Computer Science 111
Introduction to Computer Science I

Boston University, Spring 2025

The slides in this book are based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College.

Introduction to Computer Science I

Course Overview

Computer Science 111
Boston University

Welcome to CS 111!

Computer science is not so much the science of computers as it is the science of solving problems using computers.

Eric Roberts

- This course covers:
  - the process of developing algorithms to solve problems
  - the process of developing computer programs to express those algorithms
  - other topics from computer science and its applications

Computer Science and Programming

- There are many different fields within CS, including:
  - software systems
  - computer architecture
  - networking
  - programming languages, compilers, etc.
  - theory
  - AI
- Experts in many of these fields don't do much programming!
- *However, learning to program will help you to develop ways of thinking and solving problems used in all fields of CS.*

A Breadth-Based Introduction

- Five major units:
  - weeks 0-4: computational problem solving and "functional" programming
  - weeks 4-6: a look "under the hood" (digital logic, circuits, etc.)
  - weeks 6-8: imperative programming
  - weeks 8-11: object-oriented programming
  - weeks 12-end: topics from CS theory
- In addition, short articles on other CS-related topics.
- Main goals:
  - to develop your computational problem-solving skills
    - including, but not limited to, coding skills
  - to give you a sense of the richness of computer science

A *Rigorous* Introduction

- Intended for:
  - CS, math, and physical science concentrators
  - others who want a rigorous introduction
  - no programming background required,
    but can benefit people with prior background
- Allow for 10-15 hours of work per week
  - start work early!
- Other alternatives include:
  - CS 101: overview of CS
  - CS 103: the Internet
  - CS 108: programming with applications for non-majors
  - DS 100: programming, data modeling and visualization
  - for more info:
    http://www.bu.edu/cs/courses/divisional-study-courses

Course Materials

- **Required**: *The CS 111 Coursepack*
  - use it during pre-lecture and lecture – need to fill in the blanks!
  - PDF version is available on Blackboard
  - recommended: get it printed
    - one option: FedEx Office (Cummington & Comm Ave)
    - to order, email `usa5012@fedex.com`
- **Required** in-class software: Top Hat Pro platform
  - used for pre-lecture quizzes and in-lecture exercises
  - also used periodically for location-based attendance
  - create your account and purchase a subscription ASAP
- **Optional** textbook: *CS for All*
  by Alvarado, Dodds, Kuenning, and Libeskind-Hadas

Traditional Lecture Classes

- The instructor summarizes what you need to know.
- Readings are assigned, but may not actually be done!
- Dates back to before the printing press.

- Many technological developments since then!

Limitations of the Traditional Approach

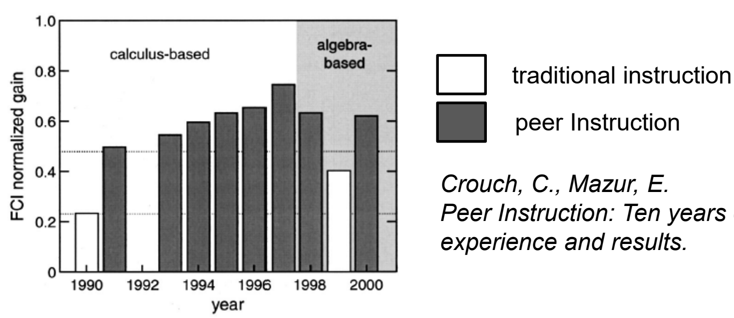
- You get little or no immediate feedback.
- Research shows that little is learned from passive listening.
  - need to actively engage with the material
- Homework provides active engagement, but in-class engagement provides added benefits.

Lectures in this Class

- Based on an approach called *peer instruction*.
  - developed by Eric Mazur at Harvard
- Basic process:
  1. Question posed (possibly after a short intro)
  2. Solo vote on Top Hat (no discussion yet)
  3. Small-group discussions (in teams of 3)
     - explain your thinking to each other
     - come to a consensus
  4. Group vote on Top Hat
     - each person in the group should enter the same answer
  5. Class-wide discussion

Benefits of Peer Instruction

- It promotes active engagement.
- You get immediate feedback about your understanding.
- I get immediate feedback about your understanding!
- It promotes increased learning.
  - explaining concepts to others benefits you!

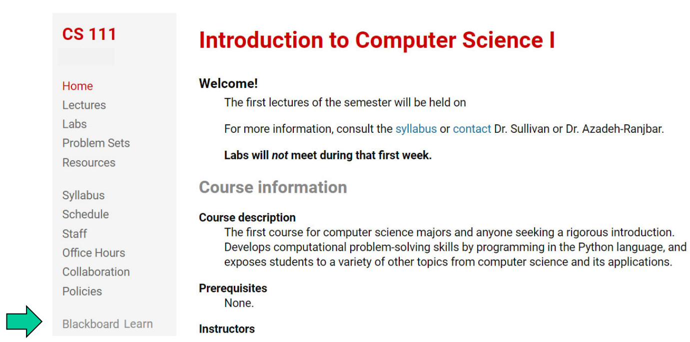


Crouch, C., Mazur, E.
Peer Instruction: Ten years of experience and results.

Preparing for Lecture

- Short video(s) and/or readings
  - fill in the blanks as you watch the videos!
- Short online reading quiz on Top Hat
  - complete **by 10 a.m.** of the day of lecture (unless noted otherwise)
  - won't typically be graded for correctness
  - your work should show that you've prepared for lecture
  - **no late submissions accepted**
- Preparing for lecture is essential!
  - gets you ready for the lecture questions and discussions
  - we won't cover everything in lecture

Course Website
www.cs.bu.edu/courses/cs111



- *not* the same as the Blackboard site for the course
- use Blackboard to access info. on:
  - the pre-lecture videos/readings
  - the pre-lecture quizzes
  - list of pages covered in lecture

posted by 36 hours before lecture

Labs

- Will help you prepare for and get started on the assignments
- Will also reinforce essential skills
- **ASAP: Complete Lab 0** (on the course website)
  - short tasks to prepare you for the semester

Assignments

- Weekly problem sets
  - most have two parts:
    - part I due by 11:59 p.m. on Thursday
    - part II due by 11:59 p.m. on Sunday
- Final project (worth 1.5 times an ordinary assignment)
- Can submit up to 24 hours late with a 10% penalty.
- No submissions accepted after 24 hours.

Collaboration

- Two types of homework problems:
  - individual-only: must complete on your own
  - pair-optional: can complete alone or with one other student
- For both types of problems:
  - may discuss the main ideas with others
  - may ***not* view** another student/pair's work
  - may ***not* show** your work to another student/pair
  - don't give a student unmonitored access to your laptop
  - don't consult solutions in books or online
  - **don't use tools that automate coding/problem-solving**
  - don't post your work where others can view it
- ***Students who engage in misconduct can face serious repercussions (see the syllabus).***

Grading

1. Weekly problem sets + final project (25%)
2. Exams
   - two midterms (30%) – **Wed nights 6:30-7:45**; no makeups!
   - final exam (35%)
     - can replace lowest problem set and lowest midterm
     - **wait until you hear its dates/times from me;**
       initial info posted by Registrar will likely be incorrect;
       make sure you're available for the entire exam period!
3. Participation (10%)

To pass the course, you must have
a passing PS average
and a passing exam average.

Participation

- Full credit if you:
  - earn 85% of points for pre-lecture and in-lecture questions
  - make 85% of the lecture-attendance votes
  - attend 85% of the labs
- If you end up with x% for a given component where x < 85, you will get x/85 of the possible points.
- This policy is designed to allow for occasional absences for special circumstances.
- If you need to miss a lecture:
  - watch its recording ASAP (available on Blackboard)
  - keep up with the pre-lecture tasks and the assignments
  - **do not email your instructor!**

Course Staff

- **Instructors:** David Sullivan (A1 lecture)
  Tiago Januario (B1 lecture)
- **Teaching Assistants (TAs)**
  plus **Undergrad Course Assistants (CAs)**
  - see the course website for names and photos:
    http://www.cs.bu.edu/courses/cs111/staff.shtml
- Office-hour calendar:
  http://www.cs.bu.edu/courses/cs111/office_hours.shtml
- For questions: **post on Piazza** or cs111-staff@cs.bu.edu

Algorithms

- In order to solve a problem using a computer, you need to come up with one or more *algorithms*.
- An algorithm is a step-by-step description of how to accomplish a task.
- An algorithm must be:
  - *precise:* specified in a clear and unambiguous way
  - *effective:* capable of being carried out

Programming

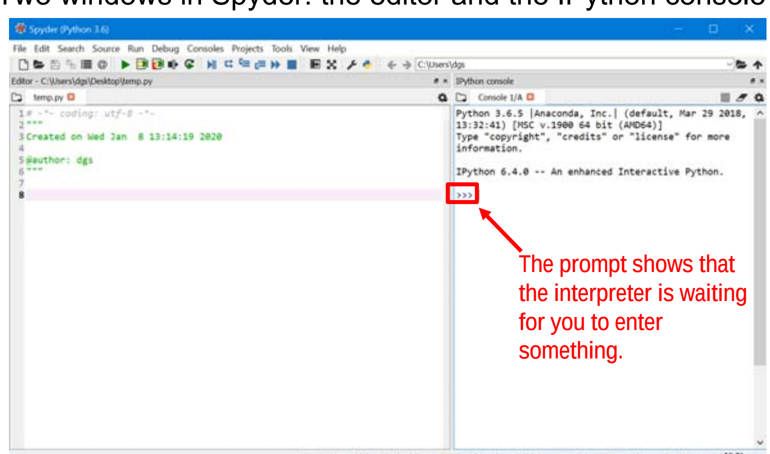
- Programming involves expressing an algorithm in a form that a computer can interpret.
- We will use the Python programming language.
  - one of many possible languages
  - widely used
  - relatively simple to learn
- The key concepts of the course transcend this language.
- You can use any version of Python **3**
  - ***not*** Python 2
  - see Lab 0 for details

Pre-Lecture
Getting Started With Python

Computer Science 111
Boston University

Interacting with Python

- We're using Python **3** (*not* 2).
  - see Lab 0 for how to install and configure Spyder
- Two windows in Spyder: the editor and the IPython console



Arithmetic in Python

- Numeric operators include:
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `**` exponentiation
  - `%` modulus: gives the remainder of a division

- Examples:

```
>>> 6 * 7
42
>>> 2 ** 4
16
>>> 17 % 2
1
>>> 17 % 3
_________
```

Arithmetic in Python (cont.)

- The operators follow the standard order of operations.
  - example: multiplication before addition
- You can use parentheses to force a different order.
- Examples:

```
>>> 2 + 3 * 5
____________
>>> (2 + 3) * 5
____________
```

Data Types

- Different kinds of values are stored and manipulated differently.
- Python *data types* include:
  - integers
    - example: `451`
  - floating-point numbers
    - numbers that include a decimal
    - example: `3.1416`

Data Types and Operators

- There are really <u>two sets</u> of numeric operators:
  - one for integers (ints)
  - one for floating-point numbers (floats)
- In most cases, the following rules apply:
  - if *at least one* of the operands is a float, the result is a float
  - if *both* of the operands are ints, the result is an int
- One exception: division!
- Examples:

Two Types of Division

- The `/` operator *always* produces a float result.
  - examples:

```
>>> 5 / 3
1.6666666666666667
>>> 6 / 3
```

____

Two Types of Division (cont.)

- There is a separate `//` operator for *integer* division.

```
>>> 6 // 3
2
```

- Integer division *discards* any fractional part of the result:

```
>>> 11 // 5
2
>>> 5 // 3
```

____

- Note that it does *not* round!

Another Data Type

- A *string* is a sequence of characters/symbols
  - surrounded by single or double quotes
  - examples: `"hello"` `'Picobot'`

Pre-Lecture
Program Building Blocks: Variables, Expressions, Statements

Computer Science 111
Boston University

Variables

- Variables allow us to store a value for later use:

```
>>> temp = 77
>>> (temp - 32) * 5 / 9
25.0
```

Expressions

- *Expressions* produce a value.
  - We *evaluate* them to obtain their value.
- They include:
  - *literals* ("hard-coded" values):
    ```
    3.1416
    'Picobot'
    ```
  - variables
    ```
    temp
    ```
  - combinations of literals, variables, and operators:
    ```
    (temp - 32) * 5 / 9
    ```

Evaluating Expressions with Variables

- When an expression includes variables, they are first replaced with their current value.
- Example:

```
(temp - 32) * 5 / 9
( 77  - 32) * 5 / 9
      45    * 5 / 9
           225   / 9
               25.0
```

Statements

- A *statement* is a command that carries out an action.
- A *program* is a sequence of statements.

```
quarters = 2
dimes = 3
nickels = 1
pennies = 4
cents = quarters*25 + dimes*10 + nickels*5 + pennies
print('you have', cents, 'cents')
```

Assignment Statements

- *Assignment statements* store a value in a variable.

```
temp = 20
```

- General syntax:

  variable = expression

= is known as the *assignment operator*

- Steps:
  1) evaluate the expression on the right-hand side of the =
  2) assign the resulting value to the variable on the left-hand side of the =
- Examples:

```
quarters = 10
quarters_val = 25 * quarters
               25 * 10
               250
```

Assignment Statements (cont.)

- We can change the value of a variable by assigning it a new value.

- Example:

`num1 = 100` `num2 = 120`	num1: 100	num2: 120
`num1 = 50`	num1:	num2: 120
`num1 = num2 * 2`	num1:	num2: 120
`num2 = 60`	num1:	num2:

Assignment Statements (cont.)

- An assignment statement does not create a permanent relationship between variables.

- ***You can only change the value of a variable by assigning it a new value!***

Assignment Statements (cont.)

- A variable can appear on both sides of the assignment operator!
- Example:

`sum = 13` `val = 30`	sum 13	val 30

`sum = sum + val`
`      13  +  30`
`        43`
	sum	val 30

`val = val * 2`	sum	val

Creating a Reusable Program

- Put the statements in a text file.

```
# a program to compute the value of some coins

quarters = 2       # number of quarters
dimes = 3
nickels = 1
pennies = 4

cents = quarters*25 + dimes*10 + nickels*5 + pennies
print('you have', cents, 'cents')
```

- Program file names should have the extension `.py`
  - example: `coins.py`

Print Statements

- print statements display one or more values on the screen
- Basic syntax:

  `print(`*expr*`)`

  or

  `print(`$expr_1$`, `$expr_2$`, `... $expr_n$`)`

  where each *expr* is an expression

- Steps taken when executed:
  1) the individual expression(s) are evaluated
  2) the resulting values are displayed on the same line, *separated by spaces*
- To print a blank line, omit the expressions:

```
print()
```

Print Statements (cont.)

- Examples:
  - first example:

```
print('the results are:', 15 + 5, 15 - 5)

      'the results are:'    20      10
```

output: `the results are: 20 10`

(note that the quotes around the string literal are *not* printed)

  - second example:

```
cents = 89
print('you have', cents, 'cents')
```

output: ______________________

Variables and Data Types

- The `type` function gives us the type of an expression:

```
>>> type('hello')
<class 'str'>
>>> type(5 / 2)
<class 'float'>
```

- Variables in Python do *not* have a fixed type.
  - examples:

```
>>> temp = 25.0
>>> type(temp)
<class 'float'>
>>> temp = 77
>>> type(temp)
______________
```

Python Basics

Computer Science 111
Boston University

What is the output of the following program?

```
x = 15
name = 'Picobot'
x = x // 2
print('name', x, type(x))
```

What about this program?

```
x = 15
name = 'Picobot'
x = 7.5
print(name, 'x', type(x))
```

What are the values of the variables after the following code runs?

```
x = 5
y = 6
x = y + 3
z = x + y
x = x + 2
```

x	y	z
5		
5	6	

Complete this table to keep track of the values of the variables!

What are the values of the variables after the following code runs?

```
x = 5
y = x ** 2
z = x % 3
x + 2
```

x	y	z

On paper, make a table for the values of your variables!

Pre-Lecture
Strings

Computer Science 111
Boston University

Strings: Numbering the Characters

- The position of a character within a string is known as its *index*.
- There are two ways of numbering characters in Python:
  - from left to right, starting from 0

```
 0 1 2 3 4
'Perry'
```

  - from right to left, starting from -1

```
 -5 -4 -3 -2 -1
'Perry'
```

  - `'P'` has an index of 0 or -5
  - `'y'` has an index of ____________

String Operations

- Indexing: *string*[*index*]

```
>>> name = 'Picobot'
>>> name[1]
'i'
>>> name[-3]
__________
```

- Slicing (extracting a substring): *string*[*start* : *end*]

```
>>> name[0:2]
'Pi'
>>> name[1:-1]
_________
>>> name[1:]
'icobot'
>>> name[:4]
_________
```

from this index

up to but ***not including*** this index

String Operations (cont.)

- Concatenation: *string1* + *string2*

```
>>> word = 'program'
>>> plural = word + 's'
>>> plural
'programs'
```

- Duplication: *string* * *num_copies*

```
>>> 'ho!' * 3
'ho!ho!ho!'
```

- Determining the length: `len`(*string*)

```
>>> name = 'Perry'
>>> len(name)
5
>>> len('')   # an empty string - no characters!
0
```

Skip-Slicing

- Slices can have a third number: *string*[*start* : *end* : *stride_length*]

```
s = 'boston university terriers'
     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
>>> s[0:8:2]
'bso '             # note the space at the end!
```

Skip-Slicing (cont.)

- Slices can have a third number: *string*[*start* : *end* : *stride_length*]

```
s = 'boston university terriers'
     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
>>> s[5:0:-1]
```

Pre-Lecture
Lists

Computer Science 111
Boston University

Lists

- Recall: A string is a sequence of characters.
  ```
  'hello'
  ```
- A list is a sequence of *arbitrary* values (the list's *elements*).
  ```
  [2, 4, 6, 8]
  ['CS', 'math', 'english', 'psych']
  ```
- A list can include values of different types:
  ```
  ['Star Wars', 1977, 'PG', [35.9, 460.9]]
  ```

List Ops == String Ops (more or less)

```
>>> majors = ['CS', 'math', 'english', 'psych']
>>> majors[2]
'english'
>>> majors[1:3]
____________
>>> len(majors)
____________
>>> majors + ['physics']
['CS', 'math', 'english', 'psych', 'physics']
>>> majors[::-2]
_______________
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Terriers'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
____________________
```

```
>>> info[2:]
____________________
>>> info[-1][-1]
460.9
>>> info[0][-4]
____________________
```

Strings and Lists

Computer Science 111
Boston University

What is the value of s after the following code runs?

```
s = 'abc'

s = ('d' * 3) + s

s = s[2:-2]
```

Fill in the blank to make the code print `'compute!'`

```
subject = 'computer science!'
verb = _______________
print(verb)
```

Skip-Slicing

- Slices can have a third number: *string*[*start* : *end* : *stride_length*]

```
s = 'boston university terriers'
       0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
>>> s[0:8:2]
'bso '             # note the space at the end!
>>> s[5:0:-1]
'notso'
>>> s[  :  :  ]   # what numbers do we need?
'viti'
>>> s[12:21:8] + s[21::3]     # what do we get?
```

What is the output of the following program?

```
mylist = [1, 2, [3, 4, 5]]
print(mylist[1], mylist[1:2])
```

Note the difference!

- For a string, both slicing and indexing produce a string:

```
>>> s = 'Terriers'
>>> s[1:2]
'e'
>>> s[1]
'e'
```

- For a list:
  - slicing produces a list
  - indexing produces a single element – may or may not be a list

```
>>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
>>> info[1:2]
[1977]
>>> info[1]
1977
>>> info[-1]
[35.9, 460.9]
```

```
>>> ___________________
35.9
```

How could you fill in the blank to produce `[105, 111]`?

```
intro_cs = [101, 103, 105, 108, 109, 111]

dgs_courses = ______________________
```

A. `intro_cs[2:3] + intro_cs[-1:]`

B. `intro_cs[-4] + intro_cs[5]`

C. `intro_cs[-4] + intro_cs[-1:]`

D. more than one of the above

E. none of the above

Extra practice from the textbook authors!

```
pi = [3,1,4,1,5,9]
L = [ 'pi', "isn't", [4,2] ]
M = 'You need parentheses for chemistry !'
     0   4   8   12  16  20  24  28  32
```

Part 1

What is `len(pi)`

What is `len(L)`

What is `len(L[1])`

What is `pi[2:4]`

What slice of `pi` is `[3,1,4]`

What slice of `pi` is `[3,4,5]`

Part 2

What is `L[0]`

These two are different!

What is `L[0:1]`

What is `L[0][1]`

What slice of `M` is `'try'`? is `'shoe'`?

What is `M[9:15]`

What is `M[::5]`

Extra! What are `pi[0]*(pi[1] + pi[2])` and `pi[0]*(pi[1:2] + pi[2:3])` ?

These two are different, too…

Pre-Lecture
Introduction to Functions

Computer Science 111
Boston University

Defining a Function

the function's name

x is the input or *parameter*

```
def triple(x):
    return 3*x
```

must indent

this line specifies what the function outputs (or *returns*) – in this case, 3 times the input

Multiple Lines, Multiple Parameters

```
def circle_area(diam):
    """ Computes the area of a circle
        with a diameter diam.
    """
    radius = diam / 2
    area = 3.14159 * (radius**2)
    return area

def rect_perim(l, w):
    """ Computes the perimeter of a rectangle
        with length l and width w.
    """
    return 2*l + 2*w
```

What is the output of this code?

```
def calculate(x, y):
    a = y
    b = x + 1
    return a + b + 3


print(calculate(3, 2))
```

x	y	a	b
3	2		

(complete the rest on the next slide)

The values in the function call are assigned to the parameters.

In this case, it's as if we had written:

```
x = 3
y = 2
```

What is the output of this code? (cont.)

x	y	a	b

```
def calculate(x, y):
    a = y
    b = x + 1
    return a + b + 3


print(calculate(3, 2))
```

The output/return value:

- is sent back to where the function call was made
- replaces the function call

The program picks up where it left off when the function call was made.

Intro. to Functions

Computer Science 111
Boston University

Functions With String Inputs

```
def undo(s):
    """ Adds the prefix "un" to the input s. """
    return 'un' + s

def redo(s):
    """ Adds the prefix "re" to the input s. """
    return 're' + s
```

- Examples:

```
>>> undo('plugged')

>>> undo('zipped')

>>> redo('submit')

>>> redo(undo('zipped'))
```

What is the output of this program?

```
def mystery1(t):
    return t[::-1]

def mystery2(t):
    return t[0] + t[-1]

s = 'terriers'
mystery1(s)
print(mystery2(s))
```

A. `ts`

B. `st`

C. `sreirret`
`ts`

D. `sreirret`
`st`

What is the output of this code?

```
def calculate(x, y):
    a = y
    b = x + 1
    return a * b - 3


print(calculate(4, 1))
```

x	y	a	b

Practice Writing a Function

- Write a function `avg_first_last(values)` that:
  - takes a list `values` that has at least one element
  - returns the average of the first and last elements
  - examples:

```
>>> avg_first_last([2, 6, 3])
2.5                     # average of 2 and 3 is 2.5
>>> avg_first_last([7, 3, 1, 2, 4, 9])
8.0                     # average of 7 and 9 is 8.0
```

```
def avg_first_last(values):
    first = _____________
    last = _____________
    return _______________
```

Returning vs. Printing

- Our previous function *returns* the result:

```
def avg_first_last(values):
    ...
    return ____________
```

- Would it be equivalent to print the result?

```
def avg_first_last(values):
    ...
    print(_____________)
```

- If the function prints instead of returning,
  you can't do something like this:

```
avg = avg_first_last([5, 7, 9, 10, 12])
print('The result is', avg)
```

More Practice

- Write a function `middle_elem(values)` that:
  - takes a list `values` that has at least one element
  - returns the element in the middle of the list
    - when there are two middle elements, return the one closer to the end
  - examples:

```
>>> middle_elem([2, 6, 3])
6
>>> middle_elem([7, 3, 1, 2, 4, 9])
2
```

```
def middle_elem(values):
    middle_index = ____________
    return ______________
```

Pre-Lecture
Making Decisions: Conditional Execution

Computer Science 111
Boston University

Conditional Execution

- Conditional execution allows your code to *decide* whether to do something, based on some condition.
  - example:

```
def abs_value(x):
    """ returns the absolute value of input x """
    if x < 0:
        x = -1 * x
    return x
```

  - examples of calling this function from the Shell:

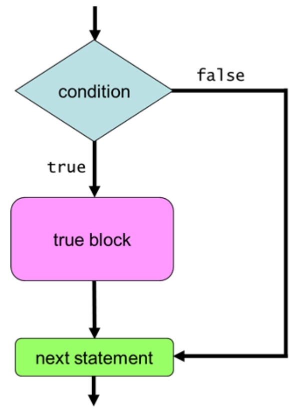
```
>>> abs_value(-5)
5
>>> abs_value(10)
__________________
```

Simple Decisions: `if` Statements

- Syntax:

```
if condition:
    true block
```

where:
- *condition* is an expression that is true or false
- *true block* is one or more indented statements



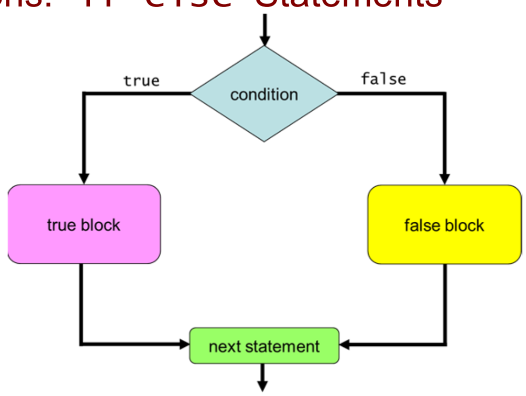
- Example:

```
def abs_value(x):
    if x < 0:
        x = -1 * x     # true block
    return x
```

Two-Way Decisions: `if-else` Statements

- Syntax:

```
if condition:
    true block
else:
    false block
```



- Example:

```
def pass_fail(avg):
    if avg >= 60:
        grade = 'pass'     # true block
    else:
        grade = 'fail'     # false block
    return grade
```

Expressing Simple Conditions

- Python provides a set of *relational operators* for making comparisons:

operator	name	examples
`<`	less than	`val < 10` `price < 10.99`
`>`	greater than	`num > 60` `state > 'Ohio'`
`<=`	less than or equal to	`average <= 85.8`
`>=`	greater than or equal to	`name >= 'Jones'`
`==` *(don't confuse with =)*	equal to	`total == 10` `letter == 'P'`
`!=`	not equal to	`age != my_age`

Boolean Values and Expressions

- A condition has one of two values: `True` or `False`.

```
>>> 10 < 20
True
>>> "Jones" == "Baker"
False
```

- `True` and `False` are *not* strings.
  - they are literals from the `bool` data type

```
>>> type(True)
<class 'bool'>
>>> type(30 > 6)
```

____

- An expression that evaluates to `True` or `False` is known as a *boolean expression.*

Forming More Complex Conditions

- Python provides *logical operators* for combining/modifying boolean expressions:

name	example and meaning
`and`	`age >= 18 and age <= 35` `True` *if both conditions are* `True`, *and* `False` *otherwise*

`or`	`age < 3 or age > 65`
	`True` *if one or both of the conditions are* `True`;
	`False` *if both conditions are* `False`

`not`	`not (grade > 80)` `True` *if the condition is* `False`, *and* `False` *if it is* `True`

A Word About Blocks

- A block can contain multiple statements.

```
def welcome(class):
    if class == 'frosh':
        print('Welcome to BU!')
        print('Have a great four years!')
    else:
        print('Welcome back!')
        print('Have a great semester!')
        print('Be nice to the frosh students.')
```

- A new block *begins* whenever we *increase* the amount of indenting.
- A block *ends* when we either:
  - reach a line with *less* indenting than the start of the block
  - reach the end of the program

Multi-Way Decisions

- The following function doesn't work.

avg	grade

```
def letter_grade(avg):
    if avg >= 90:
        grade = 'A'
    if avg >= 80:
        grade = 'B'
    if avg >= 70:
        grade = 'C'
    if avg >= 60:
        grade = 'D'
    else:
        grade = 'F'
    return grade
```

- example:

```
>>> letter_grade(95)
```

Multi-Way Decisions (cont.)

- Here's a fixed version:

avg	grade

```
def letter_grade(avg):
    if avg >= 90:
        grade = 'A'
    elif avg >= 80:
        grade = 'B'
    elif avg >= 70:
        grade = 'C'
    elif avg >= 60:
        grade = 'D'
    else:
        grade = 'F'
    return grade
```

- example:

```
>>> letter_grade(95)
```

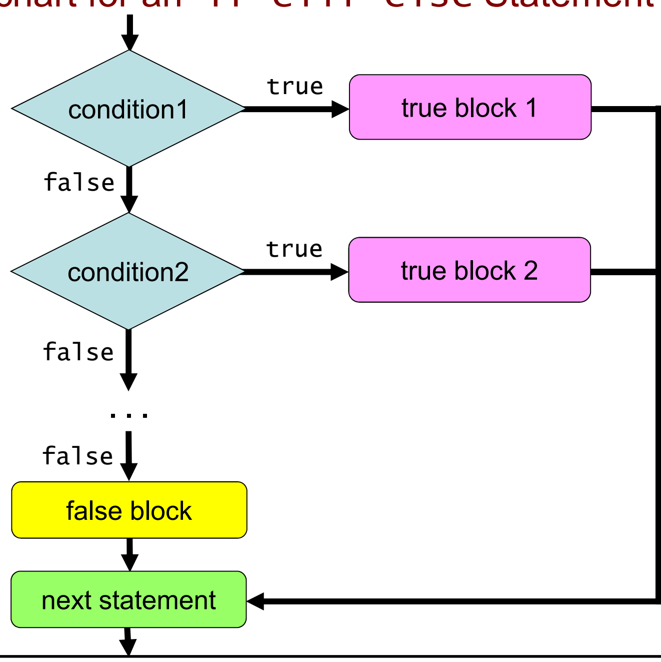
Multi-Way Decisions: `if-elif-else` Statements

- Syntax:

```
if condition1:
    true block for condition1
elif condition2:
    true block for condition2
elif condition3:
    true block for condition3
…
else:
    false block
```

- The conditions are evaluated in order. The true block of the *first* true condition is executed.
- If none of the conditions are true, the false block is executed.

Flowchart for an `if-elif-else` Statement



Making Decisions: Conditional Execution

Computer Science 111
Boston University

Making Decisions

- **One-way:** deciding whether or not to do something

```
if x < 0:
    print('x is negative')
    x = -1 * x
```

- **Two-way:** choosing among two options

```
if x < 0:
    print('x is negative')
    x = -1 * x
else:
    print('x is non-negative')
```

Recall: A Word About Blocks

- A block can contain multiple statements.

```
def welcome(class):
    if class == 'frosh':
        print('Welcome to BU!')
        print('Have a great four years!')
    else:
        print('Welcome back!')
        print('Have a great semester!')
        print('Be nice to the frosh students.')
```

- A new block *begins* whenever we *increase* the amount of indenting.
- A block *ends* when we either:
  - reach a line with *less* indenting than the start of the block
  - reach the end of the program

Nesting

- We can "nest" one conditional statement in the true block or false block of another conditional statement.

```
def welcome(class):
    if class == 'frosh':
        print('Welcome to BU!')
        print('Have a great four years!')
    else:
        print('Welcome back!')
        if class == 'senior':
            print('Have a great last year!')
        else:
            print('Have a great semester!')
        print('Be nice to the frosh students.')
```

What is the output of this program?

```
x = 5
if x < 15:
    if x > 8:
        print('one')
    else:
        print('two')
else:
    if x > 2:
        print('three')
```

What does this print? (note the changes!)

```
x = 5
if x < 15:
    if x > 8:
        print('one')
    else:
        print('two')
if x > 2:
    print('three')
```

What does this print? (note the new changes!)

```
x = 5
if x < 15:
    if x > 8:
        print('one')
else:
    print('two')
if x > 2:
    print('three')
```

How many lines does this print?

```
x = 5
if x == 8:
    print('how')
elif x > 1:
    print('now')
elif x < 20:
    print('wow')
print('cow')
```

How many lines does this print?

```
x = 5
if x == 8:
    print('how')
if x > 1:
    print('now')
if x < 20:
    print('wow')
print('cow')
```

What is the output of this code?

```
def mystery(a, b):
    if a == 0 or a == 1:
        return b
    return a * b


print(mystery(0, 5))
```

Common Mistake When Using and / or

```
def mystery(a, b):
    if a == 0 or 1:      # this is problematic
        return b
    return a * b

print(mystery(0, 5))
```

- When using and / or, both sides of the operator should be a boolean expression that could stand on its own.

boolean		*boolean*		*boolean*		*integer*
`a == 0`	`or`	`a == 1`		`a == 0`	`or`	`1`
	(do this)				*(don't do this)*	

- Unfortunately, Python *doesn't* complain about code like the problematic code above.
  - but it won't typically work the way you want it to!

Avoid Overly Complicated Code

- The following also involves decisions based on a person's age:

```
age = ...   # let the user enter his/her age
if age < 13:
    print('You are a child.')
elif age >= 13 and age < 20:
    print('You are a teenager.')
elif age >= 20 and age < 30:
    print('You are in your twenties.')
elif age >= 30 and age < 40:
    print('You are in your thirties.')
else:
    print('You are really old.')
```

- How could it be simplified?

Pre-Lecture
Variable Scope

Computer Science 111
Boston University

Local Variables

```
def mystery(x, y):
    b = x - y        # b is a local var of mystery
    return 2*b       # we can access b here

c = 7
mystery(5, 2)
print(b + c)         # we can't access b here
```

- When we assign a value to a variable inside a function, we create a *local variable*.
  - it "belongs" to that function
  - it can't be accessed outside of that function
- The parameters of a function are also limited to that function.
  - example: the parameters x and y above

Global Variables

```
def mystery(x, y):
    b = x - y
    return 2*b + c    # works, but not recommended

c = 7                 # c is a global variable
mystery(5, 2)
print(b + c)          # we can access c here
```

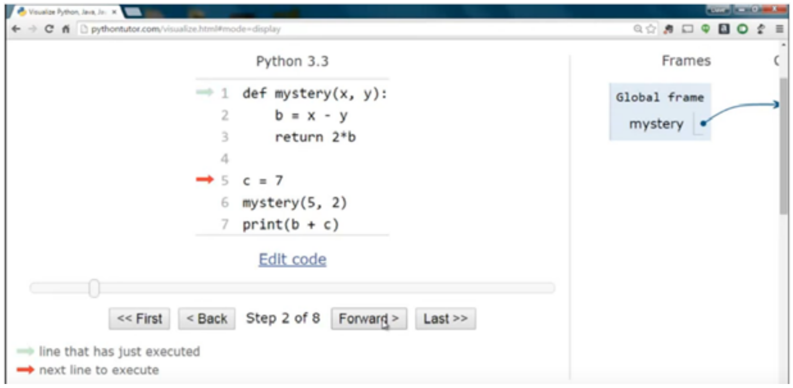
- When we assign a value to a variable *outside* of a function, we create a *global variable*.
  - it belongs to the *global scope*
- A global variable can be used anywhere in your program.
  - in code that is outside of any function
  - in code inside a function (but this is not recommended!)

Different Variables With the Same Name!

```
def mystery(x, y):
    b = x - y         # this b is local
    return 2*b        # we access the local b here

b = 1                 # this b is global
c = 7
mystery(5, 2)
print(b + c)          # we access the global b here
```

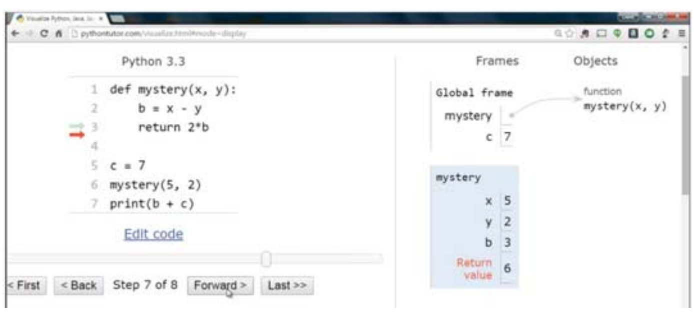
- The program above has two different variables called b.
  - one local variable
  - one global variable
- When this happens, the *local* variable has priority inside the function to which it belongs.

Python Tutor



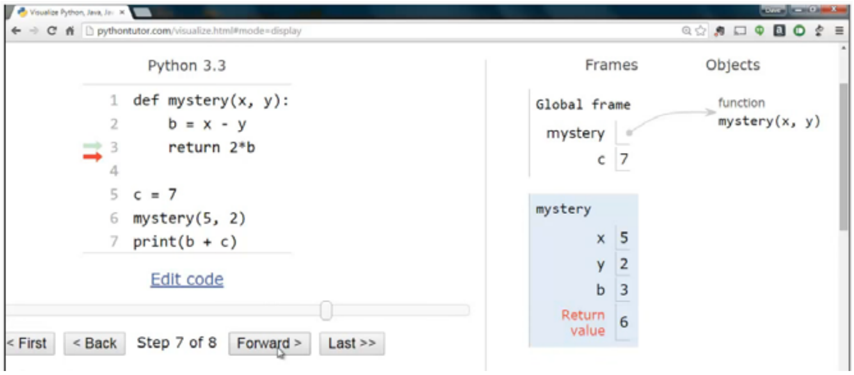
- Python Tutor allows us to trace through a program's execution.
  - use the *Forward* button
- The red arrow shows the next line to execute.
- The pale arrow shows the line that was just executed.

