# Computer Science 111
## Introduction to Computer Science I

Boston University, Fall 2023

Unit 1: Functional Programming in Python

<table>
<thead>
<tr>
<th>Course Overview</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python Basics</td>
<td>pre-lecture: 12 / in-lecture: 24</td>
</tr>
<tr>
<td>Strings and Lists</td>
<td>27 / 32</td>
</tr>
<tr>
<td>Intro. to Functions</td>
<td>36 / 39</td>
</tr>
<tr>
<td>Making Decisions: Conditional Execution</td>
<td>43 / 49</td>
</tr>
<tr>
<td>Variable Scope; Functions Calling Functions</td>
<td>55 / 61</td>
</tr>
<tr>
<td>A First Look at Recursion</td>
<td>66 / 72</td>
</tr>
<tr>
<td>More Recursion</td>
<td>75</td>
</tr>
<tr>
<td>Recursive Design</td>
<td>79 / 82</td>
</tr>
<tr>
<td>List Comprehensions (and the range function)</td>
<td>86 / 89</td>
</tr>
<tr>
<td>More Recursive Design</td>
<td>94</td>
</tr>
<tr>
<td>Lists of Lists; ASCII Codes and the Caesar Cipher</td>
<td>100 / 106</td>
</tr>
<tr>
<td>Algorithm Design</td>
<td>113</td>
</tr>
</tbody>
</table>

Unit 2: Looking "Under the Hood"

| Binary Numbers | 120 / 128 |
| Binary Arithmetic Revisited | 135 |
| Gates and Circuits | 146 / 150 |
| Minterm Expansion | 157 / 160 |
| Circuits for Arithmetic; Modular Design | 167 |

Unit 3: Imperative Programming in Python

| Definite Loops; Cumulative Computations | 175 / 183 |
| Indefinite Loops (plus User Input) | 190 / 195 |
| Program Design with Loops | 203 |
| Nested Loops | 210 / 213 |
| References and Mutable Data | 224 / 232 |
| 2-D Lists; References Revisited | 243 / 247 |

Unit 4: Object-Oriented Programming in Python

| Using Objects; Working with Text Files | 255 / 262 |
| Classes and Methods | 270 / 281 |
| More Object-Oriented Programming; Comparing and Printing Objects | 292 / 296 |
| Dictionaries and Markov Models | 303 |
| Board Objects for Connect Four | 314 |
| Inheritance | 320 |
| AI for Connect Four | 325 |

Unit 5: Topics from CS Theory

| Finite-State Machines | 340 |
| Algorithm Efficiency and Problem Hardness | 353 |

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

- Less rigorous alternatives include:
  - CS 101: overview of CS
  - CS 103: the Internet
  - CS 105: databases and data mining
  - CS 108: programming with a focus on web apps
  - 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
  - create your account and purchase a subscription ASAP (see Lab 0 for more details)

- 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 posted by 36 hours before lecture
  • the pre-lecture quizzes
  • list of pages covered in 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

- At a minimum, students who engage in misconduct will have their final grade reduced by one letter grade.
  - e.g., from a B to a C

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:
  • make 85% of the Top Hat votes over the entire semester
  • attend 85% of the labs
  (voting from outside classroom and voting for someone else are **not** allowed!)

• 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 – including isolation for Covid-19.

• 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 and C1)
  Tiago Januario (B1)

• **Teaching Assistants (TAs)**
  **plus Undergrad Course Assistants (CAs)**
  • see the course website for names and photos:
  https://www.cs.bu.edu/courses/cs111/staff.shtml

• Office-hour calendar:
  https://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

The prompt shows that the interpreter is waiting for you to enter something.
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 two sets 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.

```python
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.

```python
temp = 20
```

- General syntax:

  ```
  variable = expression
  ```

- 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:

```python
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 | num1 100 | num2 120 |
  | num2 = 120 |           |          |

  | num1 = 50  | num1    | num2 120 |
  |           |          |          |

  | num1 = num2 * 2 | num1    | num2 120 |
  | num2 = 60     |          |          |

**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:

  \[
  \begin{array}{c|c|c}
  \text{sum} = 13 & \text{sum} & 13 \\
  \text{val} = 30 & \text{val} & 30 \\
  \hline
  \text{sum} = \text{sum} + \text{val} & \text{sum} & 43 \\
  \hline
  \text{val} = \text{val} \times 2 & \text{val} & \text{val} \\
  \end{array}
  \]

Creating a Reusable Program

- Put the statements in a text file.

```python
# 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:
  \[
  \text{print}(\text{expr}) \\
  \text{or} \\
  \text{print}(\text{expr}_1, \text{expr}_2, \ldots, \text{expr}_n)
  \]
  where each \textit{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, \textit{separated by spaces}

- To print a blank line, omit the expressions:
  \[
  \text{print}() 
  \]

Print Statements (cont.)

- Examples:
  - first example:
    \[
    \text{print}('the results are:', 15 + 5, 15 - 5)
    \]
    'the results are:' 20 10
    output: \textit{the results are: 20 10}
    (note that the quotes around the string literal are \textit{not} printed)

  - second example:
    \[
    \text{cents} = 89 \\
    \text{print}('you have', \text{cents}, 'cents')
    \]
    output: __________________________
Variables and Data Types

• The `type` function gives us the type of an expression:
  ```python
  >>> type('hello')
  <class 'str'>
  >>> type(5 / 2)
  <class 'float'>
  ```

• Variables in Python do not have a fixed type.
  • examples:
    ```python
    >>> 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?

```python
x = 15
name = 'Picobot'
x = x // 2
print('name', x, type(x))
```
What about this program?

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

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Complete this table to keep track of the values of the variables!
What are the values of the variables after the following code runs?

\[
\begin{align*}
  x &= 5 \\
  y &= x ** 2 \\
  z &= x \% 3 \\
  x + 2 \\
\end{align*}
\]

On paper, make a table for the values of your variables!
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:** \texttt{string[index]}
  
  ```python
  >>> name = 'Picobot'
  >>> name[1]
  'i'
  >>> name[-3]
  _____
  ```

- **Slicing (extracting a substring):** \texttt{string[start:end]}
  
  ```python
  >>> name[0:2]
  'Pi'
  >>> name[1:-1]
 _____ 
  >>> name[1:]
  'icobot'
  >>> name[:4]
  _____
  ```

String Operations (cont.)

- **Concatenation:** \texttt{string1 + string2}
  
  ```python
  >>> word = 'program'
  >>> plural = word + 's'
  >>> plural
  'programs'
  ```

- **Duplication:** \texttt{string * num\_copies}
  
  ```python
  >>> 'ho!' * 3
  'ho!ho!ho!' 
  ```

- **Determining the length:** \texttt{len(string)}
  
  ```python
  >>> name = 'Perry'
  >>> len(name)
  5
  >>> len('')  # an empty string – no characters!
  0
  ```
Skip-Slicing

• Slices can have a third number: \[ \text{string[start:end:stride_length]} \]

\[
\begin{array}{c}
s = \text{'boston university terriers'} \\
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
\end{array}
\]

\[
\begin{array}{c}
>>> s[0:8:2] \\
'bso' \quad \# \text{note the space at the end!}
\end{array}
\]

Skip-Slicing (cont.)

• Slices can have a third number: \[ \text{string[start:end:stride_length]} \]

\[
\begin{array}{c}
s = \text{'boston university terriers'} \\
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
\end{array}
\]

\[
\begin{array}{c}
>>> s[5:0:-1]
\end{array}
\]
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[2:]
  >>> info[2]
  1977
  >>> info[1]
  1977
  >>> info[-1][-1]
  460.9
  >>> info[-1]
  460.9
  >>> info[-1][-1]
  >>> info[0][-4]
  ['Star wars', 1977, 'PG', [35.9, 460.9]]
Strings and Lists

Computer Science 111
Boston University

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

```python
s = 'abc'
s = ('d' * 3) + s
s = s[2:-2]
```
Fill in the blank to make the code print 'compute!'

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

---

**Skip-Slicing**

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

```python
s = 'boston university terriers'
```  

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

```python
>>> s[5:0:-1]
'notso'
```

```python
>>> s[::]  # what numbers do we need?
'viti'
```

```python
>>> s[12:21:8] + s[21::3]  # what do we get?
```
What is the output of the following program?

```python
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:
  ```python
  >>> 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
  ```python
  >>> info = ['Star Wars', 1977, 'PG', [35.9, 460.9]]
  >>> info[1:2]  # You cannot slice a list
  1977
  >>> info[1]    # Indexing produces a single element
  1977
  >>> info[-1]   # Slicing produces a single element
  [35.9, 460.9]
  >>> info[1:2]  # Slicing produces a list
  [1977]
  >>> info[1:2]  # Slicing produces a list
  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:]
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 !'

<table>
<thead>
<tr>
<th>Part 1</th>
<th>Part 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is len(pi)</td>
<td>What is L[0]</td>
</tr>
<tr>
<td>What is len(L)</td>
<td>What is L[0:1]</td>
</tr>
<tr>
<td>What is len(L[1])</td>
<td>What is L[0][1]</td>
</tr>
<tr>
<td>What is pi[2:4]</td>
<td>What slice of M is 'try'?</td>
</tr>
<tr>
<td>What slice of pi is [3,1,4]</td>
<td></td>
</tr>
<tr>
<td>What slice of pi is [3,4,5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>What is M[9:15]</td>
</tr>
<tr>
<td></td>
<td>What is M[::5]</td>
</tr>
</tbody>
</table>

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

def triple(x):
    return 3*x

x is the input or parameter

the function's name

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))

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

print(calculate(3, 2))
Functions With String Inputs

```python
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)
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))
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:
    ```python
    >>> 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
    ```

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

Returning vs. Printing

• Our previous function `returns` the result:
  ```python
def avg_first_last(values):
    ...
    return __________
```

• Would it be equivalent to print the result?
  ```python
def avg_first_last(values):
    ...
    print(_____________
```

• If the function prints instead of returning, you can't do something like this:
  ```python
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:
  ```python
  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:
  ```python
  >>> abs_value(-5)
  5
  >>> abs_value(10)
  ```
Simple Decisions: if Statements

- Syntax:
  ```python
  if condition:
      true block
  ```
  where:
  - `condition` is an expression that is true or false
  - `true block` is one or more indented statements

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

Two-Way Decisions: if-else Statements

- Syntax:
  ```python
  if condition:
      true block
  else:
      false block
  ```

- Example:
  ```python
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:

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
<td><code>val &lt; 10</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>price &lt; 10.99</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
<td><code>num &gt; 60</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>state &gt; 'Ohio'</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
<td><code>average &lt;= 85.8</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
<td><code>name &gt;= 'Jones'</code></td>
</tr>
<tr>
<td><code>==</code></td>
<td>equal to</td>
<td><code>total == 10</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>letter == 'P'</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal to</td>
<td><code>age != my_age</code></td>
</tr>
</tbody>
</table>

Boolean Values and Expressions

- A condition has one of two values: True or False.
  ```python
  >>> 10 < 20
  True
  >>> "Jones" == "Baker"
  False
  ```

- True and False are not strings.
  - they are literals from the bool data type
    ```python
    >>> type(True)
    <class 'bool'>
    >>> type(30 > 6)
    <class 'bool'>
    ```

- 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:

<table>
<thead>
<tr>
<th>name</th>
<th>example and meaning</th>
<th>True if both conditions are True, and False otherwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>age &gt;= 18 and age &lt;= 35</td>
<td>True if both conditions are True, and False otherwise</td>
</tr>
<tr>
<td>or</td>
<td>age &lt; 3 or age &gt; 65</td>
<td>True if one or both of the conditions are True; False if both conditions are False</td>
</tr>
<tr>
<td>not</td>
<td>not (grade &gt; 80)</td>
<td>True if the condition is False, and False if it is True</td>
</tr>
</tbody>
</table>

A Word About Blocks

- A block can contain multiple statements.

```python
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.

