CS 1124 Media Computation

Lecture 10.2 October 29, 2008

Steve Harrison

Virtual Machines

- interpreted, compiled and virtual machines

Why do we write programs?

- One reason we write programs is to be able to do the same thing over-and-over again, without having to rehash the same steps in Photoshop each time.

Which one leads to shorter time overall?

- Interpreted version:
  - **100 times**
    - doGraphics(["b 100 200","b 101 200","b 102 200","l 102 200 102 300","l 102 300 200 300"]) involving interpretation and drawing each time.
- Compiled version
  - **1 time makeGraphics(["b 100 200","b 101 200","b 102 200","l 102 200 102 300","l 102 300 200 300"])**
    - Takes as much time (or more) as intepreting.
    - But only *once*
  - **100 times running the very small graphics program.**

Applications are compiled

- Applications like Photoshop and Word are written in languages like C or C++
  - **These languages are then compiled down to machine language.**
  - **That stuff that executes at a rate of 1.5 billion bytes per second.**
- Jython programs are interpreted.
  - **Actually, they're interpreted twice!**

Java programs typically don’t compile to machine language.

- Recall that every processor has its *own* machine language.
  - **How, then, can you create a program that runs on any computer?**
- The people who invented Java also invented a *make-believe processor*—a *virtual machine*.
  - **It doesn’t exist anywhere.**
  - **Java compiles to run on the virtual machine**
    - The Java Virtual Machine (JVM)

What good is it to run only on a computer that doesn't exist?!?

- Machine language is a *very* simple language.
- A program that *interprets* the machine language of some computer is not hard to write.

```
def VMinterpret(program):
  for instruction in program:
    if instruction == 1:  #It's a load
     ...
    if instruction == 2:  #It's an add
      ...
```

Java runs on everything…

- Everything that has a JVM on it!
- Each computer that can execute Java has an *interpreter* for the Java machine language.
  - **That interpreter is usually compiled to machine language, so it's very fast.**
- Interpreting Java machine is pretty easy
  - **Takes only a small program**
- Devices as small as wristwatches can run Java VM interpreters.

What happens when you execute a Python statement in JES

- Your statement (like "show(canvas)") is *first* compiled to Java!
  - **Really! You're actually running Java, even though you wrote Python!**
- Then, the Java is compiled into Java virtual machine language.
  - **Sometimes appears as a .class or .jar file.**
- *Then*, the virtual machine language is interpreted by the JVM program.
  - **Which executes as a machine language program (a .exe)**

Is it any wonder that Python programs in JES are slower?

- Photoshop and Word simply execute.
  - **At 1.5 Ghz and faster!**
- Python programs in JES are compiled, then compiled, then interpreted.
  - **Three layers of software before you get down to the real speed of the computer!**
- It only works at all because 1.5 *billion* is a *REALLY* big number!

Why interpret?

- For us, to have a command area.
  - **Compiled languages don’t typically have a command area where you can print things and try out functions.**
  - **Interpreted languages help the learner figure out what’s going on.**
- For others, to maintain portability.
  - **Java can be compiled to machine language.**
    - In fact, some VMs will actually compile the virtual machine language for you while running—no special compilation needed.
  - **But once you do that, the result can only run on one kind of computer.**
  - **Programs for Java (.jar files typically) can be moved from any kind of computer to any other kind of computer and just work.**

