CS 1124 Media Computation

Lecture 10.1 October 27, 2008

Steve Harrison

TODAY

- from split to databases
- modules
- Objects
- interpreted vs. compiled

TODAY

- from split to databases
- modules
- Objects
- interpreted vs. compiled

From last time: split

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

Extended Split Example

```
def phonebook():
  return """
Mary:893-0234:Realtor:
Fred:897-2033:Boulder crusher:
Barney:234-2342:Professional bowler:"""

def phones():
  phones = phonebook()
  phonelist = phones.split('\n')
  newphonelist = []
  for list in phonelist:
    newphonelist = newphonelist +
      [list.split(":")]
  return newphonelist
```

```
def findPhone(person):
  for people in phones():
    if people[0] == person:
      print "Phone number
for",person,"is",people[
1]
```

Running the Phonebook

```
>>> print phonebook()

Mary:893-0234:Realtor:
Fred:897-2033:Boulder crusher:
Barney:234-2342:Professional bowler:
>>> print phones()
[[''], ['Mary', '893-0234', 'Realtor', ''], ['Fred', '897-2033', 'Boulder crusher', ''],
    ['Barney', '234-2342', 'Professional bowler', '']]
>>> findPhone('Fred')
Phone number for Fred is 897-2033
```

TODAY

- from split to databases
- modules
- Objects
- interpreted vs. compiled

Adding new capabilities: Modules

- What we need to do is to add capabilities to Python that we haven't seen so far.
- We do this by *importing* external *modules.*
- A module is a file with a bunch of additional functions and objects defined within it.
  - **Some kind of module capability exists in virtually every programming language.**
- By *importing* the module, we make the module's capabilities available to our program.
  - **Literally, we are evaluating the module, as if we'd typed them into our file.**

An interesting module: Random

```
>>> import random
>>> for i in range(1,10):
...     print random.random()
...
0.82113693141939328
0.6354266779703246
0.9460060163520159
0.904615696559684
0.33500464463254187
0.08124982126940594
0.0711481376807015
0.725521730734604 8
0.2920541211845866
```

Randomly choosing words from a list

```
>>> for i in range(1,5):
...     print random.choice(["Here", "is", "a", "list", "of",
    "words", "in","random","order"])
...
list
a
Here
list
```

Randomly generating language

- Given a list of nouns,
  verbs that agree in tense and number,
  and object phrases that all match the verb,
- We can randomly take one from each to make sentences.

Random sentence generator

```
import random

def sentence():
  nouns = ["Mark", "Adam", "Angela", "Larry", "Jose", "Matt", "Jim"]
  verbs = ["runs", "skips", "sings", "leaps", "jumps", "climbs", "argues",
    "giggles"]
  phrases = ["in a tree", "over a log", "very loudly", "around the bush", "while
    reading the Technique"]
  phrases = phrases + ["very badly", "while skipping","instead of grading",
    "while typing on the CoWeb."]
  print random.choice(nouns), random.choice(verbs), random.choice(phrases)
```

Running the sentence generator

```
>>> sentence()
Jose leaps while reading the Technique
>>> sentence()
Jim skips while typing on the CoWeb.
>>> sentence()
Matt sings very loudly
>>> sentence()
Adam sings in a tree
>>> sentence()
Adam sings around the bush
>>> sentence()
Angela runs while typing on the CoWeb.
>>> sentence()
Angela sings around the bush
>>> sentence()
Jose runs very badly
```

TODAY

- more dot.notation() examples
- modules
- Objects
- interpreted vs. compiled

History of Objects: Where they came from

- Start of the Story: Late 60's and Early 70's
  - **Windows are made of glass, mice are undesirable rodents**
  - **Good programming = Procedural Abstraction**
    - That’s basically what we’ve been doing—procedural-oriented programming
    - It’s essentially Verb-oriented
      - **We’re defining “How to” swap backgrounds, increase red, decrease volume, etc.**

Object-oriented programming

- First goal: Define and describe the *objects* of the world
  - **Noun-oriented**
  - **Focus on the domain of the program**
  - **The object-oriented analyst asks herself: “The program I’m trying to write relates to the real world in some way. What are the things in the real world that this program relates to?”**
- Example: Imagine you’re building an O-O Banner
  - **What are the objects?**
  - **Students, transcripts, classes, catalog, major-requirements, grades, rooms…**

