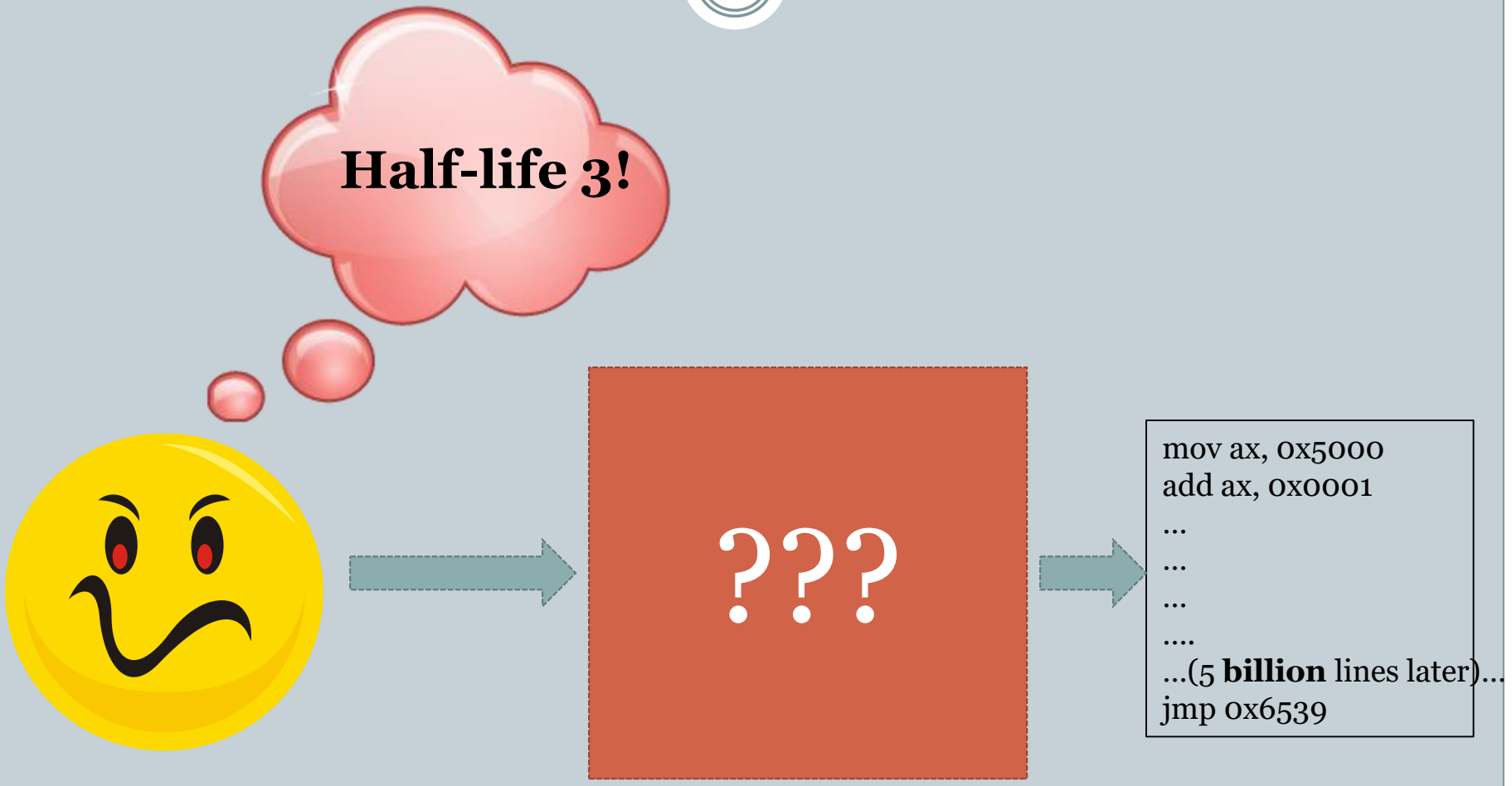
# High-level languages + Python

## REFERENCE: CHAPTER 1

(I'll try to put these at the Beginning of most lectures – it's from the optional textbook on the syllabus)