```python
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.

```python
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:

```python
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

```
condition1
  true
  true block 1
false
condition2
  true
  true block 2
false
  ...
false
false block
false
next 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.

```python
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.

```python
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?

```python
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!)

```python
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 new changes!)

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

How many lines does this print?

```python
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?

```python
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?

```python
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.

<table>
<thead>
<tr>
<th>boolean</th>
<th>boolean</th>
<th>boolean</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == 0</td>
<td>a == 1</td>
<td>a == 0</td>
<td>or 1</td>
</tr>
<tr>
<td>(do this)</td>
<td>(don't do this)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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:

```python
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

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
mystery(5, 2)
print(b + c)         # we access the global b here

c = 7
print(b + c)

• 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 in this program?

Frames for Variables (cont.)

- What is the output of this fixed version of the program?
Finding the Distance Between Two Points

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

Diagram from: math.about.com
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)
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?

```python
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))
Tracing Function Calls

```python
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)
```

Full Trace of First Example

```python
def quadruple(y):
    # 3. local y = 8
    y = 4 * y  # 4. local y = 4 * 8 = 32
    return y   # 5. return local y's value

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!

```python
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
    \[ n! = n \times (n-1)! \]
Functions Calling Themselves: *Recursion!* (cont.)

```python
def fac(n):
    if n <= 1:
        return 1  # base case
    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)*

```python
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!

<table>
<thead>
<tr>
<th>Fac(n)</th>
<th>n:</th>
<th>rest:</th>
<th>return value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac(5)</td>
<td>n:</td>
<td>rest:</td>
<td>return value:</td>
</tr>
<tr>
<td>fac(4)</td>
<td>n:</td>
<td>rest:</td>
<td>return value:</td>
</tr>
<tr>
<td>fac(3)</td>
<td>n:</td>
<td>rest:</td>
<td>return value:</td>
</tr>
<tr>
<td>fac(2)</td>
<td>n:</td>
<td>rest:</td>
<td>return value:</td>
</tr>
<tr>
<td>fac(1)</td>
<td>n:</td>
<td>rest:</td>
<td>return value:</td>
</tr>
</tbody>
</table>
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
Recursively Raising a Number to a Power

```python
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...

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

```python
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

Recall: Functions Calling Themselves: *Recursion!*

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

- 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!

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

Recursion does almost all of the rest of the problem!
You specify one step at the end.

You handle the base case – the easiest case!

How many times will mylen() be called?

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

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

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

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

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

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))
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?

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')
How recursion works...

```python
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
```

Debugging Technique: Adding Temporary prints

```python
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:
  ```python
  >>> 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?

```python
list1 = [30, 2, 18]          list2 = [5, 12, 25, 2]

largest element = 18          largest element = 10

mymax(list1) \rightarrow _____    mymax(list2) \rightarrow _____
```

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

```python
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!*

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Values</th>
<th>Max In Rest</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mymax([10, 12, 5, 8])</td>
<td>[10, 12, 5, 8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mymax([12, 5, 8])</td>
<td>[12, 5, 8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mymax()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mymax()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the stack frames!
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
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:
  >>> replace('boston', 'o', 'e')
  result: '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`?

```python
s1 = 'a_____
 s2 = 'r_____
```

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

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

---

Complete This Function Together!

```python
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
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:

```python
>>> 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...

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

This takes on each value

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

Examples of LCs

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

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

__________________________
List Comprehensions (LCs)

• Syntax:

\[
\begin{align*}
&[\text{expression for variable in sequence}] \\
&\text{or} \\
&[\text{expression for variable in sequence if boolean}]
\end{align*}
\]

• Examples:

\[
\begin{align*}
&[0, 1, 2, 3, 4, 5] \\
&[2*x \text{ for } x \text{ in } \text{range}(6) \text{ if } x \% 2 == 0] \\
&[0, 4, 8] \\
&[y \text{ for } y \text{ in } ['how', 'now', 'brown'] \text{ if } \text{len}(y) == 3]
\end{align*}
\]
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

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

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

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

```
3*x
define the
```

and 3*x is output for each one

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

More Examples

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

```
[8, 10, 12, 14]
```

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

```
['g', 's']
```

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

```
[0, 1, 2, 3, 4, 5]
```

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

```
[1, 3, 5]
```

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

```
[False, True, False, True, False]
```

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

```
[1, 5]
```

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

```
6
```
What is the output of this code?

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

LC Puzzles! – Fill in the blanks

```python
>>> [__________ 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'])
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)

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 ________ / _________

>>> avg_len(['commonwealth', 'avenue'])
9.0

>>> avg_len(['keep','calm','and','code','on'])
3.4
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?

```python
def remove_vowels(s):
    if s == '':  # base case
        return ________
    else:         # recursive case
        rem_rest = ________________
        # do our one step!
        ...
```

Consider this original call...

```python
def remove_vowels(s):
    if s == '':  # base case
        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')

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?*

```python
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

```python
>>> 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

\texttt{rem\_all(10, [3, 5, 10, 7, 10])} \hspace{1cm} # first value is \textit{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...?

\textit{What is our one step?}

\texttt{rem\_all(10, [10, 3, 5, 10, 7])} \hspace{1cm} # first value \textit{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...?

\textit{What is our one step?}

\begin{verbatim}
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 ________________
\end{verbatim}
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
  ```python
  >>> max([4, 10, 2])
  10
  >>> max(['all', 'students', 'love', 'recursion'])
  'students'
  ```

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

- Recall that the elements of a list can themselves be lists:
  \[
  \begin{bmatrix}
  [124, 'Jaws'],
  [150, 'Lincoln'],
  [115, 'E.T.'],
  \end{bmatrix}
  \]

- 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]
     ```
Here’s a function that works for an arbitrary list of words:

```python
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]
```
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.

<table>
<thead>
<tr>
<th>character</th>
<th>ASCII value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>65</td>
</tr>
<tr>
<td>'B'</td>
<td>66</td>
</tr>
<tr>
<td>'C'</td>
<td>67</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Converting Between Characters and Numbers

**ASCII values**

| 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 |
|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| a  | b  | c  | d   | e   | f   | g   | h   | i   | j   | k   | l   | m   | n   | o   | p   | q   | r   | s   | t   | u   | v   | w   | x   | y   | z   |

| 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| A  | B  | C  | D  | E  | F  | G  | H  | I  | J  | K  | L  | M  | N  | O  | P  | Q  | R  | S  | T  | U  | V  | W  | X  | Y  | Z  |

**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**

```python
>>> ord('e')
101
>>> chr(101)
'e'
>>> ord('G')
71
>>> chr(71)
'G'
```

**Encryption**

Original message: 
'my password is foobar'

Encrypted message: 
'pb sdvvzrug lv irredu'
Caesar Cipher Encryption

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

   \[
   \text{abcdefghijklmnopqrstuvwxyz}
   \rightarrow
   \text{ABCDEFGHIJKLMNOPQRSTUVWXYZ}
   \]

• Example: a shift of 3

   \[
   \begin{align*}
   'a' & \rightarrow 'd' & 'A' & \rightarrow 'D' \\
   'b' & \rightarrow 'B' \\
   'c' & \rightarrow 'C' \\
   \end{align*}
   \]

   etc.

   \[
   \text{original message} \quad \rightarrow \quad \text{encrypted message}
   \]

   'my password is foobar' \rightarrow 'pb sdvvezug lv irredu'

• Non-alphabetic characters are left alone.

• We "wrap around" as needed.

   \[
   \begin{align*}
   'x' & \rightarrow 'a' \\
   'y' & \rightarrow 'A' \\
   \end{align*}
   \]

   etc.

Implementing a Shift in Python

• `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

```python
>>> max([578.7, 596.0, 586.9])
result: 596.0
```

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

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

```python
>>> 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
    
    \[
    \text{words} = ['\text{aliens}', 'z\text{ap}', 'h\text{azy}', 'c\text{ode}']
    \]
  - a `scrabble_score()` function like the one from PS 2

- To find the best word:
  - form a list of lists using a list comprehension
    
    \[
    \text{scored_words} = [[\text{scrabble_score(w)}, w] \text{ for } w \text{ in } \text{words}]
    \]
  
    ## for the above words, we get the following:

  - use `max()` to get the best [score, word] sublist:
    
    \[
    \text{bestpair} = \text{max(scored_words)}
    \]
  
    ## for the above words, we get the following:

  - use indexing to extract the word: `bestpair[1]`

```
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.
  \[
  \text{abcdefghijklmnopqrstuvwxyz} \rightarrow \text{bcdefghijklmnopqrstuvwxyz}
  \]

• Example: a shift of 3
  'a' \rightarrow 'd'
Caesar Cipher in PS 3

- You will write an encipher function:
  ```python
  >>> 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
  ```python
  >>> encipher('XYZ xyz', 3)
  result: 'ABC abc
  ```

What Should This Code Output?

```python
secret = encipher('Caesar? Wow!', 5)
print(secret)
```
Caesar Cipher with a Shift/Rotation of 13

- 'a' \rightarrow 'n'
- 'b' \rightarrow 'o'
- 'c' \rightarrow 'p'

etc.