Mid Term Review

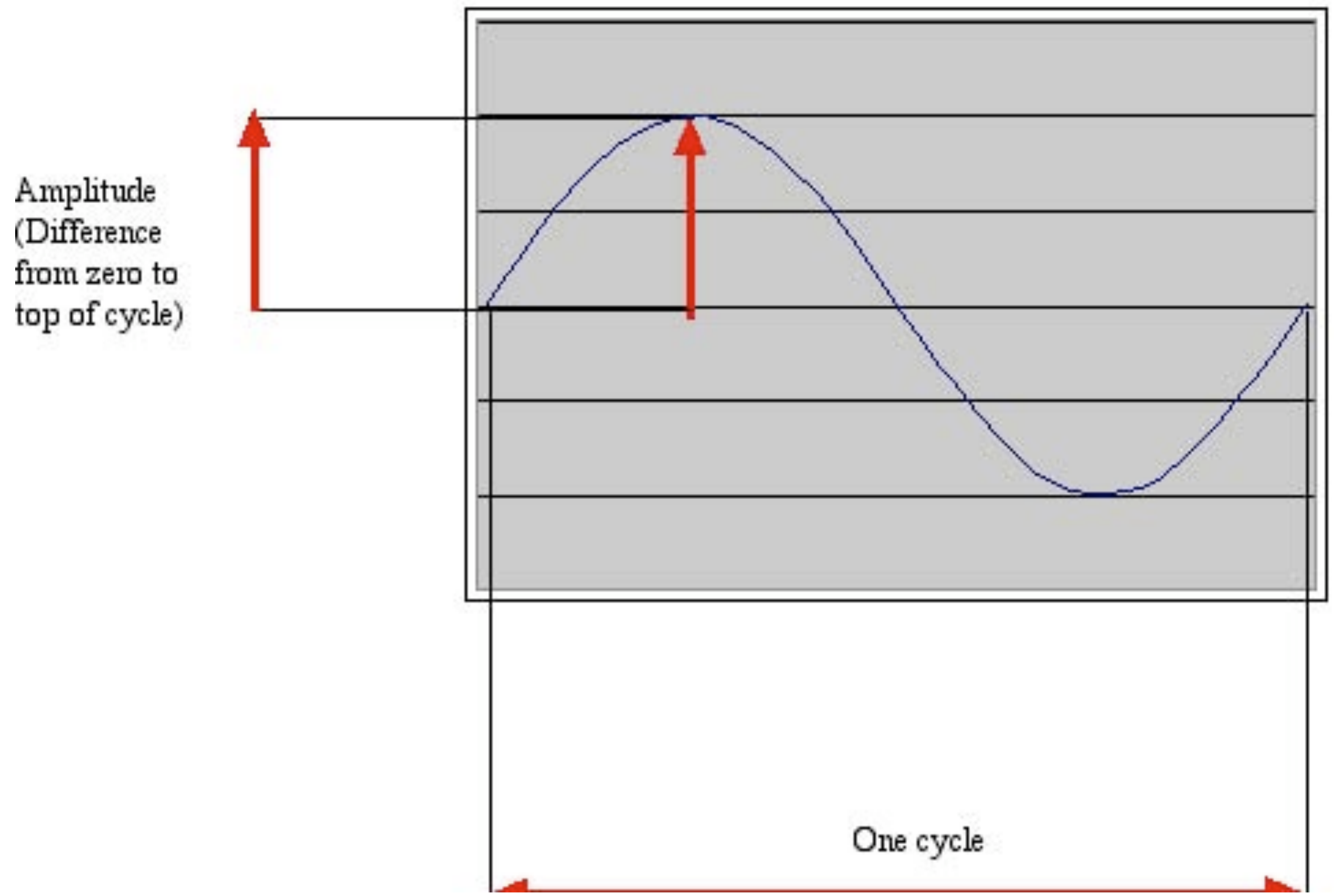
- Sound
    - samples
- Text
    - arrays and lists
    - object.method()
- Design and Problem-Solving
- HTML
- Recursion

Mid Term Review

- Sound
  - samples
- Text
  - arrays and lists
  - object.method()
- Design and Problem-Solving
- HTML
- Recursion

How sound works: Acoustics, the physics of sound

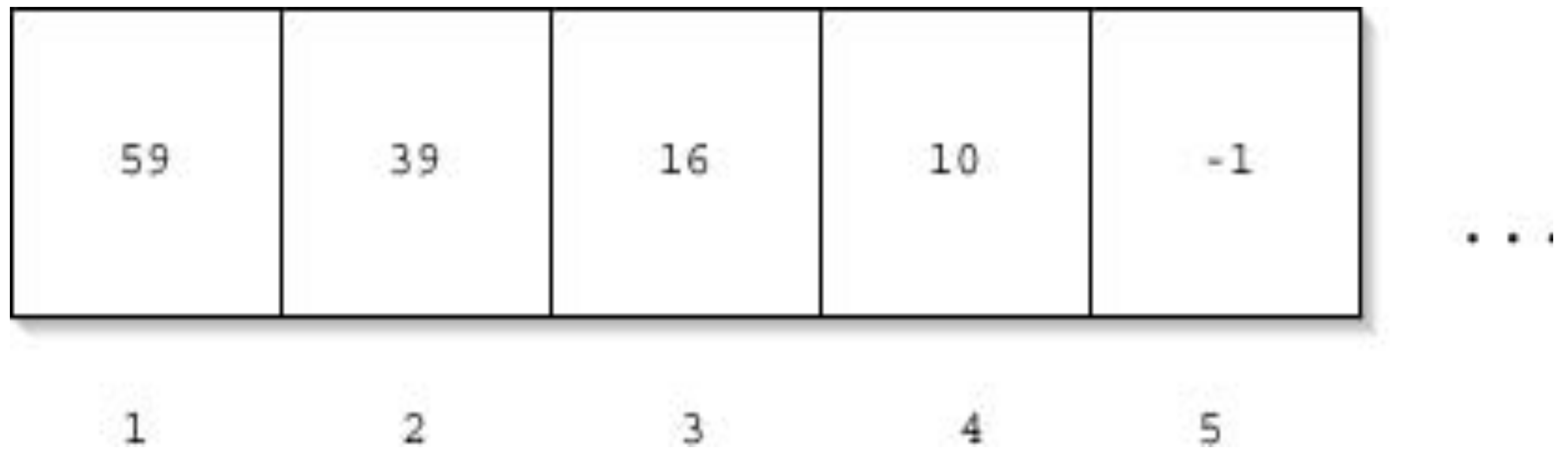
- Sounds are waves of air pressure
  - **Sound comes in cycles**
  - **The frequency of a wave is the number of cycles per second (cps), or Hertz**
    - (Complex sounds have more than one frequency in them.)
  - **The amplitude is the maximum height of the wave**



Sounds as arrays

- Samples are just stored one right after the other in the computer's memory
- That's called an *array*
  - **It's an especially efficient (quickly accessed) memory structure**
  - **each sample is two bytes**