# Game Programming...???



# In the old days…the missing piece was a person!

```
VOUT            LDA  #$20
                STA  ScanLine           ; We're assuming scanline $20.
                STA  WSYNC
                STA  HMOVE              ; Move sprites horizontally.
VOUT_VB         LDA  INTIM
                BNE  VOUT_VB            ; Wait for INTIM to time-out.
                STA  WSYNC
                STA  CXCLR              ; Clear collision latches
                STA  VBLANK             ; End vertical blank
                TSX
                STX  TMPSTK             ; Save stack pointer
                LDA  #$02
                STA  CTRLPF             ; Double, instead of reflect.
                LDX  KLskip
Vskip1  STA  WSYNC              ; Skip a few scanlines...
                DEX
                BNE  Vskip1
                LDA  KLskip
                CMP  #$0E               ; "No Score" value of KLskip
                BEQ  Vmain
```

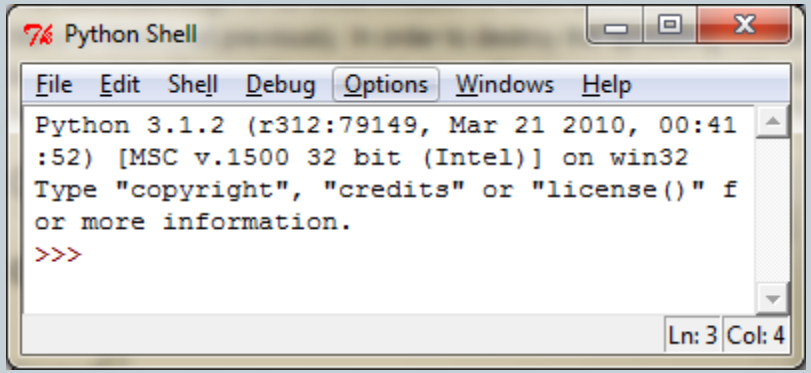

Ted Dabney (L) and Nolan Bushnell (R)

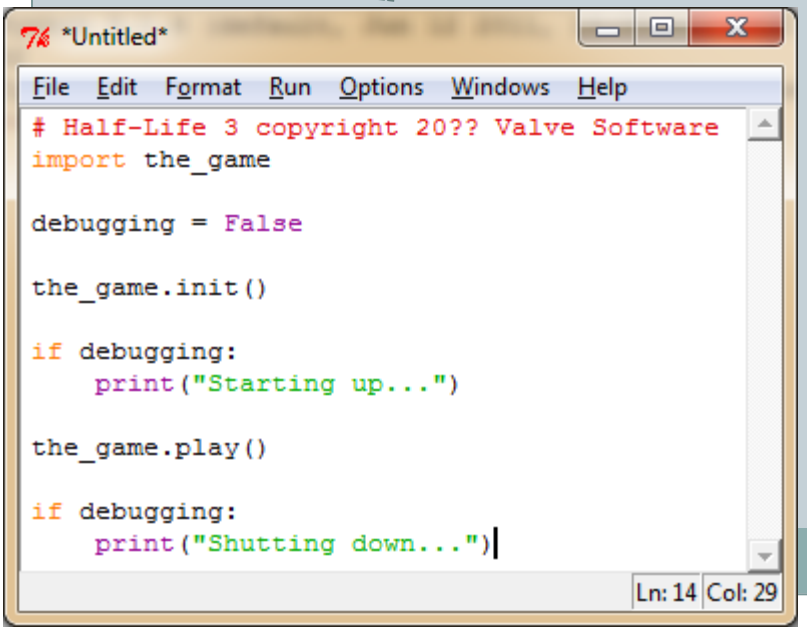
This is a 20-line snippet. The complete program has 1988 lines! See http://www.bjars.com/source/Combat.asm for the full source code (very well commented!)

