
 2. when one list empty, just copy rest of other list

MERGE SORT

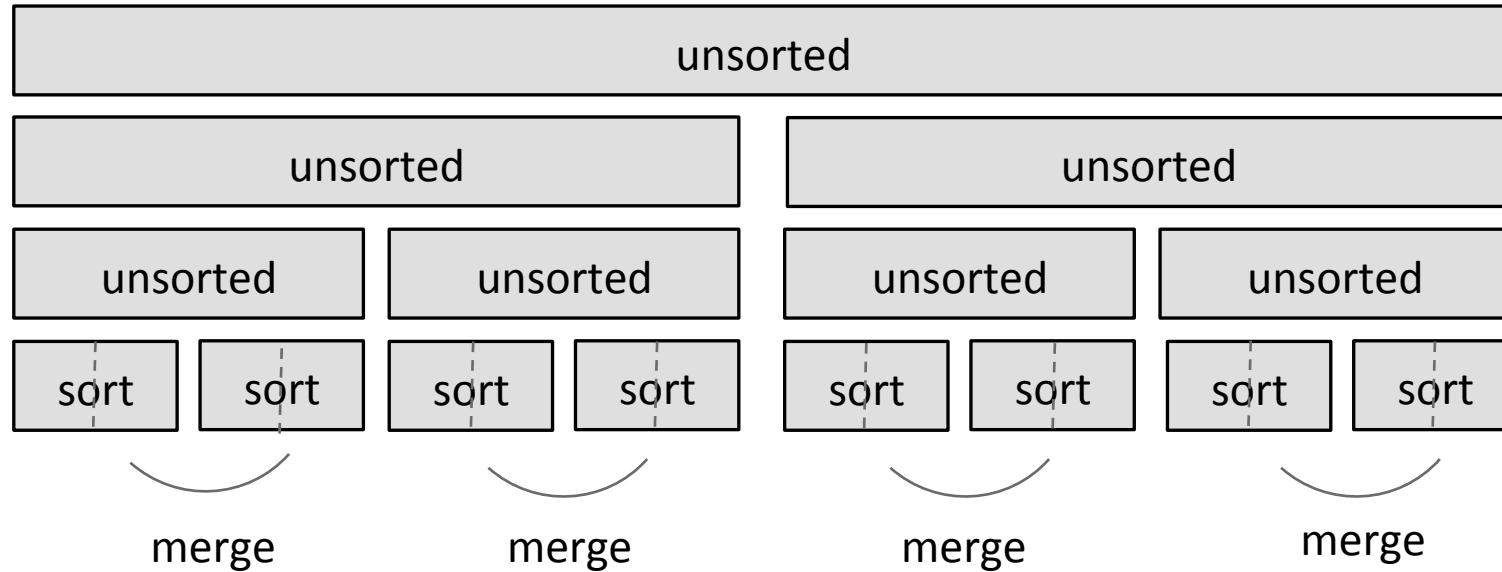
- divide and conquer



- **split list in half** until have sublists of only 1 element

MERGE SORT

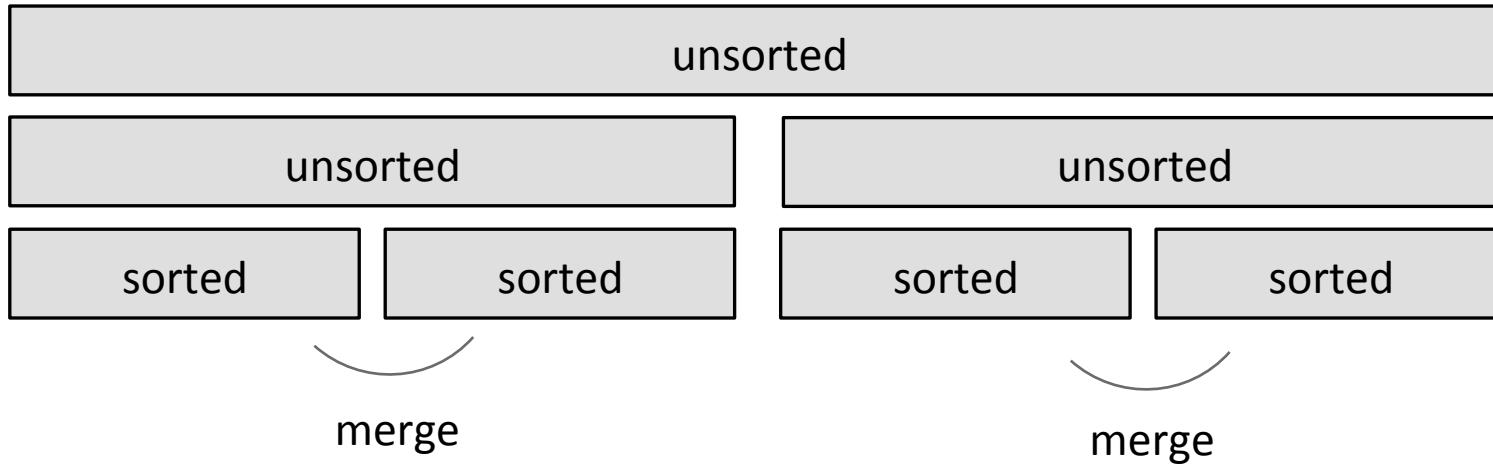
- divide and conquer



- merge such that **sublists will be sorted after merge**

MERGE SORT

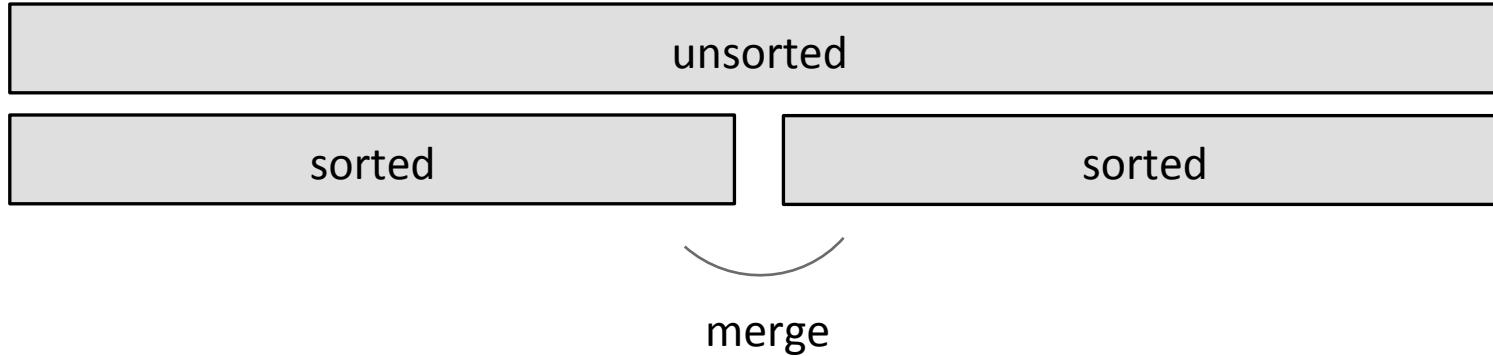
- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer – done!

sorted

EXAMPLE OF MERGING

Left in list 1	Left in list 2	Compare	Result
[15,12,18,19,20]	[2,3,4,17]	1, 2 → []	
[5,12,18,19,20]	[2,3,4,17]	5, 2 → [1]	
[5,12,18,19,20]	[3,4,17]	5, 3 → [1,2]	
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

MERGING SUBLISTS STEP

```
def merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

- left and right sublists are ordered
- move indices for sublists depending on which sublist holds next smallest element

when right sublist is empty

when left sublist is empty

COMPLEXITY OF MERGING SUBLISTS STEP

- go through two lists, only one pass
- compare only **smallest elements in each sublist**
- $O(\text{len(left)} + \text{len(right)})$ copied elements
- $O(\text{len(longer list)})$ comparisons
- **linear in length of the lists**

MERGE SORT ALGORITHM

-- RECURSIVE

```
def merge_sort(L):
    if len(L) < 2:
        return L[:]