Alan Kay

- U. Utah PhD student in 1966
  - **Studied Sketchpad, the first object-oriented drawing program**
  - **Studied Simula, a programming language designed to make simulations (e.g., to allow you to simulate a factory floor to see if the flow of materials worked well before you actually built it)**
- Saw “objects” as the future of computer science
  - **The way to build software better, more robustly, handle complexity better.**

Kay's Insights

- Think of the "Computer" as collection of Networked Computers
  - **Each one does its own thing and just communicates with others**
- All software is simulating the real world
  - **Therefore, it should be "noun-oriented" since the world is filled with nouns.**
- Biology as model for objects

Birth of Objects

- Objects as models of real world entities
- Objects as Cells
  - **Independent, indivisible, interacting—in standard ways**
  - **Scales well**
    - Complexity: Distributed responsibility
    - Robustness: Independent
    - Supporting growth: Same mechanism everywhere
    - Reuse: Provide services, just like in real world
- Created various languages to try out idea:
  - **A. Kay: Smalltalk & Squeak**
  - **others: Java, objective C, …**

We’ve been doing object-oriented programming already

- You’ve been using objects already, everywhere.
- Pictures, sounds, samples, colors—these are all objects.
- The functions that we’ve been providing merely cover up the underlying objects.

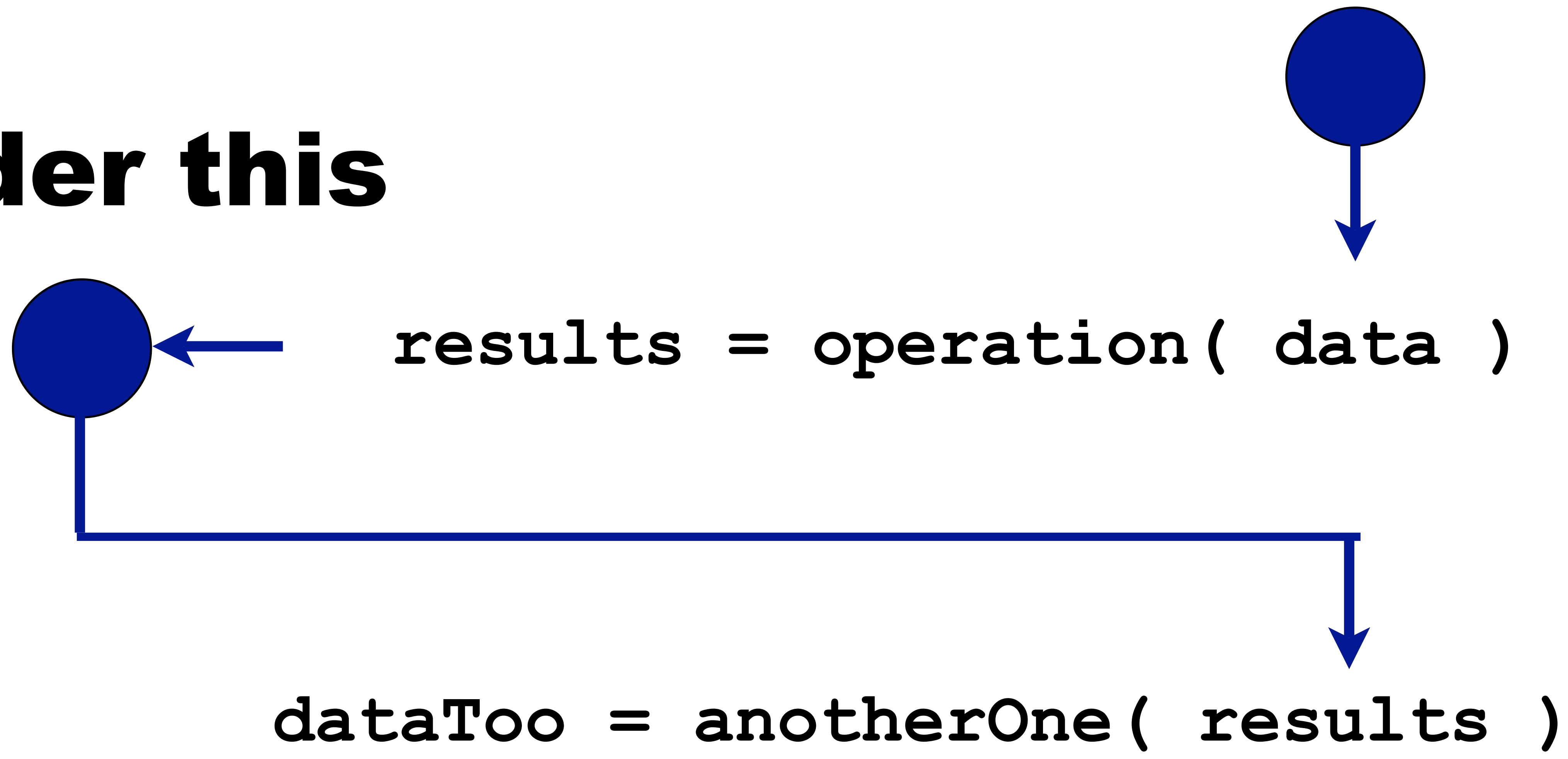
Using picture as an object

```
>>>
  pic=makePicture(getMediaPath("barbara.jp
  g"))
>>> pic.show()
```