Frames for Variables



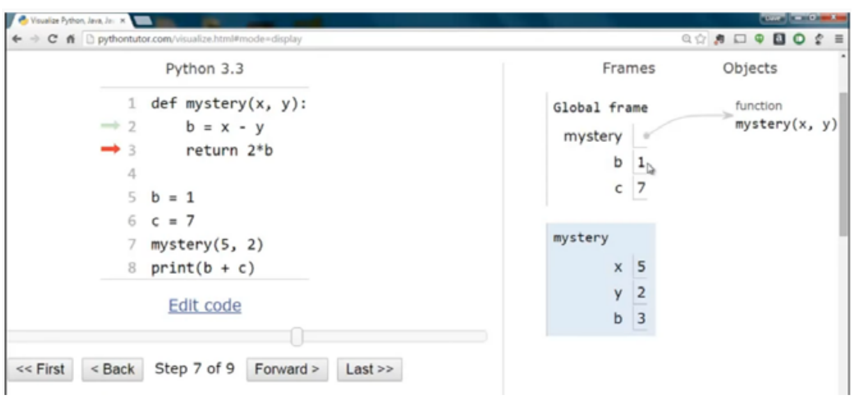
- Variables are stored in blocks of memory known as *frames.*
  - stored in a region of memory known as the *stack*
- Global variables are stored in the *global frame*.
- Each function call gets a frame for its local variables.
  - goes away when the function returns

Frames for Variables (cont.)



- Where is the error is this program?

Frames for Variables (cont.)



- What is the output of this fixed version of the program?

Pre-Lecture
Functions Calling Functions

Computer Science 111
Boston University

Finding the Distance Between Two Points

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

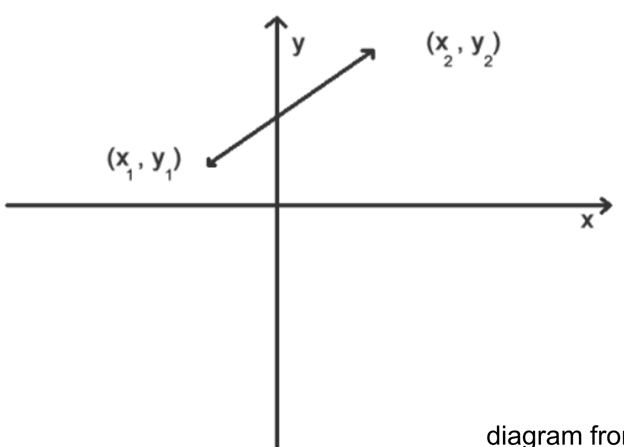


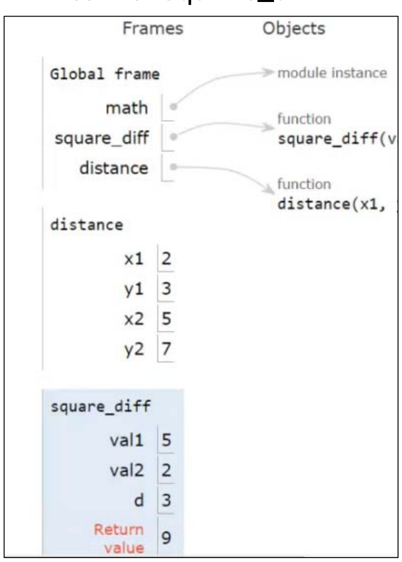
diagram from:
math.about.com

A Program for Computing Distance

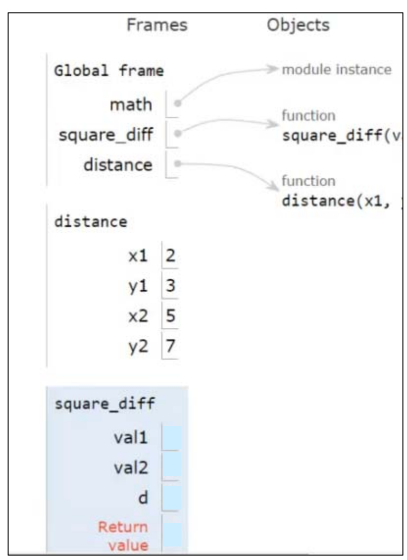
```
import math

def square_diff(val1, val2):
    """ returns the square of val1 – val2 """
    d = val1 - val2
    return d ** 2

def distance(x1, y1, x2, y2):
    """ returns the distance between two points
        in a plane with coordinates (x1, y1)
        and (x2, y2)
    """
    d = square_diff(x2, x1) + square_diff(y2, y1)
    dist = math.sqrt(d)
    return dist

dist = distance(2, 3, 5, 7)
print('distance between (2, 3) and (5, 7) is', dist)
```

Tracing the Program in Python Tutor

- stack frames during the 1st call to `square_diff`:



- fill in the stack frame for the 2nd call:



Variable Scope
Functions Calling Functions

Computer Science 111
Boston University

What is the output of this code?

```
def mystery2(a, b):
    x = a + b
    return x + 1

x = 8
mystery2(3, 2)
print(x)
```

What is the output of this code? (version 2)

```
def mystery2(a, b):
    x = a + b
    return x + 1

x = 8
mystery2(3, 2)
print(x)
```

A Note About Globals

- It's not a good idea to access a global variable inside a function.
  - for example, you shouldn't do this:

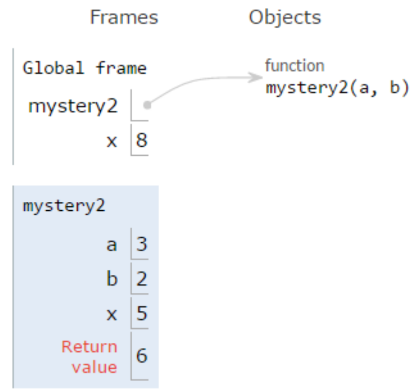
```
def average3(a, b):
    total = a + b + c    # accessing a global c
    return total/3

c = 8
print(average3(5, 7))
```

- Instead, you should pass it in as a parameter/input:

```
def average3(a, b, c):
    total = a + b + c    # accessing input c
    return total/3

c = 8
print(average3(5, 7, c))
```

Recall: Frames and the Stack

- Variables are stored in blocks of memory known as *frames.*
- Each function call gets a frame for its local variables.
  - goes away when the function returns
- Global variables are stored in the global frame.
- The *stack* is the region of the computer's memory in which the frames are stored.
  - thus, they are also known as *stack frames*



What is the output of this code?

```
def quadruple(y):
    y = 4 * y
    return y

y = 8
quadruple(y)

print(y)
```

How could we change this to see the return value?

```
def quadruple(y):
    y = 4 * y
    return y

y = 8
quadruple(y)

print(y)
```

What is the output of this program?

```
def demo(x):
   return x + f(x)

def f(x):
   return 11*g(x) + g(x//2)

def g(x):
   return -1 * x

print(demo(-4))
```

```
demo
  x = -4
  return -4 + f(-4)
```

```
f
  x = -4
  return 11*g(-4) + g(-4//2)
```

```
g                    frame for 1st call
  x = -4
  return -1 * x
```

```
g                    frame for 2nd call
  x =
  return -1 * x
```

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

foo

x	y

global

x	y

output

Full Trace of First Example

```
def quadruple(y):      # 3. local y = 8
    y = 4 * y          # 4. local y = 4 * 8 = 32
    return y           # 5. return local y's value
           32
y = 8               # 1. global y = 8
quadruple(y)        # 2. pass in global y's value
                    # 6. return value is thrown away!
print(y)            # 7. print global y's value,
                    #    which is unchanged!
```

You ***can't*** change the value of a variable by passing it into a function!

Pre-Lecture
A First Look at Recursion

Computer Science 111
Boston University

Functions Calling Themselves: *Recursion!*

```
def fac(n):
    if n <= 1:
        return 1
    else:
        return n * fac(n - 1)
```

- Recursion solves a problem by reducing it to a *simpler* or *smaller* problem *of the same kind*.
  - the function calls itself to solve the smaller problem!
- We take advantage of *recursive substructure*.
  - the fact that we can define the problem *in terms of itself*
    `n! = n * (n-1)!`

Functions Calling Themselves: *Recursion!* (cont.)

```
def fac(n):
    if n <= 1:           } base case
        return 1
    else:
        return n * fac(n - 1)    } recursive case
```

- One recursive call leads to another...

```
fac(5) = 5 * fac(4)
       = 5 * 4 * fac(3)
       = ...
```

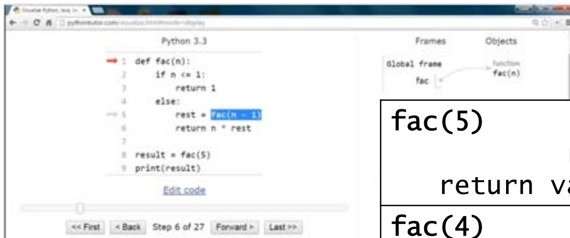
- We eventually reach a problem that is small enough to be solved directly – a *base case.*
  - stops the recursion
  - make sure that you always include one!

Alternative Version of `fac(n)`

```
def fac(n):
    if n <= 1:
        return 1

    else:
        rest = fac(n - 1)
        return n * rest
```

- Many people find this easier to read/write/understand.
- Storing the result of the recursive call will occasionally make the problem easier to solve.
- It also makes your recursive functions easier to trace and debug.
- ***We highly recommend that you take this approach!***

Tracing Recursion in Python Tutor



Fill in the stack frames!

fac(5)	n: rest: return value:
fac(4)	n: rest: return value:
fac(3)	n: rest: return value:
fac(2)	n: rest: return value:
fac(1)	n: rest: return value:

Pre-Lecture
Using Recursion, Part I

Computer Science 111
Boston University

Recursively Processing a List or String

- Sequences are recursive!
  - a string is a character followed by a string...
  - a list is an element followed by a list...
- Let `s` be the sequence (string or list) that we're processing.
- Do one step!
  - use `s[0]` to access the initial element
  - do something with it
- Delegate the rest!
  - use `s[1:]` to get the rest of the sequence.
  - make a recursive call to process it!

Recursively Finding the Length of a String

```
def mylen(s):
    """ returns the number of characters in s
        input s: an arbitrary string
    """
    if                          # base case


    else:
```

- Ask yourself:

(base case) When can I determine the length of s *without* looking at a smaller string?

(recursive substructure) How could I use the length of ***anything smaller*** than s to determine the length of s?

How recursion works...

```
mylen('wow')
  s = 'wow'
  len_rest = mylen('ow')
  return
```

```
mylen('ow')
  s =
  len_rest =
  return
```

4 different stack frames, each with its own s and `len_rest`

The final result gets built up *on the way back* from the base case!

Recursively Raising a Number to a Power

```
def power(b, p):
    """ returns b raised to the p power
        inputs: b is a number (int or float)
                p is a non-negative integer
    """
    if                    # base case


    else:
```

- Ask yourself:

(base case) When can I determine b^p *without* determining a smaller power?

(recursive substructure) How could I use ***anything smaller*** than b^p to determine b^p?

How recursion works...

```
power(3, 3)
  b = 3, p = 3
  pow_rest = power(3, 2)
  return
```

```
power(3, 2)
  b = 3, p = 2
  pow_rest =
  return
```

4 different stack frames, each with its own b, p and `pow_rest`

The final result gets built up *on the way back* from the base case!

A First Look at Recursion

Computer Science 111
Boston University

Recall: Functions Calling Themselves: *Recursion!*

```
def fac(n):
    if n <= 1:
        return 1                   } base case
    else:
        fac_rest = fac(n - 1)      } recursive case
        return n * fac_rest
```

- One recursive call leads to another...
- We eventually reach a problem that is small enough to be solved directly – a *base case.*
  - stops the recursion
  - make sure that you always include one!

Let Recursion Do the Work For You!

```
def fac(n):
    if n <= 1:
        return 1
    else:
        fac_rest = fac(n-1)
        return n * fac_rest
```

You handle the base case – the easiest case!

Recursion does <u>almost</u> all of the rest of the problem!

You specify one step at the end.

How many times will `mylen()` be called?

```
def mylen(s):
    if s == '':             # base case
        return 0
    else:                   # recursive case
        len_rest = mylen(s[1:])
        return len_rest + 1

print(mylen('step'))
```

```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

```
mylen('ep')
  s = 'ep'
  len_rest = mylen('p')
```

```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```

Fill in the rest of the stack frames!

What is the output of this program?

```
def foo(x, y):
  if x <= y:
     return y
  else:
     return x + foo(x-2,y+1)

print(foo(9, 2))
```

Fill in the stack frames!
(use as many as you need)

foo(9, 2)	x: y:
foo( )	x: y:
	x: y:
	x: y:
	x: y:

More Recursion!

Computer Science 111
Boston University

Designing a Recursive Function

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*
2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
3. Solve the smaller problem using a recursive call!
   - store its result in a variable
4. Do your one step.
   - build your solution from the result of the recursive call
   - use *concrete cases* to figure out what you need to do

A Recursive Function for Counting Vowels

```
def num_vowels(s):
    """ returns the number of vowels in s
        input s: a string of lowercase letters
    """
    # We'll design this together!
```

- Examples of how it should work:

```
>>> num_vowels('compute')
3
>>> num_vowels('now')
1
```

- The `in` operator will be helpful:

```
>>> 'fun' in 'function'
True
>>> 'i' in 'team'
False
```

Design Questions for `num_vowels()`

(base case) When can I determine the # of vowels in `s` *without* looking at a smaller string?

(recursive substructure) How could I use the solution to ***anything smaller*** than `s` to determine the solution to `s`?

a

r

```
total # of vowels
=
```

```
total # of vowels
=
```

How Many Lines of This Function Have a Bug?

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        num_rest = num_vowels(s[0:])
        if s[0] in 'aeiou':
            return 1
        else:
            return 0
```

After you make your group vote, fix the function!

What value is eventually assigned to num_rest? (i.e., what does the recursive call return?)

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        num_rest = num_vowels(________________)
        ...


num_vowels('aha')
```

```
num_vowels('aha')
  s = 'aha'
  num_rest = ??
```

How recursion works...

```
num_vowels('aha')
  s = 'aha'
  num_rest = num_vowels('ha')
```

```
num_vowels(       )
  s =
  num_rest =
```

Debugging Technique: Adding Temporary prints

```
def num_vowels(s):
    print('beginning call for', s)
    if s == '':
        print('base case returns 0')
        return 0
    else:
        num_rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            print('call for', s, 'returns', 1 + num_rest)
            return 1 + num_rest
        else:
            print('call for', s, 'returns', 0 + num_rest)
            return 0 + num_rest
```

Pre-Lecture
Using Recursion, Part II

Computer Science 111
Boston University

Recursively Finding the Largest Element in a List

- `mymax(values)`
  - input: a *non-empty* list
  - returns: the largest element in the list
- examples:

```
>>> mymax([5, 8, 10, 2])
10
>>> mymax([30, 2, 18])
30
```

Design Questions for `mymax()`

(base case) When can I determine the largest element in a list without needing to look at a smaller list?

(recursive case) How could I use the largest element in a smaller list to determine the largest element in the entire list?

```
list1 = [30, ______]
```
largest element = 18

```
list2 = [5, ______]
```
largest element = 10

```
mymax(list1) → _______
mymax(list2) → _______
```

1. compare the first element to largest element in the rest of the list
2. return the larger of the two

Let the recursive call handle the rest of the list!

Recursively Finding the Largest Element in a List

```
def mymax(values):
    """ returns the largest element in a list
        input: values is a *non-empty* list
    """
    if                                  # base case


    else:                               # recursive case
```

Tracing Recursion in Python Tutor

Fill in the stack frames!

```
mymax([10, 12, 5, 8])
       values: [10, 12, 5, 8]
  max_in_rest:
 return value:
```

```
mymax([12, 5, 8])
       values: [12, 5, 8]
  max_in_rest:
 return value:
```

```
mymax(              )
       values:
  max_in_rest:
 return value:
```

```
mymax(              )
       values:
  max_in_rest:
 return value:
```

Practicing Recursive Design

Computer Science 111
Boston University

Recall: Recursively Finding the Largest Element in a List

- `mymax(vals)`
  - input: a *non-empty* list
  - returns: the largest element in the list
- examples:

```
>>> mymax([5, 8, 10, 2])
result: 10

>>> mymax([30, 2, 18])
result: 30
```

How many times will `max_rest` be returned?

```
def mymax(vals):
    if len(vals) == 1:          # base case
        return vals[0]
    else:                       # recursive case
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest     # how many times?

print(mymax([5, 30, 10, 8]))
```

How recursion works...

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax(__________)

mymax(             )
  vals =
  max_rest = mymax(__________)
```

Recall: Designing a Recursive Function

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*
2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
3. Solve the smaller problem using a recursive call!
   - store its result in a variable
4. Do your one step.
   - build your solution on the result of the recursive call
   - use *concrete cases* to figure out what you need to do

Recursively Replacing Characters in a String

- `replace(s, old, new)`
  - inputs: a string `s`
    two characters, `old` and `new`
  - returns: a version of `s` in which all occurrences of `old` are replaced by `new`
- examples:

```
                old     new
>>> replace('boston', 'o', 'e')
result: 'besten'
```

```
'boston'
'besten'
```

```
>>> replace('banana', 'a', 'o')
result: 'bonono'

>>> replace('mama', 'm', 'd')

result: ____________
```

Design Questions for `replace()`

(base case) When can I determine the "replaced" version of s *without* looking at a smaller string?

(recursive case) How could I use the "replaced" version of a smaller string to get the "replaced" version of s?

s1 = 'a[ ]'

s2 = 'r[ ]'

```
replace(s1, 'a', 'o')
=
```

```
replace(s2, 'e', 'i')
=
```

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return _____
    else:
        # recursive call handles rest of string
        repl_rest = replace(______, old, new)

        # do your one step!
        if ________________:
            return ________________
        else:
            return ________________
```

Use our concrete cases!

replace('always', 'a', 'o')
return 'o' + soln to rest of string

replace('recurse!', 'e', 'i')
return 'r' + soln to rest of string

More Recursive Design!

Computer Science 111
Boston University

Removing Vowels From a String

- `remove_vowels(s)` - removes the vowels from the string `s`, returning its "vowel-less" version!

```
>>> remove_vowels('recursive')
'rcrsv'
>>> remove_vowels('vowel')
'vwl'
```

- Can we take the usual approach to recursive string processing?
  - base case: empty string
  - delegate `s[1:]` to the recursive call
  - we're responsible for handling `s[0]`

How should we fill in the blanks?

```
def remove_vowels(s):
    if s == '':         # base case
        return __________
    else:               # recursive case
        rem_rest = __________________

        # do our one step!
        ...
```

Consider this original call…

```
def remove_vowels(s):
    if s == '':
        return ________
    else:
        rem_rest = ______________________

        # do our one step!
        ...
remove_vowels('recurse')
```

What value is eventually assigned to `rem_rest`? (i.e., what does the recursive call return?)

```
def remove_vowels(s):
    if s == '':
        return ________
    else:
        rem_rest = ______________________

        # do our one step!
        ...

remove_vowels('recurse')
```

```
remove_vowels('recurse')
  s = 'recurse'
  rem_rest = ??
```

What should happen after the recursive call?

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
```

- In our one step, we take care of `s[0]`.
  - we build the solution to the larger problem on the solution to the smaller problem (in this case, `rem_rest`)
  - does what we do depend on the value of `s[0]`?

Consider Concrete Cases

```
remove_vowels('after')      # s[0] is a vowel
```

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

```
remove_vowels('recurse')    # s[0] is not a vowel
```

- what is its solution? '
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

remove_vowels()

```
def remove_vowels(s):
    """ returns the "vowel-less" version of s
        input s: an arbitrary string
    """
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        if s[0] in 'aeiou':

            return ________________
        else:

            return ________________
```

More Recursive Design! `rem_all()`

- `rem_all(elem, values)`
  - inputs: an arbitrary value (`elem`) and a list (`values`)
  - returns: a version of `values` in which *all* occurrences of `elem` in `values` (if any) are removed

```
>>> rem_all(10, [3, 5, 10, 7, 10])
[3, 5, 7]
```

More Recursive Design! `rem_all()`

- Can we take the usual approach to processing a list recursively?
  - base case: empty list
  - delegate `values[1:]` to the recursive call
  - we're responsible for handling `values[0]`
- What are the possible cases for our part (`values[0]`)?
  - does what we do with our part depend on its value?

Consider Concrete Cases

`rem_all(10, [3, 5, 10, 7, 10])` # first value is *not* a match

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

`rem_all(10, [10, 3, 5, 10, 7])` # first value *is* a match

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

rem_all()

```
def rem_all(elem, values):
    """ removes all occurrences of elem from values
    """
    if values == []:
        return _______
    else:
        rem_rest = rem_all(______, ________)

        if _________________:
            return _________________
        else:
            return _________________
```

Pre-Lecture
List Comprehensions

Computer Science 111
Boston University

Generating a Range of Integers

- `range(low, high)`: allows us to work with the range of integers from `low` to `high-1`
  - to see the result produced by `range()` use the `list()` function
  - if you omit `low`, the range will start at 0
- Examples:

```
>>> list(range(3, 10))
[3, 4, 5, 6, 7, 8, 9]

>>> list(range(20, 30))
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]

>>> list(range(8))
```

____

List Comprehensions

[*expression* for *variable* in *sequence*]

this *"runner"* variable can have ***any*** name...

x takes on each value

```
>>> [3*x for x in [0,1,2,3,4,5]]
```

and 3*x is output for each one

```
[0, 3, 6, 9, 12, 15]
```

Examples of LCs

```
                     [1, 2, 3, 4]
>>> [111*n for n in range(1, 5)]
[111, 222, 333, 444]

>>> [s[0] for s in ['python', 'is', 'fun!']]

______________________
```

List Comprehensions (LCs)

- Syntax:

  [*expression* for *variable* in *sequence*]

  or

  [*expression* for *variable* in *sequence* if *boolean*]

- Examples:

```
                       [0, 1, 2, 3, 4, 5]
>>> [2*x for x in range(6) if x % 2 == 0]
[0, 4, 8]

>>> [y for y in ['how', 'now', 'brown'] if len(y) == 3]

____________________________________
```

List Comprehensions

Computer Science 111
Boston University

Another Useful Built-In Function

- `sum(list)`: computes & returns the sum of a list of numbers

```
>>> sum([4, 10, 2])
16
```

Recall: List Comprehensions

[*expression* for *variable* in *sequence*]

this *"runner"* variable can have ***any*** name...

x takes on each value

```
>>> [3*x for x in [0,1,2,3,4,5]]
```

and 3*x is output for each one

```
[0, 3, 6, 9, 12, 15]
```

More Examples

```
>>> [n - 2 for n in range(10, 15)]

>>> [s[-1]*2 for s in ['go', 'terriers!']]

>>> [z for z in range(6)]

>>> [z for z in range(6) if z % 2 == 1]

>>> [z % 4 == 0 for z in [4, 5, 6, 7, 8]]

>>> [1 for x in [4, 5, 6, 7, 8] if x % 4 == 0]

>>> sum([1 for x in [4, 5, 6, 7, 8] if x % 4 == 0])
```

What is the output of this code?

```
lc = [x for x in range(5) if x**2 > 4]

print(lc)
```

LC Puzzles! – Fill in the blanks

```
>>> [__________ for x in range(4)]
[0, 14, 28, 42]

>>> [_________ for s in ['boston', 'university', 'cs']]
['bos', 'uni', 'cs']

>>> [__________ for c in 'compsci']
['cc', 'oo', 'mm', 'pp', 'ss', 'cc', 'ii']

>>> [__________ for x in range(20, 30) if ____________]
[20, 22, 24, 26, 28]

>>> [__________ for w in ['I', 'like', 'ice', 'cream']]
[1, 4, 3, 5]
```

LCs vs. Raw Recursion

```
# raw recursion
def mylen(seq):
   if seq == '' or seq == []:
       return 0
   else:
       len_rest = mylen(seq[1:])
       return 1 + len_rest

# using an LC
def mylen(seq):
   lc = [1 for x in seq]
   return sum(lc)

# here's a one-liner!
def mylen(seq):
   return sum([1 for x in seq])
```

LCs vs. Raw Recursion (cont.)

```
# raw recursion
def num_vowels(s):
    if s == '':
        return 0
    else:
        num_in_rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + num_in_rest
        else:
            return 0 + num_in_rest

# using an LC
def num_vowels(s):
   lc = [1 for c in s if c in 'aeiou']
   return sum(lc)

# here's a one-liner!
def num_vowels(s):
   return sum([1 for c in s if c in 'aeiou'])
```

What list comprehension(s) would work here?

```
def num_odds(values):
    """ returns the number of odd #s in a list
        input: a list of 0 or more integers
    """
    lc = ____________________________________
    return sum(lc)
```

Fill in the Blanks

```
def avg_len(wordlist):
    """ returns the average length of the strings
        in wordlist as a float
        input: a list of 1 or more strings
    """
    lc = [_________ for ____ in __________]

    return ________ / _________



                     12           6
>>> avg_len(['commonwealth', 'avenue'])
9.0
                 4       4      3      4     2
>>> avg_len(['keep','calm','and','code','on'])
3.4
```

Pre-Lecture
`max()`, `min()`, and Lists of Lists

Computer Science 111
Boston University

`max()` and `min()`

- `max(values)`: returns the largest value in a list of values
```
>>> max([4, 10, 2])
10
>>> max(['all', 'students', 'love', 'recursion'])
'students'
```

- `min(values)`: returns the smallest value in a list of values
```
>>> min([4, 10, 2])
2
>>> min(['all', 'students', 'love', 'recursion'])
```
____

Lists of Lists

- Recall that the elements of a list can themselves be lists:

```
[[124, 'Jaws'], [150, 'Lincoln'], [115, 'E.T.']]
```

- When you apply `max()`/`min()` to a list of lists, the comparisons are based on the ***first*** element of each sublist:

```
>>> max([[124, 'Jaws'], [150, 'Lincoln'], [115, 'E.T.']])
[150, 'Lincoln']

>>> min([[124, 'Jaws'], [150, 'Lincoln'], [115, 'E.T.']])
```

____

Problem Solving Using LCs and Lists of Lists

- Sample problem: finding the ***shortest*** word in a list of words.

```
words = ['always', 'come', 'to', 'class']
```

1. Use a list comprehension to build a list of lists:

```
scored_words = [[len(w), w] for w in words]
# for the above words, we get:
```

2. Use min/max to find the correct sublist:

```
min_pair = min(scored_words)
# for the above words, we get:
```

3. Use indexing to extract the desired value from the sublist:

```
min_pair[1]
```

Problem Solving Using LCs and Lists of Lists (cont.)

- Here's a function that works for an arbitrary list of words:

```
def shortest_word(words):
    """ returns the shortest word from the input
        list of words
    """
    scored_words = [[len(w), w] for w in words]
    min_pair = min(scored_words)
    return min_pair[1]
```

Pre-Lecture
ASCII Codes and the Caesar Cipher

Computer Science 111
Boston University

ASCII
American Standard Code for Information Interchange

- Strings are sequences of characters. `'hello'`
- Individual characters are actually stored as integers.
- ASCII specifies the mapping between characters and integers.

character	ASCII value
'A'	65
'B'	66
'C'	67
...	
'a'	97
'b'	98
'c'	99
...	

Converting Between Characters and Numbers

ASCII values

abcdefghijklmnopqrstuvwxyz

97 99 101 103 105 107 109 111 113 115 117 119 122

ABCDEFGHIJKLMNOPQRSTUVWXYZ

65 67 69 71 73 75 77 79 81 83 85 87 90

Conversion functions

`ord(c)` — ***input:*** a one-character string, c
returns: an integer, the ASCII value of c

`chr(n)` — ***input:*** an integer ASCII value
returns: the one-character string for that ASCII value

Examples

```
>>> ord('e')
101

>>> ord('G')
71
```

```
>>> chr(101)
'e'

>>> chr(71)
'G'
```

Encryption

original message → *encrypted message*

```
'my password is foobar'  →  'pb sdvvzrug lv irredu'
```

Caesar Cipher Encryption

- Each letter is shifted/"rotated" forward by some number of places.

  `abcdefghijklmnopqrstuvwxyz`

- Example: a shift of 3

```
'a' → 'd'            'A' → 'D'
'b' →                'B' →
'c' →                'C' →
```

  etc.

  original message *encrypted message*

  `'my password is foobar'` → `'pb sdvvzrug lv irredu'`

- Non-alphabetic characters are left alone.

- We "wrap around" as needed.

```
'x' → 'a'            'X' → 'A'
'y' →                'Y' →
```

  etc.

Implementing a Shift in Python

ASCII values														
abcdefghijklmnopqrstuvwxyz	97	99	101	103	105	107	109	111	113	115	117	119	122	
ABCDEFGHIJKLMNOPQRSTUVWXYZ	65	67	69	71	73	75	77	79	81	83	85	87	90	

- `ord()` and addition gives the ASCII code of the shifted letter:

```
>>> ord('b')
98
>>> ord('b') + 3      # in general, ord(c) + shift
101
```

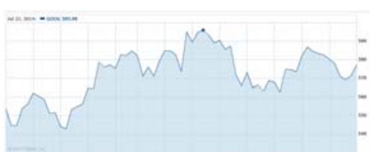
- `chr()` turns it back into a letter:

```
>>> chr(ord('b') + 3)
'e'
```

`max()`, `min()`, and Lists of Lists; ASCII Codes and the Caesar Cipher

Computer Science 111
Boston University

Finding a Maximum Stock Price



```
>>> max([578.7, 596.0, 586.9])
           'jun'    'jul'    'aug'
result: 596.0
```

- To determine the month in which the max occurred, use a *list of lists*!

```
>>> max([[578.7, 'jun'], [596.0, 'jul'], [586.9, 'aug']])
result:

>>> max([['jun', 578.7], ['jul', 596.0], ['aug', 586.9]])
result:
```

Finding the Best Scrabble Word

- Assume we have:
  - a list of possible Scrabble words
    ```
    words = ['aliens', 'zap', 'hazy', 'code']
    ```
  - a `scrabble_score()` function like the one from PS 2

- To find the best word:
  - form a ***list of lists*** using a list comprehension
    ```
    scored_words = [[scrabble_score(w), w] for w in words]
    ## for the above words, we get the following:
    ```
  - use `max()` to get the best `[score, word]` sublist:
    ```
    bestpair = max(scored_words)
    ## for the above words, we get the following:
    ```
  - use indexing to extract the word: `bestpair[1]`

best_word()

```
def best_word(words):
    """ returns the word from the input list of words
        with the best Scrabble score
    """
    scored_words = [[scrabble_score(w), w] for w in words]
    bestpair = max(scored_words)
    return bestpair[1]
```

How Would Your Complete This Function?

```
def longest_word(words):
    """ returns the string that is the longest
        word from the input list of words
    """
    scored_words = ___________________________
    bestpair = max(scored_words)
    return ________________
```

Recall: Caesar Cipher Encryption

- Each letter is shifted/"rotated" forward by some number of places.

  abcdefghijklmnopqrstuvwxyz

- Example: a shift of 3
  `'a'` → `'d'`

Caesar Cipher in PS 3

- You will write an `encipher` function:

```
>>> encipher('hello!', 1)
result: 'ifmmp!'
>>> encipher('hello!', 2)
result: 'jgnnq!'
>>> encipher('hello!', 4)
result: 'lipps!'
```

- "Wrap around" as needed.
  - upper-case letters wrap to upper; lower-case to lower

```
>>> encipher('XYZ xyz', 3)
result: 'ABC abc'
```

What Should This Code Output?

```
secret = encipher('Caesar? Wow!', 5)
print(secret)
```

Caesar Cipher with a Shift/Rotation of 13

- `'a'` → `'n'` `'b'` → `'o'` `'c'` → `'p'` etc.

  `'n'` → `'a'` `'o'` → `'b'` `'p'` → `'c'`

- Using `chr()` and `ord()`:

```
>>> chr(ord('a') + 13)
result: 'n'
>>> chr(ord('P') + 13 - 26)      # wrap around!!
result: 'C'
```

- Can use the following to determine if `c` is lower-case:

```
    if 'a' <= c <= 'z':
```

- Can use the following to determine if `c` is upper-case:

```
    if 'A' <= c <= 'Z':
```

Caesar Cipher with a Shift/Rotation of 13

```
def rot13(c):
    """ rotate c forward by 13 characters,
        wrapping as needed; only letters change
    """
    if 'a' <= c <= 'z':          # lower-case
        new_ord = ord(c) + 13
        if new_ord > ord('z'):
            new_ord = ______________
    elif 'A' <= c <= 'Z':        # upper-case
        new_ord = ord(c) + 13
        if __________________:
            _______________________
    else:                        # non-alpha
        __________________

    return _______________
```

Deciphering an Enciphered Text

- You will write a function for this as well.

```
>>> decipher('Bzdrzq bhogdq? H oqdedq Bzdrzq rzkzc.')
result: 'Caesar cipher? I prefer Caesar salad.'

>>> decipher('Bomebcsyx sc pexnkwoxdkv')
result: 'Recursion is fundamental'

>>> decipher('gv vw dtwvg')
???
```

- `decipher` only takes a string.
  - no shift/rotation amount is given!
- How can it determine the correct "deciphering"?

```
decipher('gv vw dtwvg')
```

All possible decipherings

```
gv vw dtwvg
hw wx euxwh
ix xy fvyxi
jy yz gwzyj
kz za hxazk
la ab iybal
mb bc jzcbm
nc cd kadcn
od de lbedo
pe ef mcfep
qf fg ndgfq
rg gh oehgr
sh hi pfihs
ti ij qgjit
uj jk rhkju
vk kl silkv
wl lm tjmlw
xm mn uknmx
yn no vlony
zo op wmpoz
ap pq xnqpa
bq qr yorqb
cr rs zpsrc
ds st aqtsd
et tu brute
fu uv csvuf
```

Score them all

```
[0, 'gv vw dtwvg'],
[2, 'hw wx euxwh'],
[2, 'ix xy fvyxi'],
[0, 'jy yz gwzyj'],
[2, 'kz za hxazk'],
[4, 'la ab iybal'],
[0, 'mb bc jzcbm'],
[1, 'nc cd kadcn'],
[4, 'od de lbedo'],
[3, 'pe ef mcfep'],
[0, 'qf fg ndgfq'],
[2, 'rg gh oehgr'],
[2, 'sh hi pfihs'],
[0, 'cr rs zpsrc'],
[1, 'ds st aqtsd'],
[4, 'et tu brute'],
[3, 'fu uv csvuf']
```

Need to quantify "Englishness" so that max() will yield the "most English" phrase.

```
decipher('gv vw dtwvg')
```

All possible decipherings

```
gv vw dtwvg
hw wx euxwh
ix xy fvyxi
jy yz gwzyj
kz za hxazk
la ab iybal
mb bc jzcbm
nc cd kadcn
od de lbedo
pe ef mcfep
qf fg ndgfq
rg gh oehgr
sh hi pfihs
ti ij qgjit
uj jk rhkju
vk kl silkv
wl lm tjmlw
xm mn uknmx
yn no vlony
zo op wmpoz
ap pq xnqpa
bq qr yorqb
cr rs zpsrc
ds st aqtsd
et tu brute
fu uv csvuf
```

Score them all

max!

```
[0, 'gv vw dtwvg'],
[2, 'hw wx euxwh'],
[2, 'ix xy fvyxi'],
[0, 'jy yz gwzyj'],
[2, 'kz za hxazk'],
[4, 'la ab iybal'],
[0, 'mb bc jzcbm'],
[1, 'nc cd kadcn'],
[4, 'od de lbedo'],
[3, 'pe ef mcfep'],
[0, 'qf fg ndgfq'],
[2, 'rg gh oehgr'],
[2, 'sh hi pfihs'],
[3, 'ti ij qgjit'],
[2, hkju'],
ilkv'],
mlw'],
nmx'],
ny'],
oz'],
xnqpa'],
'bq qr yorqb'],
[0, 'cr rs zpsrc'],
[1, 'ds st aqtsd'],
[4, 'et tu brute'],
[3, 'fu uv csvuf']
```

A score based on # of vowels doesn't work for this phrase.

```
decipher('gv vw dtwvg')
```

All possible decipherings

```
gv vw dtwvg
hw wx euxwh
ix xy fvyxi
jy yz gwzyj
kz za hxazk
la ab iybal
mb bc jzcbm
nc cd kadcn
od de lbedo
pe ef mcfep
qf fg ndgfq
rg gh oehgr
sh hi pfihs
ti ij qgjit
uj jk rhkju
vk kl silkv
wl lm tjmlw
xm mn uknmx
yn no vlony
zo op wmpoz
ap pq xnqpa
bq qr yorqb
cr rs zpsrc
ds st aqtsd
et tu brute
fu uv csvuf
```

Scores

```
[6.9e-05, 'gv vw dtwvg'],
[3.6e-05, 'hw wx euxwh'],
[1.4e-07, 'ix xy fvyxi'],
[8.8e-11, 'jy yz gwzyj'],
[7.2e-10, 'kz za hxazk'],
[0.01503, 'la ab iybal'],
[3.7e-08, 'mb bc jzcbm'],
[0.00524, 'nc cd kadcn'],
[0.29041, 'od de lbedo'],
[0.00874, 'pe ef mcfep'],
[7.3e-07, 'qf fg ndgfq'],
[0.06410, 'rg gh oehgr'],
[0.11955, 'sh hi pfihs'],
[3.1e-06, 'ti ij qgjit'],
[1.1e-08, 'ui hkju'],
[2.6e ilkv'],
mlw'],
nmx'],
ny'],
op wmpoz'],
07, 'ap pq xnqpa'],
[5.7e-08, 'bq qr yorqb'],
[0.00024, 'cr rs zpsrc'],
[0.45555, 'et tu brute'],
```

max!

A score based on letter frequencies/ probabilities does!

Algorithm Design

Computer Science 111
Boston University

Helper Functions

- When designing a function, it often helps to write a separate *helper function* for a portion of the overall task.
- We've seen this before:
  - `scrabble_score()` called `letter_score()`

```
def letter_score(letter):
    if letter in 'aeilnorstu':
        return 1
    ...

def scrabble_score(word):
    if ...
        ...
    else:
        score_rest = scrabble_score(...)
        return letter_score(...) + ...
```

  - other places as well!

In PS 3: Jotto Score

- `jscore(s1, s2)`
  - returns the number of characters in `s1` that are shared by `s2`
  - the positions and the order of the characters do *not* matter
  - repeated letters are counted multiple times

- Examples:
  - `jscore('diner', 'syrup')` → 1
  - `jscore('always', 'bananas')` → 3
  - `jscore('always', 'walking')` → 3

What will this call return?