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':

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 oqedeq Bzdrzq rzkzc.
  result: 'Caesar cipher? I prefer Caesar salad.'

  >>> decipher('Bomebcsyx sc pexnwoxdkv')
  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')

Score them all

max! [4, 'la ab iybal']

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

max! [0.45555, 'et tu brute']

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('always', 'walking') work?

- will jscore('always', '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

  ```python
  >>> 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

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

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

```python
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

```python
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.

<table>
<thead>
<tr>
<th>binary</th>
<th>decimal integer</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100001</td>
<td>97</td>
<td>'a'</td>
</tr>
<tr>
<td>01100010</td>
<td>70</td>
<td>'F'</td>
</tr>
</tbody>
</table>

Representing Integers in Decimal

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

176

- 1 hundred + 7 tens + 6 ones
- \(1 \times 10^2 + 7 \times 10^1 + 6 \times 10^0\)
Representing Integers in Binary

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

\[
10110000
\]

\[
1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0
\]

\[
128 + 0 + 32 + 16 + 0 + 0 + 0 + 0
\]

also 176!

What Does the Rightmost Bit Tell Us?

• 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

\[
\begin{array}{c}
1 \\
1 \\
\hline
12537 \\
+ 9272 \\
\hline
21809
\end{array}
\]

Adding Binary Numbers

\[
\begin{array}{c}
01110 \\
+ 11100 \\
\hline
110010
\end{array}
\]
Multiplying Binary Numbers

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\times & & & & 1 & 1 & 1 & 0 \\
\hline
& & & & 0 & 0 & 0 & 0 & 0 \\
& & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
& & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
& + & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
\hline
& & & & & & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

Hint: 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 \(10110100_2 = 180_{10}\)
- 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:
    \[
    \begin{array}{cccccccc}
    b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
    0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
    \end{array}
    \]

  • For each bit that is 1, add \(2^n\), where \(n = \) the bit number
    • example:
      \[
      \begin{array}{cccccccc}
      b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
      0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
      \end{array}
      \]
      
      decimal value = \(2^6 + 2^5 + 2^3 + 2^2 + 2^0\)
      
      \[
      \begin{array}{cccccccc}
      b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
      0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
      \end{array}
      \]
      \[
      \begin{array}{cccccccc}
      b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
      0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
      \end{array}
      \]

• 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?

\[
90 \text{ (decimal)} \rightarrow \text{_____} \text{ (binary)}
\]

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

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

an x denotes a "hidden" bit that we aren't revealing
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 \(10110100_2 = 180_{10}\)

Left-shifting by 1 doubles the value of a number.

In Python, we can apply the left-shift operator (<<) to any integer:

```python
>>> 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:

```python
>>> 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 \texttt{dec_to_bin()} Should Work...

\begin{verbatim}
\texttt{dec_to_bin(13)}
\begin{verbatim}
n = 13
bin\_rest = \texttt{dec_to_bin(6)}
\end{verbatim}
\end{verbatim}

\begin{verbatim}
\texttt{dec_to_bin(6)}
\begin{verbatim}
n = 6
bin\_rest =
\end{verbatim}
\end{verbatim}

\begin{verbatim}
\texttt{dec_to_bin(\_)}
\begin{verbatim}
n =
bin\_rest =
\end{verbatim}
\end{verbatim}

Binary to Decimal: Right-to-Left

- Here again, we can use a \texttt{right-to-left} approach.
- For example:

\begin{quote}
'1101' = ?
\end{quote}

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

- \textit{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

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

**How bin_to_dec() Should Work...**

```python
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

\[
\begin{array}{c}
1 & 1 & 1 \\
\hline
01110 \\
+ & 11100 \\
\hline
101010
\end{array}
\]

Add these two binary numbers \textit{WITHOUT} converting to decimal!

\[
\begin{array}{c}
101101 \\
+ & 1110 \\
\hline
111010
\end{array}
\]

\textbf{Hint:}
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 $\rightarrow$ 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:

  $\begin{array}{c}
  1 \\
  101010 \\
  + 001001 \\
  \hline
  110011
  \end{array}$

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

  ```python
  add_bitwise('101010', '001001')
  ```
PS 4: add_bitwise

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

- It should look something like this:

```python
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:

  ```python
  add_bitwise('100', '010')
  ```

How recursion works: add_bitwise(b1, b2)

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

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

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

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

  ```python
  add_bitwise('', '')
  b1: ''  b2: ''
  base case: return ''
  ```
How recursion works: add_bitwise(b1, b2)

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

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

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

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

```python
add_bitwise('', '')
b1: ''  b2: ''
base case: 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.

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

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

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

```python
add_bitwise('100', '010')
b1: '100'  b2: '010'
sum_rest = '11'
if ...
  return
```
The Tricky Part of `add_bitwise(b1, b2)`

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

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('101', '011')</code></td>
<td><code>b1: '101' b2: '011'</code></td>
</tr>
<tr>
<td><code>sum_rest</code></td>
<td><code>= add_bitwise('10', '01')</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('10', '01')</code></td>
<td><code>b1: '10' b2: '01'</code></td>
</tr>
<tr>
<td><code>sum_rest</code></td>
<td><code>= add_bitwise('1', '0')</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('1', '0')</code></td>
<td><code>b1: '1' b2: '0'</code></td>
</tr>
<tr>
<td><code>sum_rest</code></td>
<td><code>= add_bitwise('', '')</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('', '')</code></td>
<td><code>b1: '' b2: ''</code></td>
</tr>
<tr>
<td><code>base case</code></td>
<td><code>return ''</code></td>
</tr>
</tbody>
</table>

- Changing the rightmost bits to 1

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

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

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('101', '011')</code></td>
<td><code>b1: '101' b2: '011'</code></td>
</tr>
<tr>
<td><code>sum_rest</code></td>
<td><code>= add_bitwise('10', '01')</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('10', '01')</code></td>
<td><code>b1: '10' b2: '01'</code></td>
</tr>
<tr>
<td><code>sum_rest</code></td>
<td><code>= add_bitwise('1', '0')</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('1', '0')</code></td>
<td><code>b1: '1' b2: '0'</code></td>
</tr>
<tr>
<td><code>sum_rest</code></td>
<td><code>= add_bitwise('', '')</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_bitwise('', '')</code></td>
<td><code>b1: '' b2: ''</code></td>
</tr>
<tr>
<td><code>base case</code></td>
<td><code>return ''</code></td>
</tr>
</tbody>
</table>
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!
  
  \[
  \begin{array}{c}
  \text{1} \\
  101 \\
  + \text{ 011} \\
  \text{110} \\
  \end{array}
  \]

- 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:
  
  \[
  \begin{array}{c}
  \text{1} \\
  101010 \\
  + \text{ 001001} \\
  \text{110011} \\
  \end{array}
  \]

- 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

\[
\begin{array}{c}
101101 \\
\times \quad 1110 \\
\hline
000000 \\
1011010 \\
10110100 \\
+ \quad 101101000 \\
\hline
1001110110
\end{array}
\]

Multiply these binary numbers WITHOUT converting to decimal!

\[
\begin{array}{c}
1101 \\
\times \quad 11 \\
\hline
529 \\
\times \quad 42 \\
\hline
1058 \\
+ \quad 2116 \\
\hline
22218
\end{array}
\]

\textbf{Hint:} Do you remember this algorithm? It's the same!
Recall: Finding the Largest Element in a List

```python
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?

```python
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:])
```
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 = $2^n - 1$ $\leftarrow$ 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

```
<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**OR Gate**

**OR** outputs 1 if any input is 1

```
<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
| 1      | 1      | 1       |```
NOT Gate

**NOT** reverses its input

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOT 0</td>
</tr>
<tr>
<td>1</td>
<td>NOT 1</td>
</tr>
</tbody>
</table>

From Gates to Circuits

- We combine logic gates to form larger circuits.

- Example: what is the output when $x = 0$ and $y = 1$?
A circuit is a boolean function – a function of bits!

- takes one or more bits as inputs
- produces the appropriate bit(s) as output

**A Truth Table for a Circuit**

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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!

![Diagram showing the process of computation with input voltages 42 and 9, computational result 51, and output voltages 101010 and 110111.](image)
All Computation Involves Functions of Bits!

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

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x   y   z   w</td>
<td>AND(x, y, z, w)</td>
</tr>
<tr>
<td>0   0   0   0</td>
<td>0</td>
</tr>
<tr>
<td>0   0   0   1</td>
<td>0</td>
</tr>
<tr>
<td>...12 more rows not shown...</td>
<td>0</td>
</tr>
<tr>
<td>1   1   1   0</td>
<td>0</td>
</tr>
<tr>
<td>1   1   1   1</td>
<td>1</td>
</tr>
</tbody>
</table>

OR Gate (with four inputs)

**OR** outputs 1 if any input is 1

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x   y   z   w</td>
<td>OR(x, y, z, w)</td>
</tr>
<tr>
<td>0   0   0   0</td>
<td>0</td>
</tr>
<tr>
<td>0   0   0   1</td>
<td>1</td>
</tr>
<tr>
<td>...12 more rows not shown...</td>
<td>1</td>
</tr>
<tr>
<td>1   1   1   0</td>
<td>1</td>
</tr>
<tr>
<td>1   1   1   1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fifteen 0s</th>
<th>one 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>only</td>
<td>all</td>
</tr>
<tr>
<td>outputs 1</td>
<td>any</td>
</tr>
<tr>
<td>inputs</td>
<td></td>
</tr>
<tr>
<td>output</td>
<td></td>
</tr>
</tbody>
</table>
Circuit Building Blocks: Logic Gates

AND outputs 1 only if ALL inputs are 1

OR outputs 1 if ANY input is 1

NOT reverses its input

- 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

NOT Gate

NOT reverses its input

input \( \rightarrow \) \( x \) \( \rightarrow \) output

\[
\begin{array}{c|c|c}
\text{input} & \text{output} \\
\hline
x & \text{NOT}(x) \\
\hline
0 & 1 \\
1 & 0 \\
\end{array}
\]
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?

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
We need only these three building blocks to compute anything at all!

Claim

We'll prove this next time!

Extra Practice: Recursive Bitwise AND

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

  \[
  \begin{array}{c}
  111010 \\
  101011 \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \hline
  101010 \\
  \hline
  \end{array}
  \]

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

  \[
  \begin{array}{c}
  10101001 \\
  11011 \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \hline
  00001001 \\
  \hline
  \end{array}
  \]
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?