show(pic) or pic.show()

what is the difference?

Consider this



```
results = operation( data )

dataToo = anotherOne( results )
```

What if **`operation`** needs to calculate some temporary value needed or **`anotherOne`**?

What if we mess up and pass the wrong data to **`anotherOne`**?

● = data

State information shared between functions.

What if...?

- What if instead of having functions like this:
  - **result = function(data)**
- ...we instead put the data round the function?
  - **result = data.function()**
- And what if the data could keep variables stored internally, like local variables in a function?
- Then...
  - **data.operation()**
  - **data.anotherOne()**

We move the data to the outside

```
results = operation( data )
```

HEY! What is “state”?

```
results = data.operation( )
```

Now we can keep state between calls.
The results of one operation can be available to the next call.

For now, we will see how to use objects, later we will define new ones.

State …

- Have you ever used “undo”?
- “State” is a stable condition where all the variables have known contents.

More methods than functions

- In general, there are many more *methods* defined in JES than there are *functions*.
- Most specifically, there are a whole bunch of methods for drawing onto a picture that aren't defined as functions.
  - **It's easier to deal with the complexity at the level of methods than functions.**
  - **The names for the functions get more and more complicated, where polymorphism lets them be simple and contextualized.**

TODAY

- from split to databases
- modules
- Objects
- interpreted vs. compiled

Big speed differences

- Many of the techniques we’ve learned take *no time at all* in other applications
- Select a figure in Word.
  - **It’s automatically inverted as fast as you can wipe.**
- Color changes in Photoshop happen *as you change the slider*
  - **Increase or decrease red? Play with it and see it happen just as fast as you can move the slider.**

Where does the speed go?