```
jscore('recursion', 'explorations')
```

Jotto Score: Consider Concrete Cases

`jscore('always', 'walking')`

- what is its solution?
- what is the next smaller subproblem?
  - will `jscore('lways', 'alking')` work?
  - will `jscore('lways', 'walking')` work?
  - what should we do instead?

Removing the First Occurrence of an Element from a List

- `rem_first(elem, values)`
  - inputs: an arbitrary value (`elem`) and a list (`values`)
  - returns: a version of `values` in which ***only the first*** occurrence of `elem` in `values` (if any) is removed

```
>>> rem_first(10, [3, 5, 10, 7, 10])
[3, 5, 7, 10]
```

- We'll write this function together in lecture.
- On the problem set, you will:
  - adapt it to work with strings
  - use it as a helper function for `jscore()`

Look Familiar?

- `rem_all(elem, values)`
  - inputs: an arbitrary value (`elem`) and a list (`values`)
  - returns: a version of `values` in which *all* occurrences of `elem` in `values` (if any) are removed

```
>>> rem_all(10, [3, 5, 10, 7, 10])
[3, 5, 7]
```

- `rem_first(elem, values)`
  - inputs: an arbitrary value (`elem`) and a list (`values`)
  - returns: a version of `values` in which ***only the first*** occurrence of `elem` in `values` (if any) is removed

```
>>> rem_first(10, [3, 5, 10, 7, 10])
[3, 5, 7, 10]
```

We can adapt `rem_all()` to get `rem_first()`...

```
def rem_all(elem, values):
    """ removes all occurrences of elem from
        values
    """
    if values == []:
        return []
    else:
        rem_rest = rem_all(elem, values[1:])

        if values[0] == elem:
            return rem_rest
        else:
            return [values[0]] + rem_rest
```

Consider Concrete Cases!

`rem_first(10, [3, 5, 10, 7, 10])`

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

`rem_first(10, [10, 3, 5, 10, 7])`

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

Use the concrete cases to fill in the blanks…

```
def rem_first(elem, values):
    """ removes the first occurrence of elem from
        values
    """
    if values == []:
        return []
    else:
        rem_rest = rem_first(elem, values[1:])

        if values[0] == elem:

            return ____________________
        else:

            return ____________________
```

A Recursive Palindrome Checker

- A *palindrome* is a string that reads the same forward and backward.
  - examples: "radar", "mom", "abcddcba"
- Let's write a function that determines if a string is a palindrome:

```
>>> is_pal('radar')
True
>>> is_pal('abccda')
False
```

- We need more than one base case. What are they?
- How should we reduce the problem in the recursive call?

Consider Concrete Cases!

`is_pal('radar')`

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

`is_pal('modem')`

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

A Recursive Palindrome Checker

```
def is_pal(s):
    """ returns True if s is a palindrome
        and False otherwise.
        input s: a string containing only letters
                 (no spaces, punctuation, etc.)
    """
    if len(s) <= 1:   # empty string or one letter
        return ____________
    elif __________________:
        return __________
    else:             # recursive case
        is_pal_rest = ____________________

        # do our one step!
```

Pre-Lecture
Binary Numbers

Computer Science 111
Boston University

Bits and Bytes

- Everything stored in a computer is essentially a binary number.
  `0110110010100111`
- Each digit in a binary number is one *bit.*
  - a single 0 or 1
  - based on two voltages: "low" = 0, "high" = 1
- One *byte* is 8 bits.
  - example: 01101100

Bits of Data

- A given set of bits can have more than one meaning.

binary	decimal integer	character
01100001	97	'a'
01000110	70	'F'

Representing Integers in Decimal

- In base 10 (decimal), each column represents a power of 10.

HUNDREDs column
TENs column
ONEs column

176

1 hundred + 7 tens + 6 ones

$1*10^2 + 7*10^1 + 6*10^0$

Representing Integers in Binary

- In base 2 (binary), each column represents a power of 2.

128's column
SIXTY-FOURs column
THIRTY-TWOs column
SIXTEENs column
EIGHTs column
FOURs column
TWOs column
ONEs column

10110000

$1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 0*2^0$

128 + 0 + 32 + 16 + 0 + 0 + 0 + 0

also 176!

What Does the Rightmost Bit Tell Us?

128's column
SIXTY-FOURs column
THIRTY-TWOs column
SIXTEENs column
EIGHTs column
FOURs column
TWOs column
ONEs column

10110000

- If the rightmost bit is 0, the number is ___________.
- If the rightmost bit is 1, the number is ___________.

Binary to Decimal (On Paper)

- Number the bits from right to left
  - example:

0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

- For each bit that is 1, add 2^n, where n = the bit number
  - example:

0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

decimal value = $2^6 + 2^4 + 2^3 + 2^2 + 2^0$

____

Decimal to Binary (On Paper)

- Go in the reverse direction: determine which powers of 2 need to be added together to produce the decimal number.

- Start with the largest power of 2 less than or equal to the number, and work down from there.
  - example: what is 53 in binary?
    - 32 is the largest power of 2 <= 53: 53 = 32 + 21
    - now, break the 21 into powers of 2: 53 = 32 + 16 + 5
    - now, break the 5 into powers of 2: 53 = 32 + 16 + 4 + 1
    - 1 is a power of 2 (2^0), so we're done: 53 = 32 + 16 + 4 + 1
      = $2^5 + 2^4 + 2^2 + 2^0$
      = 110101

Pre-Lecture
Binary Arithmetic

Computer Science 111
Boston University

Binary Addition Fundamentals

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10
- 1 + 1 + 1 = 11

Adding Decimal Numbers

```
   1  1
   12537
+   9272
--------
   21809
```

Adding Binary Numbers

```
   01110
+  11100
--------
```

Multiplying Binary Numbers

```
    101101
*     1110
----------
    000000
   1011010
  10110100
+ 101101000
----------
1001110110
```

Hint:

```
  529
*  42
-----
 1058
+2116
-----
22218
```

Do you remember this algorithm? It's the same!

Shifting Bits to the Left

- A left-shift:
  - moves every bit of a binary number to the left
  - adds a 0 in the right-most place
- For example: $1011010_2 = 90_{10}$
  - a left-shift by 1 gives $10110100_2 = 180_{10}$

    base 2 base 10

- Left-shifting by 1 doubles the value of a number.
- In Python, we can apply the left-shift operator (<<) to any integer:

```
>>> print(75 << 1)
```

______

Shifting Bits to the Right

- A right-shift:
  - moves every bit of a binary number to the right
  - the rightmost bit is lost!
- For example: $1011010_2 = 90_{10}$
  - a right-shift by 1 gives $101101_2 = 45_{10}$
- Right-shifting by 1 halves the value of a number (using integer division).
- In Python, we can apply the right-shift operator (>>) to any integer:

```
>>> print(15 >> 1)
```

______

Binary Numbers

Computer Science 111
Boston University

Recall: Binary to Decimal (On Paper)

- Number the bits from right to left
  - example:

0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

- For each bit that is 1, add 2^n, where n = the bit number
  - example:

0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

decimal value = $2^6 + 2^4 + 2^3 + 2^2 + 2^0$

$64 + 16 + 8 + 4 + 1 = 93$

  - another example: what is the integer represented by 1001011?

Recall: Decimal to Binary (On Paper)

- Go in the reverse direction: determine which powers of 2 need to be added together to produce the decimal number.
- Start with the largest power of 2 less than or equal to the number, and work down from there.
  - example: what is 53 in binary?
    - 32 is the largest power of 2 <= 53: 53 = 32 + 21
    - now, break the 21 into powers of 2: 53 = 32 + 16 + 5
    - now, break the 5 into powers of 2: 53 = 32 + 16 + 4 + 1
    - 1 is a power of 2 (2^0), so we're done: 53 = 32 + 16 + 4 + 1 $= 2^5 + 2^4 + 2^2 + 2^0$ = 110101

Which of these is a correct *partial* binary representation of the decimal integer 90?

90 (decimal) → ______ (binary)

A. 101xxx1
B. 111xxx1
C. 101xxx0
D. 111xxx0
E. none of these

an x denotes a "hidden" bit that we aren't revealing

Hint: You shouldn't need to perform the full conversion (i.e., you shouldn't need to determine the hidden bits)!

Recall: Shifting Bits to the Left

- A left-shift:
  - moves every bit of a binary number to the left
  - adds a 0 in the right-most place
- For example: $1011010_2 = 90_{10}$
  - a left-shift by 1 gives $10110100_2 = 180_{10}$

  base 2 base 10
- Left-shifting by 1 doubles the value of a number.
- In Python, we can apply the left-shift operator (<<) to any integer:

```
>>> print(75 << 1)
150
>>> print(5 << 2)
```

Recall: Shifting Bits to the Right

- A right-shift:
  - moves every bit of a binary number to the right
  - the rightmost bit is lost!
- For example: $1011010_2 = 90_{10}$
  - a right-shift by 1 gives $101101_2 = 45_{10}$
- Right-shifting by 1 halves the value of a number (using integer division).
- In Python, we can apply the right-shift operator (>>) to any integer:

```
>>> print(15 >> 1)
7
>>> print(120 >> 2)
```

Recall: Decimal to Binary (On Paper)

90 = 64 + 26
= 64 + 16 + 10
= 64 + 16 + 8 + 2
= $2^6 + 2^4 + 2^3 + 2^1$
= 1011010

- This is a **left-to-right** conversion.
  - we begin by determining the leftmost digit
- The first step is tricky to perform computationally, because we need to determine the largest power.

Decimal to Binary: Right-to-Left

- We can use a **right-to-left** approach instead.
- For example: let's convert 139 to binary:

$$139 = \underbrace{???????}1$$

The rightmost bit must be 1. Why?

If the remaining bits were on their own (without the rightmost bit), what number would they represent?

Decimal to Binary: Right-to-Left (cont.)

```
                139  =  ? ? ? ? ? ? ? 1
139 >> 1  →      69  =  ? ? ? ? ? ?
 69 >> 1  →      34  =  ? ? ? ? ?
 34 >> 1  →      17  =  ? ? ? ?
 17 >> 1  →       8  =  ? ? ?
  8 >> 1  →       4  =  ? ?
  4 >> 1  →       2  =  ?
  2 >> 1  →       1  =
-------------------------------
                139  =
```

`dec_to_bin()` Function

- `dec_to_bin(n)`
  - takes an integer `n`
  - should return a *string* representation of `n`'s binary value

```
>>> dec_to_bin(139)
'10001011'
>>> dec_to_bin(13)
'1101'
```

How `dec_to_bin()` Should Work...

```
dec_to_bin(13)
  n = 13
  bin_rest = dec_to_bin(6)

dec_to_bin(6)
  n = 6
  bin_rest =

dec_to_bin(  )
  n =
  bin_rest =

dec_to_bin(  )
  n =
```

Binary to Decimal: Right-to-Left

- Here again, we can use a **right-to-left** approach.
- For example:

`'1101' = ?`

If we knew the decimal value of these bits, how could we use it?

What should we do with the rightmost bit?

- ***Devise an algorithm together!***

`bin_to_dec()` Function

- `bin_to_dec(b)`
  - takes a string `b` that represents a binary number
  - should return an *integer* representation of `b`'s decimal value

```
>>> bin_to_dec('10001011')
139
>>> dec_to_bin('1101')
13
```

How `bin_to_dec()` Should Work...

```
bin_to_dec('1101')
  b = '1101'
  dec_rest = bin_to_dec('110')

  bin_to_dec('110')
    b = '110'
    dec_rest =

    bin_to_dec(      )
      b =
      dec_rest =

      bin_to_dec(      )
        b =
```

Binary Arithmetic Revisited

Computer Science 111
Boston University

Recall: Binary Addition Fundamentals

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10
- 1 + 1 + 1 = 11

Recall: Adding Binary Numbers

```
 1 1 1
  01110
+ 11100
-------
 101010
```

Add these two binary numbers ***WITHOUT*** converting to decimal!

```
  101101
+   1110
--------
```

Hint:

```
   1
  529
+ 742
-----
 1271
```

Do you remember this algorithm? It's the same!

Recall: It's All Bits!

- Another example: text

`'terriers'`

`0111010001100101011100100111001001101001011001010111001001110011`

8 ASCII characters, 8 bits each → 64 bits

- *All* types of data are represented in binary.
  - images, sounds, movies, floating-point numbers, etc...
- ***All computation*** involves manipulating bits!

It's All Bits! (cont.)

- Example: to add 42 + 9, the computer does *bitwise addition*:

```
   1
  101010
+ 001001
--------
  110011
```

- In PS 4, you'll write a Python function for this.

```
add_bitwise('101010', '001001')
```

PS 4: add_bitwise

- `add_bitwise(b1, b2)`
  b1 and b2 are *strings* representing binary #s

```
   1
  101010
+ 001001
  ------
  110011
```

- It should look something like this:

```
def add_bitwise(b1, b2):
    if ...      # base case #1

    elif ...    # base case #2

    else:       # recursive case
        sum_rest = add_bitwise(b1[:-1],b2[:-1])
        if ...
             # rest of recursive case
```

- Let's trace through a concrete case:
  `add_bitwise('100', '010')`

How recursion works: add_bitwise(b1, b2)

- Recall: we get a separate stack frame for each call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10','01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'   b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''   b2: ''
base case: return ''
```

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'    b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'    b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'    b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''    b2: ''
base case: return ''
```

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'    b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'    b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'    b2: '0'
sum_rest = ''
if ...
    return
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = '1'
if ...
    return
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = '11'
if ...
    return
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

The Tricky Part of `add_bitwise(b1, b2)`

- We again end up with a series of recursive calls:

changing the rightmost bits to 1

```
add_bitwise('101', '011')
b1: '101'    b2: '011'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'    b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'    b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''    b2: ''
base case: return ''
```

The Tricky Part of `add_bitwise(b1, b2)`

- We again build our solution on our way back from the base case:

```
add_bitwise('101', '011')
b1: '101'    b2: '011'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'    b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'    b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''    b2: ''
base case: return ''
```

The Tricky Part of `add_bitwise(b1, b2)`

- **What do we need to do differently here?**

```
add_bitwise('101', '011')
b1: '101'   b2: '011'
sum_rest = '11'  # same as before
if ...
    ???
```

- We need to carry!

```
     1
    101
 +  011
 -------
    110
  +  1
   1000
```

- We need to add 11 + 1 to get 100.
  - how can we do this addition?

It's All Bits! (cont.)

- Example: to add 42 + 9, the computer does *bitwise addition*:

```
     1
   101010
 + 001001
 --------
   110011
```

- In PS 4, you'll write a Python function for this.
  `add_bitwise('101010', '001001')`
- ***In PS 5, you'll design a circuit for it!***
  - ***more on this next time***
- ***You'll also design a circuit for binary multiplication!***

Recall: Multiplying Binary Numbers

```
     101101
*      1110
-----------
     000000
    1011010
   10110100
+ 101101000
-----------
 1001110110
```

Multiply these binary numbers ***WITHOUT*** converting to decimal!

```
   1101
*    11
-------
```

Hint:

```
   529
*   42
------
  1058
+2116
------
 22218
```

Do you remember this algorithm? It's the same!

Recall: Finding the Largest Element in a List

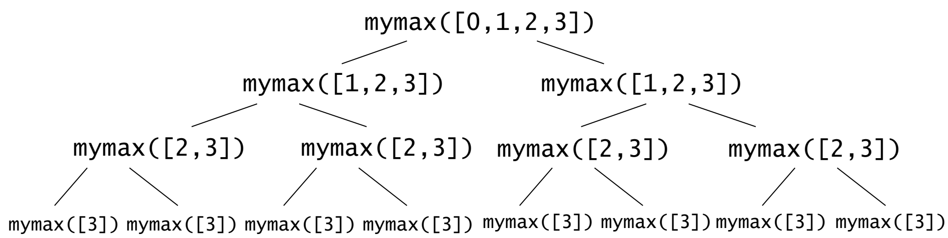
```
def mymax(values):
    """ returns the largest element in a list
        input: values is a *non-empty* list
    """
    if len(values) == 1:       # base case
        return values[0]
    else:                      # recursive case
        max_in_rest = mymax(values[1:])
        if values[0] > max_in_rest:
            return values[0]
        else:
            return max_in_rest
```

What's Wrong (If Anything) With This Alternative?

```
def mymax(values):
    """ returns the largest element in a list
        input: values is a *non-empty* list
    """
    if len(values) == 1:       # base case
        return values[0]
    else:                      # recursive case
        # max_in_rest = mymax(values[1:])
        if values[0] > mymax(values[1:]):
            return values[0]
        else:
            return mymax(values[1:])
```

How recursion works...

```
def mymax(values):
    if len(values) == 1:
        return values[0]
    else:
        if values[0] > mymax(values[1:]):
            return values[0]
        else:
            return mymax(values[1:])
```



number of calls for a list of length 4 = 15

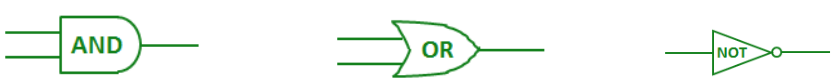
number of calls for a list of length n = $2^n - 1$ ← gets big fast!!! *exponential* growth

Pre-Lecture
Gates and Circuits

Computer Science 111
Boston University

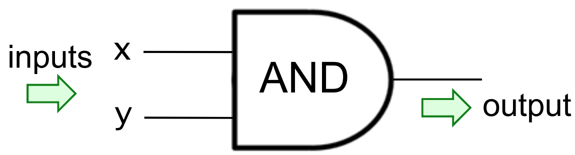
Bits as Boolean Values

- When designing a circuit, we think of bits as boolean values:
  - 1 = True
  - 0 = False
- In Python, we've used *logic operators* (`and`, `or`, `not`) to build up boolean expressions.
- In circuits, there are corresponding *logic gates*.



AND Gate

AND outputs 1 only
if ***all*** inputs are 1



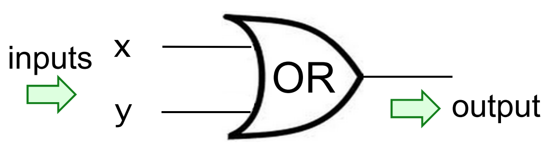
AND's function:

inputs x	inputs y	output x AND y
0	0	
0	1	
1	0	
1	1	

truth table

OR Gate

OR outputs 1 if
any input is 1

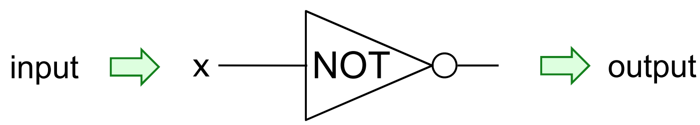


OR's function:

inputs x	inputs y	output x OR y
0	0	
0	1	
1	0	
1	1	

NOT Gate

NOT reverses
its input

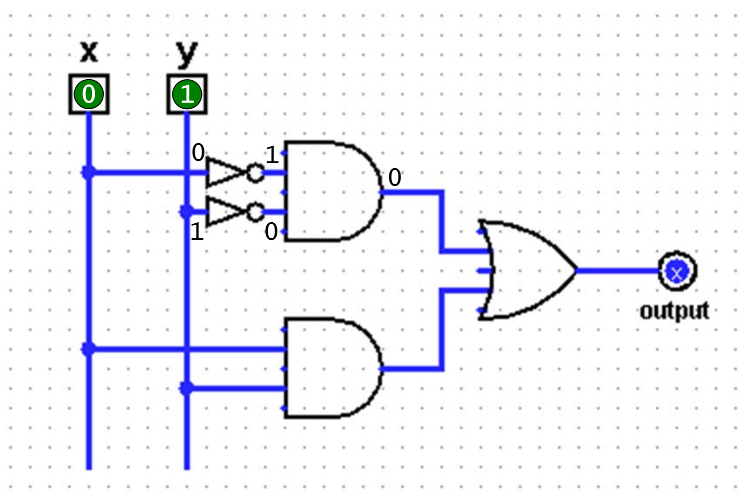


NOT's function:

input x	output NOT x
0	
1	

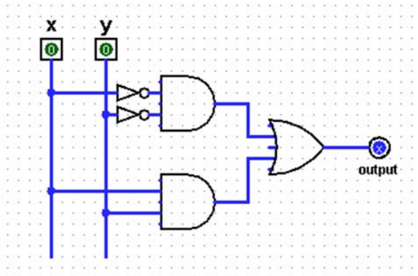
From Gates to Circuits

- We combine logic gates to form larger circuits.



- Example: what is the output when x = 0 and y = 1?

A Truth Table for a Circuit



inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

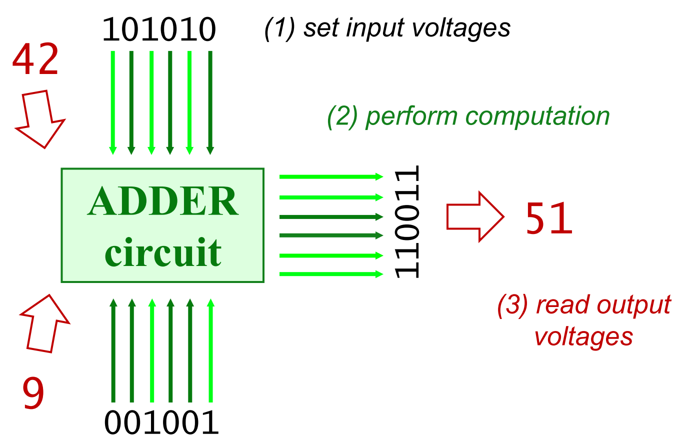
- A circuit is a boolean function – a function of bits!
  - takes one or more bits as inputs
  - produces the appropriate bit(s) as output

Gates and Circuits

Computer Science 111
Boston University

How Computation Works

- In a computer, each bit is represented as a *voltage.*
  - 1 is +5 volts, 0 is 0 volts
- Computation is the deliberate combination of those voltages!



All Computation Involves *Functions* of Bits!

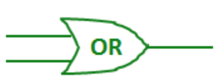
binary inputs **A** and **B** → output, **A+B**

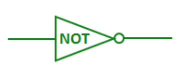
bitwise addition function

A	**B**	**A+B**
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

Recall: Bits as Boolean Values

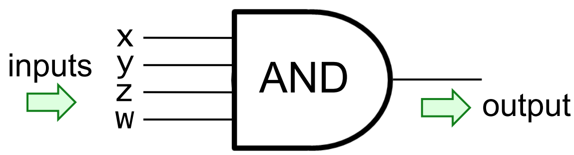
- When designing a circuit, we think of bits as boolean values:
  - 1 = True
  - 0 = False
- In Python, we've used *logic operators* (`and`, `or`, `not`) to build up boolean expressions.
- In circuits, there are corresponding *logic gates*.





AND Gate (with *four* inputs)

AND outputs 1 only
if ***all*** inputs are 1



AND's function:

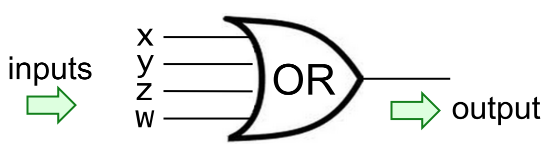
inputs				output
x	y	z	w	AND(x,y,z,w)
0	0	0	0	0
0	0	0	1	0
…12 more rows not shown…				0
1	1	1	0	0
1	1	1	1	1

fifteen 0s

one 1

OR Gate (with *four* inputs)

OR outputs 1 if
any input is 1



OR's function:

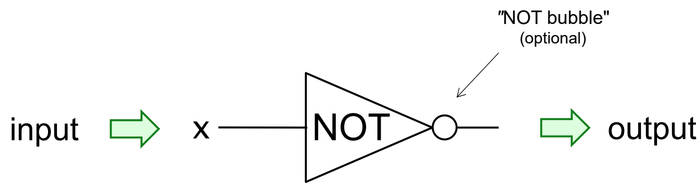
inputs				output
x	y	z	w	OR(x,y,z,w)
0	0	0	0	0
0	0	0	1	1
…12 more rows not shown…				1
1	1	1	0	1
1	1	1	1	1

one 0

fifteen 1s

NOT Gate

NOT reverses its input



NOT's function:

input	output
x	NOT(x)
0	1
1	0

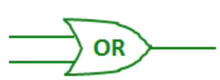
Circuit Building Blocks: Logic Gates

AND outputs 1 only if **ALL** inputs are 1

AND

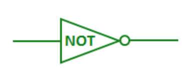
OR outputs 1 if **ANY** input is 1

OR



NOT reverses its input

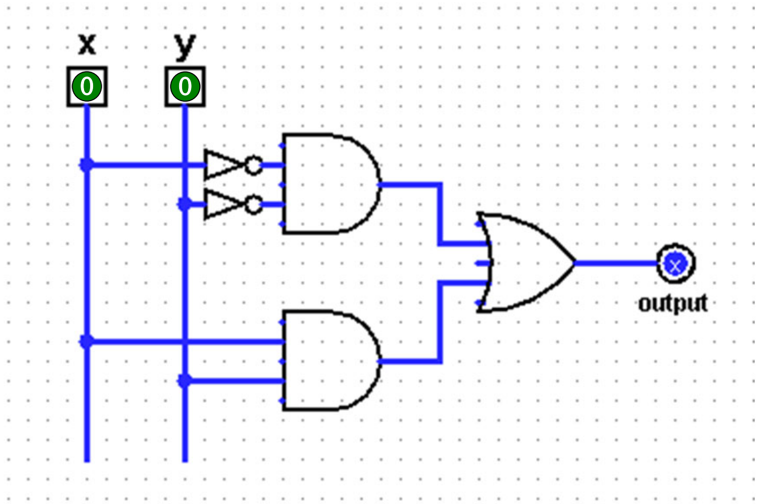
NOT



- They each define a boolean function – a function of bits!
  - take one or more bits as inputs
  - produce the appropriate bit as output
  - the function can be defined by means of a *truth table*

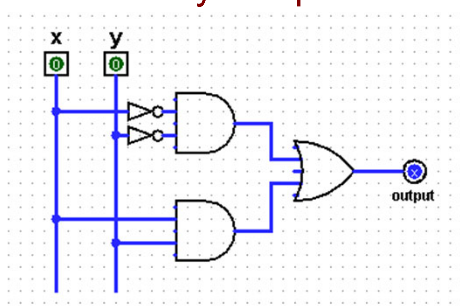
From Gates to Circuits (Second Example)

- We combine logic gates to form larger circuits.



- Example: what is the output when x = 0 and y = **0**?

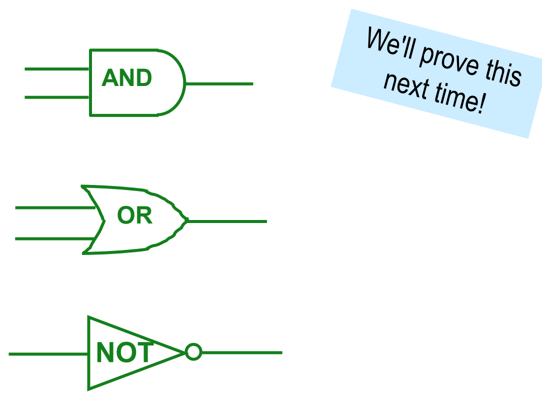
Which column correctly completes the truth table?



inputs		
x	y	output
0	0	
0	1	
1	0	
1	1	

Claim

We need only these three building blocks to compute anything at all!



Extra Practice: Recursive Bitwise AND

- To take the bitwise AND of two binary numbers, we:
  - line them up
  - AND together each pair of bits:

```
 111010
 101011
 ------
 101010
```

- If one number has more bits, those bits are effectively ANDed with 0s:

```
 10101001
    11011
 --------
 00001001
```

Extra Practice: Recursive Bitwise AND

- Write a recursive function `bitwise_and(b1, b2)`
  - examples:

```
>>> bitwise_and('110', '010')
'010'
>>> bitwise_and('1001', '1100')
'1000'
>>> bitwise_and('1011001', '1100')
'0001000'
>>> bitwise_and('1101', '')
'0000'
```

- You will need more than one base case.
- You need to process the bitstrings *from right to left.* Why?

Extra Practice: Recursive Bitwise AND

```
def bitwise_and(b1, b2):
    """ computes bitwise AND of bitstrings b1 and b2
    """
    if ______________________:
        return ______________________
    elif ________________________:
        return ______________________
    # other elif if needed

    else:
        and_rest = ______________________________
        # do your one step below!
```

Pre-Lecture
Minterm Expansion

Computer Science 111
Boston University

Boolean Notation

- Recall:

inputs x	y	output x AND y
0 *	0	= 0
0 *	1	= 0
1 *	0	= 0
1 *	1	= 1

inputs x	y	output x OR y
0 +	0	= 0
0 +	1	= 1
1 +	0	= 1
1 +	1	= 1

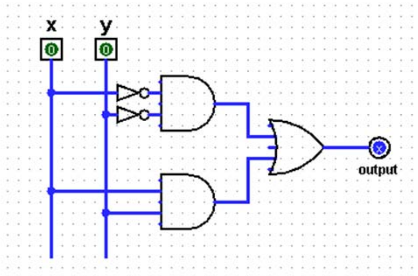
input x	output NOT x
0	1
1	0

- In boolean notation:
  - x AND y is written as multiplication: xy
  - x OR y is written as addition: x + y
  - NOT x is written using a bar: $\bar{x}$

- Example:

(x AND y) OR (x AND (NOT y)) ←→ ____________

Boolean Expressions for Truth Tables



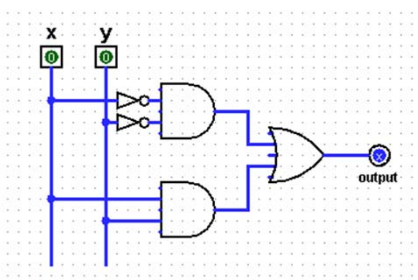
inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

- This truth table/circuit can be summarized by the expression:

$$\bar{x}\bar{y} + xy$$

inputs		output	$\bar{x}\bar{y} + xy$
x	y		
0	0	1	1*1 + 0*0 = 1
0	1	0	1*0 + 0*1 = 0
1	0	0	0*1 + 1*0 = 0
1	1	1	0*0 + 1*1 = 1

Boolean Expressions for Truth Tables



inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

- This truth table/circuit can be summarized by the expression:

$$\bar{x}\bar{y} + xy$$

- This expression is the *minterm expansion* of this truth table.
  - one *minterm* for each row that has an output of 1
  - combined using OR

Building a Minterm Expansion for a Boolean Function

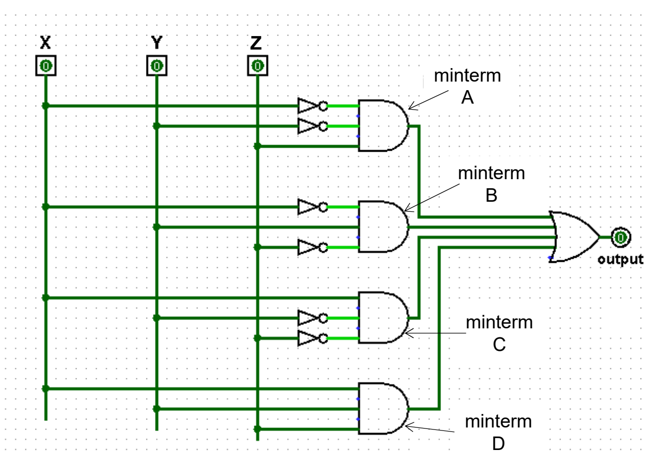
1. If you don't have it, create the truth table.
2. Delete the rows with an output of 0.
3. Create a minterm for each remaining row (the ones with an output of 1):
   - AND the input variables together
   - if a variable has a 0 in that row, negate it
   - example: minterm for the 2nd row
     $\overline{x}\,\overline{y}z$
4. OR the minterms together.

inputs			output
x	y	z	
~~0~~	~~0~~	~~0~~	~~0~~
0	0	1	1
0	1	0	1
~~0~~	~~1~~	~~1~~	~~0~~
1	0	0	1
~~1~~	~~0~~	~~1~~	~~0~~
~~1~~	~~1~~	~~0~~	~~0~~
1	1	1	1

Minterm Expansion → Circuit

minterm expansion =

$$\underbrace{\overline{x}\,\overline{y}z}_{\text{minterm A}} + \underbrace{\overline{x}y\overline{z}}_{\text{minterm B}} + \underbrace{x\overline{y}\,\overline{z}}_{\text{minterm C}} + \underbrace{xyz}_{\text{minterm D}}$$

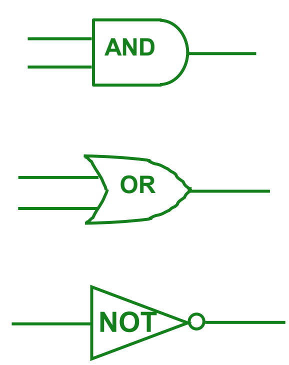


Minterm Expansion

Computer Science 111
Boston University

Claim

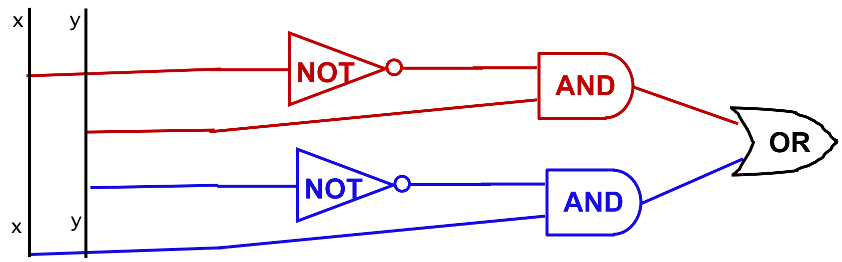
We need only these three building blocks to compute anything at all!



Constructive Proof!

① Specify a **truth table** defining ***any*** function you want.

input		output
x	y	fn(x,y)
0	0	0
0	1	1
1	0	1
1	1	0



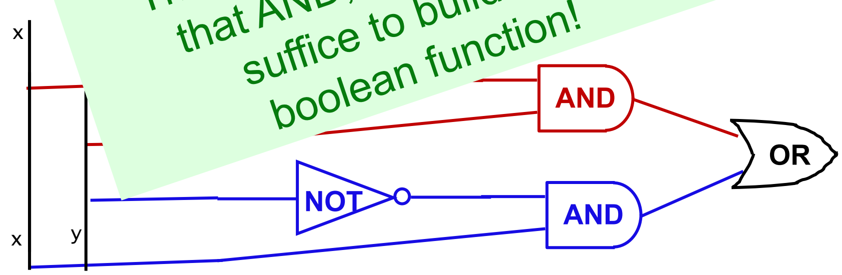
② For each input row whose output needs to be 1, build an **AND** circuit that outputs 1 *only for that specific input!*

③ **OR** them all together .

Constructive Proof!

① Specify a **truth table** defining ***any*** function you want.

input		output
x	y	fn(x,y)
0	0	0
0	1	1
1	0	
1		



② For each input row whose output needs to be 1, build an **AND** circuit t outputs 1 *only for c input!*

This is a constructive proof that AND, OR, and NOT suffice to build any boolean function!

Constructive Proof!

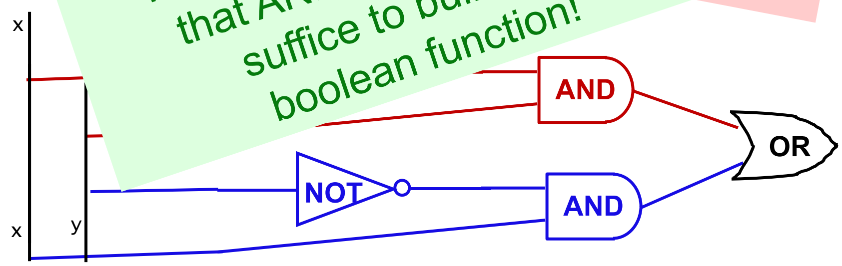
(1) Specify a **truth table** defining ***any*** function you ...

input		fn
x	y	fn
0	0	
0	1	1
1	0	1
1	1	

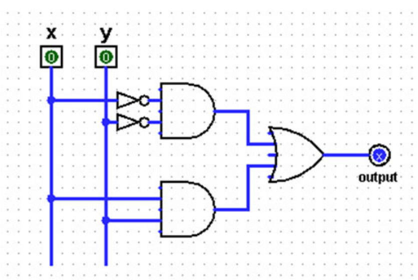
(2) For each input row whose output needs to be 1, build an **AND** circuit th... t outputs 1 ... *input*!

But... **ALL** functions are just boolean functions: because everything is in binary!

This is a co... that AND, OR, ... suffice to build a... boolean function!



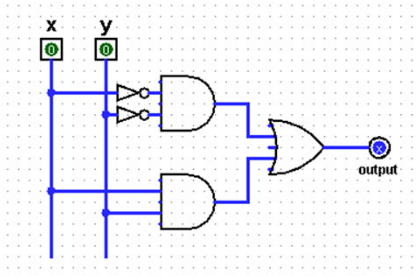
Revisiting Our Earlier Circuit...



inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

- The top AND gate implements which row of the truth table?
- The bottom AND gate implements which row?

Recall: Boolean Expressions for Truth Tables



inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

- This truth table/circuit can be summarized by the expression:

$$\bar{x}\bar{y} + xy$$

- This expression is the *minterm expansion* of this truth table.
  - one *minterm* for each row that has an output of 1
  - combined using OR

Building a Minterm Expansion for a Boolean Function

ex: `greater_than_4(x, y, z)`
→ 1 if the 3-digit binary number xyz > 4
→ 0 otherwise

for example:

- `greater_than_4(1, 1, 0)` → 1 (True)
  Why?
- `greater_than_4(0, 1, 1)` → 0 (False)
  because $011_2 = 3_{10}$, and 3 is *not* > 4

Building a Minterm Expansion for a Boolean Function

ex: `greater_than_4(x, y, z)`
→ 1 if the 3-digit binary number `xyz` > 4
→ 0 otherwise

1. If you don't have it, create the truth table.
2. Delete the rows with an output of 0.
3. Create a minterm for each remaining row (the ones with an output of 1):
   - AND the input variables together
   - if a variable has a 0 in that row, negate it
4. OR the minterms together.

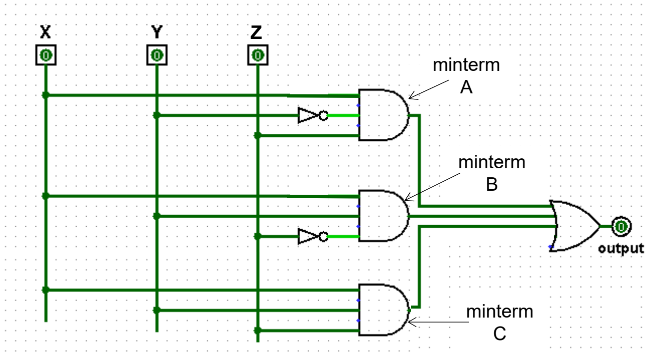
	inputs			output
dec	x	y	z	
0:	0	0	0	0
1:	0	0	1	0
2:	0	1	0	0
3:	0	1	1	0
4:	1	0	0	0
5:	1	0	1	1
6:	1	1	0	1
7:	1	1	1	1

Minterm Expansion → Circuit

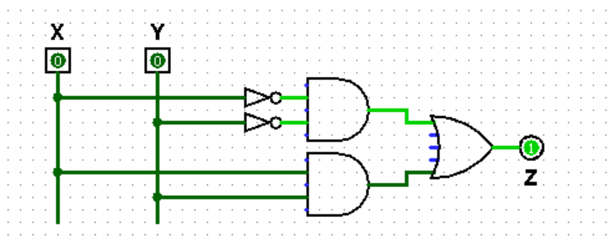
minterm expansion =

$x\bar{y}z + xy\bar{z} + xyz$

minterm A, minterm B, minterm C

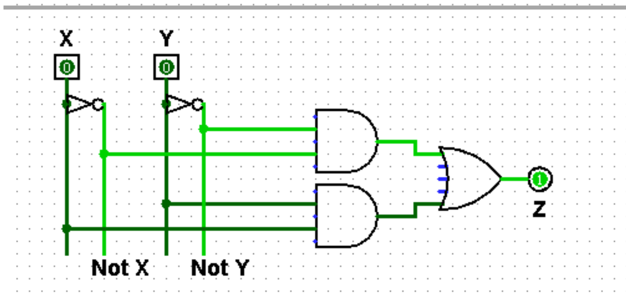


Adding "Rails" for the NOT of Each Input



Here's a circuit that we looked at earlier.

It tests whether x == y.

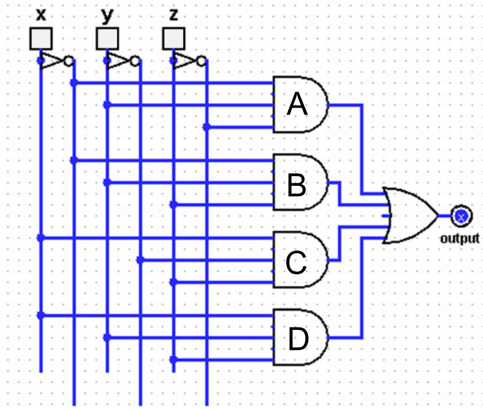


Here's an alternative version that adds "rails" for Not X and Not Y.

In some cases (but not this one!), doing so can reduce the number of NOT gates.

Which AND gate corresponds to row 3 of the table?

	input			output
	x	y	z	
row 0	0	0	0	
row 1	0	0	1	
row 2	0	1	0	
row 3	0	1	1	
row 4	1	0	0	
row 5	1	0	1	
row 6	1	1	0	
row 7	1	1	1	



- Complete the rest of the truth table.
- What is its minterm expansion as a formula/expression?
- If the inputs represent a three-bit integer, what property of integers does the circuit compute?

What is the minterm expansion of this truth table?

inputs			output
x	y	z	
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A. $yz + xz + xy$

B. $xyz + x\overline{y}\overline{z} + \overline{x}y\overline{z} + \overline{x}\overline{y}z$

C. $\overline{x}\overline{y}z + \overline{x}y\overline{z} + x\overline{y}\overline{z} + xyz$

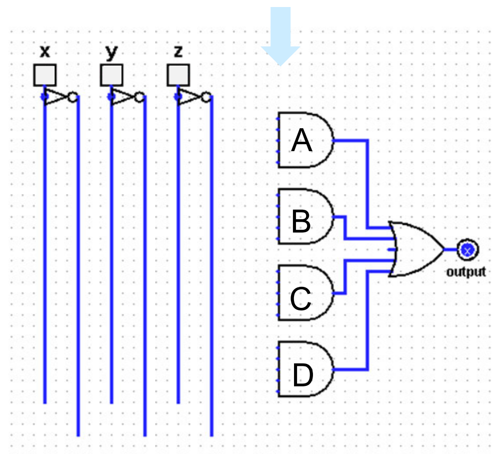
D. $\overline{x}\overline{y}\overline{z} + \overline{x}yz + x\overline{y}z + \overline{x}\overline{y}z$

E. none of the above

Extra Practice: DIY!

Add the wires needed to build a circuit for the truth table at left…

x	y	z	output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



- What is the minterm expansion formula?
- What is the circuit testing for (i.e., when does it output a 1)?

Pre-Lecture
Definite Loops in Python

Computer Science 111
Boston University

for Loops

- A `for` statement is one way to create a loop in Python.
  - allows us to *repeat* one or more statements.
- Example:

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```

the body of the loop

will output:

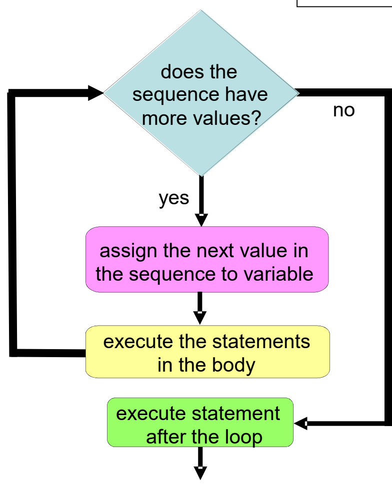
```
Warning
1
Warning
2
Warning
3
```

for Loops (cont.)

- General syntax:

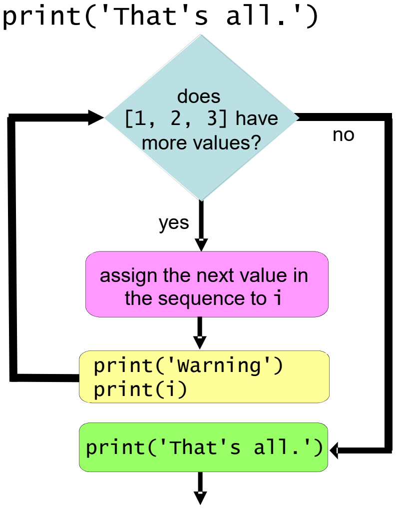
```
for variable in sequence:
    body of the loop
```

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```



Executing Our Earlier Example
(with one extra statement)

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):        # [0, 1, 2, ..., N – 1]
    body of the loop
```