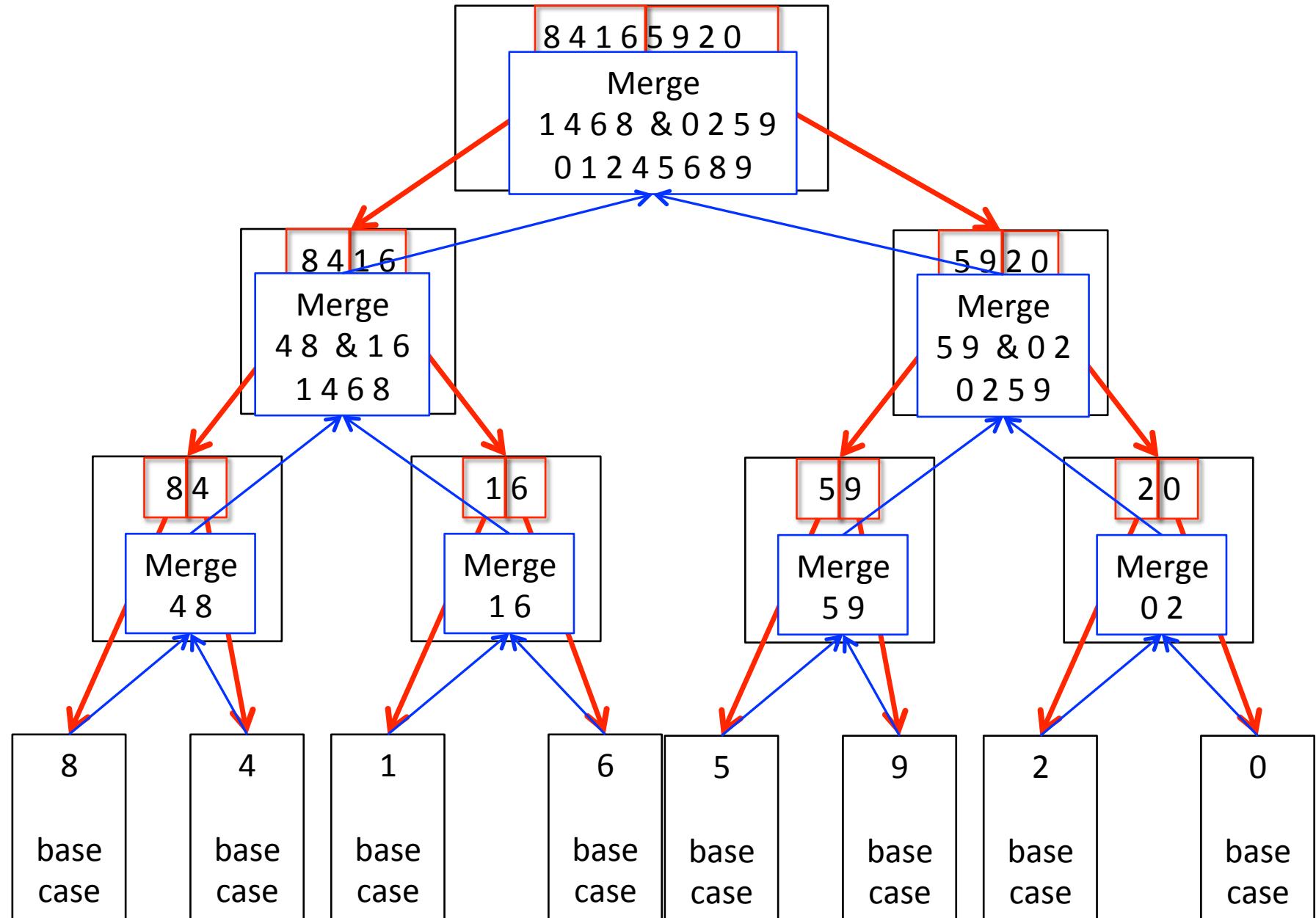
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

base case

divide

conquer with
the merge step

- **divide list** successively into halves
- depth-first such that **conquer smallest pieces down one branch** first before moving to larger pieces



COMPLEXITY OF MERGE SORT

- at **first recursion level**
 - $n/2$ elements in each list
 - $O(n) + O(n) = O(n)$ where n is $\text{len}(L)$
- at **second recursion level**
 - $n/4$ elements in each list
 - two merges $\rightarrow O(n)$ where n is $\text{len}(L)$
- each recursion level is $O(n)$ where n is $\text{len}(L)$
- **dividing list in half** with each recursive call
 - $O(\log(n))$ where n is $\text{len}(L)$
- overall complexity is **$O(n \log(n))$ where n is $\text{len}(L)$**

SORTING SUMMARY

-- n is $\text{len}(L)$

- bogo sort
 - randomness, unbounded $O()$
- bubble sort
 - $O(n^2)$
- selection sort
 - $O(n^2)$
 - guaranteed the first i elements were sorted
- merge sort
 - $O(n \log(n))$
- $O(n \log(n))$ is the fastest a sort can be

WHAT HAVE WE SEEN IN 6.0001?