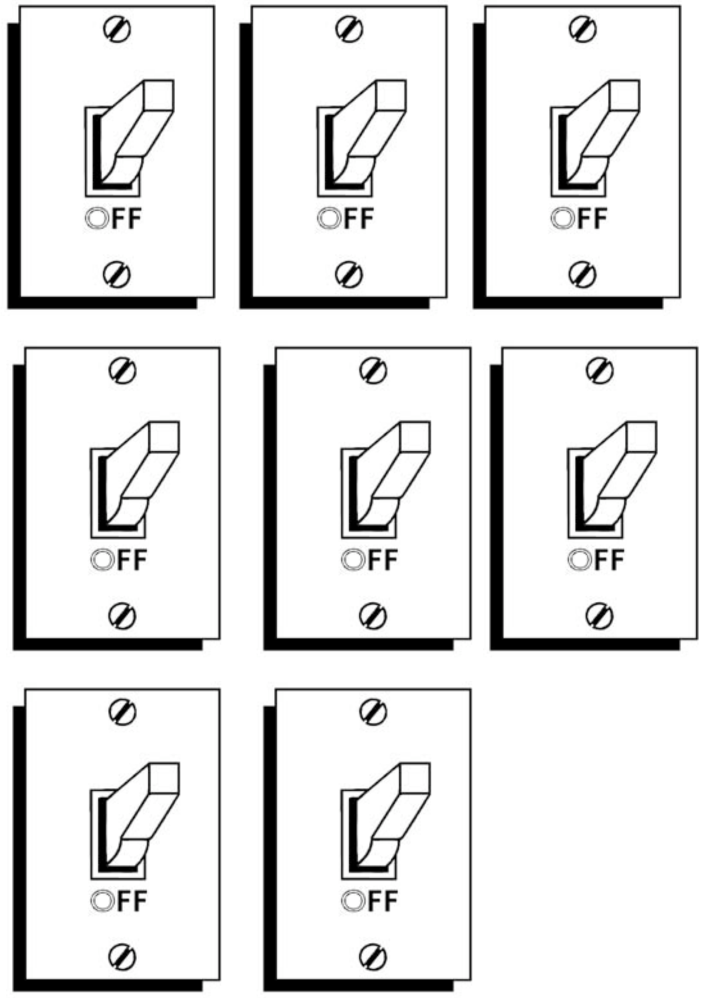
- Is it that Photoshop is so fast?
- Or that Jython is so slow?
- It's some of both—it's not a simple problem with an obvious answer.
- We'll consider in these two lectures two issues:
  - **How fast can computers get**
  - **What's not computable, no matter how fast you go**

What a computer really understands

- Computers really do not understand Python, nor Java, nor any other language.
- The basic computer only understands one kind of language: *machine language.*
  - **Machine language consists of instructions to the computer expressed in terms of values in bytes.**
  - **These instructions tell the computer to do very low-level activities.**

Machine language trips the right switches

- The computer doesn't really *understand* machine language.
- The computer is just a machine, with lots of switches that make data flow this way or that way.
- Machine language is just a bunch of switch settings that cause the computer to do a bunch of other switch settings.
- We *interpret* those switchings to be addition, subtraction, loading, and storing.
  - **In the end, it's all about encoding.**



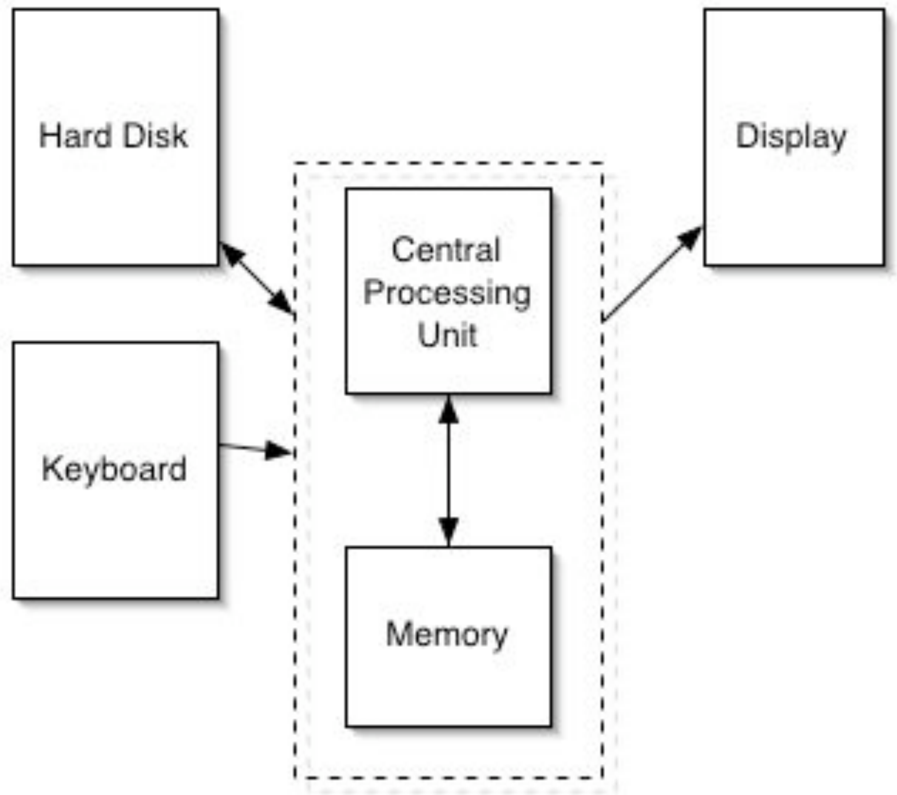
A byte of switches

Assembler and machine language

- Machine language looks just like a bunch of numbers.
- *Assembler language* is a set of words that corresponds to the machine language.
  - **It's a one-to-one relationship.**
  - **A word of assembler equals one machine language instruction, typically.**
    - (Often, just a single byte.)