- Example:

```
for i in range(3):        # [0, 1, 2]
    print('I'm feeling loopy!')
```

outputs:

```
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
```

continued on next slide

Simple Repetition Loops (cont.)

- To repeat a loop's body *N* times:

```
for i in range(N):        # [0, 1, 2, ..., N – 1]
    body of the loop
```

- Example:

```
for i in range(5):
    print('I'm feeling loopy!')
```

outputs:

`for` Loops Are Definite Loops

- *Definite* loop = a loop in which the number of repetitions is *fixed* before the loop even begins.

- In a `for` loop, # of repetitions = `len(`*sequence*`)`

```
for variable in sequence:
    body of the loop
```

Pre-Lecture
Cumulative Computations;
Element-Based vs. Index-Based Loops

Computer Science 111
Boston University

Python Shortcuts

- Consider this code:

```
age = 13
age = age + 1
            13 + 1
               14
```

- Instead of writing

```
age = age + 1
```

  we can just write

```
age += 1
```

Python Shortcuts (cont.)

shortcut	equivalent to
var += *expr*	*var* = *var* + (*expr*)
var -= *expr*	*var* = *var* - (*expr*)
var *= *expr*	*var* = *var* * (*expr*)
var /= *expr*	*var* = *var* / (*expr*)
var //= *expr*	*var* = *var* // (*expr*)
var %= *expr*	*var* = *var* % (*expr*)
var **= *expr*	*var* = *var* ** (*expr*)

where *var* is a variable
expr is an expression

- **Important:** the = must come *after* the other operator.
  - += is correct
  - =+ is not!

Using a Loop to Sum a List of Numbers

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result

print(sum([10, 20, 30, 40, 50]))
```

x	result

Cumulative Computations (cont.)

```
def sum(vals):
    result = 0        # the accumulator variable
    for x in vals:
        result += x   # gradually accumulates the sum
    return result

print(sum([10, 20, 30, 40, 50]))
```

Element-Based for Loop

```
vals = [3, 15, 17, 7]

         x
```

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

Index-Based for Loop

```
           vals[0] vals[1] vals[2] vals[3]
vals = [3,   15,     17,     7]
        0    1       2       3
                 i
```

```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

Tracing an Index-Based Cumulative Sum

```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result

print(sum([10, 20, 30, 40, 50]))
```

i	vals[i]	result

Circuits for Arithmetic; Modular Design ; A First Look at Loops

Computer Science 111
Boston University

2-Bit Binary Addition

- The truth table is at right.
  - 4 bits of input
  - 3 bits of output
- In theory, we could use the minterm-expansion approach to create ***3 circuits***.
  - one for each output bit
- It ends up being overly complicated.
  - more gates than are really needed
- Instead, we'll take advantage of two things:
  - our elementary-school bitwise-addition algorithm
  - modular design!

binary inputs **A** and **B**		output, **A+B**
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110
A	**B**	

A Full Adder

- Recall our bitwise algorithm:

```
       0 1 1
      101101
+     001110
------------
      111011
```

- A *full adder* adds only one column.

- It takes 3 bits of input:
  - x and y – one bit from each number being added
  - c_{in} – the carry bit *into* the current column

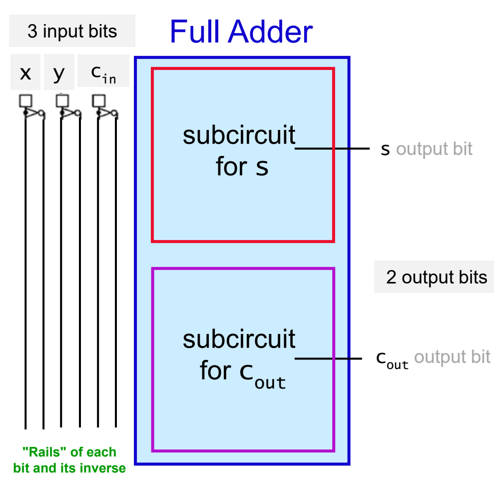
- It produces 2 bits of output:
  - s – the bit from the sum that goes at the bottom of the column
  - c_{out} – the carry bit *out of* the current column
    - it becomes the c_{in} of the next column!

inputs			outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

How many AND gates will you need in all?

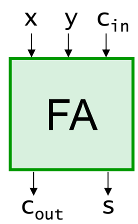
Create a separate minterm expansion/circuit **for each output bit**!

inputs			outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

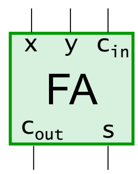


Modular Design

- Once we have a full adder, we can treat it as an *abstraction* – a "black box" with 3 inputs and two outputs.

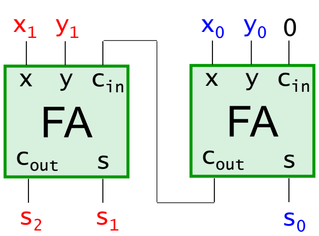


- Here's another way to draw it:



Modular Design (cont.)

- To add 2-bit numbers, combine two full adders!



$$\begin{array}{r} x_1 x_0 \\ +\ y_1 y_0 \\ \hline s_2 s_1 s_0 \end{array}$$

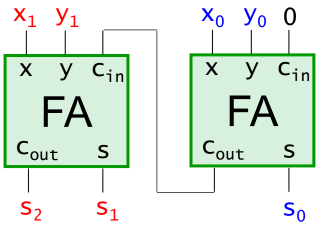
- Produces what is known as a *2-bit ripple-carry adder.*
- To add larger numbers, combine even more FAs!

In PS 5, you'll build a 4-bit version!

- More efficient than an adder built using minterm expansion.
  - 16-bit minterm-based adder: need several *billion* gates
  - 16-bit ripple-carry adder: only need *hundreds* of gates

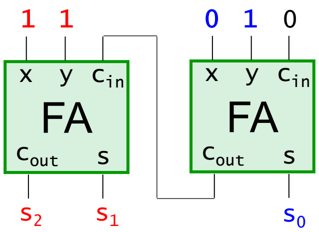
2-Bit Ripple-Carry Adder

- Schematic:



$$\begin{array}{r} x_1x_0 \\ +\ y_1y_0 \\ \hline s_2s_1s_0 \end{array}$$

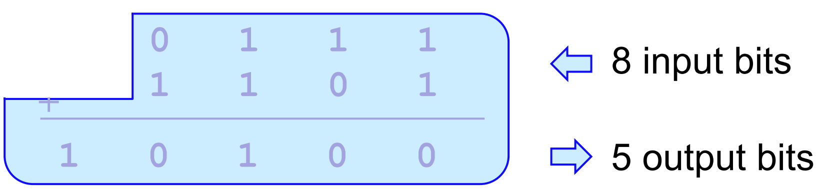
- Here's an example computation:



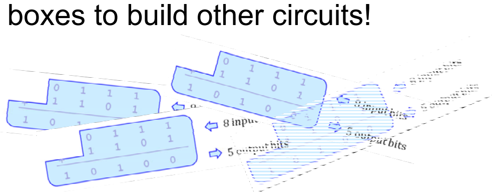
$$\begin{array}{r} 10 \\ +\ \ 11 \\ \hline \end{array}$$

More Modular Design!

- Once you build a 4-bit ripple-carry adder, you can treat it as a "black box".



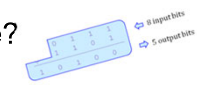
- Use these boxes to build other circuits!



Also in PS 5: Building a 4 x 2 Multiplier

		1	1	0	1	first factor (4 bits)
x				1	0	second factor (2 bits)
		0	0	0	0	2 partial products
	1	1	0	1		
	1	1	0	1	0	final answer

- How could you use a 4-bit ripple-carry adder here?



- What other smaller circuit might we want to build first so that we can use it as part of the 4 x 2 multiplier?

Two Key Components of a Computer

program + data

CPU ⟷ **RAM**

central processing unit — random access memory

CPU:
- all computation happens here
- adders, multipliers, etc.
- small number of *registers* for storing values

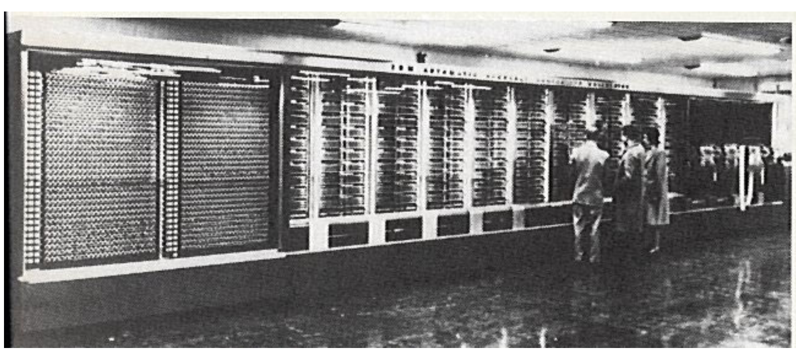
RAM:
- lots of room for storage
- no computation happens here

- Program instructions are stored *with the data* in RAM.
- Instructions and data are transferred back and forth between RAM and the CPU.

von Neumann Architecture

- John von Neumann was the one who proposed storing programs in memory.

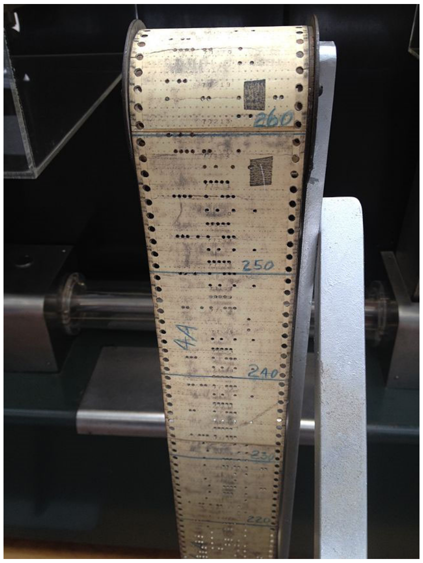
Early Computers



The Mark I: Howard Aiken, Grace Hopper, et al.; Harvard, the 1940s/50s

- In the first computers, programs were stored *separately* from the data.

Early Computers (cont.)

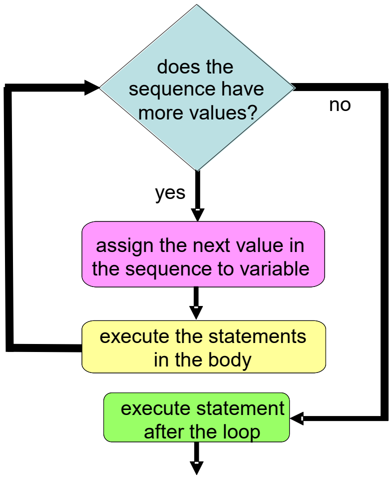


an external program tape for the Mark I

Recall: Executing a `for` Loop

```
for variable in sequence:
    body of the loop
```

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```



Another Example

- What would this code output?

```
for val in [2, 4, 6, 8, 10]:
    print(val * 10)
print(val)
```

- Use a table to help you:

more?	val	output/action

Definite Loops in Python (cont.)

Computer Science 111
Boston University

Recall: Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):          # [0, 1, 2, ..., N – 1]
    body of the loop
```

- What would this loop do?

```
for i in range(8):
    print('I'm feeling loopy!')
```

Simple Repetition Loops (cont.)

- Another example:

```
for i in range(7):
    print(i * 5)
```

how many repetitions?

output?

To print the warning 20 times, how could you fill in the blank?

```
for i in ________________:
    print('Warning!')
```

A. `range(20)`

B. `[1] * 20`

C. `'abcdefghijklmnopqrst'`

D. either A or B would work, but not C

E. A, B or C would work

To add the numbers in the list `vals`, how could you fill in the blanks?

```
def sum(vals):
    result = 0
    for ______________________________:
        result += ____________________
    return result
```

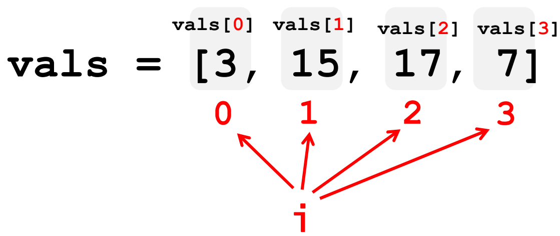
	first blank	second blank
A.	`x in vals`	`x`
B.	`x in vals`	`vals[x]`
C.	`i in range(len(vals))`	`vals[i]`
D.	either A or B would work, but not C	
E.	either A or C would work, but not B	

Option A Produces an Element-Based `for` Loop

```
vals = [3, 15, 17, 7]

     x
```

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

Option C Produces an Index-Based `for` Loop




```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

Both Versions Perform a Cumulative Computation

```
def sum(vals):
    result = 0         # the accumulator variable
    for x in vals:
        result += x    # gradually accumulates the sum
    return result

print(sum([10, 20, 30, 40, 50]))
```

What is the output of this program?

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i-1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 6, 6, 5]))
```

i	vals[i]	vals[i-1]	result

Follow-Up Questions

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i-1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 6, 6, 5]))
```

- Element-based or index-based loop?
- What does this program do in general?
- Could we easily do this with the other type of loop?

Simpler

```
vals = [3, 15, 17, 7]
```

x

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

element-based loop

More Flexible

```
       vals[0] vals[1] vals[2] vals[3]
vals = [3,     15,     17,     7]
        0      1       2       3
```

i

```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

index-based loop

More on Cumulative Computations

- Here's a loop-based factorial in Python:

```
def fac(n):
    result = 1
    for x in range(____________):   # fill in the blank
        result *= x
    return result
```

- Is this loop element-based or index-based?

Pre-Lecture
Indefinite Loops

Computer Science 111
Boston University

Staying on the Same Line When Printing

- By default, `print` puts an invisible *newline* character at the end of whatever it prints.
  - causes separate `print`s to print on different lines
- Example:

```
for i in range(7):
    print(i * 5)
```

```
0
5
10
15
20
25
30
```

Staying on the Same Line When Printing (cont.)

- To get separate `prints` to print on the same line, we can replace the newline with something else.
- Examples:

```
for i in range(7):
    print(i * 5, end=' ')
```

```
0 5 10 15 20 25 30
```

```
for i in range(7):
    print(i * 5, end=',')
```

____

`for` Loops Are *Definite* Loops

- *Definite* loop = a loop in which the number of repetitions is *fixed* before the loop even begins.
- In a `for` loop, # of repetitions = `len(`*sequence*`)`

```
for variable in sequence:
    body of the loop
```

Indefinite Loops

- Use an *indefinite loop* when the # of repetitions you need is:
  - not as obvious
  - impossible to determine before the loop begins
- Sample problem: `print_multiples(n, bound)`
  - should print all multiples of n that are less than bound
  - output for `print_multiples(9, 100)`:

```
9 18 27 36 45 54 63 72 81 90 99
```

Indefinite Loop for Printing Multiples

- Pseudocode:

```
def print_multiples(n, bound):
    mult = n
    repeat as long as mult < bound:
        print mult followed by a space
        mult = mult + n
    print a newline (go to the next line)
```

- Python:

```
def print_multiples(n, bound):
    mult = n
    while mult < bound:
        print(mult, end=" ")
        mult = mult + n
    print()

    # no value is being returned
    # function returns at the end of its block
```

Tracing a `while` Loop

- Let's trace the loop for `print_multiples(15, 70)`:

n bound

```
mult = n
while mult < bound:
    print(mult, end=' ')
    mult = mult + n
print()
```

`mult < bound`	output thus far	`mult`
		15
15 < 70 (True)	15	30
30 < 70 (True)	15 30	45
45 < 70 (True)	15 30 45	60

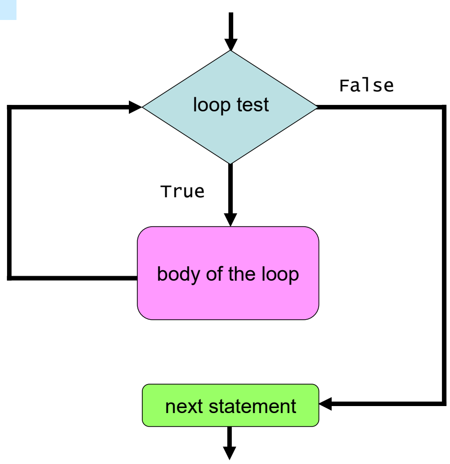
complete the rest of the table!

`while` Loops

```
while loop test:
    body of the loop
```

Steps:

1. evaluate the loop test (a boolean expression)
2. if it's `True`, execute the statements in the body, and go back to step 1
3. if it's `False`, skip the statements in the body and go to the statement after the loop



Important!

- Recall the loop in `print_multiples`:

```
mult = n
while mult < bound:
    print(mult, end=' ')
    mult = mult + n
```

- In general, a `while` loop's test includes a key "loop variable".
- We need to update that loop variable in the body of the loop.
- Failing to update it can produce an *infinite loop*!

Indefinite Loops

Computer Science 111
Boston University

Cumulative Computations with Strings

- Recall our recursive `remove_vowels` function:

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        removed_rest = remove_vowels(s[1:])
        if s[0] in 'aeiou':
            return removed_rest
        else:
            return s[0] + removed_rest
```

- Examples:

```
>>> remove_vowels('recurse')
'rcrs'
>>> remove_vowels('vowels')
'vwls'
```

Cumulative Computations with Strings (cont.)

- Here's one loop-based version:

```
def remove_vowels(s):
    result = ''              # the accumulator
    for c in s:
        if c not in 'aeiou':
            result += c      # accumulates the result
    return result
```

- Let's trace through `remove_vowels('vowels')`:

```
s = 'vowels'
 c       result
```

Recall: *Indefinite* Loops

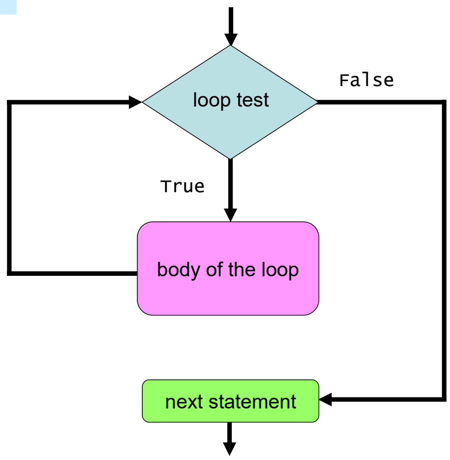
- Use an *indefinite loop* when the # of repetitions you need is:
  - not as obvious
  - impossible to determine before the loop begins
- In Python, we usually use a `while` loop for this.

Recall: while Loops

```
while loop test:
    body of the loop
```

Steps:

1. evaluate the loop test (a boolean expression)
2. if it's `True`, execute the statements in the body, and go back to step 1
3. if it's `False`, skip the statements in the body and go to the statement after the loop



Factorial Using a while Loop

- We don't need an indefinite loop, but we can still use `while`!

```
def fac(n):
    result = 1
    while n > 0:
        result *= n
        ____________      # what do we need here?
    return result
```

- Let's trace `fac(4)`:

```
n       n > 0          result
```

Factorial Three Ways!

recursion

```
def fac(n):
    if n == 0:
        return 1
    else:
        rest = fac(n-1)
        return n * rest
```

for loop

```
def fac(n):
    result = 1
    for x in range(1, n+1):
        result *= x
    return result
```

while loop

```
def fac(n):
    result = 1
    while n > 0:
        result *= n
        n = n - 1
    return result
```

Extreme Looping!

- What does this code do?

```
print('It keeps')

while True:
    print('going and')

print('Phew! Done!')
```

Choosing a Random Number

- Python's `random` module allows us to produce random numbers.
  - to use it, we need to import it:

```
import random
```

- `random.choice(vals)`
  - takes a sequence `vals`
  - randomly chooses one value from `vals` and returns it

- examples from the Shell:

```
>>> import random
>>> random.choice(range(7)) # random number from 0-6
5
>>> random.choice(range(7))
2
>>> random.choice(range(7))
4
```

Breaking Out of An Infinite Loop

```
import random

while True:
    print('Help!')
    if random.choice(range(10000)) == 111:
        break
    print('Let me out!')

print('At last!')
```

A `break` statement causes a loop to end early.

- jumps to the line that comes after the loop

- Thus, the final two lines that are printed are:

Counting the Number of Repetitions

```
import random

count = 0
while True:
    count += 1
    print('Help!')
    if random.choice(range(10000)) == 111:
        break
    print('Let me out!')

print('At last! It took', count, 'tries to escape!')
```

User Input

- Getting a *string value* from the user:

  `variable = input(prompt)` where *prompt* is a string

- Getting an *integer value*:

  `variable = int(input(prompt))`

- Getting a *floating-point value*:

  `variable = float(input(prompt))`

- Getting an arbitrary non-string value (e.g., a list):

  `variable = eval(input(prompt))`

  - eval treats a string as an expression to be evaluated

- Examples:

```
name = input('What is your name? ')
count = int(input('possible points: '))
scores = eval(input('list of scores: '))
```

Using a `while True` Loop to Get User Input

```
import math

while True:
    val = int(input('Enter a positive number: '))
    if val > 0:
        break
    else:
        print(val, 'is not positive. Try again!')

result = math.sqrt(val)
print('result =', result)
```

How many values does this loop print?

```
a = 40
while a > 2:
    a = a // 2
    print(a - 1)
```

a > 2	a	prints

For what inputs does this function return `True`?

```
def mystery(n):
    while n != 1:
        if n % 2 != 0:
            return False
        n = n // 2
    return True
```

Try tracing these two cases:

mystery(12)	mystery(8)
n	n
12	8

Program Design with Loops

Computer Science 111
Boston University

Recall: Two Types of Loops

`for`	`while`
definite loop	***indefinite loop***
For a **known** number of repetitions	For an **unknown** number of repetitions

Recall: Two Types of `for` Loops

```
vals = [3, 15, 17, 7]

   x
```

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

***element*-based loop**

```
       vals[0] vals[1] vals[2] vals[3]
vals = [3,     15,     17,     7]
        0      1       2       3

               i
```

```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

***index*-based loop**

Finding the Smallest Value in a List

- What if we needed to write a loop-based version of `min()`?

```
vals = [ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

- What strategy should we use?
- What type of loop: `for` or `while`?

How should we fill in the blank to initialize m?

```
vals = [ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m is the
"min so far"

```
def minval(vals):
    m = __________
    for x in vals:
        if x < m:
            m = x
    return m
```

Finding the *Position* of the Smallest Value

```
  0    1    2    3    4    5    6   7
[ 45, 80, 10, 30, 27, 50, 5, 15 ]
```

m = 45 → m = 10 → m = 5 →

6 should be returned

```
def minval_posn(vals):
    # initialize variable(s)

    for ______________________:
        if _________:
            # update var(s)
    return ________
```

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise
- Use a loop to check all possible divisors.
  - What are they?
  - For example, what divisors do we need to check for 41?

    `2, 3, 4, 5, 6, 7, 8, ..., 37, 38, 39, 40`
- What type of loop should we use?

Determining if a Number is Prime

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise

```
def is_prime(n):
    max_div = int(math.sqrt(n))    # max possible divisor

    # try all possible divisors
    ____________________________________:
        if ________________:
            return ________   # when can we return "early"?

    # If we get here, what must be the case?
    return ____________
```

Does this version work?

- Write a function `is_prime(n)` that:
  - returns `True` if `n` is prime
  - returns `False` otherwise

```
def is_prime(n):
    max_div = int(math.sqrt(n))   # max possible divisor

    # try all possible divisors
    for div in range(2, max_div + 1):
        if n % div == 0:
            return False
        else:
            return True
```

Another Sample Problem

- `any_below(vals, cutoff)`
  - should return `True` if *any* of the values in `vals` is < `cutoff`
  - should return `False` otherwise
- examples:
  - `any_below([50, 18, 25, 30], 20)` should return `True`
  - `any_below([50, 18, 25, 30], 10)` should return `False`
- How should this method be implemented using a loop?

```
def any_below(vals, cutoff):
    for ____ in _____________:
        if ________________:
```

Which of these works?

A.

```
def any_below(vals, cutoff):
    for x in vals:
        if x >= cutoff:
            return False

    return True
```

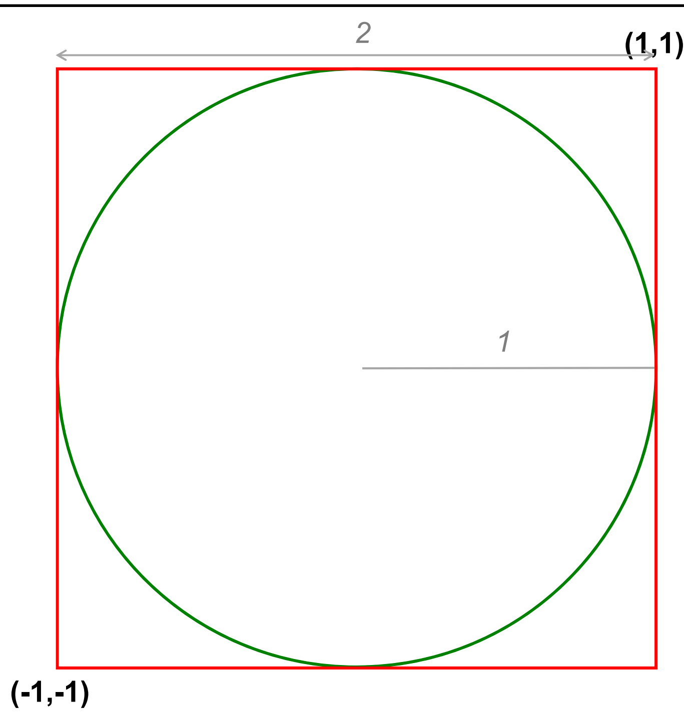
B.

```
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True

    return False
```

C.

```
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True
        else:
            return False
```

D. more than one of them



Estimating π
by dart throwing!

$$\frac{\bigcirc\ \text{area}}{\square\ \text{area}} = \frac{\pi}{4}$$

$$\pi = \frac{4 * \bigcirc\ \text{area}}{\square\ \text{area}}$$

"hits"

$$\pi \sim \frac{4 * \text{darts in} \bigcirc}{\text{darts in} \square}$$

"throws"

Loops: `for` or `while`?

`pi_one(e)` — ***e == how close to π we need to get***

`pi_two(n)` — ***n == number of darts to throw***

Which function will use which kind of loop?

Thinking in Loops

`for`	`while`
definite iteration	***indefinite iteration***
For a **known** number of repetitions	For an **unknown** number of repetitions

Pre-Lecture
Nested Loops

Computer Science 111
Boston University

Repeating a Repetition!

```
for i in range(3):          # 0, 1, 2
    for j in range(4):      # 0, 1, 2, 3
        print(i, j)
```

```
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
```

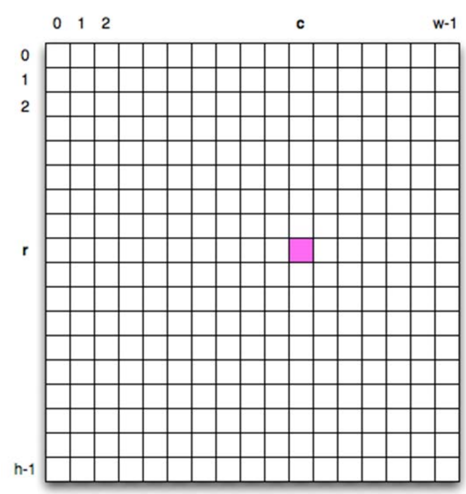
Repeating a Repetition!

```
for i in range(3):
    for j in range(4):
        print(i, j)
    print('---')
```

inner loop

outer loop

Nested loops and ***2D structure***




```
for r in range(h):
    for c in range(w):
        # process the pixel at (r, c)
```

Tracing a Nested `for` Loop

```
for i in range(5):      # [0,1,2,3,4]
    for j in range(i):
        print(i, j)
```

i	range(i)	j	value printed

Nested Loops

Computer Science 111
Boston University

Nested Loops!


```
for y in range(84):
  for m in range(12):
    for d in range(f(m,y)):
      for h in range(24):
        for mn in range(60):
          for s in range(60):
            tick()
```

How many lines are printed?

```
for i in range(5):
    for j in range(7):
        print(i, j)
```

Recall: Tracing a Nested `for` Loop

```
for i in range(5):      # [0,1,2,3,4]
    for j in range(i):
        print(i, j)
```

i	range(i)	j	value printed
0	[]	none	nothing (we exit the inner loop)
1	[0]	0	1 0
2	[0,1]	0	2 0
		1	2 1
3	[0,1,2]	0	3 0
		1	3 1
		2	3 2
4	[0,1,2,3]	0	4 0
		1	4 1
		2	4 2
		3	4 3

full output:

```
1 0
2 0
2 1
3 0
3 1
3 2
4 0
4 1
4 2
4 3
```

Second Example: Tracing a Nested `for` Loop

```
for i in range(4):
    for j in range(i, 3):
        print(i, j)
    print(j)
```

i	range(i, 3)	j	value printed

Using Loops: T.T. Securities (TTS)

Analyzes a sequence of stock prices

	day 0	day 1	day 2	day 3	day 4	day 5	day 6	day 7

```
prices = [45, 80, 10, 30, 27, 50, 5, 15]
```

You will implement a menu of options:

```
(0) Input a new list of prices
(1) Print the current list
(2) Find the latest price
(3) Find the average price
...
(8) Quit
Enter your choice:
```

Our starter code

```
def display_menu():
    """ prints a menu of options
    """
    print()
    print('(0) Input a new list of prices')
    print('(1) Print the current prices')
    print('(2) Find the latest price')
    ## Add the new menu options here.

    print('(8) Quit')
    print()

...
```

Our starter code

```
def tts():
    prices = []
    while True:
        display_menu()
        choice = int(input('Enter your choice: '))
        print()
        if choice == 0:
            prices = get_new_prices()
        elif choice == 8:
            break
        elif choice == 1:
            print_prices(prices)
        elif choice == 2:
            latest = latest_price(prices)
            print('The latest price is', latest)
        ## add code to process the other choices here
        ...
    print('See you yesterday!')
```

The remainder of the program

- Each menu option will have its own helper function.
- Each function will use one or more loops.
  - ***most of them will <u>not</u> be nested!***
- You may *not* use the built-in `sum`, `min`, or `max` functions.
  - use your own loops instead!

T.T. Securities
==
Time Travel
Securities!

```
(0) Input a new list of prices
(1) Print the current list
(2) Find the latest price
(3) Find the average price
...
(7) Your TTS investment plan
(8) Quit
Enter your choice:
```

The TTS Advantage!