(Like pixels in a picture)



Doubling the amplitude

```
def double( sound ) :
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample, value * 2)
```

Normalizing

- A few ways to think about “normalizing”:
  - **use the whole enchilada (don’t waste any bits…)**
  - **make everything use the same scale (0 to 100%)**

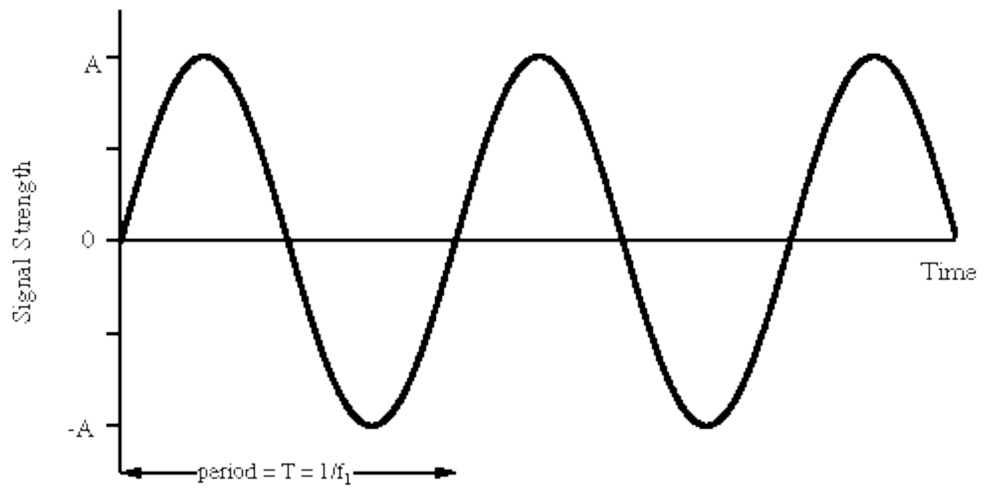
```
def normalize( sound ) :
  largest = 0
  for sample in getSamples(sound):
     largest = max( largest, getSample(sample) )
  multiplier = 32767.0 / largest
  for sample in getSamples(sound):
    setSample(sample, getSample(sample) * multiplier)
```

Ranges, home on the

- What is a range, really?
  - **a sequence**
  - **kind of like an array [1 ... N]**
    - or is it [1 ... N-1]?
- range(first element, upper bound + 1, increment)
  - **integers**
  - **first element**
  - **upper bound + 1**
    - a problem (remember black lines in EC?)
  - **increment**

Sine wave

- recipe 70

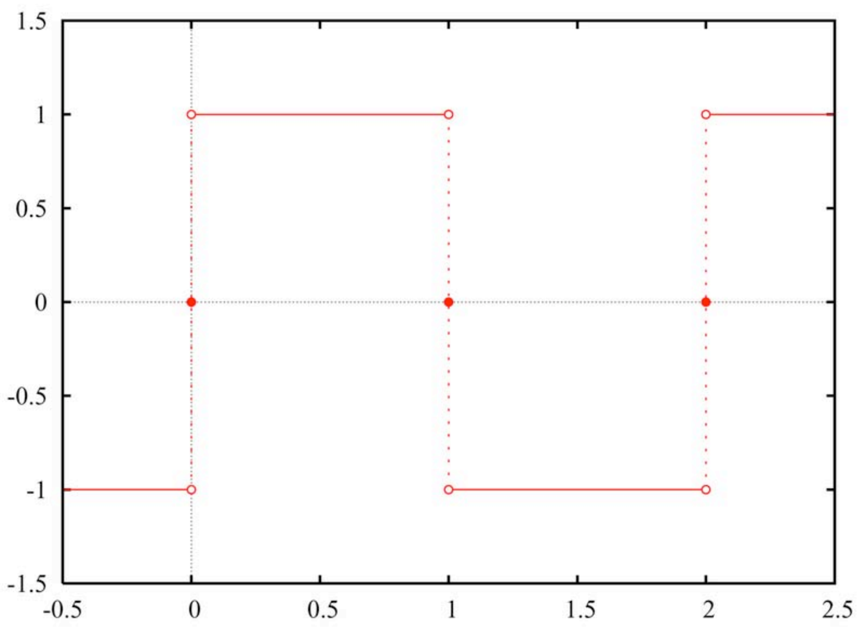


(a) Sine Wave

```
def sineWave( freq, amplitude ) :
  mySound = getMediaPath("sec1silence.wav")
  buildSin = makeSound(mySound)
  sr = getSamplingRate(buildSin)  # sampling rate
  interval = 1.0 / freq                   # interval of sample
  samplesPerCycle = interval * sr    # samples / cycle
  maxCycle = 2 * pi
  for pos in range( 1, getLength( buildSin ) + 1 ) :
    rawSample = sin(( pos / samplesPerCycle) * maxCycle)
    sampleVal = int( amplitude * rawSample )
    setSampleValueAt( buildSin, pos, sampleVal )
  return buildSin
```