```python
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:

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
<th>inputs</th>
<th>output</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x AND y</td>
<td>x</td>
<td>y</td>
<td>x OR y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>= 0</td>
<td>0</td>
<td>0</td>
<td>= 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>= 0</td>
<td>0</td>
<td>1</td>
<td>= 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>= 0</td>
<td>1</td>
<td>0</td>
<td>= 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>= 1</td>
<td>1</td>
<td>1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

• 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:
  \((x \text{ AND } y) \text{ OR } (x \text{ AND } (\text{NOT } y))\) \(\leftrightarrow\) ____________
This truth table/circuit can be summarized by the expression:

\[ \overline{xy} + xy \]

This expression is the \textit{minterm expansion} of this truth table.

- one \textit{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):
   a. AND the input variables together
   b. if a variable has a 0 in that row, negate it
   c. example: minterm for the 2nd row
      \( \overline{x}yz \)

4. OR the minterms together.

```
<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Minterm Expansion → Circuit

\[\text{minterm expansion} = xyz + \overline{x}yz + xy\overline{z} + xyz\]

```
<table>
<thead>
<tr>
<th>minterm</th>
<th>A</th>
<th>minterm B</th>
<th>minterm C</th>
<th>minterm D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>y</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>
```

```
\text{output}
```

```
<table>
<thead>
<tr>
<th>minterm A</th>
</tr>
</thead>
<tbody>
<tr>
<td>minterm B</td>
</tr>
<tr>
<td>minterm C</td>
</tr>
<tr>
<td>minterm D</td>
</tr>
</tbody>
</table>
```

CAS CS 111
Boston University, Fall 2023
159
Minterm Expansion

Computer Science 111
Boston University

Claim

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

- AND
- OR
- NOT
Constructive Proof!

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>fn(x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

   OR them all together.

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 want.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>fn(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2. For each input row whose output needs to be 1, build an AND circuit that outputs 1 for that specific input. OR them all together.

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

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

Revisiting Our Earlier Circuit...

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

• The bottom AND gate implements which row?
Recall: Boolean Expressions for Truth Tables

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

- This truth table/circuit can be summarized by the expression:
  \[ \overline{xy} + xy \]

- 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)
  \[ \rightarrow 1 \text{ if the 3-digit binary number } xyz > 4 \]
  \[ \rightarrow 0 \text{ otherwise} \]

for example:
- greater_than_4(1, 1, 0) \[ \rightarrow 1 \] (True)
  Why?
- greater_than_4(0, 1, 1) \[ \rightarrow 0 \] (False)
  because \( 011_2 = 3_{10} \), and 3 is not > 4
Building a Minterm Expansion for a Boolean Function

Ex: \texttt{greater\_than\_4(x, y, z)}
\begin{align*}
\rightarrow 1 & \text{ if the 3-digit binary number } xyz > 4 \\
\rightarrow 0 & \text{ otherwise}
\end{align*}

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.

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec</td>
<td>x y z</td>
</tr>
<tr>
<td>0:</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1:</td>
<td>0 0 1</td>
</tr>
<tr>
<td>2:</td>
<td>0 1 0</td>
</tr>
<tr>
<td>3:</td>
<td>0 1 1</td>
</tr>
<tr>
<td>4:</td>
<td>1 0 0</td>
</tr>
<tr>
<td>5:</td>
<td>1 0 1</td>
</tr>
<tr>
<td>6:</td>
<td>1 1 0</td>
</tr>
<tr>
<td>7:</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

Minterm Expansion \rightarrow Circuit

\textit{minterm expansion} =
\begin{align*}
x'y'z' + xyz' + xyz
\end{align*}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{circuit.png}
\end{figure}
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?

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( y )</td>
</tr>
<tr>
<td>row 0</td>
<td>0</td>
</tr>
<tr>
<td>row 1</td>
<td>0</td>
</tr>
<tr>
<td>row 2</td>
<td>0</td>
</tr>
<tr>
<td>row 3</td>
<td>0</td>
</tr>
<tr>
<td>row 4</td>
<td>1</td>
</tr>
<tr>
<td>row 5</td>
<td>1</td>
</tr>
<tr>
<td>row 6</td>
<td>1</td>
</tr>
<tr>
<td>row 7</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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?

<table>
<thead>
<tr>
<th>inputs</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A. $yz + xz + xy$
B. $xyz + x\bar{y}\bar{z} + \bar{x}y\bar{z} + \bar{x}\bar{y}z$
C. $\bar{x}yz + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xyz$
D. $\bar{x}\bar{y}z + \bar{x}yz + x\bar{y}z + \bar{x}y\bar{z}$
E. none of the above

Physical Logic Gates!

6 NOT gates in one chip!
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.

\[ \overline{x}y\overline{z} + x\overline{y}\overline{z} + xyz \]
Recall: Minterm Expansion $\rightarrow$ Circuit

$minterm\ expansion = \ xyz + xy\bar{z} + x\bar{y}z$

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

• What is the minterm expansion formula?

• What is the circuit testing for (i.e., when does it output a 1)?
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!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A+B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A Full Adder

- Recall our bitwise algorithm:

  \[
  \begin{array}{cccc}
  & 0 & 1 & 1 \\
  + & 1 & 0 & 1 \\
  \hline
  & 1 & 1 & 0 \\
  \end{array}
  \]

- A full adder adds only one column.

- It takes 3 bits of input:
  - x and y — one bit from each number being added
  - cin — 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
  - cout — the carry bit out of the current column
  - it becomes the cin of the next column!
How many AND gates will you need in all?

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

<table>
<thead>
<tr>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Full Adder

<table>
<thead>
<tr>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

"Rails" of each bit and its inverse

Subcircuit for s

Subcircuit for c_{out}

3 input bits

2 output bits

How many AND gates will you need in all?

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!

\[
\begin{array}{c}
\text{FA} \\
\text{FA}
\end{array}
\]

\[
\begin{array}{c}
x_1 y_1 \\
x_0 y_0 0
\end{array}
\]

\[
\begin{array}{c}
\text{cout} s \\
\text{cout} s
\end{array}
\]

\[
\begin{array}{c}
s_2 s_1 \\
s_0
\end{array}
\]

\[
\begin{array}{c}
\text{FA} \\
\text{FA}
\end{array}
\]

\[
\begin{array}{c}
x_1 x_0 \\
+ y_1 y_0
\end{array}
\]

\[
\begin{array}{c}
s_2 s_1 s_0
\end{array}
\]

• Produces what is known as a 2-bit ripple-carry adder.

• To add larger numbers, combine even more FAs!

• 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:

\[
\begin{array}{c}
x_1 y_1 \\
x_0 y_0 0
\end{array}
\]

\[
\begin{array}{c}
\text{FA} \\
\text{FA}
\end{array}
\]

\[
\begin{array}{c}
x_1 x_0 \\
+ y_1 y_0
\end{array}
\]

\[
\begin{array}{c}
s_2 s_1 s_0
\end{array}
\]

• Here’s an example computation:

\[
\begin{array}{c}
1 1 \\
0 1 0
\end{array}
\]

\[
\begin{array}{c}
\text{FA} \\
\text{FA}
\end{array}
\]

\[
\begin{array}{c}
10 \\
+ 11
\end{array}
\]
More Modular Design!

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

![Adder Diagram]

- Use these boxes to build other circuits!

Also in PS 5: Building a 4x2 Multiplier

```
  1 1 0 1
x 1 0
--
 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

CPU
- central processing unit
- all computation happens here
- adders, multipliers, etc.
- small number of registers for storing values
- Program instructions are stored with the data in RAM.
- Instructions and data are transferred back and forth between RAM and the CPU.

RAM
- random access memory
- lots of room for storage
- no computation happens here

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

will output:

```
Warning
1
Warning
2
Warning
3
```
for Loops (cont.)

- General syntax:

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

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

- Does the sequence have more values?
  - No: execute statement after the loop
  - Yes: assign the next value in the sequence to variable
        execute the statements in the body

Executing Our Earlier Example
(with one extra statement)

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

- To repeat a loop's body \( N \) times:

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

- Example:

  ```python
  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:

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

- Example:

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

  outputs:

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

```python
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:
  ```python
  age = 13
  age = age + 1
  13 + 1
  14
  ```

• Instead of writing
  ```python
  age = age + 1
  ```
we can just write
  ```python
  age += 1
  ```
### Python Shortcuts (cont.)

<table>
<thead>
<tr>
<th>shortcut</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>var += expr</td>
<td>var = var + (expr)</td>
</tr>
<tr>
<td>var -= expr</td>
<td>var = var - (expr)</td>
</tr>
<tr>
<td>var *= expr</td>
<td>var = var * (expr)</td>
</tr>
<tr>
<td>var /= expr</td>
<td>var = var / (expr)</td>
</tr>
<tr>
<td>var //= expr</td>
<td>var = var // (expr)</td>
</tr>
<tr>
<td>var %= expr</td>
<td>var = var % (expr)</td>
</tr>
<tr>
<td>var **= expr</td>
<td>var = var ** (expr)</td>
</tr>
</tbody>
</table>

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

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

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

\[
\begin{array}{ll}
x & \text{result}
\end{array}
\]
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]))

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

vals = [3, 15, 17, 7]

print(sum(vals))
Index-Based for Loop

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

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

```
i  vals[i]  result
0   3      3
1   15     18
2   17     35
3   7      42
```

Tracing an Index-Based Cumulative Sum

```python
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
0   10     10
1   20     30
2   30     60
3   40     100
4   50     150
```
Definite Loops in Python

Recall: Executing a for Loop

```
for i in [1, 2, 3]:
    print('warning')
    print(i)
```
Another Example

- What would this code output?
  ```python
  for val in [2, 4, 6, 8, 10]:
    print(val * 10)
    print(val)
  ```

- Use a table to help you:
  | more? | val | output/action |

Recall: Simple Repetition Loops

- To repeat a loop's body \( N \) times:
  ```python
  for i in range(N):
    # [0, 1, 2, ..., N–1]
    body of the loop
  ```

- What would this loop do?
  ```python
  for i in range(8):
    print('I'm feeling loopy!')
  ```
Simple Repetition Loops (cont.)

- Another example:
  ```python
  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?

```python
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?

```python
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

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

def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```
Option C Produces an Index-Based for Loop

\[
\begin{array}{c}
vals = [3, 15, 17, 7] \\
0 \quad 1 \quad 2 \quad 3 \\
i
\end{array}
\]

def sum(vals):
def sum(vals):
    result = 0
    result = 0    # the accumulator variable
    for i in range(len(vals)):
    for x in vals:
        result += vals[i]        result += x  # gradually accumulates the sum
    return result
    return result

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

Both Versions Perform a Cumulative Computation

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

print(sum([10, 20, 30, 40, 50]))
What is the output of this program?