```
prices = [45, 80, 10, 30, 27, 50, 5, 15]
```

Day	Price
0	45.00
1	80.00
2	10.00
3	30.00
4	27.00
5	50.00
6	5.00
7	15.00

Time travel into the future to find the best days to buy and sell!

What is the TTS investment plan for the prices shown here?

To be realistic, however (for the SEC), you may only sell ***after*** you buy.

Finding a minimum difference

`diff` should return the **smallest** absolute diff. between any value from `l1` and any value from `l2`.

```
                l1        l2
>>> diff([12,3,7], [6,0,5])
1
```

- How can we try all possible pairs of values?

- As we try pairs, we keep track of the min diff thus far:

```
def diff(l1, l2):
    mindiff = abs(l1[0]-l2[0])
    for x in l1:
        for y in l2:
            d = abs(x - y)
            if d < mindiff:
                mindiff = d
    return mindiff
```

What if we want the *indices* of the min-diff values?

```
                    l1        l2
>>> diff_indices([12,3,7], [6,0,5])
[2, 0]
```

index of value in `l2`

index of value in `l1`

```
def diff_indices(l1, l2):    # what needs to change?
    mindiff = abs(l1[0] - l2[0])

    for x in l1:
        for y in l2:
            d = abs(x - y)
            if d < mindiff:
                mindiff = d

    return mindiff
```

What if we want the *indices* of the min-diff values?

```
>>> diff_indices([12,3,7], [6,0,5])
[2, 0]
```

$\underbrace{[12,3,7]}_{l1}$, $\underbrace{[6,0,5]}_{l2}$

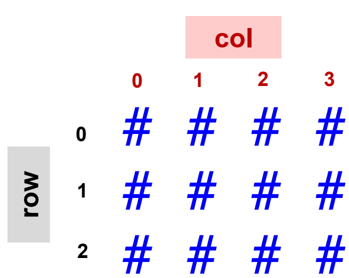
index of value in l2

index of value in l1

```
def diff_indices(l1, l2):
    mindiff = abs(l1[0] - l2[0])
    pos1 = 0
    pos2 = 0

    for i in range(len(l1)):
        for j in range(len(l2)):
            d = abs(l1[i] - l2[j])
            if d < mindiff:
                mindiff = d
                pos1 = i
                pos2 = j
    return [pos1, pos2]
```

Printing Patterns

```
for row in range(3):
    for col in range(4):
        print('#', end=' ')
    print()  # go to next line
```



Fill in the Blank #1

```
for row in range(3):
    for col in range(6):
        print(_____, end=' ')
    print()  # go to next line
```

col

row

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 4 5
```

Fill in the Blank #2

```
for row in range(3):
    for col in range(6):
        print(_____, end=' ')
    print()  # go to next line
```

col

row

```
0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
```

What is needed in the blanks to get this pattern?

```
for row in range(5):
    for col in ___________:
        print(_____, end=' ')
    print()  # go to next line
```

```
0 0 0 0 0
1 1 1 1
2 2 2
3 3
4
```

What is needed in the blank to get this pattern?

```
for row in range(3):
    for col in range(6):
        print(_____, end=' ')
    print()  # go to next line
```

```
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
```

if you have time...

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
```

ASCII art...?   How about ***ASCII video!***

http://www.asciimation.co.nz/

<u>Pre-Lecture</u>

References and Mutable Data

Computer Science 111
Boston University

Recall: Variables as Boxes

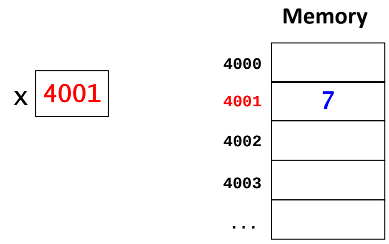
- You can picture a variable as a named "box" in memory.
- Example from an earlier lecture:

```
num1 = 100
num2 = 120
```

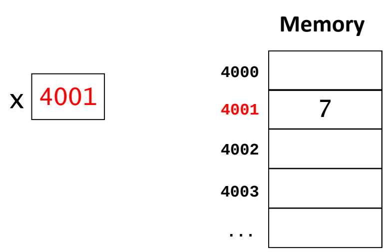
num1 $\boxed{100}$ num2 $\boxed{120}$

Variables and Values

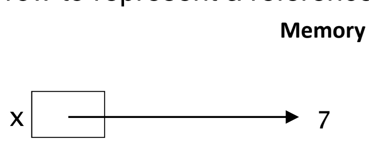
- In Python, when we assign a value to a variable, we're not actually storing the value *in* the variable.
- Rather:
  - the value is somewhere else in memory
  - the variable stores the *memory address* of the value.
- Example: `x = 7`



References

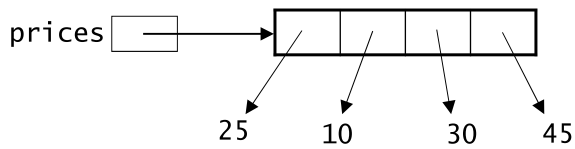


- We say that a variable stores a *reference* to its value.
  - also known as a *pointer*
- Because we don't care about the actual memory address, we use an arrow to represent a reference:



Lists and References

```
prices = [25, 10, 30, 45]
```



- When a variable represents a list, it stores a reference to the list.
- The list itself is a *collection* of references!
  - each element of the list is a reference to a value

Mutable vs. Immutable Data

- In Python, strings and numbers are *immutable*.
  - their contents/components cannot be changed
- Lists are *mutable.*
  - their contents/components *can* be changed
  - example:

```
>>> prices = [25, 10, 30, 45]
>>> prices[2] = 50
>>> print(prices)
[25, 10, 50, 45]
```

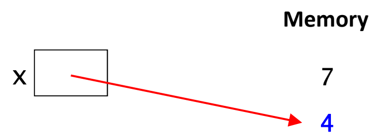
Changing a Value vs. Changing a Variable

- There's no way to change an immutable value like 7.

```
x = 7
```

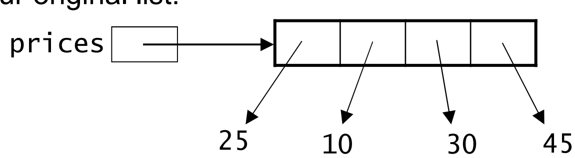
- However, we *can* use assignment to change the variable—making it refer to a different value:

```
x = 4
```



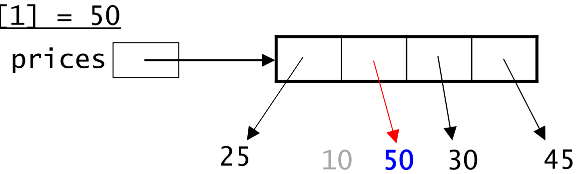
Changing a Value vs. Changing a Variable

- Here's our original list:



- Lists are mutable, so we *can* change the value (the list) by modifying its elements:

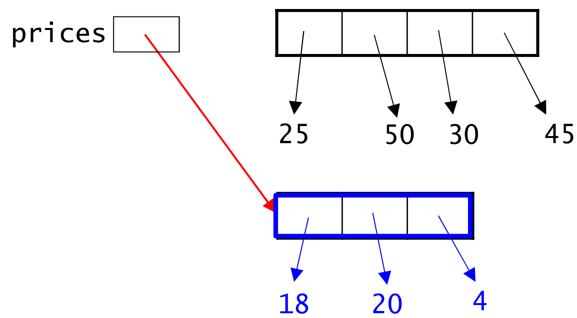
```
prices[1] = 50
```



Changing a Value vs. Changing a Variable

- We can also change the variable—making it refer to a completely different list:

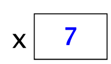
```
prices = [18, 20, 4]
```



Simplifying Our Mental Model

- When a variable represents an immutable value, it's okay to picture the value as being *inside* the variable.

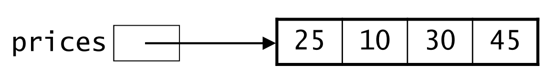
```
x = 7
```



  - a simplified picture, but good enough!

- The same thing holds for list elements that are immutable.

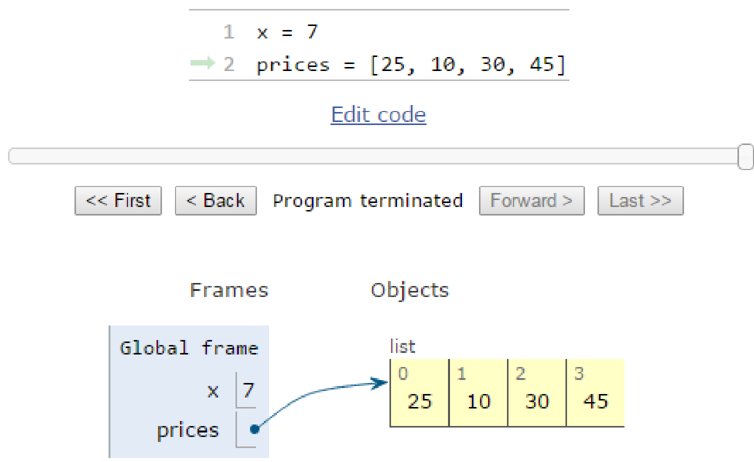
```
prices = [25, 10, 30, 45]
```



- We still need to use references for *mutable* data like lists.

Simplifying Our Mental Model (cont.)

- Python Tutor uses this simplified model, too:



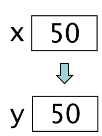
Copying Variables

- The assignment

  var2 = var1

  copies the contents of *var1* into *var2*:

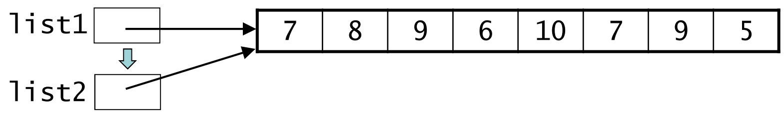
```
x = 50
y = x
```



Copying References

- Consider this example:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



- Given the lines of code above, what will the lines below print?

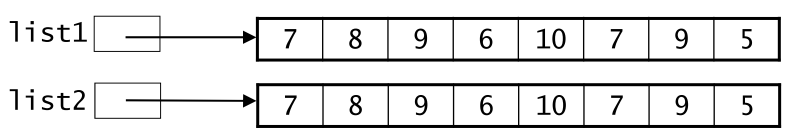
```
list2[2] = 4
print(list1[2], list2[2])
```

- Copying a list variable simply copies the reference.
- It doesn't copy the list itself!

Copying a List

- We can copy a list like this one using a full slice:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



- What will this print now?

```
list2[2] = 4
print(list1[2], list2[2])
```

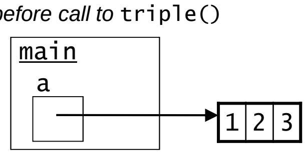
Passing a List to a Function

- When a list is passed into a function:
  - the function gets a copy of the *reference* to the list
  - it does *not* get a copy of the list itself
- Thus, if the function changes the components of the list, those changes will be there when the function returns.
- Consider the following program:

```
def main():
    a = [1, 2, 3]
    triple(a)
    print(a)


def triple(vals):
    for i in range(len(vals)):
        vals[i] = vals[i] * 3
```

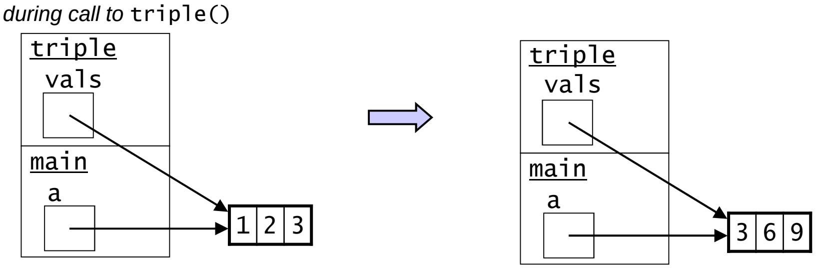
Passing a List to a Function (cont.)

before call to `triple()`

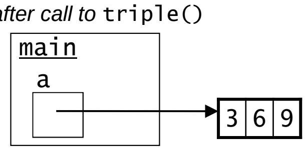



```
def main():
    a = [1, 2, 3]
    triple(a)
    print(a)      # prints [3, 6, 9]
```

during call to `triple()`



after call to `triple()`

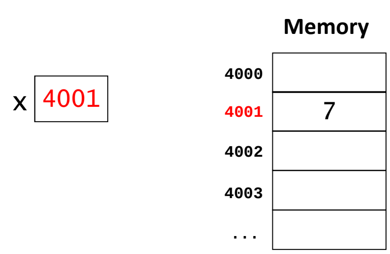


References and Mutable Data

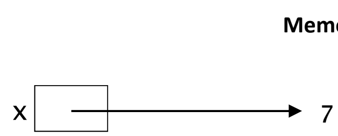
Computer Science 111
Boston University

Recall: References

x = 7

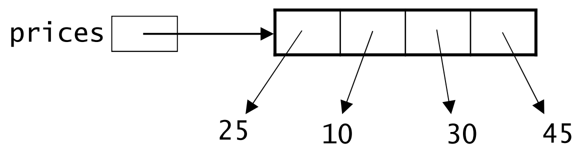


- Because we don't care about the actual memory address, we use an arrow to represent a reference:



Recall: Lists and References

```
prices = [25, 10, 30, 45]
```



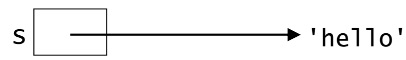
Mutable vs. Immutable Data

- In Python, strings and numbers are *immutable*.
  - their contents/components cannot be changed
- Lists are *mutable.*
  - their contents/components *can* be changed

Changing a Value vs. Changing a Variable

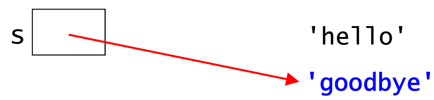
- There's no way to change an immutable *value* like `'hello'`.

`s = 'hello'`

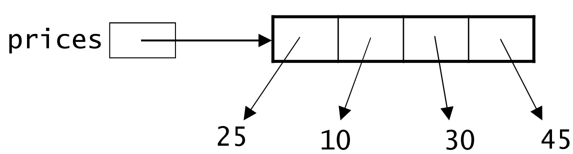


- However, we *can* change the *variable*:

`s = 'goodbye'`

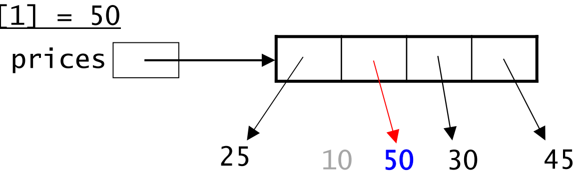


Changing a Value vs. Changing a Variable



- Lists are mutable, so we *can* change the value (the list) by modifying its elements:

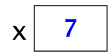
`prices[1] = 50`



Recall: Simplifying Our Mental Model

- When a variable represents an immutable value,
  it's okay to picture the value as being *inside* the variable.

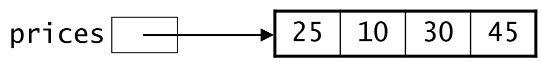
```
x = 7
```



  - a simplified picture, but good enough!

- The same thing holds for list elements that are immutable.

```
prices = [25, 10, 30, 45]
```

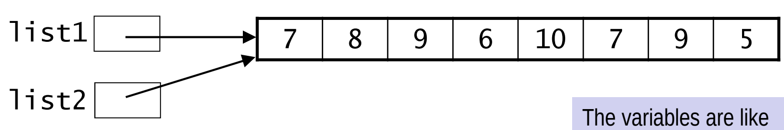


- **We still need to use references for *mutable* data like lists.**

Recall: Copying References

- Consider this example:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



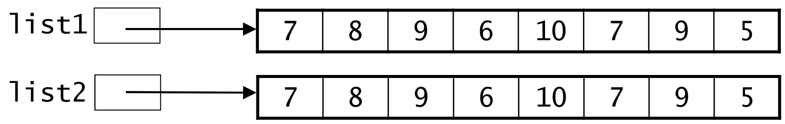
The variables are like two business cards that both have the address of the same office.

The list is the office.

Recall: Copying a List

- We can copy a list like this one using a full slice:

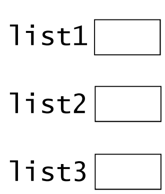
```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



The variables are like business cards for two offices at different addresses. The two offices just happen to have the same contents!

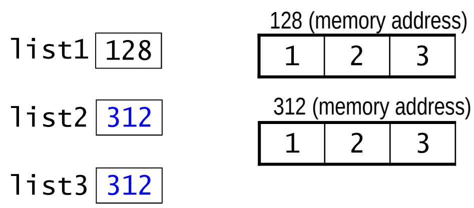
What does this program output?

```
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```



Another Way to Picture References

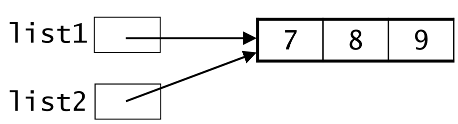
```
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```



Changing the Internals vs. Changing a Variable

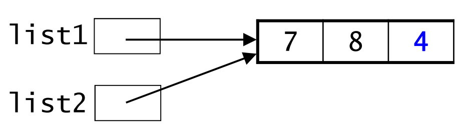
- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```



- ...if we change *the internals* of the list, both variables will "see" the change:

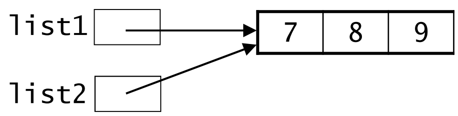
```
list2[2] = 4
print(list1)    # prints [7, 8, 4]
```



Changing the Internals vs. Changing a Variable (cont.)

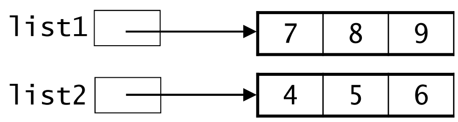
- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```



- ...if we change one of the variables *itself*, that does *not* change the other variable:

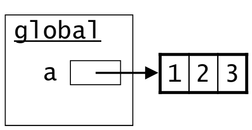
```
list2 = [4, 5, 6]
print(list1)    # prints [7, 8, 9]
```



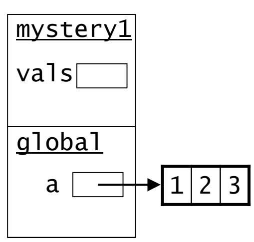
Passing a List to a Function, version 1

```
def mystery1(vals):
    vals[1] = 4    # changes the internals of the list

a = [1, 2, 3]
mystery1(a)
print(a)
```

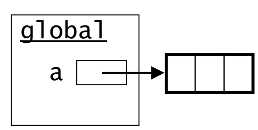
before mystery1



during mystery1

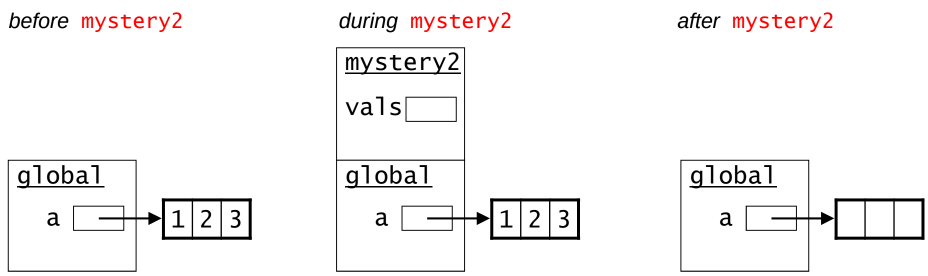


after mystery1



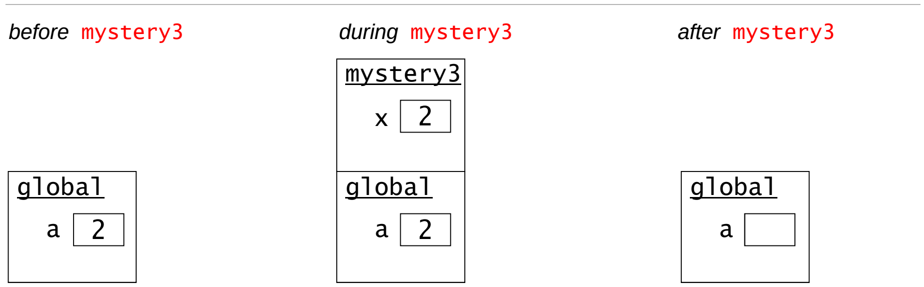
Passing a List to a Function, *version 2*

```
def mystery2(vals):
    vals = [1, 4, 3]    # changes the variable itself

a = [1, 2, 3]
mystery2(a)
print(a)
```



Passing an *Immutable* Value to a Function

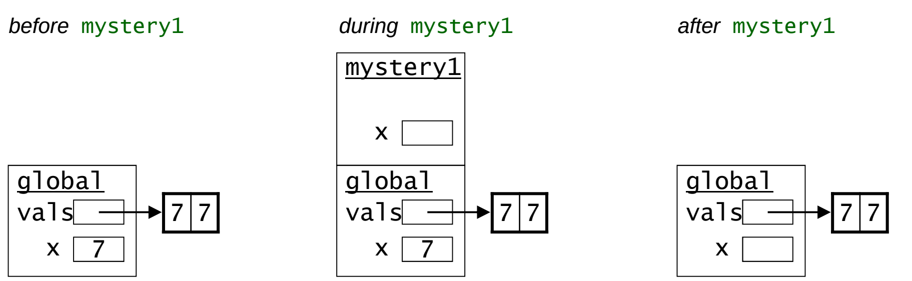
```
def mystery3(x):
    x = x * 2     # changes the variable itself

a = 2
mystery3(a)
print(a)
```



Because the value is immutable, we can think of the function getting a copy of the value.

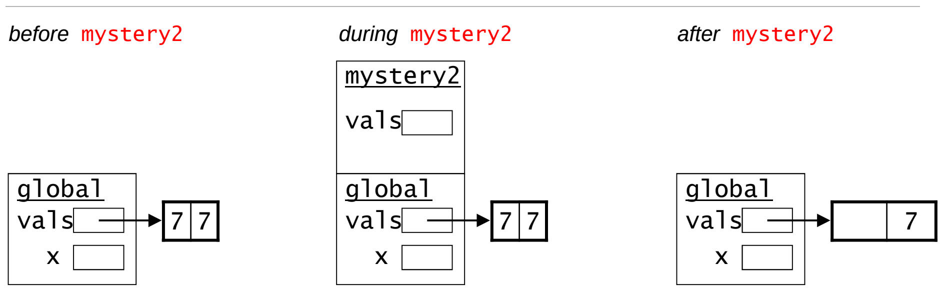
What does this program output? (part I)

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)
```



What does this program output? (part II)

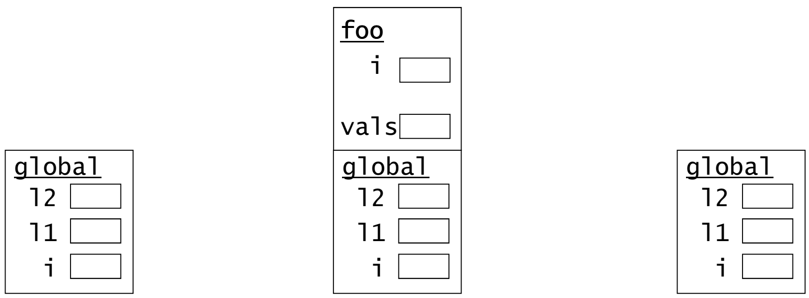
```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)
```



Extra Practice:
What does this program print?
Draw your own memory diagrams!

```
def foo(vals, i):
    i += 1
    vals[i] *= 2

i = 0
l1 = [1, 1, 1]
l2 = l1
foo(l2, i)
print(i, l1, l2)
```

before foo | *during* foo | *after* foo



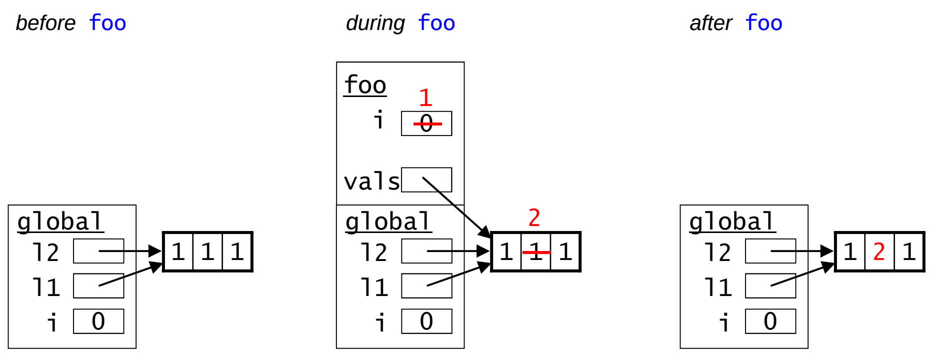
Recall Our Earlier Example...

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
mystery1(x)
mystery2(vals)
print(x, vals)
```

How can we make the *global* `x` reflect `mystery1`'s change?

Extra Practice:
What does this program print?
Draw your own memory diagrams!

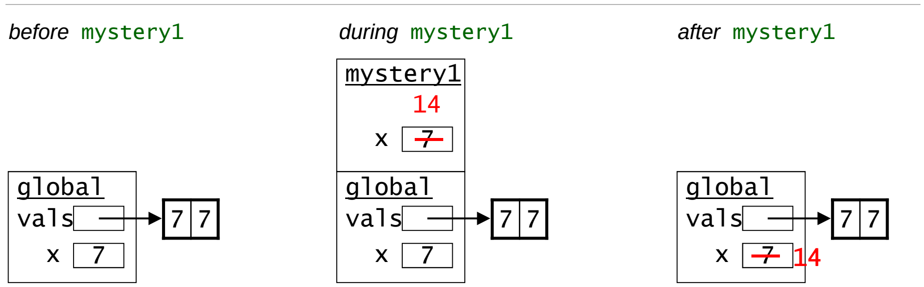
```
def foo(vals, i):
    i += 1
    vals[i] *= 2

i = 0
l1 = [1, 1, 1]
l2 = l1
foo(l2, i)
print(i, l1, l2)       # output: 0 [1, 2, 1] [1, 2, 1]
```



Recall Our Earlier Example...

How can we make the *global* x reflect mystery1's change?

```
def mystery1(x):
    x *= 2
    return x
def mystery2(vals):
    vals[0] = 111
    return vals

x = 7
vals = [7, 7]
x = mystery1(x)     # assign the return value!
mystery2(vals)
print(x, vals)
```



Pre-Lecture
2-D Lists

Computer Science 111
Boston University

2-D Lists

- Recall that a list can include sublists

```
mylist = [17, 2, [2, 5], [1, 3, 7]]
```

- To capture a rectangular table or grid of values, use a *two-dimensional* list:

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

  - a list of sublists, each with the same length
  - each sublist is one "row" of the table

Dimensions of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

`len(table)` is the # of rows in `table`

`table[r]` is the row with index `r`

`len(table[r])` is the # of elements in row `r`

`len(table[0])` is the # of columns in `table`

Picturing a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- Here's one way to picture the above list:

row indices ↓ / column indices →	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	15	8	3	16	12	7	9	5
1	6	11	9	4	1	5	8	13
2	17	3	5	18	10	6	7	21
3	8	14	13	6	13	12	8	4
4	1	9	5	16	20	2	3	9

Accessing an Element of a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

`table[r][c]` is the element at row `r`, column `c` in `table`

examples:

```
>>> print(table[2][1])
3
```

row index

column index

```
>>> table[-1][-2] = 0
```

Using Nested Loops to Process a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9   5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

```
for r in range(len(table)):
    for c in range(len(table[0])):
        # process table[r][c]
```

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] > 15:
            count += 1
print(count)
```

r	c	table[r][c]	count

2-D Lists;
References Revisited

Computer Science 111
Boston University

2-D Lists

- Recall that a list can include sublists

```
mylist = [17, 2, [2, 5], [1, 3, 7]]
```

  - what is `len(mylist)`?

- To capture a rectangular table or grid of values, use a *two-dimensional* list:

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

  - a list of sublists, each with the same length
  - each sublist is one "row" of the table

2-D Lists: Try These Questions!

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- what is `len(table)`?
- what does `table[0]` represent?
  `table[1]`?
  `table[-1]`?
- what is `len(table[0])`?
- what is `table[3][1]`?
- how would you change the `1` in the lower-left corner to a 7?

Which Of These Counts the Number of Evens?

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
```

A.
```
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
```

B.
```
count = 0
for r in len(table):
    for c in len(table[0]):
        if c % 2 == 0:
            count += 1
```

C.
```
count = 0
for r in range(len(table[0])):
    for c in range(len(table)):
        if table[r][c] % 2 == 0:
            count += 1
```

D. either A or B

E. either A or C

Using Nested Loops to Process a 2-D List

```
table = [[15, 19,  3, 16],
         [ 6, 21,  9,  4],
         [17,  3,  5, 18]]
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
print(count)
```

r	c	table[r][c]	count

Recall: Picturing a 2-D List

```
table = [[15,  8,  3, 16, 12,  7,  9,  5],
         [ 6, 11,  9,  4,  1,  5,  8, 13],
         [17,  3,  5, 18, 10,  6,  7, 21],
         [ 8, 14, 13,  6, 13, 12,  8,  4],
         [ 1,  9,  5, 16, 20,  2,  3,  9]]
```

- Here's one way to picture the above list:

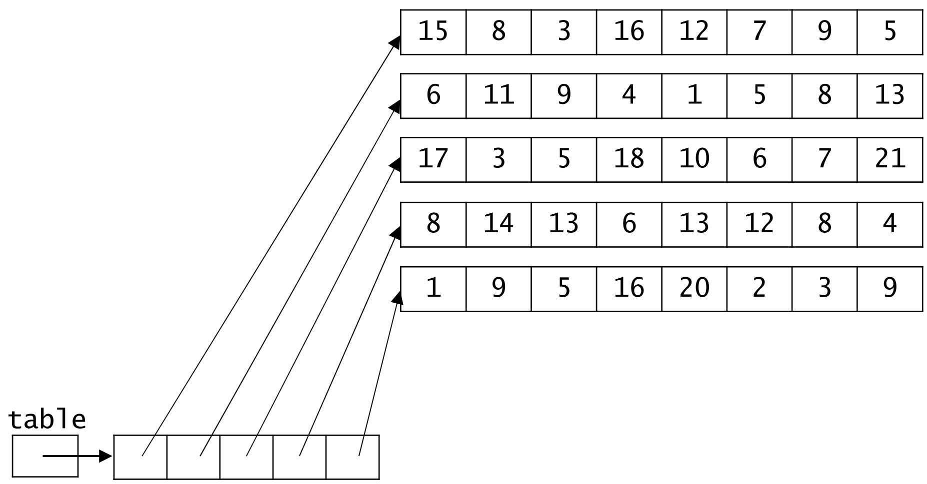
column indices

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	15	8	3	16	12	7	9	5
1	6	11	9	4	1	5	8	13
2	17	3	5	18	10	6	7	21
3	8	14	13	6	13	12	8	4
4	1	9	5	16	20	2	3	9

row indices

Picturing a 2-D List (cont)

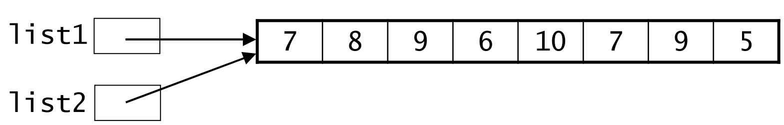
- Here's a more accurate picture:



Recall: Copying a List

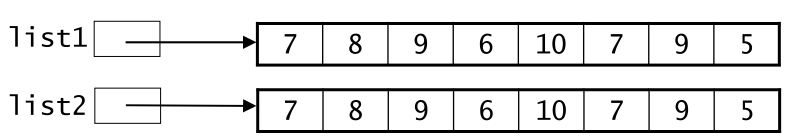
- We can't copy a list by a simple assignment:

```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```



- We can copy this list using a full slice:

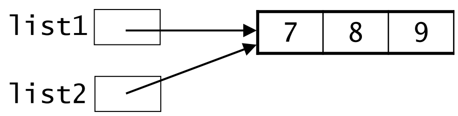
```
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```



Changing the Internals vs. Changing a Variable

- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

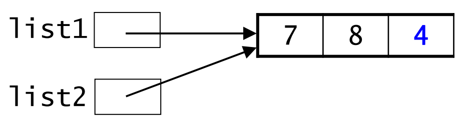


The variables are like two business cards that both have the address of the same office.

The list is the office.

- ...if we change *the internals* of the list, both variables will "see" the change:

```
list2[2] = 4
print(list1)     # prints [7, 8, 4]
```



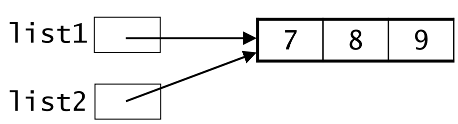
We're changing the contents of the office.

Using either business card to find the office will lead you to see the changed contents.

Changing the Internals vs. Changing a Variable (cont.)

- When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

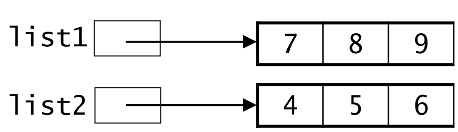


The variables are like two business cards that both have the address of the same office.

The list is the office.

- ...if we change one of the variables *itself*, that does *not* change the other variable:

```
list2 = [4, 5, 6]
print(list1)     # prints [7, 8, 9]
```

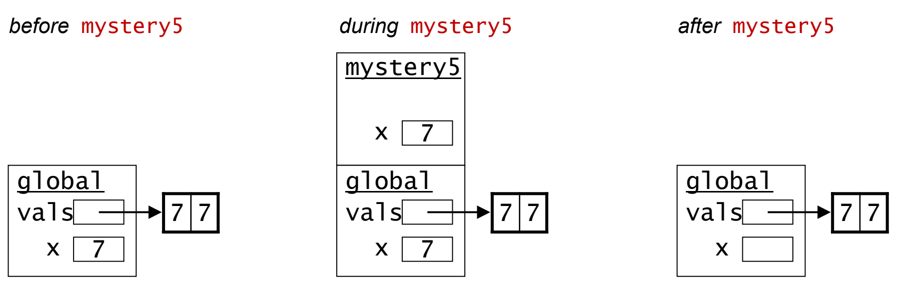


We're changing the address on one of the business cards.
It now refers to a different office.

The other business card still refers to the original unchanged office!

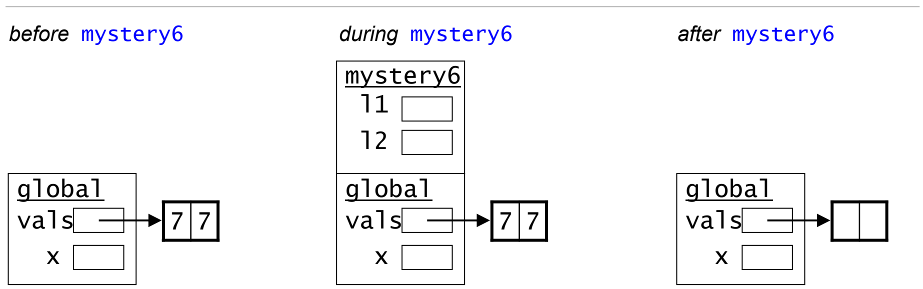
What is the output of this program? (part I)

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```



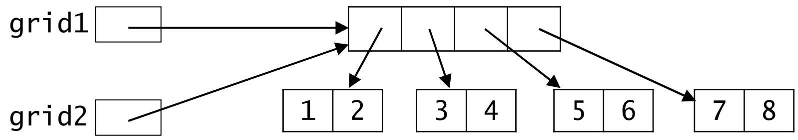
What is the output of this program? (part II)

```
def mystery5(x):
    x = x * -1
    return x
def mystery6(l1, l2):
    l1[0] = 0
    l2 = [1, 1]

x = 7
vals = [7, 7]
mystery5(x)
mystery6(vals, vals)
print(x, vals)
```



Copying a 2-D List

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
```

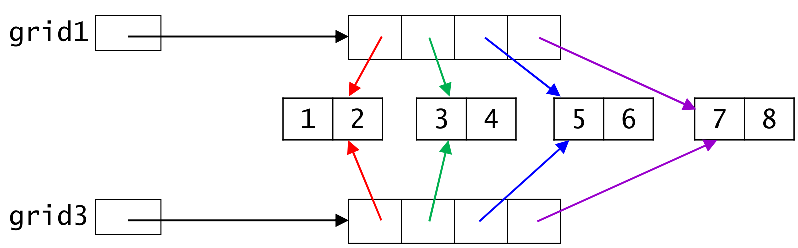


- This still doesn't copy the list: `grid2 = grid1`
  (see above)

- This doesn't either! `grid3 = grid1[:]`
  (see next slide)

A *Shallow* Copy

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
grid3 = grid1[:]
```



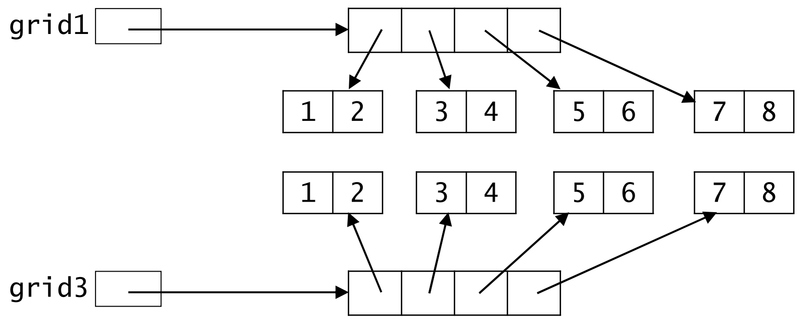
- `grid1` and `grid3` now share the same sublists.
  - known as a *shallow* copy

- What would this print?

```
grid1[1][1] = 0
print(grid3)
```

A *Deep* Copy: Nothing is Shared

```
grid1 = [[1, 2], [3, 4], [5, 6], [7, 8]]
```



- In PS 7, you'll see one way to do this.