Square wave

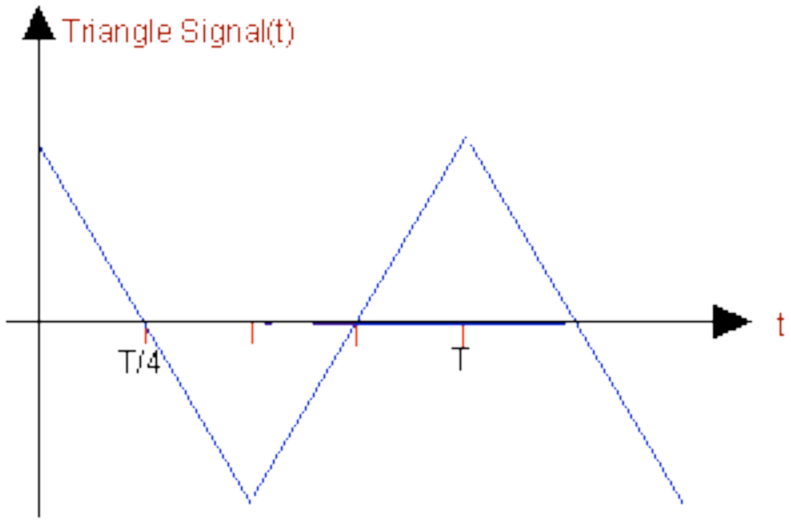
- recipe 72




```
def squareWave( freq, amplitude ) :
 mySound = getMediaPath("sec1silence.wav")
 square = makeSound(mySound)
 samplingRate = getSamplingRate(square)   # sampling rate
 seconds = 1
 interval = 1.0 * seconds / freq                     # interval of sample
 samplesPerCycle = interval * samplingRate # samples / cycle
 samplesPerHalfCycle = int(samplesPerCycle / 2)
 sampleVal = amplitude
 i = 1
 for s in range( 1, getLength( square ) + 1 ) :
  if (i > samplesPerHalfCycle):
    sampleVal = sampleVal * -1
    i = 0
  setSampleValueAt( square,s, sampleVal )
  i = i + 1
 return square
```

Triangluar wave

- recipe 73, modified




```
def triangleWave( freq ) :
  amplitude = 6000
  samplingRate = 22050                          # sampling rate
  seconds = 1
  triangle = makeEmptySound( seconds  )      # create a sound object (the book uses "sec1silence.wav")
  interval = 1.0 * seconds / freq               # interval of sample
  samplesPerCycle = interval * samplingRate # samples / cycle
  samplesPerHalfCycle = int(samplesPerCycle / 2)
  increment = int( amplitude / samplesPerHalfCycle )
  sampleVal = -amplitude
  i = 1
  for s in range( 1, samplingRate + 1 ) :
    if (i > samplesPerHalfCycle):
      increment = increment * -1
      i = 0
    sampleVal = sampleVal + increment
    setSampleValueAt( triangle, s, sampleVal )
    i = i + 1
  return triangle                                # return the sound (the book says play)
```

MIDI

- represent the sound waves
  - .wav
  - our Jython sound functions
- OR represent the "instruments"
- MIDI: *Musical Instrument Digital Interface*
  - used to connect audio (and some video) devices
    - instruments: keyboards, synthesizers, drum machines
    - synchronize events
  - more compact representation
- Jython's MIDI
  - just plays the notes (alas)
  - sounds like a piano

Mid Term Review

- Sound
  - samples
- Text
  - arrays and lists
  - object.method()
- HTML
- Design and Problem-Solving
- Recursion

Text

- Text is the universal medium
  - **We can convert any other media to a text representation.**
  - **We can convert between media formats using text.**
  - **Text is simple.**
- Text is usually processed in an *array*—a long line of characters
- We refer to one of these long line of characters as a *string*.

COMPUTER SCIENCE WAY OF THINKING

Strings

- Strings are defined with quote marks.
- Python actually supports three kinds of quotes:

```
>>> print 'this is a string'
this is a string
>>> print "this is a string"
this is a string
>>> print """this is a string"""
this is a string
```

- Use the right one that allows you to embed quote marks if you want

```
>>> phrase = "Monica's cat."
>>> print phrase
Monica's cat.
```

Encodings for strings

- Strings are just arrays of characters
- In most cases, characters are just single bytes.
  - **The ASCII encoding standard maps between single byte values and the corresponding characters**
- More recently, characters are two bytes.
  - **Unicode uses two bytes per characters so that there are encodings for glyphs (characters) of other languages**
  - **Java uses Unicode. The version of Python we are using is based in Java, so our strings are actually using Unicode.**

Backslash escapes

- “\b” is backspace
- “\n” is a newline (like pressing the Enter key)
- “\t” is a tab
- “\uXXXX” is a Unicode character, where XXXX is a code and each X can be 0-9 or A-F.
    - **http://www.unicode.org/charts/**
    - **Must precede the string with “u” for Unicode to work**

Getting parts of strings

- We use the square bracket "[]" notation to get parts of strings.
- stringVariable[n] gives you the n^{th} character in the string (but keep in mind the first one is the zero-ith)

So maybe its really the $n+1^{th}$ ...

- string[n:m] gives you the characters indexed by n through (but not including) index m.