Each kind of processor has its own machine language

- Apple computers typically use CPU *(processor)* chips called G4, G5, Intel Dual or Quad Core.
- Computers running Microsoft Windows use Intel Pentium, Dual or Quad Core processors or AMD Athlon.



Each processor understands only its *own* machine language

Assembler instructions

- Assembler instructions tell the computer to do things like:
  - **Store numbers into particular memory locations or into special locations (variables) in the computer.**
  - **Test numbers for equality, greater-than, or less-than.**
  - **Add numbers together, or subtract them.**

An example assembly language program

```
LOAD #10,R0        ; Load special variable R0 with 10
LOAD #12,R1        ; Load special variable R1 with 12
SUM R0,R1          ; Add special variables R0 and R1
STOR R1,#45        ; Store the result into memory location #45
```

Recall that we talked about memory as a long series of mailboxes in a mailroom.

Each one has a number (like #45).

The above is equivalent to Python's:

```
b = 10 + 12
```

Assembler -> Machine

```
LOAD 10,R0          ; Load special variable R0 with 10
LOAD 12,R1          ; Load special variable R1 with 12
SUM R0,R1           ; Add special variables R0 and R1
STOR R1,#45         ; Store the result into memory location #45
```

Might appear in memory as just 12 bytes:

```
01 00 10
01 01 12
02 00 01
03 01 45
```

Another Example

- LOAD R1,#65536 ; Get a character from keyboard
- TEST R1,#13 ; Is it an ASCII 13 (Enter)?
- JUMPTRUE #32768 ; If true, go to another part of the program
- CALL #16384 ; If false, call func. to process the new line

Machine Language:

```
05 01 255 255
10 01 13
20 127 255
122 63 255
```

Devices are (often) also just memory

- A computer can interact with external devices (like displays, microphones, and speakers) in lots of ways.
- Easiest way to understand it (and is often the *actual* way it's implemented) is to think about external devices as corresponding to a memory location.
  - **Store a 255 into memory location 65542, and suddenly the red component of the pixel at (101,345) on your screen is set to maximum intensity.**
  - **Everytime the computer reads memory location 897784, it's a new sample just read from the microphone.**
- So the simple loads and stores handle multimedia, too.

Machine language is executed very quickly

- A mid-range laptop these days has a *clock rate* of 1.5 Gigahertz.
- What that means *exactly* is hard to explain, but let's interpret it as processing 1.5 *billion* bytes per second.
- Those 12 bytes would execute inside the computer, then, in 12/1,500,000,000th of a second!

Applications are typically compiled

- Applications like Adobe Photoshop and Microsoft Word are *compiled*.
  - **This means that they execute in the computer as pure machine language.**
  - **They execute at that level speed.**
- However, Python, Java, Scheme, and many other languages are (in many cases) *interpreted*.
  - **They execute at a slower speed.**
  - **Why? It's the difference between translating instructions and directly executing instructions.**

An example

- Write a function **doGraphics** that will take a *list* as input. The function **doGraphics** will start by creating a canvas from the 640x480.jpg file in the mediasources folder. You will draw on the canvas according to the commands in the input list.

  Each element of the list will be a string. There will be two kinds of strings in the list:

- "b 200 120" means to draw a black dot at x position 200 y position 120. The numbers, of course, will change, but the command will always be a "b". You can assume that the input numbers will always have three digits.
- "l 000 010 100 200" means to draw a line from position (0,10) to position (100,200)

  So an input list might look like: ["b 100 200","b 101 200","b 102 200","l 102 200 102 300"] (but have any number of elements).