Pre-Lecture
Using Objects

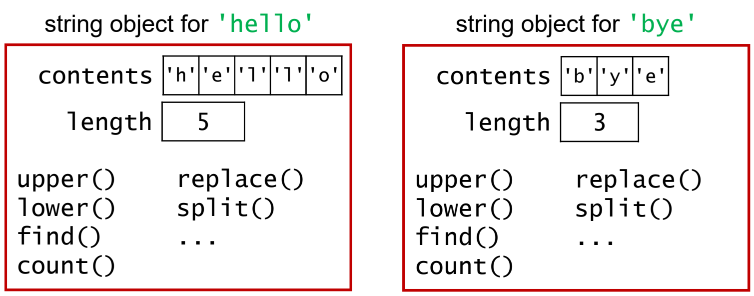
Computer Science 111
Boston University

What Is An Object?

- An object is a construct that groups together:
  - one or more data values (the object's *attributes*)
  - one or more functions that operate on those data values (known as the object's *methods*)

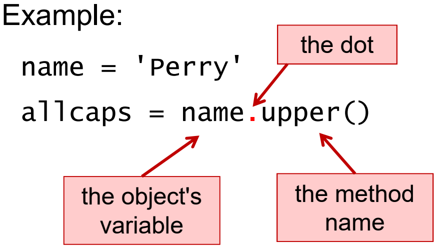
Strings Are Objects

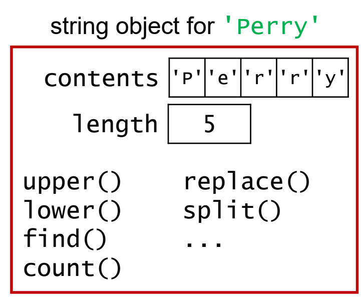
- In Python, a string is an object.
  - ***attributes:***
    - the characters in the string
    - the length of the string
  - ***methods:*** functions inside the string that we can use to operate on the string



Calling a Method

- An object's methods are inside the object, so we use *dot notation* to call them.
- Example:





- Because a method is inside the object, it is able to access the object's attributes.

String Methods (partial list)

- `s.upper()`: return a copy of `s` with all uppercase characters
- `s.lower()`: return a copy of `s` with all lowercase characters

```
>>> name = 'Perry'
>>> name.lower()
'perry'
>>> name
'Perry'        # original string is unchanged!
```

- `s.find(sub)`: return the index of the first occurrence of the substring `sub` in the string `s` (-1 if not found)
- `s.count(sub)`: return the number of occurrences of the substring `sub` in the string `s` (0 if not found)
- `s.replace(target, repl)`: replace all occurrences of the substring `target` in `s` with the substring `repl`

Splitting a String

- The `split()` method breaks a string into a list of substrings.

```
>>> name = '       Martin Luther   King    '
>>> name.split()
['Martin', 'Luther', 'King']
>>> components = name.split()
>>> components[0]
'Martin'
```

- By default, it uses *whitespace characters* (spaces, tabs, and newlines) to determine where the splits should occur.
- You can specify a different separator:

```
>>> date = '11/10/2014'
>>> date.split('/')
```

____

Pre-Lecture
Working with Text Files

Computer Science 111
Boston University

Text Files

- A text file can be thought of as one long string.
- The end of each line is stored as a newline character (`'\n'`).
- Example: the following three-line text file

```
Don't forget!

Test your code fully!
```

is equivalent to the following string:

```
'Don't forget!\n\nTest your code fully!\n'
```

Opening a Text File

- Before we can read from a text file, we need to *open* a connection to the file.
- Example:

```
f = open('reminder.txt', 'r')
```

  where:
  - `'reminder.txt'` is the name of the file we want to read
  - `'r'` indicates that we want to read from the file
- Doing so creates an object known as a *file handle*.
  - we use the file handle to perform operations on the file

Processing a File Using Methods

- A file handle is an object.
- We can use its methods to process a file.

`reminder.txt`

```
Don't forget!

Test your code fully!
```

```
>>> f = open('reminder.txt', 'r')
>>> f.readline()
"Don't forget!\n"
>>> f.readline()
'\n'
>>> f.readline()
'Test your code fully!\n'
>>> f.readline()

______________
>>> f = open('reminder.txt', 'r')   # start over at top
>>> f.read()
"Don't forget!\n\nTest your code fully!\n"
```

Processing a File Using a `for` Loop

- We often want to read and process a file one line at a time.
- We could use `readline()` inside a loop, but...
  we don't know how many lines there are!
- Python makes it easy!

```
for line in file-handle:
    # code to process line goes here
```

  - reads one line at a time and assigns it to `line`
  - continues looping until there are no lines left

Processing a CSV File

- CSV = comma-separated values
  - each line is one *record*
  - the *fields* in a given record are separated by commas

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

Processing a CSV File

```
file = open('courses.txt')

count = 0
for line in file:
    line = line[:-1]
    fields = line.split(',')
    if fields[0] == 'CS':
        print(fields[0],fields[1])
        count += 1
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

Complete the rest of the table!

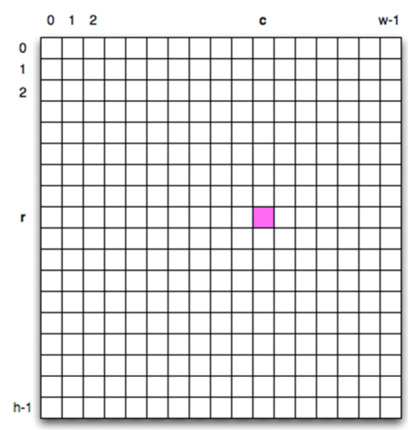
line	fields	output	count
			0
'CS,111,MWF 10-11\n'			
'CS,111,MWF 10-11'	['CS','111','MWF 10-11']	CS 111	1
'MA,123,TR 3-5\n'			
'MA,123,TR 3-5'	['MA','123','TR 3-5']	*none*	1

Using Objects;
Working with Text Files

Computer Science 111
Boston University

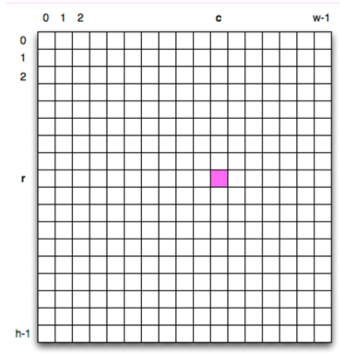
Image Processing

- An image is a 2-D collection of *pixels*.
  - h rows, w columns
- The pixel at position (r, c) tells you the color of the image at that location.
- We'll load an image's pixels into a 2-D list and process it:




```
pixels = load_pixels('my_image.png') # get a 2-D list!
h = len(pixels)
w = len(pixels[0])
for r in range(h):
    for c in range(w):
        # process pixels[r][c] in some way
```

Pixels



- Each pixel is represented by a list of 3 integers that specify its color:

  `[red, green, blue]`

  - example: the pink pixel at right has color

    `[240, 60, 225]`

  - known as RGB values
  - each value is between 0-255

- Other examples:
  - pure red: `[255, 0, 0]`
  - pure green: `[0, 255, 0]`
  - pure blue: `[0, 0, 255]`
  - white: `[255, 255, 255]`
  - black: `[0, 0, 0]`

Recall: String Methods (partial list)

- `s.lower()`: return a copy of `s` with all lowercase characters
- `s.upper()`: return a copy of `s` with all uppercase characters
- `s.find(sub)`: return the index of the first occurrence of the substring `sub` in the string `s` (-1 if not found)
- `s.count(sub)`: return the number of occurrences of the substring `sub` in the string `s` (0 if not found)
- `s.replace(target, repl)`: return a new string in which all occurrences of `target` in `s` are replaced with `repl`

Examples of Using String Methods

```
>>> chant = 'We are the Terriers!'
>>> chant.upper()

>>> chant.lower()

>>> chant.replace('e', 'o')
```

Recall: Splitting a String

- The `split()` method breaks a string into a list of substrings.

```
>>> name = 'Martin Luther King'
>>> name.split()
['Martin', 'Luther', 'King']
>>> components = name.split()
>>> components[0]
'Martin'
```

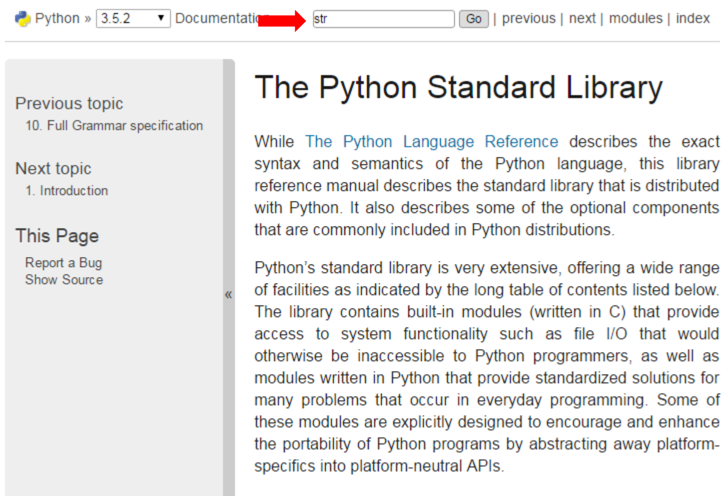
- By default, it uses *whitespace characters* (spaces, tabs, and newlines) to determine where the splits should occur.

- You can specify a different separator:

```
>>> date = '11/10/2014'
>>> date.split('/')
['11', '10', '2014']
```

Discovering What An Object Can Do

- Use the documentation for the **Python Standard Library**:
  docs.python.org/3/library



What is the output of this program?

```
s = '    programming    '
s = s.strip()
s.upper()
s = s.split('r')
print(s)
```

Recall: Processing a File Using a `for` Loop

- We often want to read and process a file one line at a time.
- We could use readline() inside a loop, but...
  - what's the problem we would face?
- Python makes it easy!

```
for line in file-handle:
    # code to process line goes here
```

  - reads one line at a time and assigns it to `line`
  - continues looping until there are no lines left

How Should We Fill in the Blank?

```
file = open('courses.txt', 'r')

count = 0
for line in file:
    line = line[:-1]
    fields = ____________________
    if fields[0] == 'CS':
        print(fields[0],fields[1])
        count += 1
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

Recall: Processing a CSV File

```
file = open('courses.txt', 'r')

count = 0
for line in file:
    line = line[:-1]
    fields = line.split(',')
    if fields[0] == 'CS':
        print(fields[0],fields[1])
        count += 1
```

courses.txt

```
CS,111,MWF 10-11
MA,123,TR 3-5
CS,105,MWF 1-2
EC,100,MWF 2-3
...
```

line	fields	output	count
			0
'CS,111,MWF 10-11\n'			
'CS,111,MWF 10-11'	['CS','111','MWF 10-11']	CS 111	1
'MA,123,TR 3-5\n'			
'MA,123,TR 3-5'	['MA','123','TR 3-5']	*none*	1
...			

(see the pre-lecture video for more!)

Closing a File

- When you're done with a file, close your connection to it:

  ```
  file.close()      # file is the file handle
  ```

  - another example of a method inside an object!
- This isn't crucial when reading from a file.
- It *is* crucial when writing to a file, which we'll do later.
  - text that you write to file may not make it
    to disk until you close the file handle!

Extracting Relevant Data from a File

- Assume that the results of a track meet are summarized in a comma-delimited text file (a *CSV file*) that looks like this:

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Len Lightning,BU,half-mile,2:15:00
Tom Turtle,UMass,half-mile,4:00:00
```

- We'd like to have a function that reads in such a results file and extracts just the results for a particular school.
  - example:

```
>>> extract_results('track_results.txt', 'BU')
Mike Mercury mile 4:50:00
Len Lightning half-mile 2:15:00
```

Extracting Relevant Data from a File

```
def extract_results(filename, target_school):
    file = open(filename, 'r')

    for line in file:
        line = line[:-1]      # chop off newline at end

        # fill in the rest of the loop body...
        # when you find a match for target_school,
        # print the athlete, event, and time.





    file.close()
```

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Len Lightning,BU,half-mile,2:15:00
Tom Turtle,UMass,half-mile,4:00:00
```

Handling Schools with No Records

- We'd like to print a message when the target school does not appear in the file.
- Would this work?

```
def extract_results(filename, target_school):
    file = open(filename, 'r')

    for line in file:
        line = line[:-1]      # chop off newline at end

        fields = line.split(',')

        if fields[1] == target_school:
            print(fields[0], fields[2], fields[3])
        else:
            print(target_school, 'not found')

    file.close()
```

Handling Schools with No Records (cont.)

- Another option: use a variable to count the matches we find.
- Would this work?

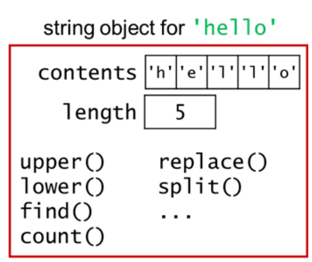
```
def extract_results(filename, target_school):
    file = open(filename, 'r')

    count = 0
    for line in file:
        line = line[:-1]      # chop off newline at end

        fields = line.split(',')
        if fields[1] == target_school:
            print(fields[0], fields[2], fields[3])
            count += 1
        if count == 0:
            print(target_school, 'not found')

    file.close()
```

Pre-Lecture
Classes:
Defining New Types of Objects

Computer Science 111
Boston University

Objects, Objects, Everywhere!

- *Recall:* Strings are objects with:
  - *attributes* – data values inside the object
  - *methods* – functions inside the object



- In fact, *everything* in Python is an object!
  - integers
  - floats
  - lists
  - booleans
  - file handles
  - ...

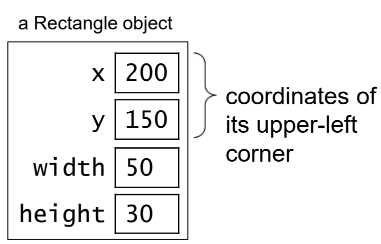
Classes

- A *class* is a blueprint – a definition of a data type.
  - specifies the attributes and methods of that type
- Objects are built according to the blueprint provided by their class.
  - they are "values" / *instances* of that type
  - use the `type` function to determine the class:

```
>>> type(111)
<class 'int'>
>>> type(3.14159)
<class 'float'>
>>> type('hello!')
_________________
>>> type([1, 2, 3])
_________________
```

Creating Your Own Classes

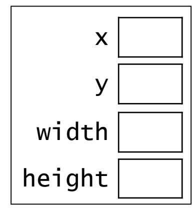
- In an *object-oriented* programming language, you can define your own classes.
  - your own types of objects
  - your own data types!
- Example: let's say that we want objects that represent rectangles.



- A `Rectangle` object could have methods for:
  - computing its area, perimeter, etc.
  - growing it (changing its dimensions), moving it, etc.

An Initial `Rectangle` Class

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """

    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

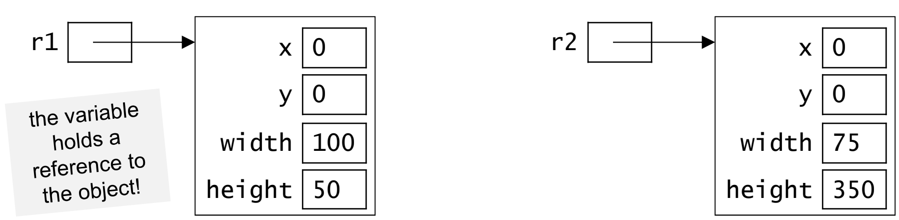


- `__init__` is the ***constructor.***
  - it's used to create new objects
  - it specifies the attributes
- Inside its methods, an object refers to itself as `self`!

Constructing and Using an Object

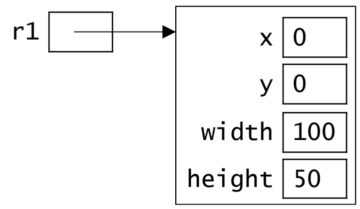
```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

```
>>> r1 = Rectangle(100, 50)     # calls __init__!
>>> r2 = Rectangle(75, 350)     # construct another one!
```



Accessing and Modifying an Object's Attributes

```
>>> r1 = Rectangle(100, 50)
```



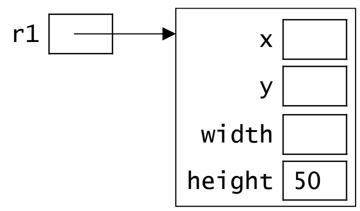
- Access the attributes using *dot notation:*

```
>>> r1.width
100
>>> r1.height
50
```

- Modify them as you would any other variable:

```
>>> r1.x = 25
>>> r1.y = 10
>>> r1.width *= 2
```

Fill in the updated field values.

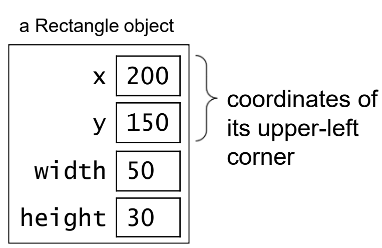


Pre-Lecture
Defining Methods

Computer Science 111
Boston University

Our Initial `Rectangle` Class

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """

    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```



Client Programs

- Our `Rectangle` class is *not* a program.
- Instead, it will be used by code defined elsewhere.
  - referred to as *client programs* or *client code*
- More generally, when we define a new type of object, we create a building block that can be used in other code.
  - just like the objects from the built-in classes: `str`, `list`, `int`, etc.
  - our programs have been clients of those classes!

Initial Client Program

```
# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
area1 = r1.width * r1.height
print('area =', area1)

print('r2:', r2.width, 'x', r2.height)
area2 = r2.width * r2.height
print('area =', area2)

# grow both Rectangles
r1.width += 50
r1.height += 10
r2.width += 5
r2.height += 30

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Using Methods to Capture an Object's Behavior

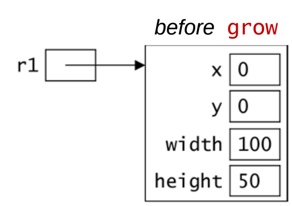
- Rather than having the client grow the `Rectangle` objects, we'd like to give each `Rectangle` object the ability to grow itself.
- We do so by adding a method to the class:

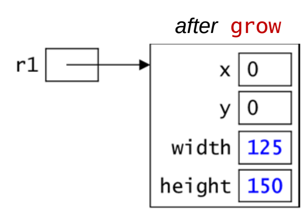
```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

Calling a Method

```
class Rectangle:
    ...
    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

```
>>> r1 = Rectangle(100, 50)

>>> r1.grow(25, 100)

>>> r1.width
125

>>> r1.height

____________
```





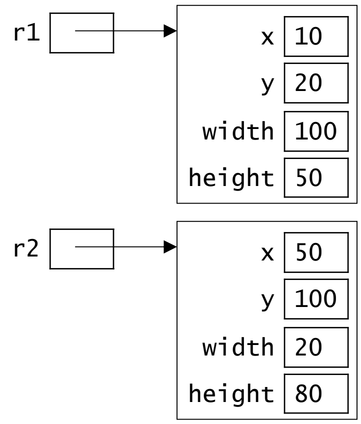
Another Example of a Method

- Here's a method for getting the area of a `Rectangle`:

```
def area(self):
    return self.width * self.height
```

- Sample method calls:

```
>>> r1.area()
5000
>>> r2.area()
__________
```



  - we're asking `r1` and r2 to give us their areas
  - nothing in the parentheses because the necessary info. is in the objects' attributes!

Second Version of our `Rectangle` Class

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """
    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight


    def area(self):
        return self.width * self.height
```

Original Client Program...

```
# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
area1 = r1.width * r1.height
print('area =', area1)

print('r2:', r2.width, 'x', r2.height)
area2 = r2.width * r2.height
print('area =', area2)

# grow both Rectangles
r1.width += 50
r1.height += 10
r2.width += 5
r2.height += 30

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Simplified Client Program

```
# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
print('area =', r1.area())

print('r2:', r2.width, 'x', r2.height)
print('area =', r2.area())

# grow both Rectangles
r1.grow(50, 10)
r2.grow(5, 30)

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

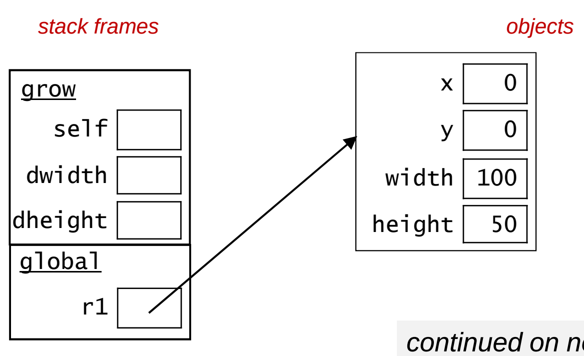
Methods That Modify an Object

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """
    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
        # why don't we need a return?

    def area(self):
       return self.width * self.height
```

Methods That Modify an Object

```
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

complete the diagram

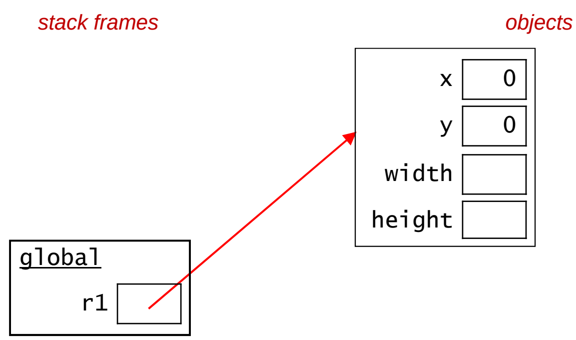


continued on next slide

Methods That Modify an Object (cont.)

```
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

output: ________________



Classes: Defining New Types of Objects

Computer Science 111
Boston University

Another Analogy

- A class is like a cookie cutter.
  - specifies the "shape" that all objects of that type should have

- Objects are like the cookies.
  - created with the "shape" specified by their class

Recall: An Initial `Rectangle` Class

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """

    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

- What is `__init__` used for?

- How many attributes do `Rectangle` objects have?

The Need to Import

- When client code is in a separate file, it needs to import the contents of the file with the class definition:

```
# assume this is in a file named rectangle.py
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """
    def __init__(self, init_width, init_height):
            self.x = 0
            self.y = 0
            self.width = init_width
            self.height = init_height
```

```
# client code in a different file
from rectangle import *

r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)
...
```

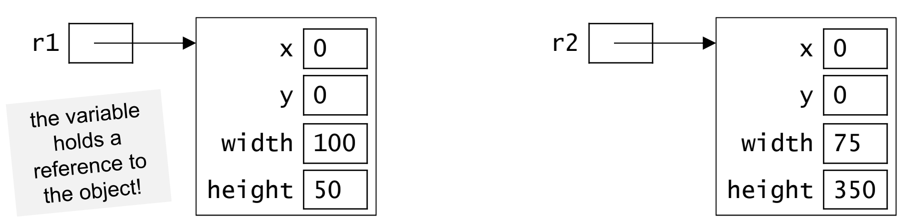
Initial Client Program

```
from rectangle import *

# construct two Rectangle objects
r1 = Rectangle(100, 50)     # what function is being called?
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
area1 = r1.width * r1.height
print('area =', area1)

print('r2:', r2.width, 'x', r2.height)
area2 = r2.width * r2.height
print('area =', area2)

# grow both Rectangles
r1.width += 50
r1.height += 10
r2.width += 5
r2.height += 30

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Recall: Constructing and Using an Object

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

```
>>> r1 = Rectangle(100, 50)     # calls __init__!
>>> r2 = Rectangle(75, 350)     # construct another one!
```



Recall: Second Version of our `Rectangle` Class

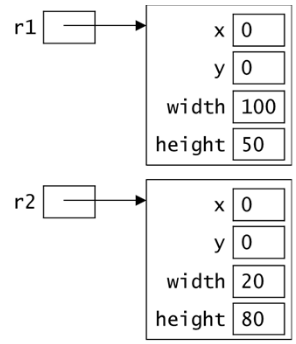
```
# assume this is in rectangle.py

class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """
    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight

    def area(self):
       return self.width * self.height
```

Recall: Simplified Client Program

```
from rectangle import *

# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
print('area =', r1.area())

print('r2:', r2.width, 'x', r2.height)
print('area =', r2.area())

# grow both Rectangles
r1.grow(50, 10)
r2.grow(5, 30)

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Be Objective!

```
class Rectangle:
    ...
    def grow(self, dwidth, dheight):
        ...

    def area(self):
        ...
```




```
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)
```

- Give an expression for:
  - the width of `r1`:
  - the height of `r2`:
- Write an assignment that changes `r1`'s x-coordinate to 50:
- Write a method call that:
  - increases `r2`'s width by 5 and height by 10:
  - gets `r1`'s area:

Method vs. Function

- Our `area` ***method*** is part of the `Rectangle` class:

```
class Rectangle:
    ...
    def area(self):         # methods have a self
        return self.width * self.height
```

  - thus, it is inside `Rectangle` objects
  - sample call:

```
r.area()
```

- Here's a ***function*** that takes two `Rectangle` objects as inputs:

```
def total_area(r1, r2):    # functions don't
    return r1.area() + r2.area()
```

  - it is *not* part of the class and is *not* inside `Rectangle` objects
  - sample call:

```
total_area(r, other_r)
```

  - it is a client of the `Rectangle` class!

Which of these is a correct perimeter method?

A.
```
def perimeter(self, width, height):
    return 2*width + 2*height
```

B.
```
def perimeter():
    return 2*self.width + 2*self.height
```

C.
```
def perimeter(self):
    return 2*self.width + 2*self.height
```

D. none of the above

Fill in the blank to call the `perimeter` method.

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height
```

```
r = Rectangle(35, 20)

perim = ______________________________
```

`scale` Method

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(______________________):
```

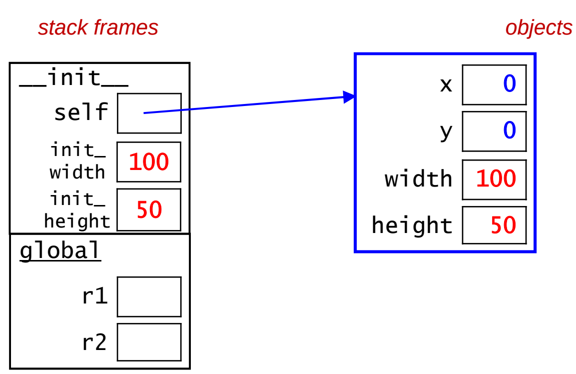
- In the space above, write a method called `scale` that scales the dimensions of a `Rectangle` by a specified factor.

  sample call:
  `r.scale(5)`

Why doesn't `scale` need to return anything?

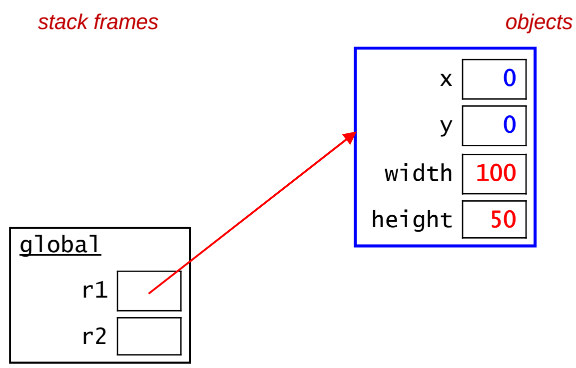
Memory Diagrams for Method Calls, part I

```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



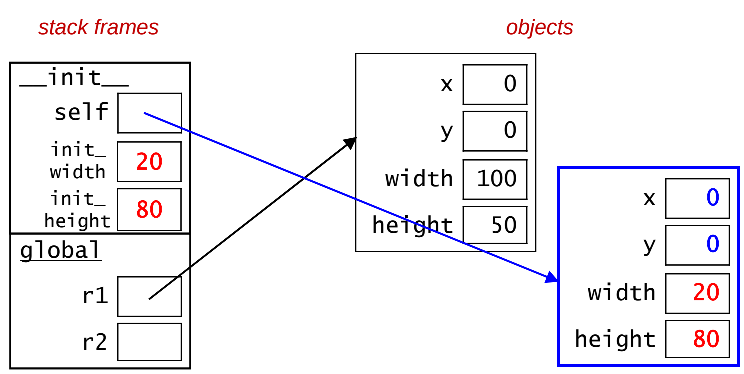
Memory Diagrams for Method Calls, part II

```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



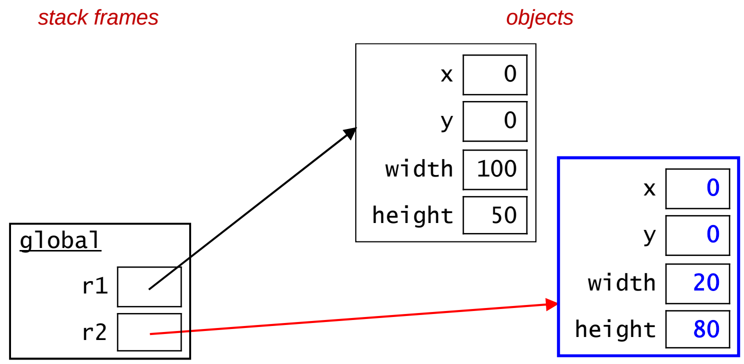
Memory Diagrams for Method Calls, part III

```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



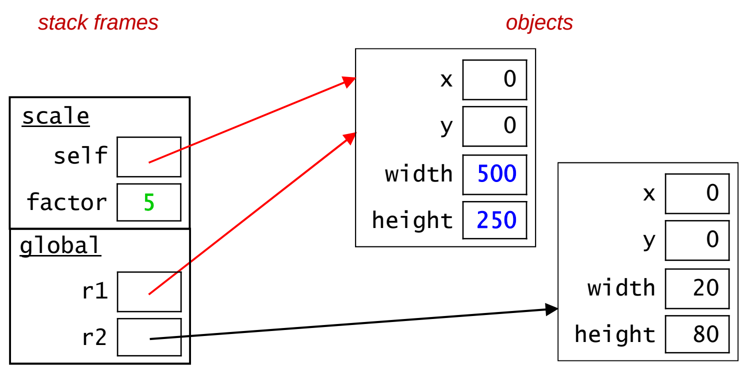
Memory Diagrams for Method Calls, part IV

```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



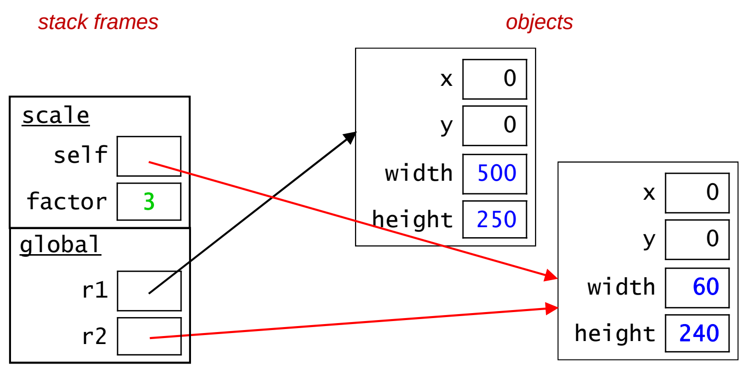
Memory Diagrams for Method Calls, part V

```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



Memory Diagrams for Method Calls, part VI

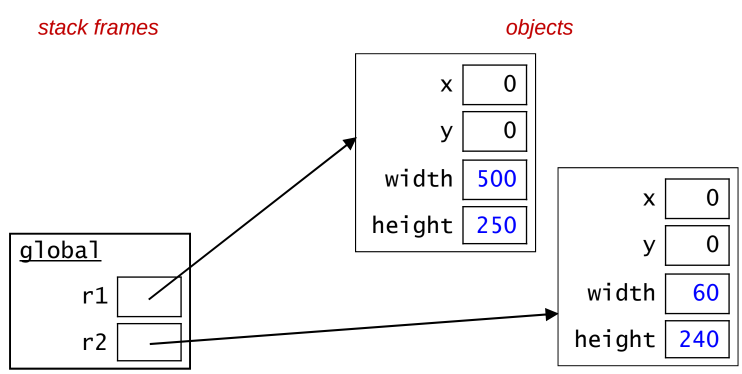
```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



Memory Diagrams for Method Calls, part VII

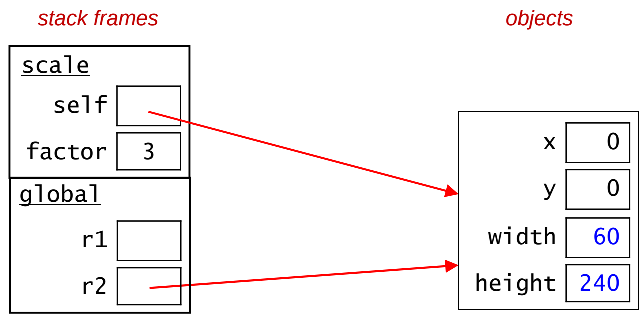
```
# Rectangle client code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```

output: 500 250 60 240



No Return Value Is Needed After a Change

- A method operates directly on the called object, so any changes it makes will be there after the method returns.
    - example: the call `r2.scale(3)` from the last slide



- `scale` gets a copy of the *reference* in `r2`
- thus, `scale's` changes to the *internals* of the object can be "seen" using `r2` after `scale` returns

Pre-Lecture
Comparing and Printing Objects

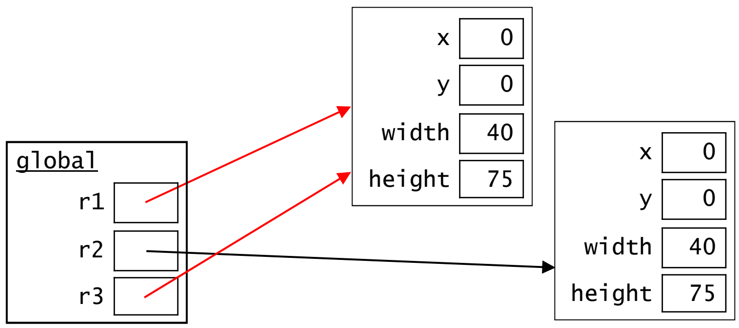
Computer Science 111
Boston University

Recall: Our `Rectangle` Class

```
# rectangle.py

class Rectangle:
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight

    def area(self):
        return self.width * self.height

    ...
```

What is the output of this client program?

```
from rectangle import *

r1 = Rectangle(40, 75)
r2 = Rectangle(40, 75)
r3 = r1

print(r1 == r2)
print(r1 == r3)
```



__eq__ (Implementing Our Own ==)

- The __eq__ method of a class allows us to implement our own version of the == operator.
- If we don't write a __eq__ method for a class, we get a default version that compares the object's memory addresses
  - see the previous example!

__eq__ Method for Our `Rectangle` Class

```
class Rectangle:
    ...          r1     r2
    def __eq__(self, other):
        if self.width == other.width and
           self.height == other.height:
            return True
        else:
            return False
```

```
>>> r1 = Rectangle(40, 75)
>>> r2 = Rectangle(40, 75)

>>> print(r1 == r2)
```

`__repr__` (Printing/Evaluating an Object)

- The `__repr__` method of a class returns a string representation of objects of that class.
- It gets called when you:
  - evaluate an object in the Shell:
    ```
    >> r1 = Rectangle(100, 80)
    >> r1                        # calls __repr__
    ```
  - apply `str()`:
    ```
    >> r1string = str(r1)    # also calls __repr__
    ```
  - print an object:
    ```
    >> print(r1)             # also calls __repr__
    ```

`__repr__` (Printing/Evaluating an Object)

- If we don't write a `__repr__` method for a class, we get a default version that isn't very helpful!

```
>>> r2 = Rectangle(50, 20)
>>> r2
<__main__.Rectangle object at 0x03247C30>
```

`__repr__` Method for Our `Rectangle` Class

```
class Rectangle:
    ...
    def __repr__(self):
        return str(self.width) + ' x ' + str(self.height)
```

- Note: the method does *not* do any printing.
- It returns a string that can then be printed or used when evaluating the object:

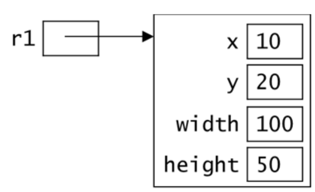
```
>>> r2 = Rectangle(50, 20)
>>> print(r2)
50 x 20
>>> r2
______________
```

More Object-Oriented Programming

Computer Science 111
Boston University

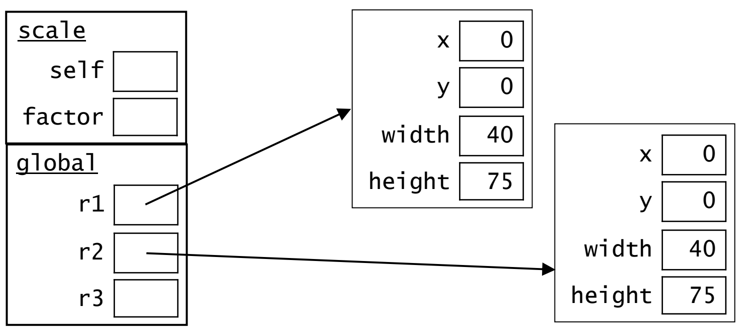
Recall: Our `Rectangle` Class

```
# rectangle.py

class Rectangle:
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(self, factor):
        self.width *= factor
        self.height *= factor
```



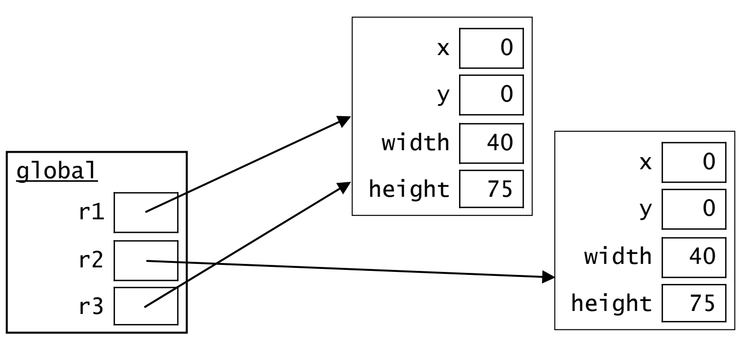
What is the output of this program?