Getting parts of strings

```
>>> helloStr = "Hello"
>>> print helloStr[1]
e
>>> print helloStr[0]
H
>>> print helloStr[2:4]
ll
```

H	e	l	l	o
0	1	2	3	4

Dot notation

- All data in Python are actually *objects*
- Objects not only store data, but they respond to special functions that only objects of the same type understand.
- We call these special functions *methods*
  - **Methods are functions known only to certain objects**
- To execute a method, you use *dot notation*
  - **objectName.method()**

Capitalize is a method known only to strings

```
>>> test="this is a test."
>>> print test.capitalize  # without the ()s a method
   will not execute
<builtin method 'capitalize'>
>>> print test.capitalize()
This is a test.
>>> print capitalize(test)
A local or global name could not be found.
NameError: capitalize
>>> print 'this is another test'.capitalize()
This is another test
>>> print 12.capitalize()
A syntax error is contained in the code -- I can't
   read it as Python.
Why?
```

Converting from strings to lists

```
>>> print letter.split(" ")
['Mr.', 'Mark', 'Guzdial', 'requests',
  'the', 'pleasure', 'of', 'your',
  'company...']
```

N.B. this split is splitting on a space. You can split on other characters too!

Lists

- We've seen lists before—that's what **range()** returns.
- Lists are very powerful structures.
  - **Lists can contain strings, numbers, even other lists.**
  - **They work very much like strings**
    - You get pieces out with []
    - You can "add" lists together
    - You can use **for** loops on them
  - **We can use them to process a variety of kinds of data.**

Useful methods to use with lists: But these don't work with strings

- **append(something)** puts something in the list at the end.
- **remove(something)** removes something from the list, if it's there.
- **sort()** puts the list in alphabetical order
- **reverse()** reverses the list
- **count(something)** tells you the number of times that something is in the list.
- **max()** and **min()** are **functions** that take a list as input and give you the maximum and minimum value in the list.

Mid Term Review

- Sound
  - samples
- Text
  - arrays and lists
  - object.method()
- HTML
- Design and Problem-Solving
- Recursion

HTML: Hypertext Markup Language

- Simple way of separating content from its display.
- A *markup language* adds *tags* to regular text to identify its parts.
- A tag in HTML is enclosed by <angle brackets>.
- Most tags have a starting tag and an ending tag.
  - **A paragraph is identified by a <p> at its start and a </p> at its end.**
  - **A heading is identified by a <h1> at its start and a </h1> at its end.**

The Simplest Web Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transition//EN"
    "http://wwww.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>The Simplest Possible Web Page</title>
</head>
<body>
<h1>A Simple Heading</h1>
<p>This is a paragraph in the simplest
possible Web page.</p>
</body>
</html>
```

Yes, that whole thing is the DOCTYPE

No, it doesn’t matter where you put new lines, or extra spaces

Parts of a Web Page

- You start with a DOCTYPE
  - **It tells browsers what kind of language you're using below.**
  - **It's gory and technical—copy it verbatim from somewhere.**
- The whole document is enclosed in <html> </html> tags.
  - **The heading is enclosed with <head> </head>**
    - That's where you put the <title> </title>
  - **The body is enclosed with <body> </body>**
    - That's where you put <h1> headings and <p> paragraphs.

A tiny tutorial on hexadecimal

- You know decimal numbers (base 10)
  - **0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16**
  - **often written as 9_{dec}**
- You've heard a little about binary (base 2)
  - **0000,0001,0010,0011,0100,0101... 0010_{bin}**
- Octal is base 8
  - **0,1,2,3,4,5,6,7,10,11,12,13,14,15,16,17,20 01_{oct}**
- Hexadecimal is base 16
  - **0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10 (16 base 10)**
  - **$1A_{hex}$**

Riddle

Riddle

- Why is Halloween logical Christmas?

Riddle

- Why is Halloween logical Christmas?
- because 31oct = 25dec

Hexadecimal colors in HTML

- #000000 is black
  - 0 for red, 0 for green, 0 for blue
  - or all bits set to 0
- #FFFFFF is white
  - 255 for red, 255 for green, 255 for blue
  - or all bits set to 1
- #FF0000 is Red
  - 255 for red (FF), 0 for green, 0 for blue
  - or 111111110000000000000000
- #0000FF is Blue
  - 0 for red, 0 for green, 255 for blue
  - or 000000001111111100000000

Emphasizing your text

- There are six levels of headings defined in HTML.
  - **<h1>…<h6>**
  - **Lower numbers are larger, more prominent.**
- Styles
  - **Emphasis, <i>Italics</i>, and Boldface**
  - **<big>Bigger font</big> and <small>Smaller font</small>**
  - **<tt>Typewriter font</tt>**
  - **<pre>Pre-formatted</pre>**
  - **<blockquote>Blockquote</blockquote>**
  - **^{Superscripts} and _{Subscripts}**

Finer control: <font>

- Can control type face, color, or size