KEY TOPICS

- ✓ ■ represent knowledge with **data structures**
- ✓ ■ **iteration and recursion** as computational metaphors
- ✓ ■ **abstraction** of procedures and data types
- ✓ ■ **organize and modularize** systems using object classes and methods
- ✓ ■ different classes of **algorithms**, searching and sorting
- ✓ ■ **complexity** of algorithms

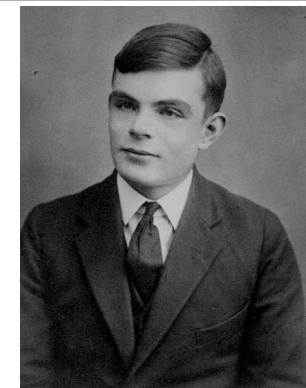
OVERVIEW OF COURSE

- ✓ ■ learn computational modes of thinking
- ✓ ■ begin to master the art of computational problem solving
- ✓ ■ make computers do what you want them to do

Hope we have started you down the path to being able to think and act like a computer scientist

WHAT DO COMPUTER SCIENTISTS DO?

- they think computationally
 - abstractions, algorithms, automated execution
- just like the three r's: reading, 'riting, and 'rithmetic – computational thinking is becoming a fundamental skill that every well-educated person will need



Alan Turing

Image in the Public Domain, courtesy of [Wikipedia Commons](#).



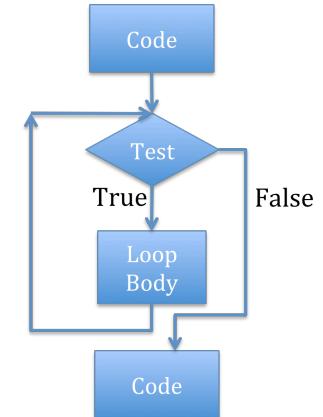
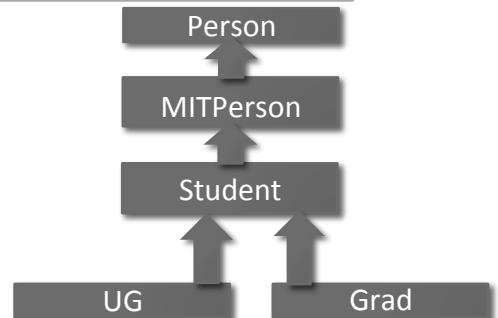
Ada Lovelace

Image in the Public Domain, courtesy of [Wikipedia Commons](#).

I ❤️ 6.0001

THE THREE A'S OF COMPUTATIONAL THINKING

- abstraction
 - choosing the right abstractions
 - operating in multiple layers of abstraction simultaneously
 - defining the relationships between the abstraction layers
- automation
 - think in terms of mechanizing our abstractions
 - mechanization is possible – because we have precise and exacting notations and models; and because there is some “machine” that can interpret our notations
- algorithms
 - language for describing automated processes
 - also allows abstraction of details
 - language for communicating ideas & processes



```
def mergeSort(L, compare = operator.lt):  
    if len(L) < 2:  
        return L[:]  
    else:  
        middle = int(len(L)/2)  
        left = mergeSort(L[:middle], compare)  
        right = mergeSort(L[middle:], compare)  
        return merge(left, right, compare)
```

ASPECTS OF COMPUTATIONAL THINKING

- how difficult is this problem and how best can I solve it?
 - theoretical computer science gives precise meaning to these and related questions and their answers
- thinking recursively
 - reformulating a seemingly difficult problem into one which we know how to solve
 - reduction, embedding, transformation, simulation

$O(\log n)$; $O(n)$;
 $O(n \log n)$;
 $O(n^2)$; $O(c^n)$

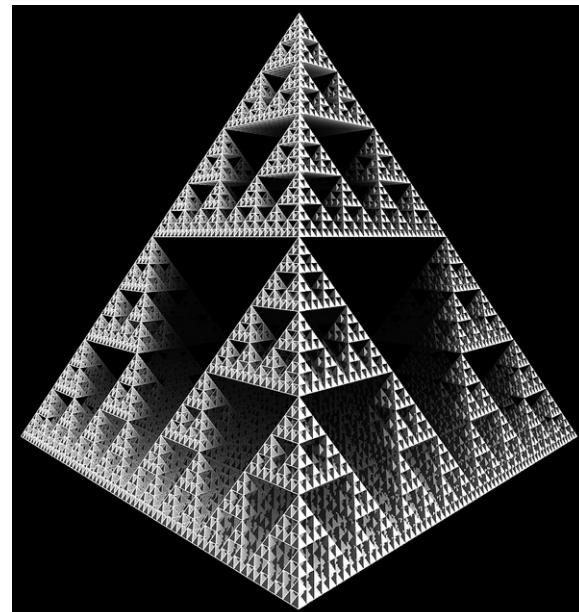


Image Licensed CC-BY, Courtesy of [Robson#](#) on Flickr.

MIT OpenCourseWare
<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.