```
from rectangle import *

r1 = Rectangle(40, 75)
r2 = Rectangle(40, 75)
r3 = r1

r1.scale(2)
print(r1.width, r2.width, r3.width)
```



What about this program?

```
from rectangle import *

r1 = Rectangle(40, 75)
r2 = Rectangle(40, 75)
r3 = r1

print(r1 == r2)
print(r1 == r3)
```



Recall: __eq__ Method for Our Rectangle Class

```
class Rectangle:
    ...
    def __eq__(self, other):
        if self.width == other.width and \
           self.height == other.height:
            return True
        else:
            return False
```

```
>>> r1 = Rectangle(40, 75)
>>> r2 = Rectangle(40, 75)

>>> print(r1 == r2)
```

Recall: __repr__ Method for Our Rectangle Class

```
class Rectangle:
    ...
    def __repr__(self):
        return str(self.width) + ' x ' + str(self.height)
```

- Note: the method does *not* do any printing.
- It returns a string that can then be printed or used when evaluating the object:

```
>>> r2 = Rectangle(50, 20)

>>> print(r2)
50 x 20

>>> r2
50 x 20

>>> str(r2)
'50 x 20'
```

```
class Rectangle:
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(self, factor):
        self.width *= factor
        self.height *= factor

    def __eq__(self, other):
        if self.width == other.width and self.height == other.height:
            return True
        else:
            return False

    def __repr__(self):
        return str(self.width) + ' x ' + str(self.height)
```

Updated `Rectangle` Class

Simplifying the Client Program Again...

```
from rectangle import *

# Construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# Print dimensions and area of each
print('r1:', r1)
print('area =', r1.area())

print('r2:', r2)
print('area =', r2.area())

# grow both Rectangles
r1.grow(50, 10)
r2.grow(5, 30)

# Print new dimensions
print('r1:', r1)
print('r2:', r2)
```

More Practice Defining Methods

- Write a method that moves the rectangle to the right by some amount.
  - sample call: `r.move_right(30)`

```
def move_right(self, amount):
```

x	200
y	150
width	50
height	30

coordinates of the upper-left corner

- Write a method that determines if the rectangle is a square.
  - return `True` if it does, and `False` otherwise
  - sample call: `r1.is_square()`

Date Class

month	11
day	11
year	1918

```
class Date:
    def __init__(self, init_month, init_day, init_year):
        """ constructor that initializes the
            three attributes
        """
        # you will write this!

    def __repr__(self):
        """This method returns a string representation for the
           object of type Date that calls it (named self).
        """
        s =  "%02d/%02d/%04d" % (self.month, self.day, self.year)
        return s

    def is_leap_year(self):
        """ Returns True if the calling object is
            in a leap year. Otherwise, returns False.
        """
        if self.year % 400 == 0:
            return True
        elif self.year % 100 == 0:
            return False
        elif self.year % 4 == 0:
            return True
        return False
```

Date Class (cont.)

- Example of how `Date` objects can be used:

```
>>> d = Date(12, 31, 2018)
>>> print(d)              # calls __repr__
12/31/2018
>>> d.advance_one()       # a method you will write
                          # nothing is returned!
>>> print(d)              # d has been changed!
01/01/2019
```

Methods Calling Other Methods

```
class Date:
    ...

    def days_in_month(self):
        """ returns the num of days in this date's month """
        numdays = [0,31,28,31,30,31,30,31,31,30,31,30,31]

        if self.is_leap_year() == True:
            numdays[2] = 29

        return numdays[self.month]
```

- The object calls `is_leap_year()` on itself!

Which call(s) does the method get wrong?

```
class Date:
    ...
    def is_before(self, other):    # buggy version!
        """ returns True if the called Date object (self)
            occurs before other, and False otherwise.
        """
        if self.year < other.year:
            return True
        elif self.month < other.month:
            return True
        elif self.day < other.day:
            return True
        else:
            return False

d1 = Date(11, 10, 2014)
d2 = Date(1, 1, 2015)
d3 = Date(1, 15, 2014)
```

Extra: Can you think of any *other* cases that it would get wrong involving these dates?

A. `d1.is_before(d2)`

B. `d2.is_before(d1)`

C. `d3.is_before(d1)`

D. more than one

Dictionaries

Computer Science 111
Boston University

Recall: Extracting Relevant Data from a File

```
def extract_results(filename, target_school):
    file = open(filename, 'r')

    for line in file:
        line = line[:-1]      # chop off newline at end

        fields = line.split(',')

        if fields[1] == target_school:
            print(fields[0], fields[2], fields[3])

    file.close()
```

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Len Lightning,BU,half-mile,2:15:00
Tom Turtle,UMass,half-mile,4:00:00
```

Another Data-Processing Task

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Len Lightning,BU,half-mile,2:15:00
Tom Turtle,UMass,half-mile,4:00:00
```

- Now we'd like to count the number of results from each school, and report all of the counts:

```
>>> school_counts('results.txt')
There are 3 schools in all.
BU has 2 result(s).
BC has 1 result(s).
UMass has 1 result(s).
```

- Python makes this easy if we use a *dictionary*.

What is a Dictionary?

- A dictionary is a set of key-value pairs.

```
>>> counts = {'BU': 2, 'UMass': 1, 'BC': 1}
```

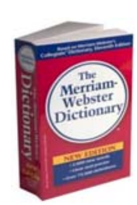
general syntax:

```
{key1: value1, key2: value2, key3: value3...}
```

- We can use the key like an index to lookup the associated value!

```
>>> counts['BU']
2
>>> counts['BC']
1
```

- It is similar to a "physical" dictionary:
  - keys = words
  - values = definitions
  - use the word to lookup its definition



Using a Dictionary

```
>>> counts = {}              # create an empty dictionary
>>> counts['BU'] = 2
```

key value

```
>>> counts['BC'] = 1
>>> counts                   # a set of key: value pairs
{'BU': 2, 'BC': 1}
>>> counts['BU']             # use the key to get the value
2
>>> counts['BC']
1
>>> counts['UMass'] = 1
>>> counts
{'BU': 2, 'UMass': 1, 'BC': 1}     # order is not fixed
```

Other Dictionary Operations

```
>>> counts = {'BU': 2, 'UMass': 1, 'BC': 1}
>>> len(counts)
3
>>> 'BU' in counts           # is 'BU' one of the keys?
True
>>> 'Harvard' in counts
False
>>> 'Harvard' not in counts
True
>>> 2 in counts
```

____

Processing All of the Items in a Dictionary

```
counts = {'BU': 2, 'UMass': 1, 'BC': 1}

for key in counts:      # get one key at a time
    print(key, counts[key])

# the above outputs:
BU 2
UMass 1
BC 1
```

- More generally:

```
for key in dictionary:
    # code to process key-value pair goes here
```

  - gets one key at a time and assigns it to key
  - continues looping until there are no keys left

Processing All of the Items in a Dictionary

```
counts = {'BU': 2, 'UMass': 1, 'BC': 1}

for key in counts:      # get one key at a time
    print(key, counts[key])
```

Fill in the rest of the table!

key	counts[key]	output
'BU'	counts['BU'] → 2	BU 2

What Is the Output?

```
d = {4: 10, 11: 2, 12: 3}

count = 0
for x in d:
    if x > 5:
        count += 1

print(count)
```

Using a Dictionary to Compute Counts

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Len Lightning,BU,half-mile,2:15:00
Tom Turtle,UMass,half-mile,4:00:00
```

```
def school_counts(filename):
    file = open(filename, 'r')
    counts = {}
    for line in file:
        fields = line.split(',')
        school = fields[1]
        if school not in counts:
            counts[school] = 1    # new key-value pair
        else:
            counts[school] += 1   # existing k-v pair
    file.close()

    print('There are', len(counts), 'schools in all.')
    for school in counts:
        print(school, 'has', counts[school], 'result(s).')
```

Another Example of Counting

```
def word_frequencies(filename):
    file = open(filename, 'r')
    text = file.read()      # read it all in at once!
    file.close()

    words = text.split()

    d = {}

    for word in words:
        if word not in d:
            d[word] = 1
        else:
            d[word] += 1

    return d
```

Shakespeare, Anyone?

```
>>> freqs = word_frequencies('romeo.txt')

>>> freqs['Romeo']
1

>>> freqs['ROMEO:']     # case and punctuation matter
47

>>> freqs['love']
12

>>> len(freqs)
2469
```

Act I of *Romeo & Juliet.*
See PS 8!

- In his plays, Shakespeare used ***31,534 distinct words!***
  http://www-math.cudenver.edu/~wbriggs/qr/shakespeare.html
- He also coined a number of words:

gust	besmirch	unreal
swagger	watchdog	superscript

http://www.pathguy.com/shakeswo.htm
http://www.shakespeare-online.com/biography/wordsinvented.html

Generate Text Based on Shakespeare!

```
>>> d = create_dictionary('romeo.txt')

>>> generate_text(d, 50)
ROMEO: Out of mine own word: If you merry! BENVOLIO:
Come, go to. She hath here comes one of the year, Come
hither, nurse. ROMEO: Well, in spite, To be gone.
BENVOLIO: For men depart.[Exeunt all Christian souls!-
Were of wine. ROMEO: Bid a sea nourish'd with their
breaths with
```

Generate Text Based on Shakespeare ...Or Anyone Else!


```
Boston University is an international,
comprehensive, private research university,
committed to educating students to be reflective,
resourceful individuals ready to live, adapt, and
lead in an interconnected world. Boston University
is committed to generating new knowledge to benefit
society.

We remain dedicated to our founding principles: that
higher education should be accessible to all and
that research, scholarship, artistic creation, and
professional practice should be conducted in the
service of the wider community—local and
international. These principles endure in the
University's insistence on the value of diversity,
in its tradition and standards of excellence, and in
its dynamic engagement with the City of Boston and
the world.

Boston University comprises a remarkable range of
undergraduate, graduate, and professional programs
built on a strong foundation of the liberal arts and
sciences. With the support and oversight of the
Board of Trustees, the University, through our
faculty, continually innovates in education and
research to ensure that we meet the needs of
students and an ever-changing world.
```

mission.txt

Generate Text Based on Shakespeare ...Or Anyone Else!

```
Boston University is an international, comprehensive, private research
university, committed to educating students to be reflective, resourceful
individuals ready to live, adapt, and lead in an interconnected world. Boston
University is committed to generating new knowledge to benefit society.

We remain dedicated to our founding principles: that higher education should be
accessible to all and that research, scholarship, artistic creation, and
professional practice should be conducted in the service of the wider community—
local and international. These principles endure in the University's insistence
on the value of diversity, in its tradition and standards of excellence, and in
its dynamic engagement with the City of Boston and the world.

Boston University comprises a remarkable range of undergraduate, graduate, and
professional programs built on a strong foundation of the liberal arts and
sciences. With the support and oversight of the Board of Trustees, the
University, through our faculty, continually innovates in education and research
to ensure that we meet the needs of students and an ever-changing world.
```

mission.txt

```
>>> d2 = create_dictionary('mission.txt')

>>> generate_text(d2, 20)
We remain dedicated to benefit society. Boston
University is an ever-changing world. Boston University
comprises a strong foundation of diversity,
```

Markov Models

- Allow us to model *any* sequence of real-world data.
  - human speech
  - written text
  - sensor data
  - etc.

- Can use the model to *generate* new sequences that are based on existing ones.

- We'll use a *first-order* Markov model.
  - each term in the sequence depends only on the *one* term that immediately precedes it

A Markov Model in Dictionary Form

```
Boston University is a comprehensive university.
It is committed to educating students to be ready
to live and to lead in an interconnected world.
It is committed to generating new knowledge.
It is amazing!
```

edited_mission.txt

sentence-start symbol

```
{'$': ['Boston', 'It', 'It', 'It'],
 'Boston': ['University'],
 'University': ['is'],
 'is': ['a', 'committed', 'committed', 'amazing!'],
 'to': ________________________________,
 'committed': ____________________________,
 ... }
```

key = a word w

value = a list of the words that follow w in the text

- *Sentence-ending words* should not be used as keys.
  - words that end with a `'.'`, `'?'`, or `'!'` (e.g., `'world.'`)

Model Creation Function

```
def create_dictionary(filename):
    # read in file and split it into a list of words

    d = {}
    current_word = '$'

    for next_word in words:
        if current_word not in d:
            d[current_word] = [next_word]
        else:
            d[current_word] += [next_word]

        # update current_word to be either
        # next_word or '$'...

    return d
```

key = a word w

value = a list of the words that follow w in the text

Model Creation Example

```
words = ['Boston', 'University', 'is', 'a', 'comprehensive',
   'university.', 'It', 'is', 'committed', ...]
d = {}
current_word = '$'

for next_word in words:
    if current_word not in d:
        d[current_word] = [next_word]
    else:
        d[current_word] += [next_word]

    # update current_word to be either next_word or '$'...
```

current_word	next_word	action taken
'$'	'Boston'	d['$'] = ['Boston']
'Boston'	'University'	d['Boston'] = ['University']
'is'	'a'	d['is'] = ['a']
'a'	'comprehensive'	d['a'] = ['comprehensive']
'comprehensive'	'university.'	d['comprehensive']=['university.']
'$'	'It'	d['$'] → ['Boston', 'It']
'It'	'is'	d['It'] = ['is']

`generate_text(word_dict, num_words)`

start with `current_word` = `'$'`

repeat `num_words` times:

- `wordlist =` words that can follow `current_word` (use the `word_dict` dictionary!)
- `next_word = random.choice(wordlist)`
- print `next_word`, followed by a space (use `end=' '`)
- update `current_word` to be either `next_word` or `'$'`

```
print()    # force a newline at the end of everything
```

Which of these could be one of the entries in d?

```
Boston University is a comprehensive university.
It is committed to educating students to be ready
to live and to lead in an interconnected world.
It is committed to generating new knowledge.
It is amazing!
```

edited_mission.txt

```
>>> d = create_dictionary('edited_mission.txt')
```

A. `'a': ['comprehensive']`

B. `'It': ['is']`

C. `'knowledge.': ['new']`

D. two of the above (which ones?)

E. A, B, and C

Using the Model to Generate New Text

```
Boston University is a comprehensive university.
It is committed to educating students to be ready
to live and to lead in an interconnected world.
It is committed to generating new knowledge.
It is amazing!
```

edited_mission.txt

- Here's a portion of our Markov model for the above text:

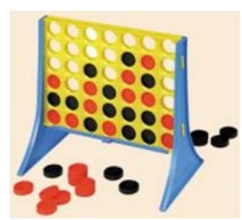
```
{'$': ['Boston', 'It', 'It', 'It'],
 'Boston': ['University'],
 'University': ['is'],
 'is': ['a', 'committed', 'committed', 'amazing!'],
 'to': ['educating','be','live','lead','generating'],
 'committed': ['to', 'to'],
 'It': ['is', 'is', 'is'], … }
```

- We use it to generate new text…

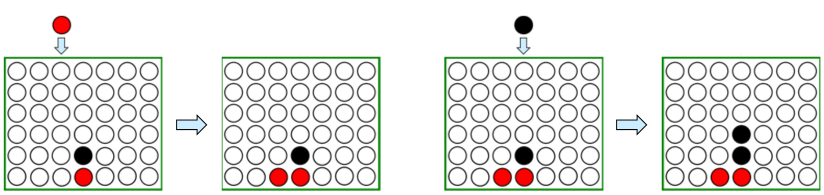
Board Objects for Connect Four

Computer Science 111
Boston University

PS 9: Connect Four!

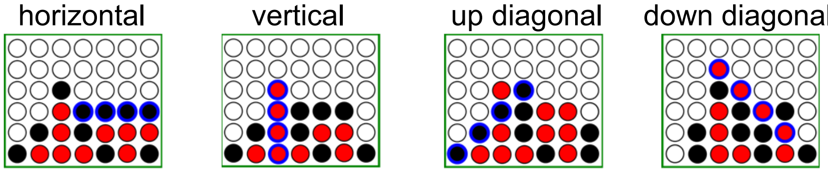


- Two players, each with one type of checker
- 6 x 7 board that stands vertically
- Players take turns dropping a checker into one of the board's columns.



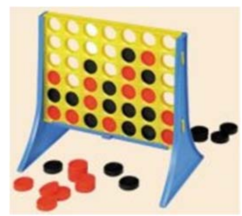
- Win == four adjacent checkers in any direction:

horizontal vertical up diagonal down diagonal



Recall: Classes and Objects

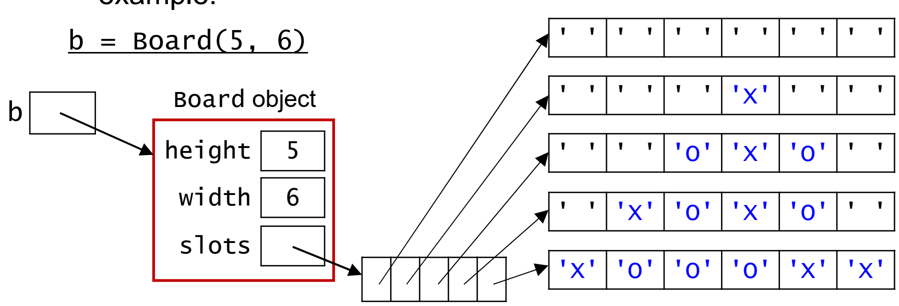
- A *class* is a blueprint – a definition of a new data type.
- We can use the class to create one or more *objects.*
  - "values" / *instances* of that type
- One thing we'll need: a `Board` class!



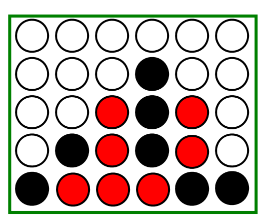
`Board` Objects

- To facilitate testing, we'll allow for dimensions other than 6 x 7.
  - example:

    <u>`b = Board(5, 6)`</u>

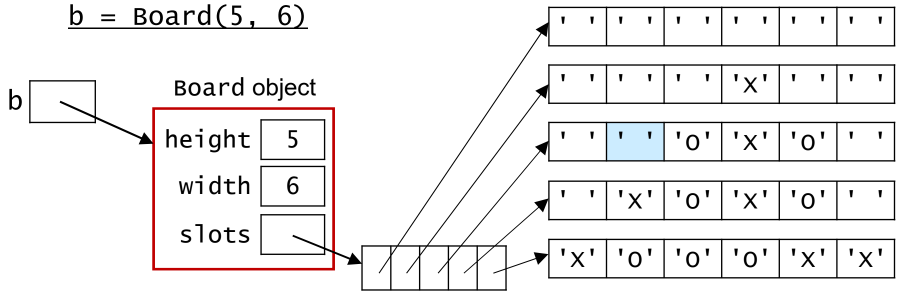


- `slots` is ***a 2-D list*** of single-character strings!
  `' '` (space) for empty slot
  `'X'` for one player's checkers
  `'O'` (not zero!) for the other's



From a Client, How Could We Set the Blue Slot to `'X'`?

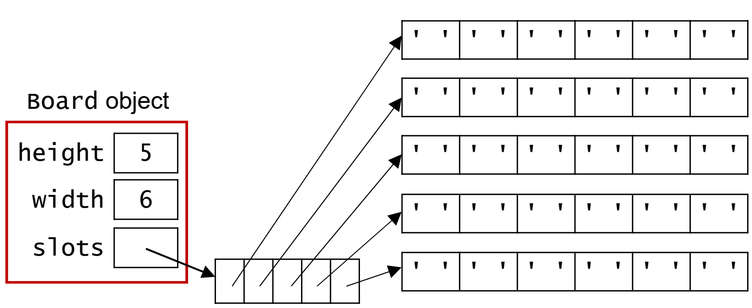
`b = Board(5, 6)`



How would you do this if the code were *inside* a `Board` method?

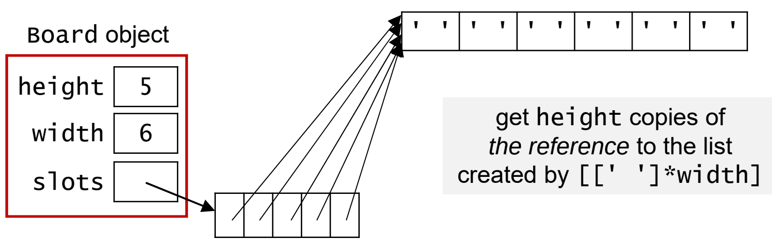
`Board` Constructor

```
class Board:
    """ a data type for a Connect Four board with
        arbitrary dimensions
    """
    def __init__(self, height, width):
        """ a constructor for Board objects """
        self.height = height
        self.width = width

        self.slots = ______________________________
```



Incorrect Board Constructor

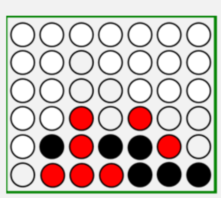
```
class Board:
    """ a data type for a Connect Four board with
        arbitrary dimensions
    """
    def __init__(self, height, width):
        """ a constructor for Board objects """
        self.height = height
        self.width = width
        self.slots = [[' ']*width] * height  doesn't work!
```



__repr__ Method

```
def __repr__(self):
    """ returns a string representation of a Board """
    s = ''    # begin with an empty string

    for row in range(self.height):
        s += '|'
        for col in range(self.width):
            s += self.slots[row][col] + '|'
        s += '\n'

    # add the row of hyphens to s
    # add the column indices to s

    return s
```

← you'll add code here!



```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X|X|
---------------
 0 1 2 3 4 5 6
```

add_checker Method

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # end of method
```

- Why don't we need a `return` statement?
  - `add_checker()`'s only purpose is to change the state of the `Board`
  - when a method changes the internals of an object, those changes will still be there after the method completes
  - thus, no return is needed!

Which of these correctly fills in the blank?

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # end of method
```

```
>>> b = Board(3, 5)    # empty Board
>>> ________________    # add 'X' to column 2
>>> print(b)
| | | | | |
| | | | | |
| | |X| | |
-----------
 0 1 2 3 4
```

A. `b.add_checker('X', 2)`

B. `add_checker(b, 'X', 2)`

C. `b = b.add_checker('X', 2)`

D. more than one of these

Your Task in add_checker()

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """

        # code to determine appropriate row goes here

        self.slots[???][col] = checker

        # no return needed!
```

```
>>> b.add_checker('O', 4)
```

```
    Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```

Which call(s) does the method *get wrong*?

```
class Board:
    ...
    def add_checker(self, checker, col):    # buggy version!
        """ adds the specified checker to column col """

        row = 0
        while self.slots[row][col] == ' ':
            row += 1

        self.slots[row][col] = checker
```

A. `b.add_checker('X', 0)`

B. `b.add_checker('O', 6)`

C. `b.add_checker('X', 2)`

D. A and B

E. A, B, and C

```
    Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```

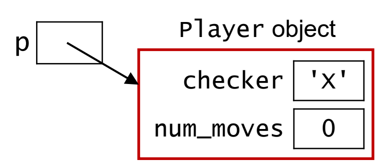
Inheritance

Computer Science 111
Boston University

Also in PS 9: A `Player` Class

```
class Player:
    def __init__(self, checker):
        ...

    def __repr__(self):
        ...

    def opponent_checker(self):
        ...

    def next_move(self, b):
        """ Get a next move for this player that is valid
            for the board b.
        """
        self.num_moves += 1

        while True:
            col = int(input('Enter a column: '))
            # if valid column index, return that integer
            # else, print 'Try again!' and keep looping
```

`p = Player('X')`



The APIs of Our `Board` and `Player` Classes

```
class Board:
   __init__(self, col)
   __repr__(self)
   add_checker(self, checker, col)
   clear(self)
   add_checkers(self, colnums)
   can_add_to(self, col)
   is_full(self)
   remove_checker(self, col)
   is_win_for(self, checker)

class Player:
   __init__(self,col)
   __repr__(self)
   opponent_checker(self)
   next_move(self, b)
```

Make sure to take full advantage of these methods in your work on PS 9!

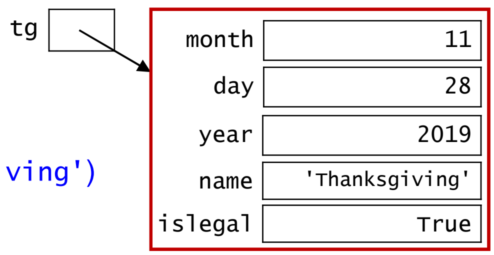
Recall: Our `Date` Class

```
class Date:
    def __init__(self, new_month, new_day, new_year):
        """ Constructor """
        self.month = new_month
        self.day = new_day
        self.year = new_year

    def __repr__(self):
        """ This method returns a string representation for the
            object of type Date that calls it (named self).
        """
        s =  "%02d/%02d/%04d" % (self.month, self.day, self.year)
        return s

    def is_leap_year(self):
        """ Returns True if the calling object is
            in a leap year. Otherwise, returns False.
        """
        if self.year % 400 == 0:
            return True
        elif self.year % 100 == 0:
            return False
        elif self.year % 4 == 0:
            return True
        return False
```

month	11
day	11
year	1918

Holidays == Special Dates!

- Each holiday has:
  - a month
  - a day
  - a year
  - a name (e.g., `'Thanksgiving'`)
  - an indicator of whether it's a legal holiday



- We want `Holiday` objects to have `Date`-like functionality:

```
>>> tg = Holiday(11, 28, 2019, 'Thanksgiving')
>>> today = Date(11, 18, 2019)
>>> tg.days_between(today)
result: 10
```

- But we want them to behave differently in at least one way:

```
>>> print(tg)
Thanksgiving (11/28/2019)
```

Let `Holiday` Inherit From `Date`!

```
class Holiday(Date):     ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        ...
```

- `Holiday` gets all of the attributes and methods of `Date`.
  - we don't need to redefine them here!
- `Holiday` is a *subclass* of `Date`.
- `Date` is a *superclass* of `Holiday`.

Constructors and Inheritance

```
class Holiday(Date):   ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value
```

```
>>> tg = Holiday(11, 28, 2019, 'Thanksgiving')
```

- `super()` provides access to the superclass of the current class.
  - allows us to call its version of `__init__`, which initializes the inherited attributes

Overriding an Inherited Method

```
class Holiday(Date):   ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = super().__repr__()  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

- To see something different when we print a `Holiday` object, we *override* (i.e., replace) the inherited version of `__repr__`.

Let Holiday Inherit From Date!

```
class Holiday(Date):   ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True    # default value

    def __repr__(self):    # overrides the inherited __repr__
        s = self.name
        mdy = super().__repr__()  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

- That's it! Everything else is inherited!
- All other `Date` methods work the same on `Holiday` objects as they do on `Date` objects!

Inheritance in PS 9

- `Player` – the superclass
  - includes fields and methods needed by all C4 players
  - in particular, a `next_move` method
  - use this class for human players
- `RandomPlayer` – a subclass for an *un*intelligent computer player
  - no new fields
  - overrides `next_move` with a version that chooses at random from the non-full columns
- `AIPlayer` – a subclass for an "intelligent" computer player
  - uses AI techniques
  - new fields for details of its strategy
  - overrides `next_move` with a version that tries to determine the best move!

AI for Connect Four

Computer Science 111
Boston University

Inheritance in PS 9

- `Player` – the superclass
  - includes fields and methods needed by all C4 players
  - in particular, a `next_move` method
  - use this class for human players
- `RandomPlayer` – a subclass for an *un*intelligent computer player
  - no new fields
  - overrides `next_move` with a version that chooses at random from the non-full columns
- `AIPlayer` – a subclass for an "intelligent" computer player
  - uses AI techniques
  - new fields for details of its strategy
  - overrides `next_move` with a version that tries to determine the best move!

"Arithmetizing" Connect Four

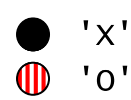
- Our `AIPlayer` assigns a score to each possible move
  - i.e., to each column
- It *looks ahead* some number of moves into the future to determine the score.
  - *lookahead* = # of future moves that the player considers
- Scoring columns:
  - **–1**: an already *full column*
  - **0**: if we choose this column, it will result in a *loss* at some point during the player's lookahead
  - **100**: if we choose this column, it will result in a *win* at some point during the player's lookahead
  - **50**: if we choose this column, it will result in *neither a win nor a loss* during the player's lookahead

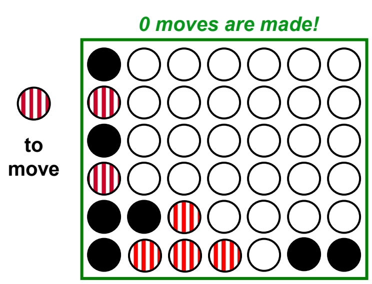
A Lookahead of 0

- A lookahead-0 player only assesses the current board (0 moves!).

LA-0 scores for ◍

-1	50	50	50	50	50	50





A Lookahead of 1

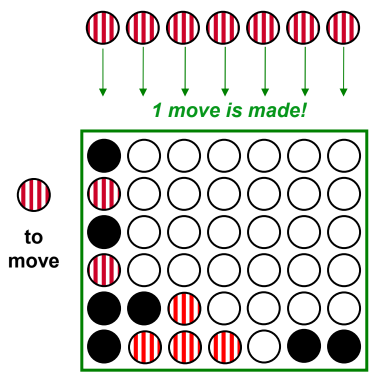
- A lookahead-1 player assesses the outcome of *only* the considered move.

LA-1 scores for ◍

-1	50	50	50	**100**	50	50

● 'X'

◍ 'O'



A lookahead-1 player will "see" an impending victory.

`next_move` will return 4!

A Lookahead of 1

- A lookahead-1 player assesses the outcome of *only* the considered move.

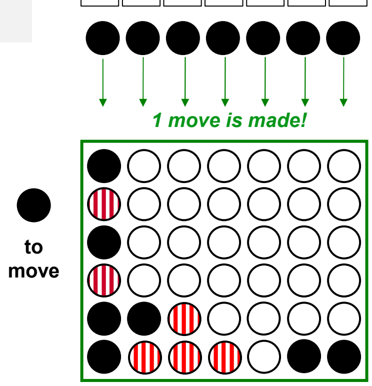
How do these scores change if it is ●'s turn instead of ◍'s?

LA-1 scores for ●

-1	50	50	50	100	50	50

● 'X'

◍ 'O'



A Lookahead of 2

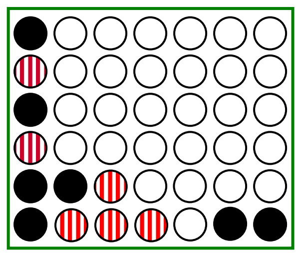
- A lookahead-2 player looks 2 moves ahead.
  - what if I make this move, and then my opponent makes *its best move*?
  - ***note:*** we assume the opponent looks ahead 2 – 1 = 1 move

LA-2 scores for ●

● 'X'
◍ 'O'

-1	0	0	0	**50**	0	0

to move



A lookahead-2 player will "see" a way to win *or* a way to block the opponent's win.

Example 2: LA-0

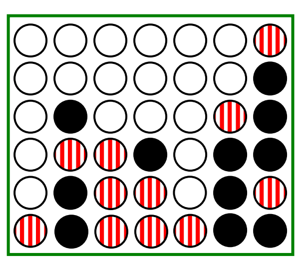
- A lookahead-0 player only assesses the current board (0 moves!).

LA-0 scores for ●

● 'X'
◍ 'O'

50	50	50	50	50	50	-1

to move



Example 2: LA-1

- A lookahead-1 player assesses the outcome of *only* the considered move.

What scores change with the increased LA?

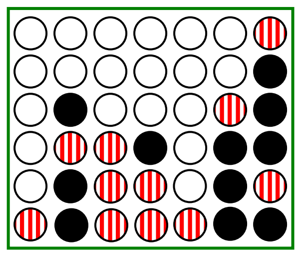
LA-1 scores for ●

50	50	50	50	50	50	-1

● 'X'

'O'

● **to move**



Example 2: LA-2

- A lookahead-2 player looks 2 moves ahead.
  - what if I make this move, and then my opponent makes *its best move*?
  - ***note:*** we assume the opponent looks ahead 2 – 1 = 1 move

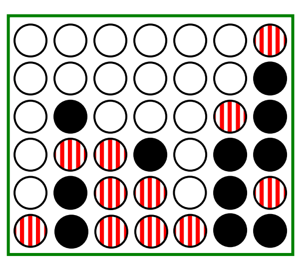
LA-2 scores for ●

						-1

● 'X'

'O'

● **to move**



LA-3!

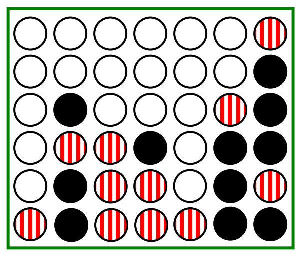
- A lookahead-3 player looks 3 moves ahead.
  - what if I make this move, and then my opponent makes its best move, *and then I make my best subsequent move?*
  - ***note:*** we assume the opponent looks ahead 3 – 1 = 2 moves

What would change?

LA-3 scores for ●

						-1

to move



'X'
'O'

Example 2: LA-0

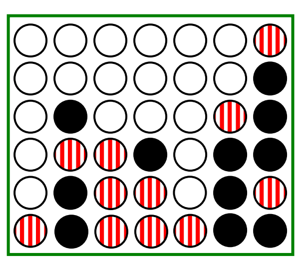
- A lookahead-0 player only assesses the current board (0 moves!).

same board, different player, same LA-0 scores

LA-0 scores for

50	50	50	50	50	50	-1

to move



'X'
'O'

Example 2: LA-1

- A lookahead-1 player assesses the outcome of *only* the considered move.

LA-1 scores for

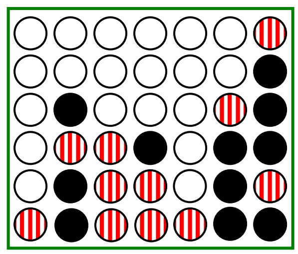
What would change?

50	50	50	50	50	50	-1

'X'

'O'

to move

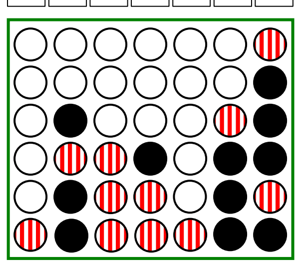


What Are the LA-2 Scores for ?

- Look 2 moves ahead. Assume the opponent looks 1 move ahead.

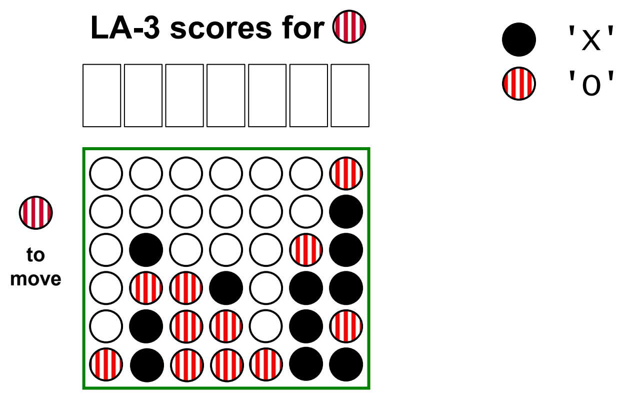
	50	50	100	50	50	50	-1	← LA-1 scores
A.	50	50	100	50	50	50	-1	← no change?
B.	0	0	100	0	0	0	-1	
C.	50	50	100	50	0	50	-1	

to move



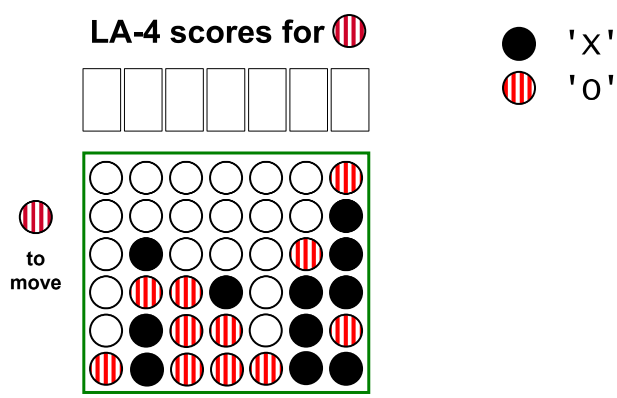
Example 2: LA-3

- A lookahead-3 player looks 3 moves ahead.
  - what if I make this move, and then my opponent makes its best move, *and then I make my best subsequent move?*
  - ***note:*** we assume the opponent looks ahead 3 – 1 = 2 moves



LA-4!

- A lookahead-4 player looks 4 moves ahead.
  - assumes the opponent looks ahead 4 – 1 = 3 moves



LA-4!

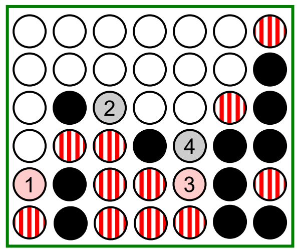
- A lookahead-4 player looks 4 moves ahead.
  - assumes the opponent looks ahead 4 – 1 = 3 moves

LA-4 scores for 'O'

0	0	100	0	0	0	-1

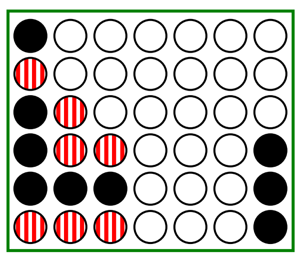
Consider column 0:
1. 'O' moves there.
2. 'X' moves to 2.
3. 'O' moves to 4 to block a diagonal win.
4. 'X' still wins horizontally!

Same thing holds for the other col's with new 0s.



'X'
'O'

What about this?



'O'
you - `self` - is playing `'O'`
'X'

	col 0	col 1	col 2	col 3	col 4	col 5	col 6
LA-0 scores for **'O'**: Looks 0 moves into the future	-1	50	50	50	50	50	50
LA-1 scores for **'O'**: Looks 1 move into the future	-1						
LA-2 scores for **'O'**: Looks 2 moves into the future	-1						
LA-3 scores for **'O'**: Looks 3 moves into the future	-1						

scores_for – the AI in AIPlayer!