One student’s solution

```
def doGraphics(mylist):
  canvas = makePicture(getMediaPath("640x480.jpg"))
  for i in mylist:
    if i[0] == "b":
      x = int(i[2:5])
      y = int(i[6:9])
      print "Drawing pixel at ",x,":",y
      setColor(getPixel(canvas, x,y),black)
    if i[0] =="l":
      x1 = int(i[2:5])
      y1 = int(i[6:9])
      x2 = int(i[10:13])
      y2 = int(i[14:17])
      print "Drawing line at",x1,y1,x2,y2
      addLine(canvas, x1, y1, x2, y2)
  return canvas
```

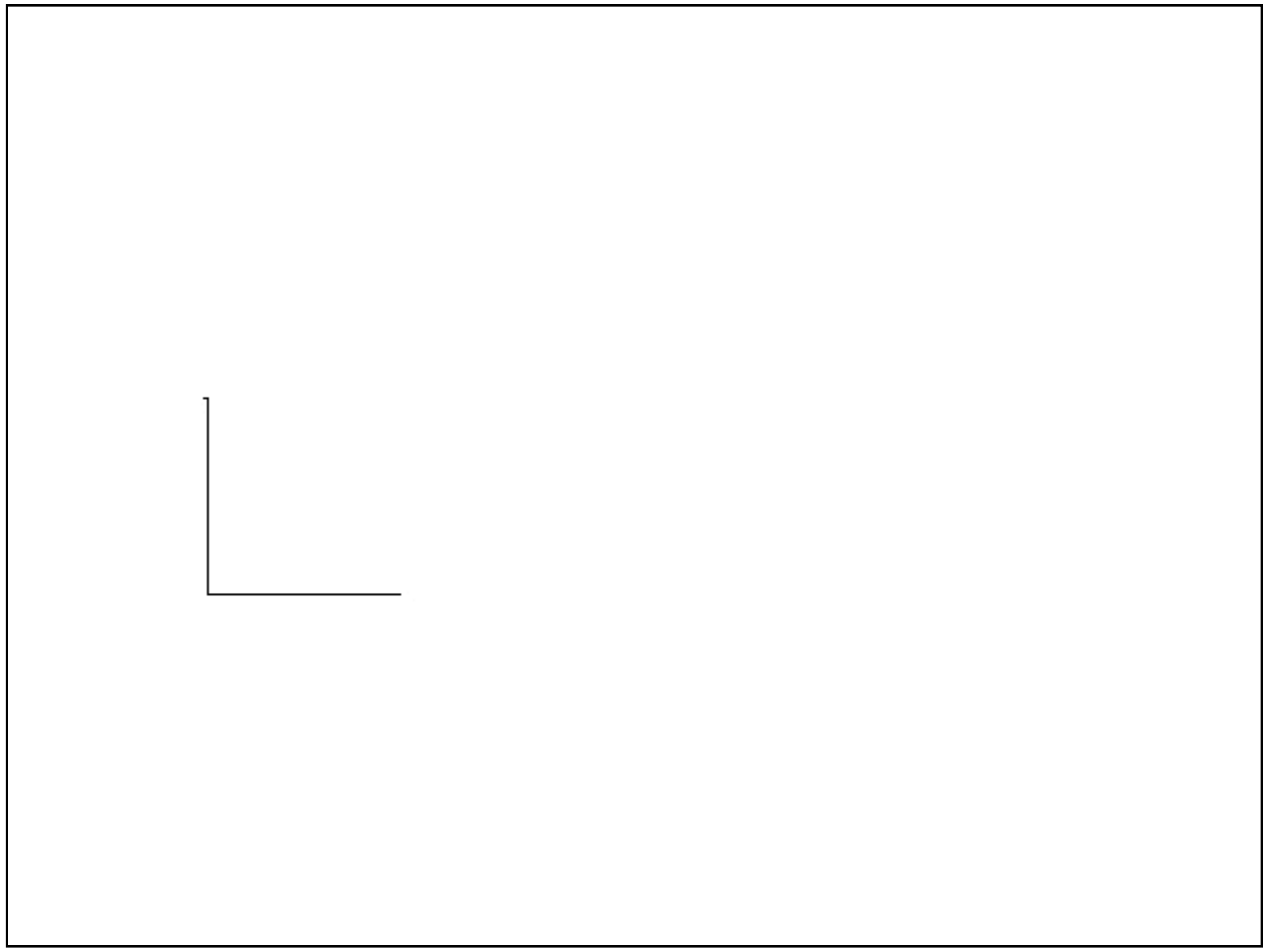
This program processes each string in the command list.