```
<body>
   <h1>A Simple Heading</h1>

   <p><font face="Helvetica">
        This is in helvetica
   </font></p>

   <p><font color="green">
        Happy Saint Patrick's Day!
   </font></p>

   <p><font size="+2">
        This is a bit bigger
   </font></p>
```

Can also use hexadecimal RGB specification here.

Breaking a line

- Line breaks are part of formatting, not content, so they were added grudgingly to HTML.
- Line breaks don't have a closing tag, so include the ending "/" inside.
  - **
**

Adding a break

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
    4.01 Transition//EN" "http://wwww.w3.org/TR/
    html4/loose.dtd">
<html>

  <head>
    <title>The Simplest Possible Web Page</title>
  </head>

  <body>

    <h1>A Simple Heading</h1>

    <p>This is a paragraph in the simplest<br />
    possible Web page.</p>
```

Adding an image

- Like break, it's a standalone tag.
  - ****
- What goes inside the quotes is the path to the image.
  - **If it's in the same directory, don't need to specify the path.**
  - **If it's in a subdirectory, you need to specify the subdirectory and the base name.**
  - **You can walk a directory by going up to a parent directory with ".."**
  - **You can also provide a complete URL to an image anywhere on the Web.**

Creating links

- Links have two main parts to them:
  - **A destination URL.**
  - **Something to be clicked on to go to the destination.**
- The link tag is “a” for “anchor”

Mark Guzdial

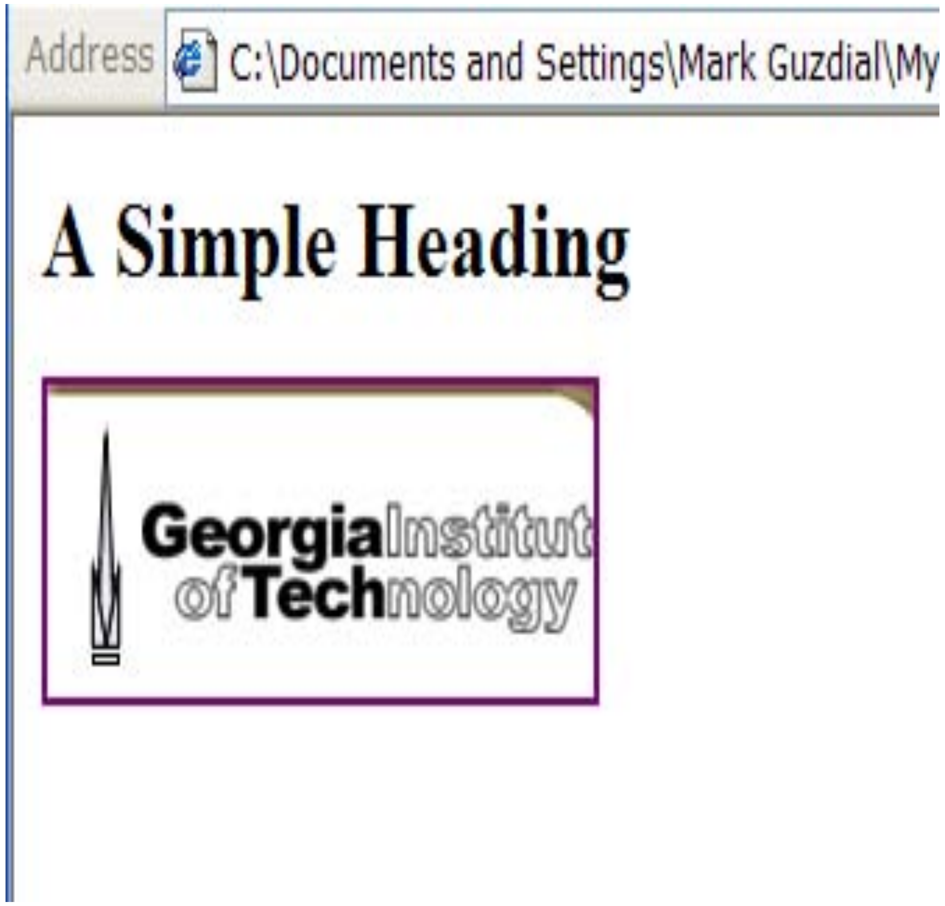
Images can be links!

```
<h1>A Simple Heading</h1>

<p>
   <a href="http://www.cc.gatech.edu/">

   <img src="http://www.cc.gatech.edu/
    images/ main_files/goldmain_01.gif" />

</a>
</p>
```



Lists (not to be confused with Jython lists…)

- Ordered lists (numbered)

  <ol>

  **
First item
**

  **
Next item
**

  **
**

- Unordered lists (bulleted)