```python
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]))
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>vals[i]</th>
<th>vals[i-1]</th>
<th>result</th>
</tr>
</thead>
</table>
```

Follow-Up Questions

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

vals = [3, 15, 17, 7]

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

vals = [3, 15, 17, 7]

---

**Simpler**

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

vals = [3, 15, 17, 7]

**More Flexible**

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

vals = [3, 15, 17, 7]

---

**More on Cumulative Computations**

- Here's a loop-based factorial in Python:

  ```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?

---

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

vals = [3, 15, 17, 7]
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 prints to print on different lines

- Example:
  ```python
  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:

```python
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 = \text{len} (sequence)

```python
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:
  ```python
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:
  ```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):

  ```python
  mult = n
  while mult < bound:
      print(mult, end=' ')
      mult = mult + n
  print()
  ```

<table>
<thead>
<tr>
<th>mult &lt; bound</th>
<th>output thus far</th>
<th>mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 &lt; 70 (True)</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>30 &lt; 70 (True)</td>
<td>15 30</td>
<td>45</td>
</tr>
<tr>
<td>45 &lt; 70 (True)</td>
<td>15 30 45</td>
<td>60</td>
</tr>
</tbody>
</table>

**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`:
  
  ```python
  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*!
Cumulative Computations with Strings

• Recall our recursive remove_vowels function:
  ```python
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:
  ```python
>>> remove_vowels('recurse')
'rcrs'
>>> remove_vowels('vowels')
'vwls'
```
Cumulative Computations with Strings (cont.)

• Here's one loop-based version:

```python
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')`:

```python
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!

```python
def fac(n):
    result = 1
    while n > 0:
        result *= n
    return result
```

- Let's trace `fac(4)`:

```
n          n > 0          result
```

CAS CS 111
Boston University, Fall 2023
### Factorial Three Ways!

<table>
<thead>
<tr>
<th>Method</th>
<th>Code</th>
</tr>
</thead>
</table>
| Recursion      | ```python
def fac(n):
    if n == 0:
        return 1
    else:
        rest = fac(n-1)
        return n * rest
``` |
| For Loop       | ```python
def fac(n):
    result = 1
    for x in range(1, n+1):
        result *= x
    return result
``` |
| While Loop     | ```python
def fac(n):
    result = 1
    while n > 0:
        result *= n
        n = n - 1
    return result
``` |

### Extreme Looping!

- What does this code do?
  ```python
  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:
    ```python
    import random
    ```
  • `random.choice(vals)`
    • takes a sequence `vals`
    • randomly chooses one value from `vals` and returns it
  • examples from the Shell:
    ```python
    >>> 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

```python
import random
while True:
    print('Help!')
    if random.choice(range(10000)) == 111:
        break
    print('Let me out!')
print('At last!')
```

• Thus, the final two lines that are printed are:
  • a break statement causes a loop to end early.
  • jumps to the line that comes after the loop.
Counting the Number of Repetitions

```python
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:
  ```python
  variable = input(prompt)  
  ```
  where `prompt` is a string

- Getting an *integer value*:
  ```python
  variable = int(input(prompt))
  ```

- Getting a *floating-point value*:
  ```python
  variable = float(input(prompt))
  ```

- Getting an arbitrary non-string value (e.g., a list):
  ```python
  variable = eval(input(prompt))
  ```
  - eval treats a string as an expression to be evaluated

- Examples:
  ```python
  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

```python
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)
```

<table>
<thead>
<tr>
<th>a &gt; 2</th>
<th>a</th>
<th>prints</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>40</td>
<td>a = 20</td>
</tr>
<tr>
<td>True</td>
<td>20</td>
<td>a = 10</td>
</tr>
<tr>
<td>True</td>
<td>10</td>
<td>a = 5</td>
</tr>
<tr>
<td>True</td>
<td>5</td>
<td>a = 2</td>
</tr>
<tr>
<td>True</td>
<td>2</td>
<td>a = 1</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For what inputs does this function return True?

```python
def mystery(n):
    while n != 1:
        if n % 2 != 0:
            return False
        n = n // 2
    return True
```

Try tracing these two cases:

<table>
<thead>
<tr>
<th>mystery(12)</th>
<th>mystery(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>
Program Design with Loops

Computer Science 111
Boston University

Recall: Two Types of Loops

for

\textbf{definite loop}

For a \textbf{known} number of repetitions

while

\textbf{indefinite loop}

For an \textbf{unknown} number of repetitions
Recall: Two Types of for Loops

\[ \text{vals} = [3, 15, 17, 7] \]

- **element-based loop**
  ```python
  def sum(vals):
      result = 0
      for x in vals:
          result += x
      return result
  ```

- **index-based loop**
  ```python
  vals = [3, 15, 17, 7]
  def sum(vals):
      result = 0
      for i in range(len(vals)):
          result += vals[i]
      return result
  ```

Finding the Smallest Value in a List

- What if we needed to write a loop-based version of `min()`?

\[ \text{vals} = [45, 80, 10, 30, 27, 50, 5, 15] \]

- What strategy should we use?

- What type of loop: for or while?
How should we fill in the blank to initialize m?

```python
def minval(vals):
    m = ∙∙∙
    for x in vals:
        if x < m:
            m = x
    return m
```

vals = [45, 80, 10, 30, 27, 50, 5, 15]

m is the "min so far"

Finding the Position of the Smallest Value

```python
def minval_posn(vals):
    # initialize variable(s)
    for ____________:
        if __________:
            # update var(s)
        return ______
```

```latex
\begin{center}
\begin{tabular}{cccccccc}
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>[45, 80, 10, 30, 27, 50, 5, 15]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{tabular}
\end{center}
```

m = 45 → m = 10 → m = 5

6 should be returned
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?

```python
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

```python
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?

```python
def any_below(vals, cutoff):
    for ____ in ____________:
        if ________________:
            ```
A. 
```python
def any_below(vals, cutoff):
    for x in vals:
        if x >= cutoff:
            return False
    return True
```

B. 
```python
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return True
    return False
```

C. 
```python
def any_below(vals, cutoff):
    for x in vals:
        if x < cutoff:
            return False
    return True
```

D. more than one of them

Which of these works?

Estimating $\pi$ by dart throwing!

$$\pi = \frac{4 \cdot \text{area}}{\text{area}}$$

$\pi \sim 4 \cdot \frac{\text{"hits"}}{\text{"throws"}}$
Loops: for or while?

pi_one(e) \quad e == how close to \pi we need to get

pi_two(n) \quad n == number of darts to throw

Which function will use which kind of loop?

Thinking in Loops

for

definite iteration

For a known number of repetitions

while

indefinite iteration

For an unknown number of repetitions
Pre-Lecture
Nested Loops

Repeating a Repetition!
```python
for i in range(3):  # 0, 1, 2
    for j in range(4):  # 0, 1, 2, 3
        print(i, j)
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Repeating a Repetition!

```python
for i in range(3):
    for j in range(4):
        print(i, j)
    print('---')
```

Outer loop: `i`  
Inner loop: `j`

```
for r in range(h):
    for c in range(w):
        # process the pixel at (r, c)
```

Nested loops and 2D structure
Tracing a Nested for Loop

```python
for i in range(5):      # [0,1,2,3,4]
    for j in range(i):
        print(i, j)
```

<table>
<thead>
<tr>
<th>i</th>
<th>range(i)</th>
<th>j</th>
<th>value printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Nested Loops

Computer Science 111
Boston University

Nested Loops!

```python
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?

```python
for i in range(5):
    for j in range(7):
        print(i, j)
```

Recall: Tracing a Nested for Loop

```python
for i in range(5):  # [0,1,2,3,4]
    for j in range(i):
        print(i, j)
```

<table>
<thead>
<tr>
<th>i</th>
<th>range(i)</th>
<th>j</th>
<th>value printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[]</td>
<td>none</td>
<td>nothing (we exit the inner loop)</td>
</tr>
<tr>
<td>1</td>
<td>[0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[0,1]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>[0,1,2]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>[0,1,2,3]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

full output:

```
1 0
2 0
2 1
3 0
3 1
3 2
4 0
4 1
4 2
4 3
```

CAS CS 111

Boston University, Fall 2023
Second Example: Tracing a Nested for Loop

```python
for i in range(4):
    for j in range(i, 3):
        print(i, j)
    print(j)
```

<table>
<thead>
<tr>
<th>i</th>
<th>range(i, 3)</th>
<th>j</th>
<th>value printed</th>
</tr>
</thead>
</table>

Using Loops: T.T. Securities (TTS)

Analyzes a sequence of stock prices

```
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()

...
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 not 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]

<table>
<thead>
<tr>
<th>Day</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>45.00</td>
</tr>
<tr>
<td>1</td>
<td>80.00</td>
</tr>
<tr>
<td>2</td>
<td>10.00</td>
</tr>
<tr>
<td>3</td>
<td>30.00</td>
</tr>
<tr>
<td>4</td>
<td>27.00</td>
</tr>
<tr>
<td>5</td>
<td>50.00</td>
</tr>
<tr>
<td>6</td>
<td>5.00</td>
</tr>
<tr>
<td>7</td>
<td>15.00</td>
</tr>
</tbody>
</table>

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.

```
>>> 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:

```python
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?

```python
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
```

```python
>>> diff_indices([12,3,7],[6,0,5])
[2, 0]
```

Index of value in l1
Index of value in l2

---

**CAS CS 111**

Boston University, Fall 2023

219
What if we want the *indices* of the min-diff values?

```python
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

```python
for row in range(3):
    for col in range(4):
        print('#', end=' ')  # go to next line
print()  # go to next line
```

```
row 0 1 2 3
  0  # # # # #
  1  # # # # #
  2  # # # # #
```
Fill in the Blank #1

```python
for row in range(3):
    for col in range(6):
        print(______, end=' ')
    print()  # go to next line
```

Fill in the Blank #2

```python
for row in range(3):
    for col in range(6):
        print(______, end=' ')
    print()  # go to next line
```
What is needed in the blanks to get this pattern?

```python
for row in range(5):
    for col in range(5):
        print(_____, end=' ')  # 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?