If the first character is “b”, then the x and y are pulled out, and a pixel is set to black.

If the first character is “l”, then the two coordinates are pulled out, and the line is drawn.

Running doGraphics()

```
>>> canvas=doGraphics(["b 100 200","b 101 200","b 102 200","l 102 200 102 300","l 102 300 200 300"])
Drawing pixel at  100 : 200
Drawing pixel at  101 : 200
Drawing pixel at  102 : 200
Drawing line at 102 200 102 300
Drawing line at 102 300 200 300
>>> show(canvas)
```



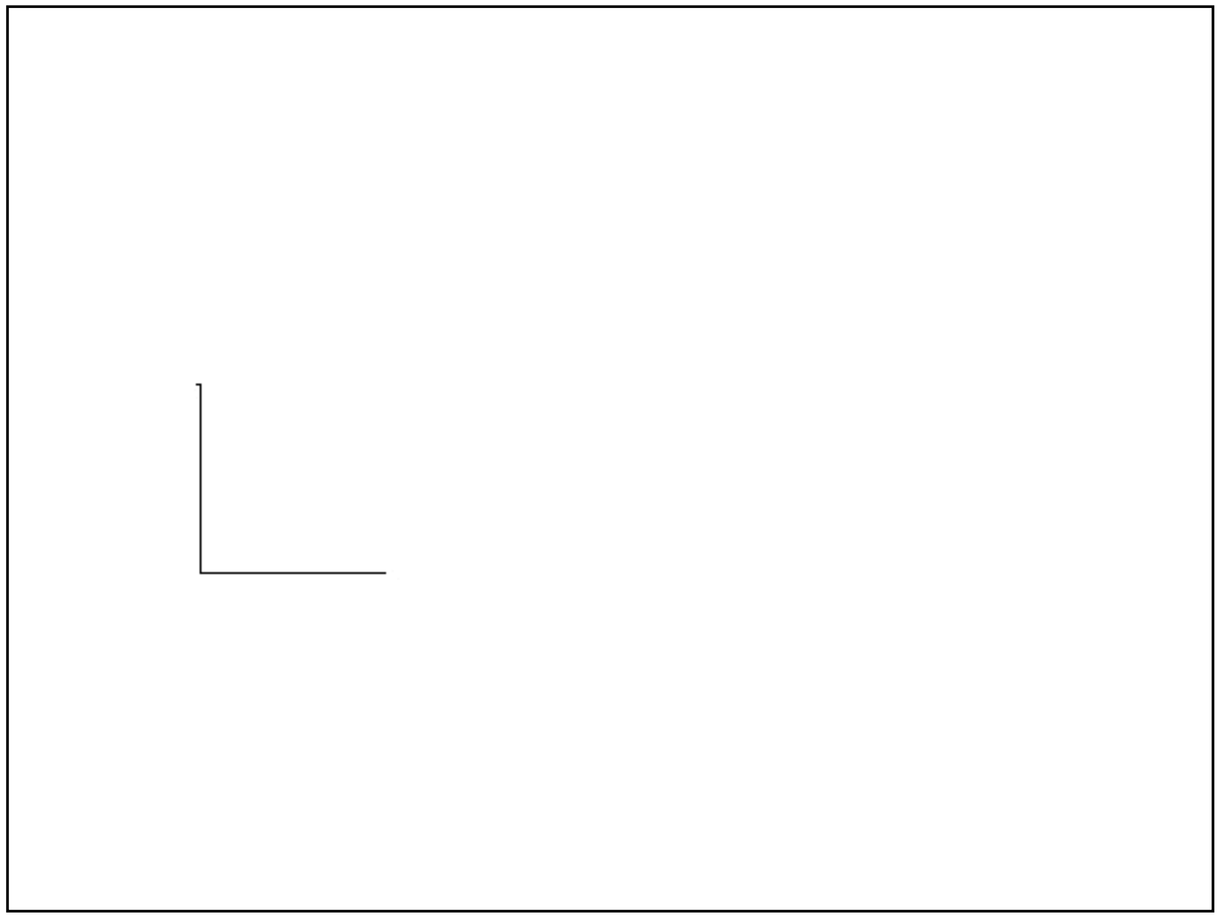
We’ve invented a new language

- ["b 100 200","b 101 200","b 102 200","l 102 200 102 300","l 102 300 200 300"] is a program in a new graphics programming language.
- Postscript, PDF, Flash, and AutoCAD are not too dissimilar from this.
  - **There’s a language that, when interpreted, “draws” the page, or the Flash animation, or the CAD drawing.**
- But it’s a *slow* language!

Would this run faster?

Does the exact same thing

```
def doGraphics():
  canvas =
     makePicture(getMediaPath("640x480.jp
     g"))
  setColor(getPixel(canvas, 100,200),black)
  setColor(getPixel(canvas, 101,200),black)
  setColor(getPixel(canvas, 102,200),black)
  addLine(canvas, 102,200,102,300)
  addLine(canvas, 102,300,200,300)
  show(canvas)
  return canvas
```



Which do you think will run faster?

```
def doGraphics(mylist):
  canvas =
     makePicture(getMediaPath("640x480.jpg"))
  for i in mylist:
    if i[0] == "b":
      x = int(i[2:5])
      y = int(i[6:9])
      print "Drawing pixel at ",x,":",y
      setColor(getPixel(canvas, x,y),black)
    if i[0] =="l":
      x1 = int(i[2:5])
      y1 = int(i[6:9])
      x2 = int(i[10:13])
      y2 = int(i[14:17])
      print "Drawing line at",x1,y1,x2,y2
      addLine(canvas, x1, y1, x2, y2)
  return canvas
```

```
def doGraphics():
  canvas = makePicture(getMediaPath("640x480.jpg"))