  <ul>

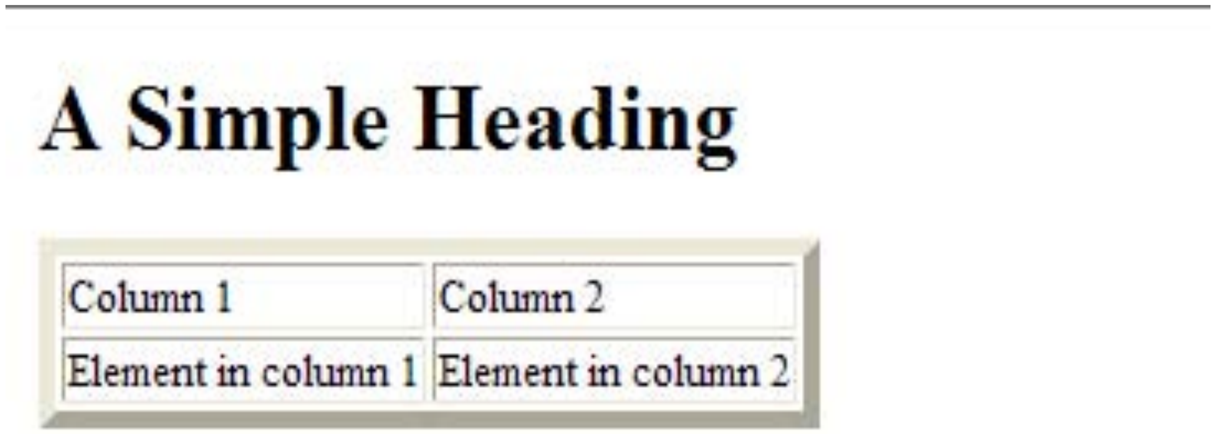
  **
First item
**

  **
Second item
**

  **
**

Tables

```
<table border="5">
<tr>
  <td>Column 1</td>
  <td>Column 2</td>
</tr>
<tr>
  <td>Element in column 1</td>
  <td>Element in column 2</td>
</tr>
</table>
```



There is lots more to HTML

- Frames
  - Can have subwindows within a window with different HTML content.
  - Anchors can have target frames.
- Divisions <div>
- Horizontal rules <hr />
  - With different sizes, colors, shading, etc.
- Applets, Javascript, CSS, etc.

Mid Term Review

- Sound
  - samples
- Text
  - arrays and lists
  - object.method()
- HTML
- Design and Problem-Solving
- Recursion

Top-down method

- Figure out what has to be done.
  - **These are called the requirements**
- Refine the *requirements* until they describe, in English, what needs to be done in the program.
  - **Keep refining until you know how to write the program code for each statement in English.**
- Step-by-step, convert the English requirements into program code.

Top-down Example

- *Write a function called pay that takes in as input a number of hours worked and the hourly rate to be paid.* Compute the gross pay as the hours times the rate. If the pay is< 100, charge a tax of 0.25 ; if the pay is >= 100 and < 300, tax rate is 0.35 ; if the pay is >=300 and < 400, tax rate is 0.45 ; if the pay is >= 400, tax rate is 0.50 ; Compute a taxable amount as tax rate * gross ; Print the gross pay and the net pay (gross – taxable amount).

Top-down Example:
Refine into steps you can code

- Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- Compute the gross pay as the hours times the rate.
- If the pay is< 100, charge a tax of 0.25
- If the pay is >= 100 and < 300, tax rate is 0.35
- If the pay is >=300 and < 400, tax rate is 0.45
- If the pay is >= 400, tax rate is 0.50
- Compute a taxable amount as tax rate * gross
- Print the gross pay and the net pay (gross – taxable amount).

Convert to program code

- √ Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- Compute the gross pay as the hours times the rate.
- If the pay is< 100, charge a tax of 0.25
- If the pay is >= 100 and < 300, tax rate is 0.35
- If the pay is >=300 and < 400, tax rate is 0.45
- If the pay is >= 400, tax rate is 0.50
- Compute a taxable amount as tax rate * gross
- Print the gross pay and the net pay (gross – taxable amount).

```
def pay(hours,rate):
```

Convert to program code

- √ Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- √ Compute the gross pay as the hours times the rate.
- If the pay is< 100, charge a tax of 0.25
- If the pay is >= 100 and < 300, tax rate is 0.35
- If the pay is >=300 and < 400, tax rate is 0.45
- If the pay is >= 400, tax rate is 0.50
- Compute a taxable amount as tax rate * gross
- Print the gross pay and the net pay (gross – taxable amount).

```
def pay(hours,rate):
  gross = hours * rate
```

Convert to program code

- √ Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- √ Compute the gross pay as the hours times the rate.
- √ If the pay is< 100, charge a tax of 0.25
- If the pay is >= 100 and < 300, tax rate is 0.35
- If the pay is >=300 and < 400, tax rate is 0.45
- If the pay is >= 400, tax rate is 0.50
- Compute a taxable amount as tax rate * gross
- Print the gross pay and the net pay (gross – taxable amount).

```
def pay(hours,rate):
  gross = hours * rate
  if pay < 100:
    tax = 0.25
```

Convert to program code

- √ Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- √ Compute the gross pay as the hours times the rate.
- √ If the pay is< 100, charge a tax of 0.25
- √ If the pay is >= 100 and < 300, tax rate is 0.35
- √ If the pay is >=300 and < 400, tax rate is 0.45
- √ If the pay is >= 400, tax rate is 0.50
- Compute a taxable amount as tax rate * gross
- Print the gross pay and the net pay (gross – taxable amount).

```
def pay(hours,rate):
  gross = hours * rate
  if pay < 100:
    tax = 0.25
  if 100 <= pay < 300:
    tax = 0.35
  if 300 <= pay < 400:
    tax = 0.45
  if pay >= 400:
    tax = 0.50
```