```python
for row in range(3):
    for col in range(6):
        print(_____, end=' ')  # 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.asciiimation.co.nz/
Recall: Variables as Boxes

- You can picture a variable as a named "box" in memory.
- Example from an earlier lecture:

  ```plaintext
  num1 = 100
  num2 = 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.
  \[ x = 7 \]

  Memory
  \[ x \rightarrow 7 \]

• However, we can use assignment to change the variable—making it refer to a different value:
  \[ x = 4 \]

  Memory
  \[ x \rightarrow 4 \]

Changing a Value vs. Changing a Variable

• Here's our original list:
  \[ \text{prices} \]

  \[ \begin{array}{cccc}
  25 & 10 & 30 & 45 \\
  \end{array} \]

• Lists are mutable, so we can change the value (the list) by modifying its elements:
  \[ \text{prices}[1] = 50 \]

  \[ \begin{array}{cccc}
  25 & 50 & 30 & 45 \\
  \end{array} \]
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]
```

```
prices
```

- The same thing holds for list elements that are immutable.

```
prices = [25, 10, 30, 45]
```

```
prices
```

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
```

- 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:

```python
1 x = 7
⇒ 2 prices = [25, 30, 30, 45]
```

Copying Variables

• The assignment

```python
var2 = var1
```

copies the contents of `var1` into `var2`:

```python
x = 50
y = x
y = 50
```
Copying References

• Consider this example:
  ```python
  list1 = [7, 8, 9, 6, 10, 7, 9, 5]
  list2 = list1
  ```

• Given the lines of code above, what will the lines below print?
  ```python
  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:
  ```python
  list1 = [7, 8, 9, 6, 10, 7, 9, 5]
  list2 = list1[:]
  ```

• What will this print now?
  ```python
  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:

```python
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.)

```
def main():
a = [1, 2, 3]
triple(a)
print(a)      # prints [3, 6, 9]
```
References and Mutable Data

Recall: References

- Because we don't care about the actual memory address, we use an arrow to represent a reference:
Recall: Lists and References

\[ \text{prices} = [25, 10, 30, 45] \]

Mutable vs. Immutable Data

- In Python, strings and numbers are \textit{immutable}.
  - their contents/components cannot be changed
- Lists are \textit{mutable}.
  - their contents/components \textit{can} be changed
Changing a Value vs. Changing a Variable

• There's no way to change an immutable value like 'hello'.

\[
\text{s = 'hello'}
\]

\[
\begin{array}{c}
\text{s} \\
\rightarrow 'hello'
\end{array}
\]

• However, we can change the variable:

\[
\text{s = 'goodbye'}
\]

\[
\begin{array}{c}
\text{s} \\
\rightarrow 'goodbye'
\end{array}
\]

Changing a Value vs. Changing a Variable

• Lists are mutable, so we can change the value (the list) by modifying its elements:

\[
\text{prices[1] = 50}
\]

\[
\begin{array}{c}
\text{prices} \\
\rightarrow 25 \quad 10 \quad 30 \quad 45
\end{array}
\]

\[
\begin{array}{c}
\text{prices} \\
\rightarrow 25 \quad \textbf{50} \quad 30 \quad 45
\end{array}
\]
Recall: Simplifying Our Mental Model

• When a variable represents an immutable value, it's okay to picture the value as being *inside* the variable.
  \[ x = 7 \]
  \[ \begin{array}{c|c}
  \text{x} & 7 \\
  \end{array} \]

  • a simplified picture, but good enough!

• The same thing holds for list elements that are immutable.
  \[ \text{prices} = [25, 10, 30, 45] \]
  \[ \begin{array}{c|c|c|c|c}
  \text{prices} & 25 & 10 & 30 & 45 \\
  \end{array} \]

  • We still need to use references for *mutable* data like lists.

Recall: Copying References

• Consider this example:
  \[ \text{list1} = [7, 8, 9, 6, 10, 7, 9, 5] \]
  \[ \text{list2} = \text{list1} \]

  \[ \begin{array}{c|c|c|c|c|c|c|c}
  \text{list1} & 7 & 8 & 9 & 6 & 10 & 7 & 9 & 5 \\
  \text{list2} & 7 & 8 & 9 & 6 & 10 & 7 & 9 & 5 \\
  \end{array} \]
Recall: Copying a List

- We can copy a list like this one using a full slice:

```python
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```

```
list1 ____________ [7 8 9 6 10 7 9 5]
list2 ____________ [7 8 9 6 10 7 9 5]
```

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?

```python
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```

```
list1 ____________ [1 2 3]
list2 ____________ [1 8 9 6 10 7 9 5]
list3 ____________ [1 8 9 6 10 7 9 5]
```
Another Way to Picture References

```python
list1 = [1, 2, 3]
list2 = list1[:]
list3 = list2
list2[1] = 7
print(list1, list2, list3)
```

```
list1 128
      1 2 3
list2 312
      1 2 3
list3 312
```

Changing the Internals vs. Changing a Variable

- When two variables hold a reference to the same list...
  ```python
  list1 = [7, 8, 9]
  list2 = list1
  list1[2] = 4
  print(list1)    # prints [7, 8, 4]
  list2
  list1
  list2
  ```

- ...if we change the internals of the list, both variables will "see" the change:
  ```python
  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...
  
  ```python
  list1 = [7, 8, 9]
  list2 = list1
  list1 |---| 7 8 9
  list2
  
  ...if we change one of the variables itself, that does not change the other variable:
  
  ```python
  list2 = [4, 5, 6]
  print(list1)    # prints [7, 8, 9]
  
  list1 |---| 7 8 9
  list2 |---| 4 5 6
  ```

Passing a List to a Function, version 1

```python
def mystery1(vals):
    vals[1] = 4  # changes the internals of the list

a = [1, 2, 3]
mystery1(a)
print(a)
```

<table>
<thead>
<tr>
<th>before mystery1</th>
<th>during mystery1</th>
<th>after mystery1</th>
</tr>
</thead>
<tbody>
<tr>
<td>global a</td>
<td>global a</td>
<td>global a</td>
</tr>
<tr>
<td></td>
<td>mystery1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vals[1]</td>
<td></td>
</tr>
<tr>
<td>[1, 2, 3]</td>
<td>[1, 2, 3]</td>
<td>[1, 2, 3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1, 2, 3]</td>
</tr>
</tbody>
</table>
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)

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]
mystery1(x)
mystery2(vals)
print(x, vals)
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]

def mystery1(x):
    x *= 2
    return x
mystery2(vals)
x = 7
vals = [7, 7]
x = mystery1(x)  # assign the return value!
mystery2(vals)
print(x, vals)
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 number of rows in `table`.
- `table[r]` is the row with index `r`.
- `len(table[r])` is the number of elements in row `r`.
- `len(table[0])` is the number 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:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>8</td>
<td>3</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>11</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>3</td>
<td>5</td>
<td>18</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>14</td>
<td>13</td>
<td>6</td>
<td>13</td>
<td>12</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>16</td>
<td>20</td>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>
```

- Column indices
- Row indices
### Accessing an Element of a 2-D List

```python
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:**

```python
>>> print(table[2][1])
3
```

```python
>>> table[-1][-2] = 0
```

### Using Nested Loops to Process a 2-D List

```python
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]]
```

```python
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

```python
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)
```

<table>
<thead>
<tr>
<th>r</th>
<th>c</th>
<th>table[r][c]</th>
<th>count</th>
</tr>
</thead>
</table>

Boston University, Fall 2023
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!

```python
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?

```python
table = [[15,  19,  3, 16],
[ 6, 21,  9,  4],
[17,  3,  5, 18]]
```

```python
count = 0
for r in range(len(table)):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1

count = 0
for r in range(len(table[0])):
    for c in range(len(table)):
        if table[r][c] % 2 == 0:
            count += 1

count = 0
for r in range(len(table[0])):
    for c in range(len(table[0])):
        if table[r][c] % 2 == 0:
            count += 1
```

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 range(len(table[0])):
       for c in range(len(table[0])):
           if table[r][c] % 2 == 0:
               count += 1

C. count = 0
   for r in range(len(table[0])):
       for c in range(len(table[0])):
           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

```python
# 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)
```

<table>
<thead>
<tr>
<th>r</th>
<th>c</th>
<th>table[r][c]</th>
<th>count</th>
</tr>
</thead>
</table>

Recall: Picturing a 2-D List

```python
# 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:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>8</td>
<td>3</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>11</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>3</td>
<td>5</td>
<td>18</td>
<td>10</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>14</td>
<td>13</td>
<td>6</td>
<td>13</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>16</td>
<td>20</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

CAS CS 111
Boston University, Fall 2023
Picturing a 2-D List (cont)

- Here's a more accurate picture:

```
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
```

Recall: Copying a List

- We can't copy a list by a simple assignment:

```python
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1
```

```
7 8 9 6 10 7 9 5
```

- We can copy this list using a full slice:

```python
list1 = [7, 8, 9, 6, 10, 7, 9, 5]
list2 = list1[:]
```

```
7 8 9 6 10 7 9 5
```
Changing the Internals vs. Changing a Variable

• When two variables hold a reference to the same list...

```
list1 = [7, 8, 9]
list2 = list1
```

<table>
<thead>
<tr>
<th>list1</th>
<th>list2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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]
```

<table>
<thead>
<tr>
<th>list1</th>
<th>list2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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
```

<table>
<thead>
<tr>
<th>list1</th>
<th>list2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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]
```

<table>
<thead>
<tr>
<th>list1</th>
<th>list2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

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)

```python
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)
```

### before mystery5  
<table>
<thead>
<tr>
<th>Global</th>
<th>vals</th>
<th>x</th>
<th>7</th>
</tr>
</thead>
</table>

### during mystery5  
<table>
<thead>
<tr>
<th>Global</th>
<th>vals</th>
<th>x</th>
<th>7</th>
</tr>
</thead>
</table>

### after mystery5  
<table>
<thead>
<tr>
<th>Global</th>
<th>vals</th>
<th>x</th>
<th>7</th>
</tr>
</thead>
</table>

# What is the output of this program? (part II)

```python
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)
```

### before mystery6  
<table>
<thead>
<tr>
<th>Global</th>
<th>vals</th>
<th>x</th>
<th>7</th>
</tr>
</thead>
</table>

### during mystery6  
<table>
<thead>
<tr>
<th>Global</th>
<th>vals</th>
<th>x</th>
<th>7</th>
</tr>
</thead>
</table>

### after mystery6  
<table>
<thead>
<tr>
<th>Global</th>
<th>vals</th>
<th>x</th>
<th></th>
</tr>
</thead>
</table>
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

<table>
<thead>
<tr>
<th>string object for 'hello'</th>
<th>string object for 'bye'</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents</td>
<td>'h' 'e' 'l' 'l' 'o'</td>
</tr>
<tr>
<td>length</td>
<td>5</td>
</tr>
<tr>
<td>methods</td>
<td>upper() replace() lower() split() find() count()</td>
</tr>
<tr>
<td>contents</td>
<td>'b' 'y' 'e'</td>
</tr>
<tr>
<td>length</td>
<td>3</td>
</tr>
<tr>
<td>methods</td>
<td>upper() replace() lower() split() find() count()</td>
</tr>
</tbody>
</table>

Calling a Method

• An object's methods are inside the object, so we use *dot notation* to call them.