  setColor(getPixel(canvas, 100,200),black)
  setColor(getPixel(canvas, 101,200),black)
  setColor(getPixel(canvas, 102,200),black)
  addLine(canvas, 102,200,102,300)
  addLine(canvas, 102,300,200,300)
  show(canvas)
  return canvas
```

Above just *draws* the picture.

The left one *figures out* (interprets) the picture, then draws it.

Could we generate that second program?

- What if we could write a function that:
  - **Inputs ["b 100 200","b 101 200","b 102 200","l 102 200 102 300","l 102 300 200 300"]**
  - **Writes a file that is the Python version of that program.**

```
def doGraphics():
  canvas = makePicture(getMediaPath("640x480.jpg"))
  setColor(getPixel(canvas, 100,200),black)
  setColor(getPixel(canvas, 101,200),black)
  setColor(getPixel(canvas, 102,200),black)
  addLine(canvas, 102,200,102,300)
  addLine(canvas, 102,300,200,300)
  show(canvas)
  return canvas
```

Introducing a compiler

```
def makeGraphics(mylist):
  file = open("graphics.py","wt")
  file.write('def doGraphics():\n')
  file.write('  canvas =
    makePicture(getMediaPath("640x480.j
    pg"))\n');
  for i in mylist:
    if i[0] == "b":
      x = int(i[2:5])
      y = int(i[6:9])
      print "Drawing pixel at ",x,":",y
      file.write('  setColor(getPixel(canvas,
      '+str(x)+','+str(y)+'),black)\n')
    if i[0] =="l":
      x1 = int(i[2:5])
      y1 = int(i[6:9])
      x2 = int(i[10:13])
      y2 = int(i[14:17])
      print "Drawing line at",x1,y1,x2,y2
      file.write('  addLine(canvas,
      '+str(x1)+','+str(y1)+','+
      str(x2)+','+str(y2)+')\n')
  file.write('  show(canvas)\n')
  file.write('  return canvas\n')
  file.close()
```

Coming Attractions

- Friday Lab
  - MidTerm II
    - open book
    - open computer
- next Monday
  - read Chapters 13, 14, & 16 (skip 12 and 13)
  - quiz due 10:00 AM