```
def scores_for(self, b):
    """ returns a list of scores - one for each col in board b
    """
    scores = [50] * b.width
    for col in range(b.width):

        ???

    return scores
```

scores_for

Suppose you're playing with LA 2...

For each column:

1) add a checker to it

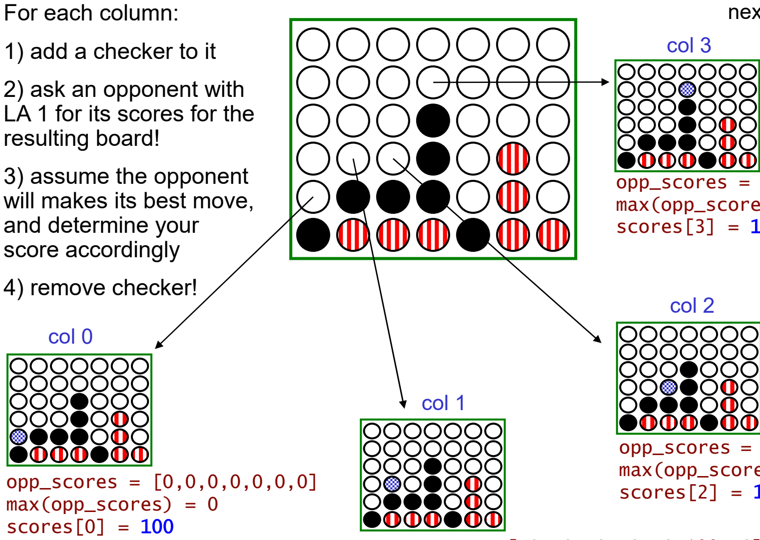
2) ask an opponent with LA 1 for its scores for the resulting board!

3) assume the opponent will makes its best move, and determine your score accordingly

4) remove checker!

(self) 'X' ●

possible next move




```
opp_scores = [0,0,0,0,0,0,0]
max(opp_scores) = 0
scores[3] = 100
```

```
opp_scores = [0,0,0,0,0,0,0]
max(opp_scores) = 0
scores[2] = 100
```

```
opp_scores = [0,0,0,0,0,0,0]
max(opp_scores) = 0
scores[0] = 100
```

A loss for my opponent is a win for me!

```
opp_scores = [50,50,50,50,50,100,50]
max(opp_scores) = 100
scores[1] = 0
```

A win for my opponent is a loss for me!

Suppose you're playing with LA 2...

scores_for

(self) 'X'

possible next move

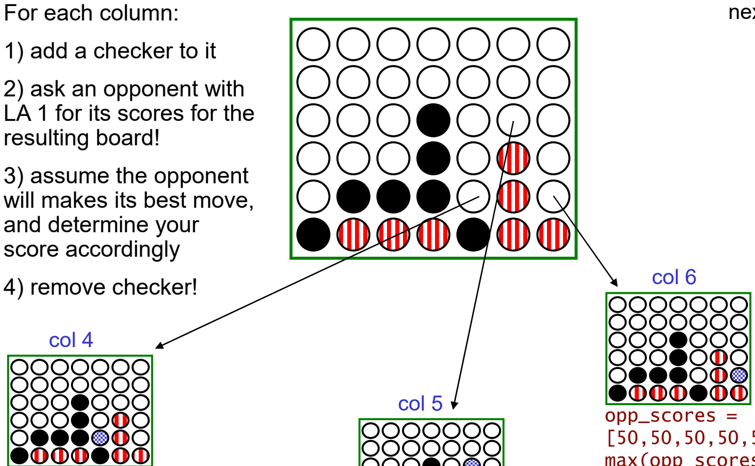
For each column:

1) add a checker to it

2) ask an opponent with LA 1 for its scores for the resulting board!

3) assume the opponent will makes its best move, and determine your score accordingly

4) remove checker!



opp_scores = [0,0,0,0,0,0,0]
max(opp_scores) = 0
scores[4] = **100**

opp_scores = [50,50,50,50,50,50,50]
max(opp_scores) = 50
scores[5] = **50**
A draw for my opponent is a draw for me!

opp_scores =
[50,50,50,50,50,**100**,50]
max(opp_scores) = **100**
scores[6] = **0**

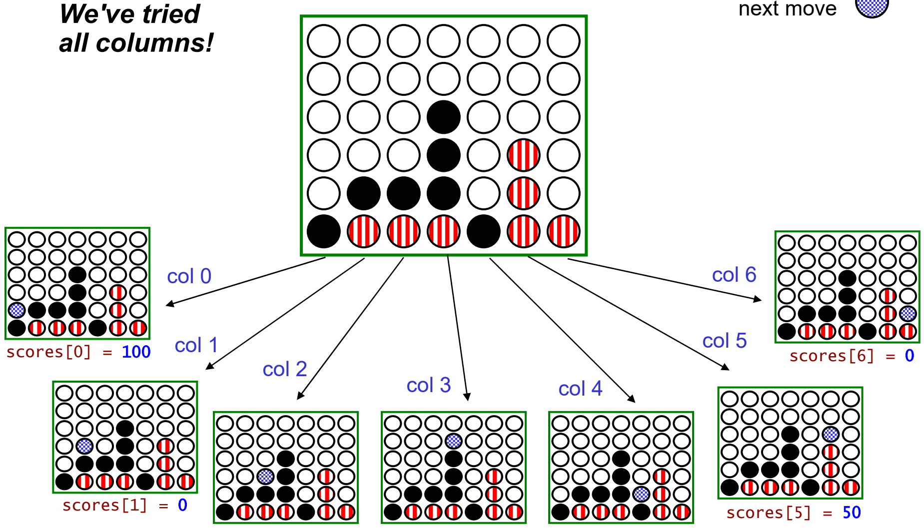
Suppose you're playing with LA 2...

scores_for

(self) 'X'

possible next move

We've tried all columns!




```
return [100, 0, 100, 100, 100, 50, 0]
```

scores_for – the AI in AIPlayer!

```
def scores_for(self, b):
    """ returns a list of scores - one for each col in board b
    """
    scores = [50] * b.width

    for col in range(b.width):
        if col is full:
            use -1 for scores[col]
        elif already win/loss:
            use appropriate score (100 or 0)
        elif lookahead is 0:
            use 50
        else:
            try col - adding a checker to it
            create an opponent with self.lookahead - 1
            opp_scores = opponent.scores_for(...)
            scores[col] = ???
            remove checker

    return scores
```

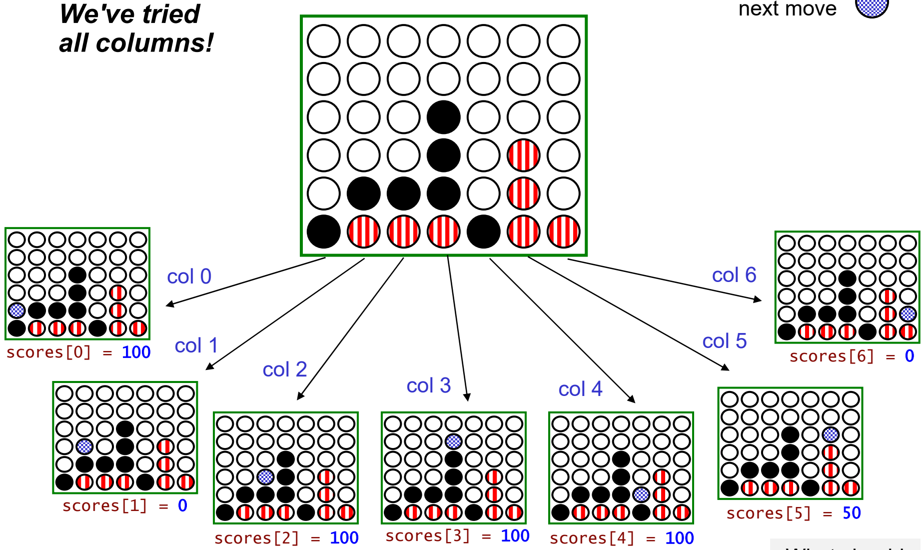
Suppose you're playing with LA 2...

scores_for

We've tried all columns!

(self) 'X'

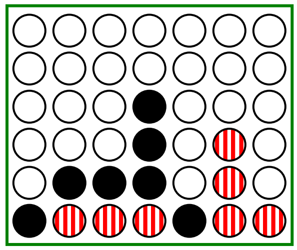
possible next move



return [100, 0, 100, 100, 100, 50, 0]

What should next_move return?

Breaking Ties



```
  0    1    2    3    4    5    6
return [100, 0, 100, 100, 100, 50, 0]
```

- possible moves: ____________
- `self.tiebreak == 'LEFT': return ____`
- `self.tiebreak == 'RIGHT': return ____`
- `self.tiebreak == 'RANDOM': choose at random!`

Recall: Inheritance in PS 9

- `Player` – a class for human Connect Four players
  - includes fields and methods needed by all C4 players
  - in particular, a `next_move` method
- `RandomPlayer` – a class for an *un*intelligent computer player
  - no new fields
  - **overrides `next_move`** with a version that chooses at random from the non-full columns
- `AIPlayer` – a class for an "intelligent" computer player
  - uses AI techniques
  - new fields for details of its strategy
  - **overrides `next_move`** with a version that tries to determine the best move!

Using the Player Classes

- Example 1: two human players

```
>>> connect_four(Player('X'), Player('O'))
```

- Example 2: human player vs. AI computer player:

```
>>> connect_four(Player('X'), AIPlayer('O', 'LEFT', 3))
```

- `connect_four()` repeatedly calls `process_move()`:

```
def connect_four(p1, p2):
    print('Welcome to Connect Four!')
    print()
    b = Board(6, 7)
    print(b)

    while True:
        if process_move(p1, b) == True:
            return b
        if process_move(p2, b) == True:
            return b
```

OOP == Object-Oriented Power!

```
def process_move(p, b):
    ...
    col = p.next_move(b)

    ...
```

- Which version of `next_move` gets called?
- It depends!
  - if `p` is a `Player` object, call `next_move` from that class
  - if `p` is a `RandomPlayer`, call that version of `next_move`
  - if `p` is an `AIPlayer`, call that version of `next_move`
- The appropriate version is automatically called!

Beware!

- Correct approach: call the `next_move` method within the object to which the variable `p` refers:

```
def process_move(p, b):
    ...
    col = p.next_move(b)
    ...
```

- In theory, we can treat `next_move` as if it were a function:

```
def process_move(p, b):
    ...
    col = Player.next_move(p, b)        # wrong!
    ...
```

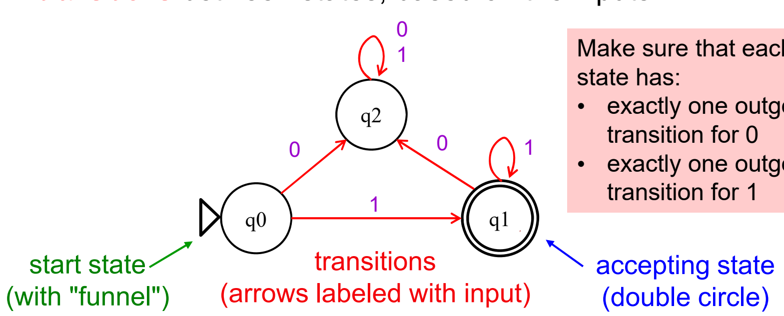
- This won't work! Why?

Finite-State Machines

Computer Science 111
Boston University

Finite State Machine (FSM)

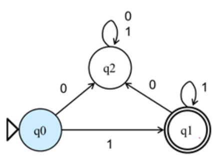
- An abstract model of computation
- Consists of:
  - one or more states (the circles)
    - *exactly one* of them is the *start / initial state*
    - *zero or more* of them can be an *accepting state*
  - a set of possible input characters (we're using {0, 1})
  - *transitions* between states, based on the inputs



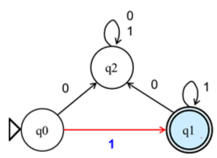
Make sure that each state has:
- exactly one outgoing transition for 0
- exactly one outgoing transition for 1

Accepting an Input Sequence

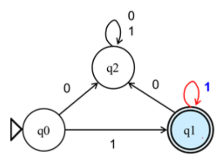
- We can use an FSM to test if an input meets some criteria.
- An FSM *accepts* an input if the transitions produced by the input leave the FSM in an accepting state.
- Example: input **111** on the FSM from the last slide



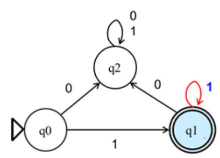
initial state



transition caused by first **1**



transition caused by next **1**

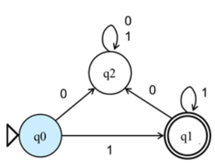


transition caused by final **1**

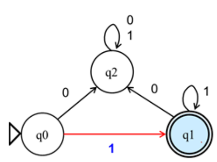
111 *is* accepted!

Rejecting an Input Sequence

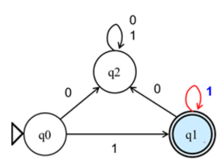
- An FSM *rejects* an input if the transitions produced by the input do *not* leave the FSM in an accepting state.
- Example: input **1101** on the FSM from the last slide



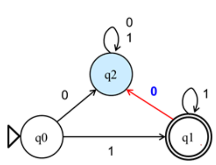
initial state



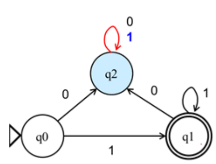
transition from first **1**



transition from next **1**



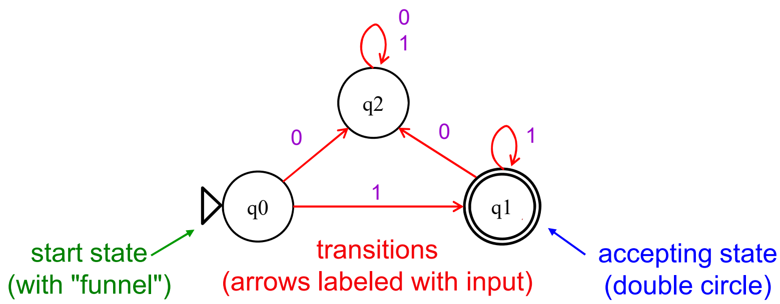
transition from **0**



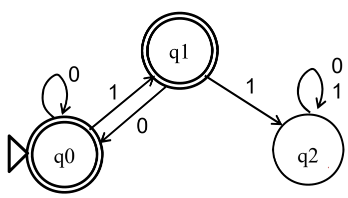
transition from final **1**

1101 is *not* accepted

Which Bit Strings Does This FSM Accept?



Which of these inputs is accepted?



A. 0000

B. 10010001

C. 00111

D. A and B

E. A and C

In general, what English phrase describes the ***inputs accepted*** by this FSM?

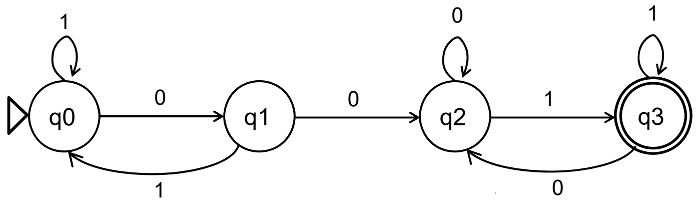
What does each state say about the ***input seen thus far***?

q0:

q1:

q2:

Which of these inputs is accepted?



A. 0101
B. 10010
C. 0011101
D. two or more
E. none of them

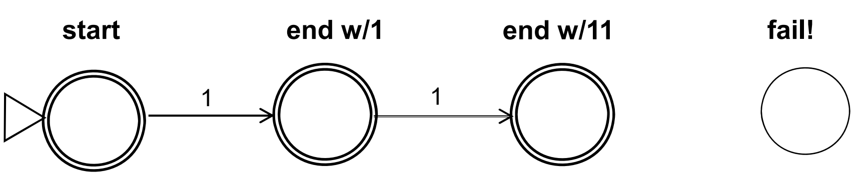
In general, what English phrase describes the ***inputs accepted*** by this FSM?

What does each state say about the ***input seen thus far***?
q0:
q1:
q2:
q3:

Add the Missing Transitions: Does Not Contain 110

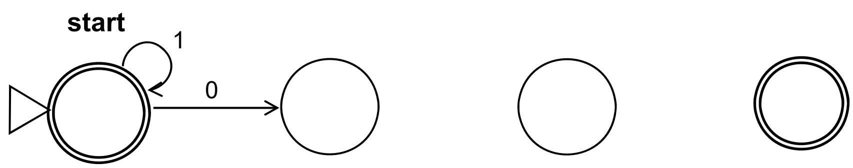
Construct a FSM accepting strings that do ***NOT*** contain the pattern **110.**

Accepted: 1010001, 00111, ... **Rejected:** 10100**110**0, 00**110**101, 1**110**, ...



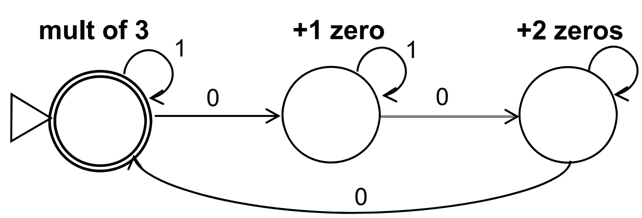
Add the Missing Transitions: Multiple-of-3 0s

Construct a FSM accepting strings in which the **number of 0s** is a **multiple of 3**.



- multiple of 3 = 0, 3, 6, 9, …
- number of 1s doesn't matter
- **accepted** strings include: 110101110, 11, 0000010
- **rejected** strings include: 101, 0000, 111011101111
- ***you may not need all four states!***

State == Set of Equivalent Input Strings



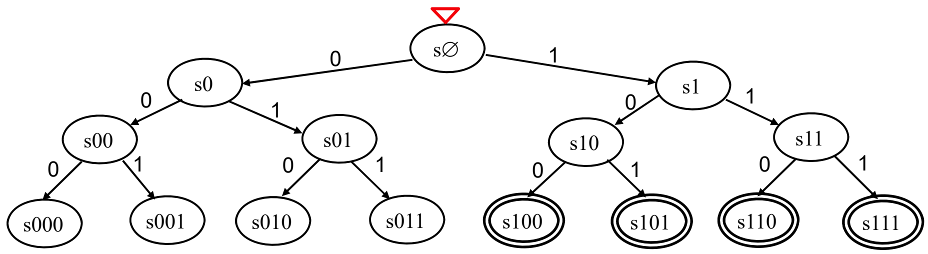
- Two input strings are ***not*** equivalent if adding the same characters to each of them produces a different outcome.
  - one of the resulting strings is accepted
  - the other is rejected
- Example: are '10' and '001' equivalent in mult-of-3-0s problem?
  '10' + '00' → '1000' (accepted)
  '001' + '00' → '00100' (rejected)
  → '10' and '001' are *not* equivalent in this problem; they *must* be in *different* states!

Third-to-Last Bit Is a 1

examples of
<u>accepted strings</u>
101100
101
00110110

Construct a FSM accepting only strings in which the third bit from the end is a 1.

In theory, we could do something like this:



Why are these accepting states?

additional transitions are needed!

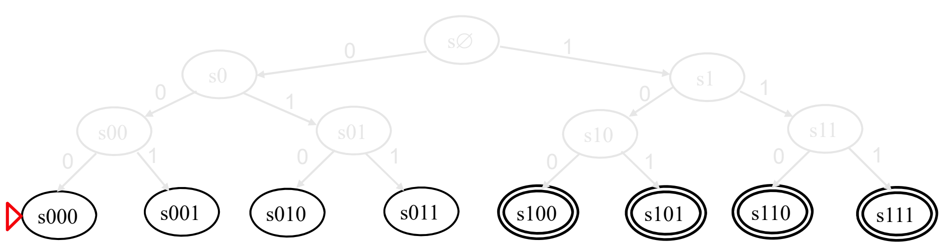
Which state should we enter if:

- we're in s111 and the next bit is a 0?
- we're in s100 and the next bit is a 1?

Third-to-Last Bit Is a 1

Construct a FSM accepting only strings in which the third bit from the end is a 1.

Because we only care about the last 3 bits, 8 states is enough!



additional transitions are needed!

Examples of equivalent states:

- ø, 0, 00, 000: we're 3 transitions away from an accepting state
- 1, 01, 001: we're 2 transitions away from an accepting state

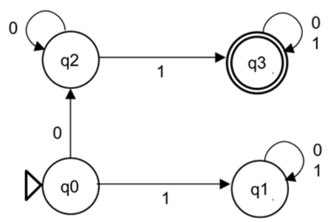
More FSM Practice!

- Construct a FSM accepting bit strings in which:
  - the **<u>first</u>** bit is 0
  - the **<u>last</u>** bit is 1
- Here are the classes of equivalent inputs:
  - empty string (q0)
  - first bit is 1 (q1)
  - first bit is 0, last bit is 0 (q2)
  - first bit is 0, last bit is 1 (q3)

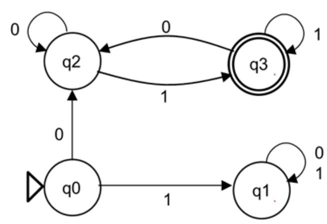
Which of these is the correct FSM?

- Construct a FSM accepting bit strings in which:
  - the **<u>first</u>** bit is 0
  - the **<u>last</u>** bit is 1

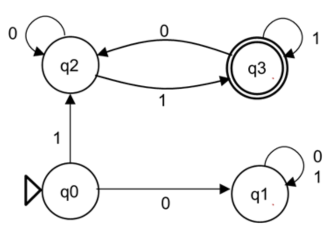
A.



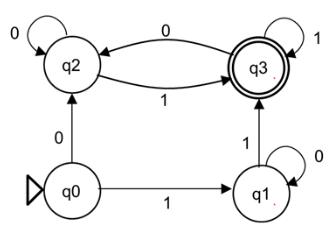
C.



B.



D.

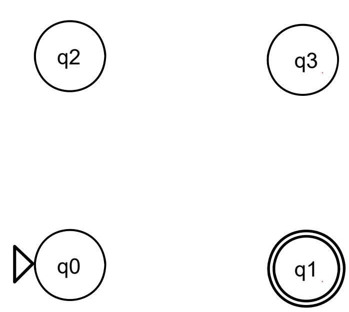


Even More Practice!

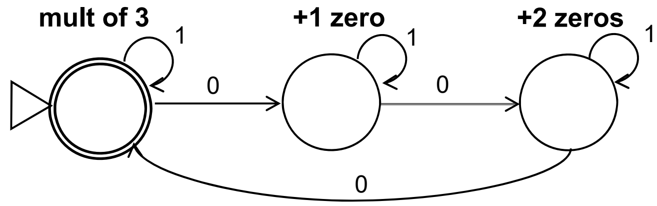
- Construct a FSM accepting bit strings in which:
  - the number of 1s is odd
  - the number of 0s is even
- ***What are the classes of equivalent inputs?***

Even More Practice!

- Construct a FSM accepting bit strings in which:
  - the number of 1s is odd
  - the number of 0s is even
- ***What are the classes of equivalent inputs?***



Recall: State == Set of Equivalent Input Strings



- Two input strings are ***not*** equivalent if adding the same characters to each of them produces a different outcome.
  - one of the resulting strings is accepted
  - the other is rejected

What About This Problem?

- Construct a FSM accepting bit strings that:
  - start with **some** number of 0s
  - followed by the **same** number of 1s
  - 01, 0011, 000111, 00001111, etc.
- ***What are the classes of equivalent inputs?***

What About This Problem?

- Construct a FSM accepting bit strings that:
  - start with **<u>some</u>** number of 0s
  - followed by the **<u>same</u>** number of 1s
  - 01, 0011, 000111, 00001111, etc.

- ***What are the classes of equivalent inputs?***

  an infinite number of them!

  n **0**s, followed by (*n*+1) or more **1**s, and/or by an alternation between groups of **1**s and **0**s – **rejected; can't recover!**

  n **0**s, followed by *n* **1**s – **accepted!** (and any further input is bad!)

  n **0**s, followed by (*n*–1) **1**s – need *one* more **1** to accept

  n **0**s, followed by (*n*–2) **1**s – need *two* more **1**s to accept

  n **0**s, followed by (*n*–3) **1**s – need *three* more **1**s to accept

  ...

- **Impossible to solve using a *finite* state machine!**

Limitations of FSMs

- Because they're finite, FSMs can only count finitely high!

Computable with FSMs

even/odd sums or differences

multiples of other integers

finite input constraints:

third digit is a 1
third-to-last digit is a 1
third digit == third-to-last digit
etc.

***Uncomputable* with FSMs**

equal numbers of two values

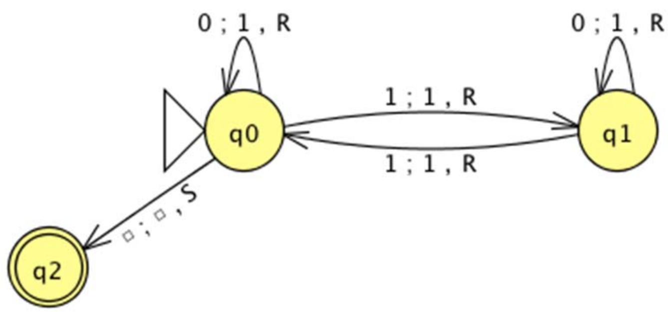
two more **1**s than **0**s or vice versa

infinite input constraints:

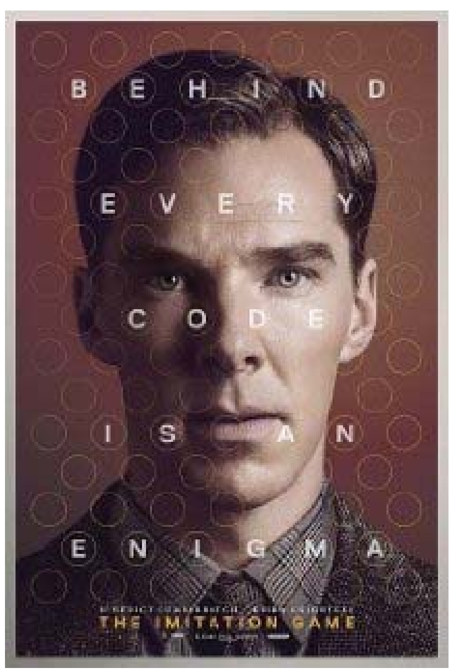
palindromes

anything modeled by a potentially unbounded `while` *loop*

A Better Machine!



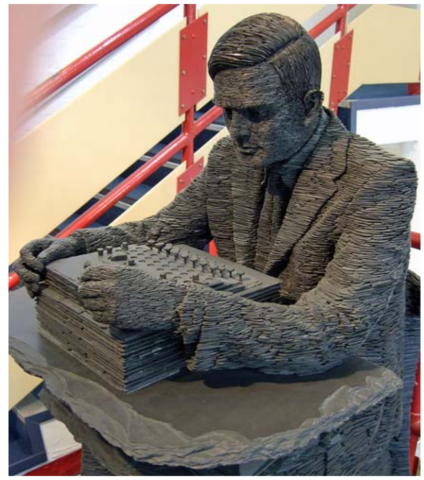
Turing Machine (TM)



Alan Turing (1912-1954)

WWII

Enigma machine ~ The axis's encryption engine



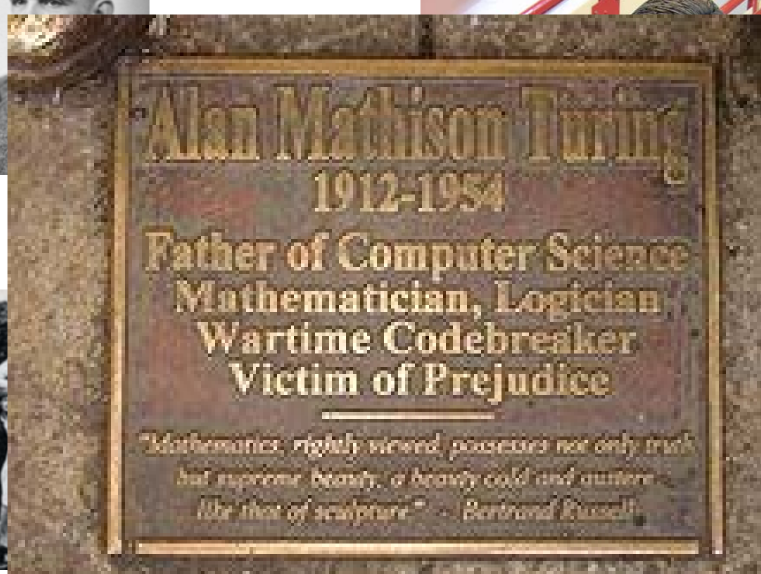
Bletchley Park

1946

Turing Award

The ACM A.M. Turing Award is an annual prize given by the Association for Computing Machinery to "an individual selected for contributions of a technical nature made to the computing community". Wikipedia

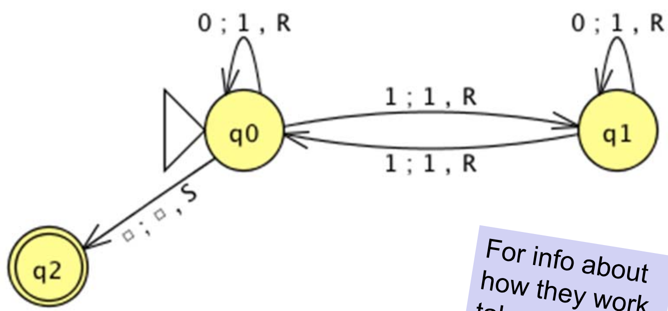
Alan Turing (1912-1954)



1946

computing community". Wikipedia

A Better Machine!



For info about how they work, take CS 332!

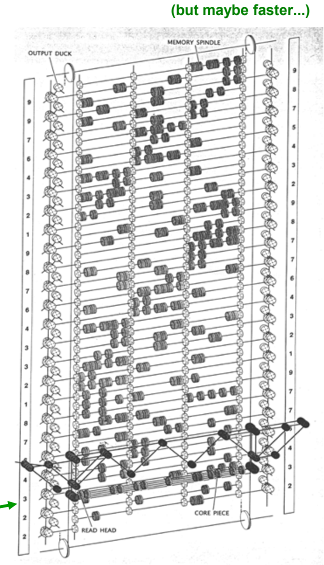
Turing Machine (TM)

So far, all known computational devices can compute only what Turing Machines can...

(but maybe faster...)

- Quantum computation
  http://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf
- Molecular computation
  http://www.arstechnica.com/reviews/2q00/dna/dna-1.html
- Parallel computers
- Integrated circuits

- Electromechanical computation
- Water-based computation
- Tinkertoy computation



Algorithm Efficiency and Problem Hardness

Computer Science 111
Boston University

Algorithm Efficiency

- This semester, we've developed algorithms for many tasks.
- For a given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is their relative *efficiency*.
  - space efficiency: how much memory an algorithm requires
  - time efficiency: how quickly an algorithm executes
    - how many "operations" it performs

Example of Comparing Algorithms

- Consider the problem of finding a phone number in a phonebook.
- Let's informally compare the time efficiency of two algorithms for this problem.

Algorithm 1 for Finding a Phone Number

```
def find_number1(person, phonebook):
    for p in range(1, phonebook.num_pages + 1):
        if person is found on page p:
            return the person's phone number

    return None
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
- What if there were 1,000,000 pages?

Algorithm 2 for Finding a Phone Number

```
def find_number2(person, phonebook):
    min = 1
    max = phonebook.num_pages

    while min <= max:
        mid = (min + max) // 2     # the middle page
        if person is found on page mid:
            return the person's number
        elif person comes earlier in phonebook:
            max = mid - 1
        else:
            min = mid + 1

    return None
```

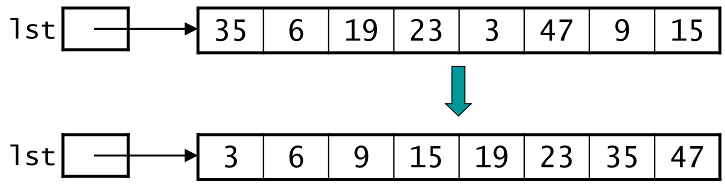
- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
- What if there were 1,000,000 pages?

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a value in a list
- Algorithm 1 is known as *sequential search*.
- Algorithm 2 is known as *binary search*.
  - only works if the items in the data collection are sorted
- For large collections of data, binary search is significantly faster than sequential search.

Sorting a Collection of Data

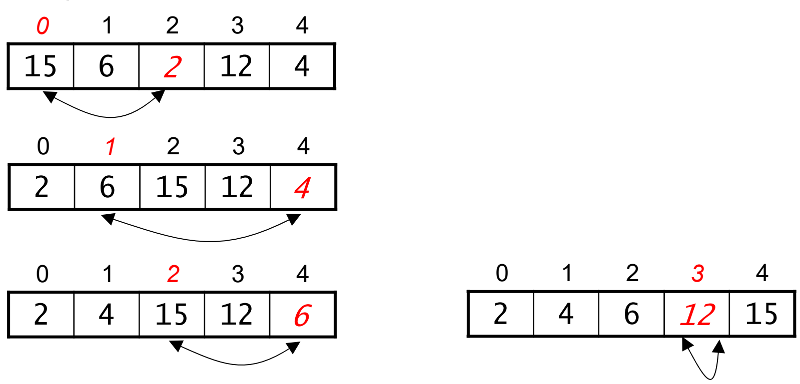
- It's often useful to be able to sort the items in a list.
- Example:



- Many algorithms have been developed for this purpose.
  - CS 112 looks at a number of them
- For large collections of data, some sorting algorithms are *much* faster than others.
  - we can see this by comparing two of them

Selection Sort

- Basic idea:
  - consider the positions in the list from left to right
  - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:



Why don't we need to consider position 4?

If we're using selection sort to sort
`[24, 8, 5, 2, 17, 10, 7]`
what will the list look like after we select
elements for the <u>first three positions</u>?

A. `[2, 5, 7, 24, 17, 10, 8]`

B. `[2, 5, 7, 8, 24, 17, 10]`

C. `[5, 8, 24, 2, 17, 10, 7]`

D. `[2, 5, 8, 24, 17, 10, 7]`

E. none of these

Quicksort

- Another possible sorting algorithm is called quicksort.
- It uses recursion to "divide-and conquer":
  - *divide:* rearrange the elements so that we end up with two sublists that meet the following criterion:
    - *each element in the left list <= each element in the right list*

  example:

12	8	14	4	6	13

  →

6	8	4	|	14	12	13

  - *conquer:* apply quicksort recursively to the sublists, stopping when a sublist has a single element
  - note: when the recursive calls return, nothing else needs to be done to "combine" the two sublists!

Comparing Selection Sort and Quicksort

- Selection sort's running time "grows proportionally to" n^2, (n = length of list).
  - make the list 2x longer → the running time will be ~4x longer
  - make the list 3x longer → the running time will be ~9x longer
  - make the list 4x longer → ???
- Quicksort's running time "grows proportionally to" $n \log_2 n$.
  - we've seen that $\log_2 n$ grows much more slowly than n
  - thus, $n \log_2 n$ grows much more slowly than n^2
- For large lists, quicksort is significantly faster than selection sort.

We use selection sort to sort a list of length 40,000, and it takes 3 seconds to complete the task.

If we now use selection sort to sort a list of length 80,000, roughly how long should it take?

Algorithm Analysis

- Computer scientists characterize an algorithm's efficiency by specifying its *growth function*.
  - the function to which its running time is roughly proportional
- We've seen several different growth functions:

```
log2n       # binary search
n           # sequential/linear search
n log2n     # quicksort
n2          # selection sort
```

- Others include:

```
cn          # exponential growth
n!          # factorial growth
```

- CS 112 develops a mathematical formalism for these functions.

How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 μsec (1×10^{-6} sec)

growth function	problem size (n) 10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	**35.7 yrs**	**36,600 yrs**

Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.

  $\log_2 n$
  n
  $n \log_2 n$
  n^2
  n^3
  etc.

  - we can solve large problem instances in a reasonable amount of time

- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.

  c^n
  $n!$

  - they can only be solved exactly for small values of n

Example of a "Hard" Problem: Map Labeling

- ***Given:*** the coordinates of a set of *point features* on a map
  - cities, towns, landmarks, etc.

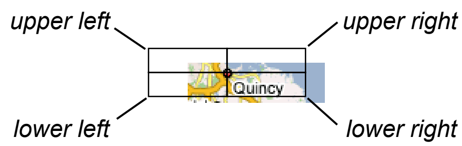
- ***Task:*** determine positions for the point features' *labels*

- Because the point features tend to be closely packed, we may get overlapping labels.

- ***Goal:*** find the labeling with the fewest overlaps

Map Labeling (cont.)

- One possible solution algorithm: ***brute force!***
  - try all possible labelings
- How long would this take?
- Assume there are only 4 possible positions for each point's label:



  - for n point features, there are 4^n possible labelings
  - thus, running time will "grow proportionally" to 4^n

exponential time!

  - example: 30 points → 4^{30} possible labelings
    - if it took 1 μsec to consider each labeling,
      it would take over 36,000 years to consider them all!

Can Optimal Map Labeling Be Done Efficiently?

- In theory, a problem like map labeling could have a yet-to-be discovered efficient solution algorithm.
- How likely is this?
- **Not very!**
- If you could solve map labeling efficiently, you could also solve many other hard problems!
  - the *NP-hard* problems
  - another example: the traveling salesperson problem in the optional reading from *CS for All*

Dealing With "Hard" Problems

- When faced with a hard problem, we resort to approaches that quickly find solutions that are "good enough".
- Such approaches are referred to as *heuristic* approaches.
  - heuristic = rule of thumb
  - no guarantee of getting the optimal solution
  - typically get a good solution

Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.
  - we can solve large problem instances in a reasonable amount of time
- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.
  - they can only be solved exactly for small values of n
- **A third class: *Impossible* problems!**
  - **can't be solved, no matter how long you wait!**
  - **referred to as *uncomputable* problems**