• Example:
  ```python
greeting = 'Hello
allcaps = greeting.upper()
```

• 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
  ```python
  >>> 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.
  ```python
  >>> 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:
  ```python
  >>> date = '11/10/2014'
  >>> date.split('/')
  ['11', '10', '2014']
  ```
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!\nTest your code fully!'
Opening a Text File

• Before we can read from a text file, we need to open a connection to the file.

• Example:
  
  \[
  f = \text{open('reminder.txt', 'r')}
  \]

  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.

  >>> 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!

```python
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
Processing a CSV File

courses.txt

<table>
<thead>
<tr>
<th>line</th>
<th>fields</th>
<th>output</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>'CS,111,MWF 10-11\n'</td>
<td>['CS', '111', 'MWF 10-11']</td>
<td>CS 111</td>
<td>1</td>
</tr>
<tr>
<td>'MA,123,TR 3-5\n'</td>
<td>['MA', '123', 'TR 3-5']</td>
<td>none</td>
<td>1</td>
</tr>
</tbody>
</table>

Complete the rest of the table!
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:

```python
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:
  \[
  \text{[red, green, blue]}
  \]
- 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

```python
>>> 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.

```python
>>> 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:

```python
>>> 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?

```python
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!
  ```python
  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?

```python
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
```

<table>
<thead>
<tr>
<th>courses.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS,111,MWF 10-11</td>
</tr>
<tr>
<td>MA,123,TR 3-5</td>
</tr>
<tr>
<td>CS,105,MWF 1-2</td>
</tr>
<tr>
<td>EC,100,MWF 2-3</td>
</tr>
</tbody>
</table>
...
Recall:Processing a CSV File

```python
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
```

<table>
<thead>
<tr>
<th>line</th>
<th>fields</th>
<th>output</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>'CS,111,MWF 10-11\n'</td>
<td>['CS','111','MWF 10-11']</td>
<td>CS 111</td>
<td>1</td>
</tr>
<tr>
<td>'CS,111,MWF 10-11'</td>
<td>['CS','111','MWF 10-11']</td>
<td>CS 111</td>
<td>1</td>
</tr>
<tr>
<td>'MA,123,TR 3-5\n'</td>
<td>['MA','123','TR 3-5']</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>'MA,123,TR 3-5'</td>
<td>['MA','123','TR 3-5']</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>none</td>
<td>1</td>
</tr>
</tbody>
</table>

(see the pre-lecture video for more!)

Closing a File

- When you're done with a file, close your connection to it:
  ```python
  file.close()  # file is the file handle
  ```
- another close() 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.
  ```python
  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.

```python
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()```

```python
>>> extract_results('track_results.txt', 'BU')
Mike Mercury mile 4:50:00
Len Lightning half-mile 2:15:00
```
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?

```python
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?

```python
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()```
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
- ...

<table>
<thead>
<tr>
<th>string object for 'hello'</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents: h e l l o</td>
</tr>
<tr>
<td>length: 5</td>
</tr>
<tr>
<td>upper()</td>
</tr>
<tr>
<td>lower()</td>
</tr>
<tr>
<td>find()</td>
</tr>
</tbody>
</table>
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:

```python
>>> 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.

```
coordinates of its upper-left corner
```

• 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!

r1
r2

the variable
holds a reference to
the object!
Accessing and Modifying an Object's Attributes

```python
>>> 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

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

coordinates of
its upper-left
corner
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

```python
# 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:

```python
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

```python
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
150
```
Another Example of a Method

• Here's a method for getting the area of a Rectangle:
  ```python
def area(self):
    return self.width * self.height
  ```

• Sample method calls:
  ```python
  >>> 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

```python
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

r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)

complete the diagram

stack frames

<table>
<thead>
<tr>
<th>grow</th>
<th>self</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

objects

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
</tr>
<tr>
<td>width</td>
<td>100</td>
</tr>
<tr>
<td>height</td>
<td>50</td>
</tr>
</tbody>
</table>

continued on next slide
Methods That Modify an Object (cont.)

```python
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

output: _________________

[Diagram showing stack frames and objects with variables x, y, width, height, and the global variable r1.]
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:

```python
# 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

```python
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

```python
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!
```

```
```

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0</td>
<td>x 0</td>
</tr>
<tr>
<td>y 0</td>
<td>y 0</td>
</tr>
<tr>
<td>width 100</td>
<td>width 75</td>
</tr>
<tr>
<td>height 50</td>
<td>height 350</td>
</tr>
</tbody>
</table>

the variable holds a reference to the object!
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)

CAS CS 111

Boston University, Fall 2023

284


class Rectangle:
    ... 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.  
```python
def perimeter(self, width, height):
    return 2*width + 2*height
```

B.  
```python
def perimeter():
    return 2*self.width + 2*self.height
```

C.  
```python
def perimeter(self):
    return 2*self.width + 2*self.height
```

D.  none of the above

Fill in the blank to call the perimeter method.

```python
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

```python
# 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

```python
# 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)
```
# Rectangle client code

```python
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)
r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```

```
height 240
width 60
y0
x0
height 250
width 500
y0
x0
```

## Memory Diagrams for Method Calls, part VI

```
output: 500 250 60 240
```

---

# Rectangle client code

```python
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)
r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```

```
height 240
width 60
y0
x0
height 250
width 500
y0
x0
```

## Memory Diagrams for Method Calls, part VII

```
output: 500 250 60 240
```

---

CAS CS 111

Boston University, Fall 2023
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

  stack frames

  objects

  `scale`
  
  `self`
  `factor 3`

  `global`
  `r1`
  `r2`

  `x 0`
  `y 0`
  `width 60`
  `height 240`

- `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
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?

```python
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

```python
class Rectangle:
    ...
    r1     r2
    def __eq__(self, other):
        if self.width == other.width and 
            self.height == other.height:
            return True
        else:
            return False
```

```python
>>> 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:
    ```python
    >> r1 = Rectangle(100, 80)
    >> r1                    # calls __repr__
    ```
  - apply str():
    ```python
    >> r1string = str(r1)    # also calls __repr__
    ```
  - print an object:
    ```python
    >> 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!

```python
>>> r2 = Rectangle(50, 20)
>>> r2
<__main__.Rectangle object at 0x03247C30>
```

__repr__ Method for Our Rectangle Class

```python
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:

```python
>>> r2 = Rectangle(50, 20)
>>> print(r2)
50 x 20
>>> r2
```

_____________
More Object-Oriented Programming

Computer Science 111
Boston University

Recall: Our Rectangle Class

```python
# 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
```

```
x 10
y 20
width 100
height 50
```
What is the output of this program?

```python
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?

```python
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

```python
class Rectangle:
...
    def __eq__(self, other):
        if self.width == other.width and 
           self.height == other.height:
            return True
        else:
            return False
```

```plaintext
>>> r1 = Rectangle(40, 75)
>>> r2 = Rectangle(40, 75)

>>> print(r1 == r2)
```

Recall: `__repr__` Method for Our `Rectangle` Class

```python
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:
  ```plaintext
  >>> r2 = Rectangle(50, 20)
  >>> print(r2)
  50 x 20
  >>> r2
  50 x 20
  >>> str(r2)
  '50 x 20'
  ```
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)

```python
def move_right(self, amount):
```

- 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

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:

```python
>>> 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)

A. d1.is_before(d2)  
B. d2.is_before(d1)  
C. d3.is_before(d1)  
D. more than one  

Extra: Can you think of any other cases that it would get wrong involving these dates?
Recall: Extracting Relevant Data from a File

```python
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:

```python
>>> 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.
  ```python
  >>> 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!
  ```python
  >>> 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

```python
>>> counts = {}          # create an empty dictionary
>>> counts['BU'] = 2
>>> 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              # order is not fixed
{'BU': 2, 'UMass': 1, 'BC': 1}
```

Other Dictionary Operations

```python
>>> 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

```python
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:

```python
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

```python
counts = {'BU': 2, 'UMass': 1, 'BC': 1}
for key in counts:  # get one key at a time
    print(key, counts[key])
```

<table>
<thead>
<tr>
<th>key</th>
<th>counts[key]</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BU'</td>
<td>counts['BU']</td>
<td>BU 2</td>
</tr>
</tbody>
</table>

Fill in the rest of the table!
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

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).')
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

• In his plays, Shakespeare used 31,534 distinct words!

• He also coined a number of words:
  gust        besmirch        unreal
  swagger     watchdog        superscript
Generate Text Based on Shakespeare!

```python
>>> 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.
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.

>>> 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!

sentence-start symbol

{ '$': ['Boston', 'It', 'It', 'It'], 'Boston': ['University'], 'University': ['is'], 'is': ['a', 'committed', 'committed', 'amazing!'], 'to': ____________________________, 'committed': ____________________________, ... }

• 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
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 '$'

<table>
<thead>
<tr>
<th>current_word</th>
<th>next_word</th>
<th>action taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>'$'</td>
<td>'Boston'</td>
<td>d['$'] = ['Boston']</td>
</tr>
<tr>
<td>'Boston'</td>
<td>'University'</td>
<td>d['Boston'] = ['University']</td>
</tr>
<tr>
<td>'is'</td>
<td>'a'</td>
<td>d['is'] = ['a']</td>
</tr>
<tr>
<td>'a'</td>
<td>'comprehensive'</td>
<td>d['a'] = ['comprehensive']</td>
</tr>
<tr>
<td>'comprehensive'</td>
<td>'university.'</td>
<td>d['comprehensive'] = ['university.']</td>
</tr>
<tr>
<td>'$'</td>
<td>'It'</td>
<td>d['$'] = ['Boston', 'It']</td>
</tr>
<tr>
<td>'It'</td>
<td>'is'</td>
<td>d['It'] = ['is']</td>
</tr>
</tbody>
</table>

---

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?

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

- 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

![Diagram of Connect Four board and examples of winning configurations]
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:
    ```python
    b = Board(5, 6)
    ```
  - 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**

```python
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 = _______________________________
```

```python
Board object
height 5
width 6
slots
```

```python
|   | |   | |   | |   |
|---|---|---|---|---|---|
|   | |   | |   | |   |
|   | |   | |   | |   |
|   | |   | |   | |   |
|   | |   | |   | |   |
|   | |   | |   | | X |
```

|   | |   | |   | |   |
|---|---|---|---|---|---|
|   | |   | |   | |   |
|   | |   | |   | |   |
|   | |   | |   | |   |
|   | |   | |   | |   |
|   | |   | |   | | X |
```
Correct 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

__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 += '
'  
    # add the row of hyphens to s
    # add the column indices to s
    return s
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)

| | | | | | |
| | | | | | |
| | | | | | |
| |O| |O| | |
| |X|O|X|X| |
| |O|O|O|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
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
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):
        self.month = new_month
        self.day = new_day
        self.year = new_year
    
    def __repr__(self):
        s = "%02d/%02d/%04d" % (self.month, self.day, self.year)
        return s

    def is_leap_year(self):
        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:

  ```python
  >>> 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:

  ```python
  >>> print(tg)
  Thanksgiving (11/28/2019)
  ```

Let `Holiday` Inherit From `Date`!

