

## COMP2720: Automating Tools for New Media

Why are PhotoShop or Visual Python so much faster than JES?

### Big speed differences

- Many of the techniques we've learned take (almost) no time at all in other applications.
- Select a figure in **Word** or **Powerpoint**.
  - 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.
- 3D animations in **Visual Python** only take milliseconds to update.
- Hugh Fisher's lecture was impressive among other things in how quickly Py3D animations were rendered

### Where does the speed go?

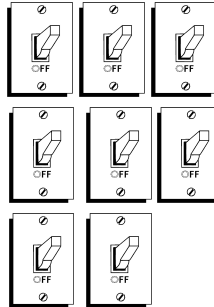
- Is it that **Visual Python** or **PhotoShop** is so fast?
  - Fast, compared to what?
- Or that **JES/Jython** is so slow?
  - Slow, compared to what?
- Or is it that **PhotoShop** is so much faster than **Jython**?
  - Yes, of course; but why?
  - It's not a simple problem with an obvious answer.
- We'll consider in this and the following lecture two issues:
  - Today: How fast can computers get?
  - Also: Are there problems that are not computable, no matter how fast the computer can go?

### What a computer *really* "understands"

- Computers really do not understand Python, nor Java, or any other easily understood 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

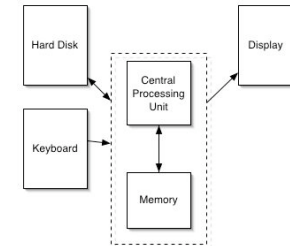
(all on = 11111111)

## Assembly and machine language

- Machine language looks just like a bunch of numbers.
- *Assembly 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 per instruction.

## Each *kind* of processor has its own machine language

- Apple computers (Mac's) typically use CPU (processor) chips called *G4, G5, Intel i386*
- Computers running Linux or Windows mainly use *Pentium* processors.
- There are other processors called *Sparc, MIPS, AMD* and many others.



Each processor understands only its *own* machine language

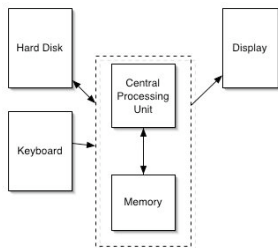
## Assembly instructions

- Assembly 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, or multiply them.

## An example assembly language program

```

LOAD #10, R0      ; Load special variable R0 with 10
LOAD #12, R1      ; Load special variable R1 with 12
ADD R0, R1        ; Add special variables R0 and R1
STORE 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`

## Assembly -> Machine language

```

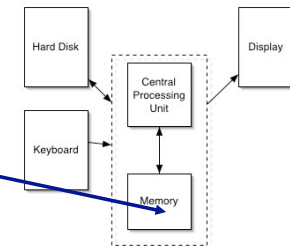
LOAD #10, R0      ; Load special variable R0 with 10
LOAD #12, R1      ; Load special variable R1 with 12
ADD R0, R1        ; Add special variables R0 and R1
STORE R1, #45     ; Store the result into memory location #45
    
```

Might appear in memory as just 12 bytes:

```

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

(e.g. these 12 bytes; right here)



## Another example

```

LOAD 65536, R1    ; 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:

```

01 255 255 01
10 01 13
20 127 255
22 63 255
    
```

The "stored program" computer (i.e. program = code **and** data) was invented by *John von Neumann*, and almost all modern computers realise von Neumann architecture

## Devices are (often) also just memory

- A computer can interact with external devices (like displays, network, keyboard, 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 location (101,345) on your screen is set to maximum intensity.
  - Every time 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.
- This is done by auxiliary processors (like graphics chips).

## Machine language is executed *very* quickly

- A mid-range laptop/desktop 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 (earlier example) would execute inside the computer, then, in 12/1,500,000,000th of a second!

## Applications are typically *compiled*

- Applications like **PhotoShop** and **Word** are written in languages like C or C++
  - These languages are then *compiled* down to machine language.
  - This means that they execute in the computer as pure machine language.
  - That stuff that executes at a rate of 1.5 billion bytes per second.
- However, Python, Java, 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.
- Jython programs are interpreted as well.
  - Actually, they're interpreted *twice!* (**J**ython = **J**ava + **P**ython)

## An example

- Assume a JES program that creates a graphic.

Write a function `doGraphics()` that will take a *list* as input. The function will start by creating an empty canvas of size 640x480 pixels. You will draw on the canvas according to the commands in the input list.

Each element of the input 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 and 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.
- "1 000 010 100 200" 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", "1 102 200 102 300"] (but have any number of elements).

## An example solution

```
def doGraphics(mylist):
    canvas = makeEmptyPicture(640,480)
    addRectFilled(canvas,1,1,640,480,white)
    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] == "1":
            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
```

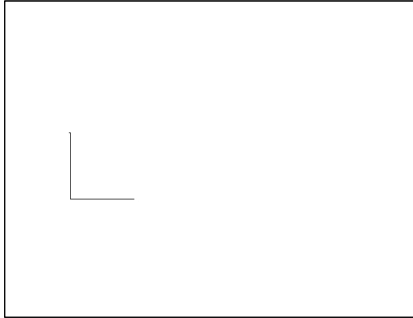
This program processes each string in the command list.

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 "1", 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)
```

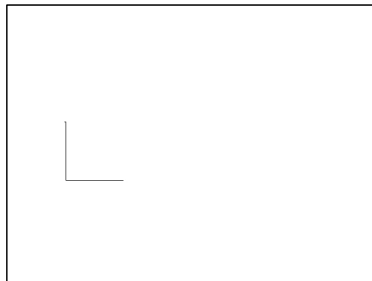


## We've invented a new language

- ["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 different 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 = makeEmptyPicture(640,480)
    addRectFilled(canvas,1,1,640,480,white)
    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 = makeEmptyPicture(640,480)
    addRectFilled(canvas,1,1,640,480,white)
    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 = makeEmptyPicture(640,480)
    addRectFilled(canvas,1,1,640,480,white)
    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 = makeEmptyPicture(640,480)
    addRectFilled(canvas,1,1,640,480,white)
    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(getMediaPath("graphics.py"),"wt")
    file.write('def doGraphics():\n')
    file.write(' canvas = makeEmptyPicture(640,480)\n')
    file.write(' addRectFilled(canvas,1,1,640,480,white)\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()
```

## 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 interpreting.
  - But only once!
- 100 times running the very small graphics program (faster).

## 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 re-do the same steps in **PhotoShop** each time.

## 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 — it's just a program.
  - 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 == 01: # It's a load
            ...
        if instruction == 02: # 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 the JVM is pretty easy.
  - Takes only a small program.
- Devices as small as wristwatches or mobile phones can run JVM 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 JVM 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!
- **Visual Python** is a compiled library written in C (but Python is still interpreted, only very small overhead).

## 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.

## What to do now

- Read section 13.1 and 13.2 in text book.
- Continue working on assignment 2.
- Work on portfolio.