Convert to program code

- √ Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- √ Compute the gross pay as the hours times the rate.
- √ If the pay is< 100, charge a tax of 0.25
- √ If the pay is >= 100 and < 300, tax rate is 0.35
- √ If the pay is >=300 and < 400, tax rate is 0.45
- √ If the pay is >= 400, tax rate is 0.50
- √ Compute a taxable amount as tax rate * gross
- Print the gross pay and the net pay (gross – taxable amount).

```
def pay(hours,rate):
  gross = hours * rate
  if pay < 100:
    tax = 0.25
  if 100 <= pay < 300:
    tax = 0.35
  if 300 <= pay < 400:
    tax = 0.45
  if pay >= 400:
    tax = 0.50
  taxableAmount = gross * tax
```

Convert to program code

- √ Write a function called **pay** that takes in as input a number of hours worked and the hourly rate to be paid.
- √ Compute the gross pay as the hours times the rate.
- √ If the pay is< 100, charge a tax of 0.25
- √ If the pay is >= 100 and < 300, tax rate is 0.35
- √ If the pay is >=300 and < 400, tax rate is 0.45
- √ If the pay is >= 400, tax rate is 0.50
- √ Compute a taxable amount as tax rate * gross
- √ Print the gross pay and the net pay (gross – taxable amount).

```
def pay(hours,rate):
 gross = hours * rate
 if pay < 100:
  tax = 0.25
 if 100 <= pay < 300:
  tax = 0.35
 if 300 <= pay < 400:
  tax = 0.45
 if pay >= 400:
  tax = 0.50
 taxableAmount = gross * tax
 print “Gross pay:”,gross
 print “Net pay:”,gross-taxableAmount
```

Why “top-down”?

- We start from the highest level of abstraction
  - **The requirements**
- And work our way down to the most specificity
  - **To the code**
- The opposite is “bottom-up”
- Top-down is the most common way that professionals design.
  - **It provides a well-defined process and can be tested throughout.**

What's "bottom-up"?

- Start with what you know, and keep adding to it until you've got your program.
- You *frequently* refer to programs you know.
  - **Frankly, you're looking for as many pieces you can steal as possible!**
- Take something and start modifying it
  - **AKA "Debugging your way into reality".**

How to understand a program

- Step 1: *Walk* the program
  - **Figure out what every line is doing, and what every variable's value is.**
  - **At least, do this for the lines that are confusing to you.**
- Step 2: *Run* the program
  - **Does it do what you think it's doing?**
- Step 3: *Check* the program
  - **Insert print statements to figure out what values are what in the program**
  - **You can also use print statements to print out values like getSampleValueAt and getRed to figure out how IF's are working.**

How to understand a program

- Use the command area!
  - **Type commands to check on values, to see how functions work.**
  - **Not sure what getSampleValueAt does? Try it!**
  - **Use showVars() to help, too.**
- Step 4: *Change* the program
  - **Now, change the program in some interesting way**
    - Instead of all pixels, do only the pixels in part of the picture
  - **Run the program again. Can you see the effect of your change?**
  - **If you can change the program and understand why your change did what it did, you understand the program**

Mid Term Review

- Sound
  - samples
- Text
  - arrays and lists
  - object.method()
- HTML
- Design and Problem-Solving
- Recursion

A very powerful idea: Recursion

- Recursion is writing functions that call *themselves*.
- When you write a recursive function, you write (at least) two pieces:
  - **What to do if the input is the smallest possible datum,**
  - **What to do if the input is larger so that you:**
    - (a) process one piece of the data
    - (b) call the function to deal with the rest.
- SEE CHAPTER 14 FOR MORE ON RECURSION

Why use functional programming and recursion?

- Can do a lot in very few lines.
- Very useful techniques for dealing with hard problems.
- *ANY* kind of loop (FOR, WHILE, and many others) can be implemented with recursion.
  - **It's the most flexible and powerful form of looping.**

Factorial -- the classic recursive function

```
def factorial( number ) :
  # the “head”
  if number == 1 :
    return number
  # the “rest”
  else :
    return number * factorial( number - 1.0 )
```

A recursive decreaseRed

```
def decreaseRed(alist):
  if alist == []:   #Empty
    return
  setRed(alist[0],
    getRed(alist[0])*0.8)
  decreaseRed(alist[1:])
```

- If the list (of pixels) is empty, don't do anything.
  - **Just return**
- Otherwise,
  - **Decrease the red in the first pixel.**
  - **Call decreaseRed on the rest of the pixels.**
- Call it like:
  >>> decreaseRed(getPixels(pic))

This actually won't work for reasonable-sized pictures—takes up too much memory in Java. The reason is each time the "rest", decreaseRed(alist[1:]), is called, it keeps a copy of the remainder alist[1:]. That gets big fast!

Mid Term Review

- Sound
  - samples
- Text
  - arrays and lists
  - object.method()
- HTML
- Design and Problem-Solving
- Recursion

Studying

- Look at programs
- Changes programs
- Write new ones

Coming Attractions

- Friday Lab
  - MidTerm II
    - open book
    - open computer
    - we will monitor internet traffic in room
    - multiple choice + 3 programs
- Monday
  - read Chapters 13, 14, & 16 (skip 12 and 15)
  - quiz due 10:00 AM
- Friday
  - HW 7 - Mind Reading Website due 10:00 AM