```python
class Holiday(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

```python
class Holiday(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
```

```python
g>>> 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

```python
class Holiday(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):
        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):
    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 unintelligent 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!
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 'X'
-1  50  50  50  50  50  50

0 moves are made!
```

0 moves are made!
A Lookahead of 1

- A lookahead-1 player assesses the outcome of only the considered move.

LA-1 scores for (X)

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>50</th>
<th>100</th>
<th>50</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>50</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

A lookahead-1 player will "see" an impending victory. `next_move` will return 4!

How do these scores change if it is \( \bigcirc \)'s turn instead of \( \bigcirc \)'s?

A Lookahead of 1

- A lookahead-1 player assesses the outcome of only the considered move.

LA-1 scores for (X)

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>50</th>
<th>100</th>
<th>50</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>50</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

1 move is made!
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

### Example 2: LA-0

- A lookahead-0 player only assesses the current board (0 moves!).
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 | -1 |

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 |

| 50 | 50 | 50 | 50 | 50 | -1 |

Note: the board and scores are placeholders for the actual game state and scoring.
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-3 scores for •
What would change?

-1
```

```
Example 2: LA-0

- A lookahead-0 player only assesses the current board (0 moves!).

```
LA-0 scores for •

-1
```

same board, different player, same LA-0 scores
Example 2: LA-1

- A lookahead-1 player assesses the outcome of only the considered move.

```
| 50 | 50 | 50 | 50 | 50 | -1 |
```

What would change?

What Are the LA-2 Scores for ⬆️?

- Look 2 moves ahead. Assume the opponent looks 1 move ahead.

A. ```
| 50 | 50 | 100 | 50 | 50 | -1 |
```

← no change?

B. ```
| 0  | 0  | 100 | 0  | 0  | -1 |
```

C. ```
| 50 | 50 | 100 | 50 | 0  | -1 |
```

← LA-1 scores
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-3 scores for  

• 'X'
• 'O'

to move

LA-4!

- A lookahead-4 player looks 4 moves ahead.
- assumes the opponent looks ahead 4 – 1 = 3 moves

LA-4 scores for  

• 'X'
• 'O'

to move
LA-4!

- A lookahead-4 player looks 4 moves ahead.
  - assumes the opponent looks ahead $4 - 1 = 3$ moves

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.

What about this?

#### LA-4 scores for 'O'

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

#### LA-0 scores for 'O'

Looks 0 moves into the future

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LA-1 scores for 'O'

Looks 1 move into the future

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LA-2 scores for 'O'

Looks 2 moves into the future

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LA-3 scores for 'O'

Looks 3 moves into the future

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
<th>col 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
scores_for – the AI in AIPlayer!

```python
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
```

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 make 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[0] = 100
A loss for my opponent is a win for me!

Opp scores = [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...

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 make its best move, and determine your score accordingly
4) remove checker!

\[
\text{scores} = [100, 0, 100, 100, 100, 50, 0]
\]

possible next move

We've tried all columns!

\[
\text{scores} = [100, 0, 100, 100, 100, 50, 0]
\]

possible next move
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...

We’ve tried
all columns!

return [100, 0, 100, 100, 100, 50, 0]
Breaking Ties

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 unintelligent 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():
  ```python
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!

```python
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:
  ```python
def process_move(p, b):
    ...
    col = p.next_move(b)
    ...
  ```

- In theory, we can treat `next_move` as if it were a function:
  ```python
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

start state
(with “funnel”)

transitions
(arrows labeled with input)

accepting state
(double circle)
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?

Start state
(with "funnel")

Accepting state
(double circle)

Transitions
(arrows labeled with input)

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: 101001100, 0011101, 1110, ...

start  end w/1  end w/11  fail!
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

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:

\[
\begin{align*}
s & \rightarrow s_0 \\
s_0 & \rightarrow 0 \rightarrow s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
s_1 & \rightarrow 0 \rightarrow s_0 \\
s_0 & \rightarrow 1 \rightarrow s_1 \\
\end{align*}
\]

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?

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

Because we only care about the last 3 bits, 8 states is enough!
More FSM Practice!

- Construct a FSM accepting bit strings in which:
  - the **first** bit is 0
  - the **last** 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 **first** bit is 0
  - the **last** bit is 1

A.  
B.  
C.  
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?
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 **some** number of 0s
  - followed by the **same** number of 1s
  - 01, 0011, 000111, 00001111, etc.

- **What are the classes of equivalent inputs?**
  - an infinite number of them!
  - $n$ 0s, followed by $(n+1)$ or more 1s, and/or by an alternation between groups of 1s and 0s – **rejected; can't recover!**
  - $n$ 0s, followed by $n$ 1s – **accepted!** (and any further input is bad!)
  - $n$ 0s, followed by $(n-1)$ 1s – need one more 1 to accept
  - $n$ 0s, followed by $(n-2)$ 1s – need two more 1s to accept
  - $n$ 0s, followed by $(n-3)$ 1s – need three more 1s to accept...

- **Impossible to solve using a finite state machine!**

Limitations of FSMs

- Because they're finite, FSMs can only count finitely high!

<table>
<thead>
<tr>
<th>Computable with FSMs</th>
<th>Uncomputable with FSMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>even/odd sums or differences</td>
<td>equal numbers of two values</td>
</tr>
<tr>
<td>multiples of other integers</td>
<td>two more 1s than 0s or vice versa</td>
</tr>
<tr>
<td>finite input constraints:</td>
<td>infinite input constraints:</td>
</tr>
<tr>
<td>third digit is a 1</td>
<td>palindromes</td>
</tr>
<tr>
<td>third-to-last digit is a 1</td>
<td>anything modeled by a potentially</td>
</tr>
<tr>
<td>third digit == third-to-last digit</td>
<td>unbounded while loop</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>
A Better Machine!

Turing Machine (TM)
Alan Turing (1912-1954)

Enigma machine — The axis’s encryption engine

Bletchley Park

1912-1954

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

1946

Alan Turing (1912-1954)

Alan Mathison Turing 1912-1954
Father of Computer Science, Mathematician, Logician, Wartime Codebreaker, Victim of Prejudice

"Mathematics, rightly viewed, possesses not only truth, but supreme beauty, a heavy cold and austere, like that of sculpture.” — Bertrand Russell

1946

computing community. Wikipedia
So far, all known computational devices can compute only what Turing Machines can...

- Quantum computation
- Molecular computation
  http://www.arstechnica.com/reviews/0q00/dna/dna-1.html
- Parallel computers
- Integrated circuits
- Electromechanical computation
- Water-based computation
- Tinkertoy computation

For info about how they work, take CS 332!
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

```python
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:

\[
\begin{array}{c}
1\text{st} \quad 35 \quad 6 \quad 19 \quad 23 \quad 3 \quad 47 \quad 9 \quad 15 \\
\end{array}
\]

\[
\begin{array}{c}
1\text{st} \quad 3 \quad 6 \quad 9 \quad 15 \quad 19 \quad 23 \quad 35 \quad 47 \\
\end{array}
\]

• 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:

\[
\begin{array}{c}
0 \quad 1 \quad 2 \quad 3 \quad 4 \\
15 \quad 6 \quad 2 \quad 12 \quad 4 \\
\end{array}
\]

\[
\begin{array}{c}
0 \quad 1 \quad 2 \quad 3 \quad 4 \\
2 \quad 6 \quad 15 \quad 12 \quad 4 \\
\end{array}
\]

\[
\begin{array}{c}
0 \quad 1 \quad 2 \quad 3 \quad 4 \\
2 \quad 4 \quad 15 \quad 12 \quad 6 \\
\end{array}
\]

\[
\begin{array}{c}
0 \quad 1 \quad 2 \quad 3 \quad 4 \\
2 \quad 4 \quad 6 \quad 12 \quad 15 \\
\end{array}
\]

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 first three positions?

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
  - **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 \( \rightarrow \) the running time will be \( \sim 4x \) longer
  • make the list 3x longer \( \rightarrow \) the running time will be \( \sim 9x \) longer
  • make the list 4x longer \( \rightarrow \) ???

• 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:
  - $\log_2 n$ # binary search
  - $n$ # sequential/linear search
  - $n \log_2 n$ # quicksort
  - $n^2$ # selection sort

- Others include:
  - $c^n$ # 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 $1 \mu$sec ($1 \times 10^{-6}$ sec)

<table>
<thead>
<tr>
<th>growth function</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>.00001 s</td>
<td>.00002 s</td>
<td>.00003 s</td>
<td>.00004 s</td>
<td>.00005 s</td>
<td>.00006 s</td>
</tr>
<tr>
<td>$n^2$</td>
<td>.0001 s</td>
<td>.0004 s</td>
<td>.0009 s</td>
<td>.0016 s</td>
<td>.0025 s</td>
<td>.0036 s</td>
</tr>
<tr>
<td>$n^3$</td>
<td>.1 s</td>
<td>3.2 s</td>
<td>24.3 s</td>
<td>1.7 min</td>
<td>5.2 min</td>
<td>13.0 min</td>
</tr>
<tr>
<td>$2^n$</td>
<td>.001 s</td>
<td>1.0 s</td>
<td>17.9 min</td>
<td>12.7 days</td>
<td>35.7 yrs</td>
<td>36,600 yrs</td>
</tr>
</tbody>
</table>
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 $4^n$
  - example: 30 points $\rightarrow 4^{30}$ possible labelings
    - if it took 1 $\mu$sec 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