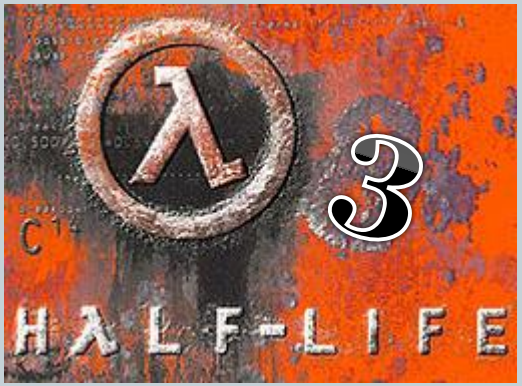
# Ready to learn assembly?

- Why don't (most) programmers write programs this way?
- What's the alternative?

# High level Programming languages to the rescue!


mov ax, 0x5000
add ax, 0x0001
...
...
...
....
...(5 **billion** lines later)...
jmp 0x6539

Halo Image from: http://ninjanerdstech.com/wiki/halo-4/

# Compilers / Interpreters

- **compiler** and **interpreter**
  - Specific to architecture and OS
  - Steps:
    - Reads  source file(s).
    - Generates machine code (CPU instructions)
- Compilers
  - Stand-alone executable (.exe file)
  - Faster to run, slow to build.
  - Cryptic run-time errors
- Interpreters
  - re-generate machine code every run
  - Instantaneous “build”, slower execution
  - Very informative run-time errors

# High Level Languages

- Name Some!
  - Interpreted:
  - Compiled:
- Why are there so many?

# Python!

- +: Easy to read/write (more English-like than many)
- +: A ton of libraries
  - e.g. pygame (www.pygame.org)
- +: Good balance between ease-of-use and power.
- -: Interpreted (so a little slower)
- -: A little harder to distribute

- Note:
  - Already installed on the lab computers
  - See the pdf on ssugames for instructions on installing it at home / on laptop

# Script vs. Interactive mode

- Interpreters usually have "interactive" mode
- [Script mode differences]
- [Demonstrate in IDLE]
- A word about IDE's (e.g. PyCharm)

# Errors

- Jason's rule-of-thumb (how much time you'll spend)
  - ~20% planning your approach
  - ~10% writing code
  - ~70% debugging / re-factoring code.
- **Syntax Errors**
- **Exceptions**
- Try these (in interactive mode):
  - print(Hello, World")
  - print(Hello, World)
  - Print("Hello, World")

# Errors, cont.

```
1  class Projectile(object):
2      def __init__(self, pos):
3          self.state = "Ready"
4          self.x = pos
5
6      def fire(self):
7          self.state = "Active"
8          self.x += "1"
9
10 def updateAll():
11     global L
12     for i in L:
13         i.fire()
14
15
16 L = []
17 L.append(Projectile(100))
18 L.append(Projectile(300))
19 updateAll()
20
```

- Call Stack:

This is the line number of the error:

We got there from here:

We got to the error from here:

The bottom section is the actual error:

This is the *type* of Error:

```
Traceback (most recent call last):
  File "C:\Temp\temp.py", line 19, in <module>
    updateAll()
  File "C:\Temp\temp.py", line 13, in updateAll
    i.fire()
  File "C:\Temp\temp.py", line 8, in fire
    self.x += "1"
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
>>>
```

# Print Function

- print(*expression*)
- expression can be:
  - A **string**
  - A **number**
  - A mathematical expression:
    - 4 + 5
    - 4 - 5
    - 4 * 5
    - 4 / 5
    - 2 ** 3
  - A comma-separated sequence of expressions
  - [sep and end]

# Comments

- [Syntax and evaluation]
- [Why are they useful?]

# Quick taste of variables

- Basically, a place to store a value.
- Use them just as you would a hard-coded (constant) value.

```
>>> x = 15
>>> print("x is", x)
     x is 15
```

- Warning: NOT a rule (as in math)
  - Just holds the *last value assigned to it.*

```
>>> width = 10
>>> height = 4
>>> area = width * height
>>> print(area)
     40
>>> width = 20
>>> print(area)
     40                (did you think it was 80? Why not?)
```

# Input function

- [As a means to pause]
- We'll see the *real* use next time…