# CSC207H: Software Design
# Lecture 3

Wael Aboelsaadat

wael@cs.toronto.edu
http://ccnet.utoronto.ca/20075/csc207h1y/
Office: BA 4261
Office hours: R 5-7

Acknowledgement: These slides are based on material by Prof. Karen Reid

# cvs: quick refresher

# Software house: what happens inside?

| | |
|---|---|
| Understand the requirements | Project Managers Group |
| Design the program | Architects Group |
| Write the program | Development/Programmer Group |
| Test the program | QA Group |
| Write Documentation | Documentation Group |
| Package/Sell/Market program | Marketing/Sales Group |

# Tools in a Software House

- ✓ Programming Language(s)
- Scripting Language(s)
- Integrated Development Environment (IDE) App
- Profiling Tools
- ✓ Version Control App (e.g. cvs)
- Quality Assurance Framework
- Software Build Management Framework
- Requirements/Feature Tracking App
- Variance Tracking App
- Architecture Tools

# Tools in a Software House

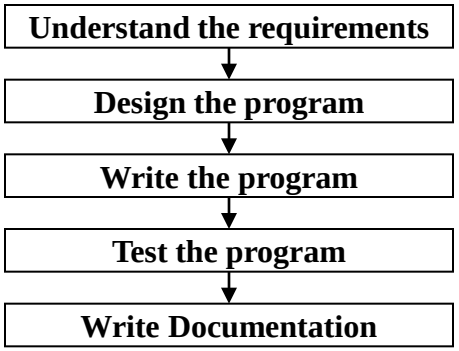
| Tool | Used By |
|---|---|
| **Programming Language** | **Programmers** |
| **Scripting Language** | **Programmers** |
| IDE | Programmers |
| Profiling tools | Programmers, QA |
| Version Control App | Programmers, QA |
| Quality Assurance Framework | Programmers, QA |
| Software Build Management Framework | Programmers |
| Requirements/Feature Tracking App | Managers, QA, Programmers, Architects |
| Variance Tracking App | Programmers, QA, Managers |
| Architecture Tools | Architects, Programmers |

# Tools in a Software House: languages

- Fundamental trade-off
  - How quickly correct code can be written
  - How quickly that code executes
- People are expensive
- Choose where to spend people time:

# What's Agile Development?

- What's the time frame here?
  - Months, weeks, years?




- Read wikipedia article:

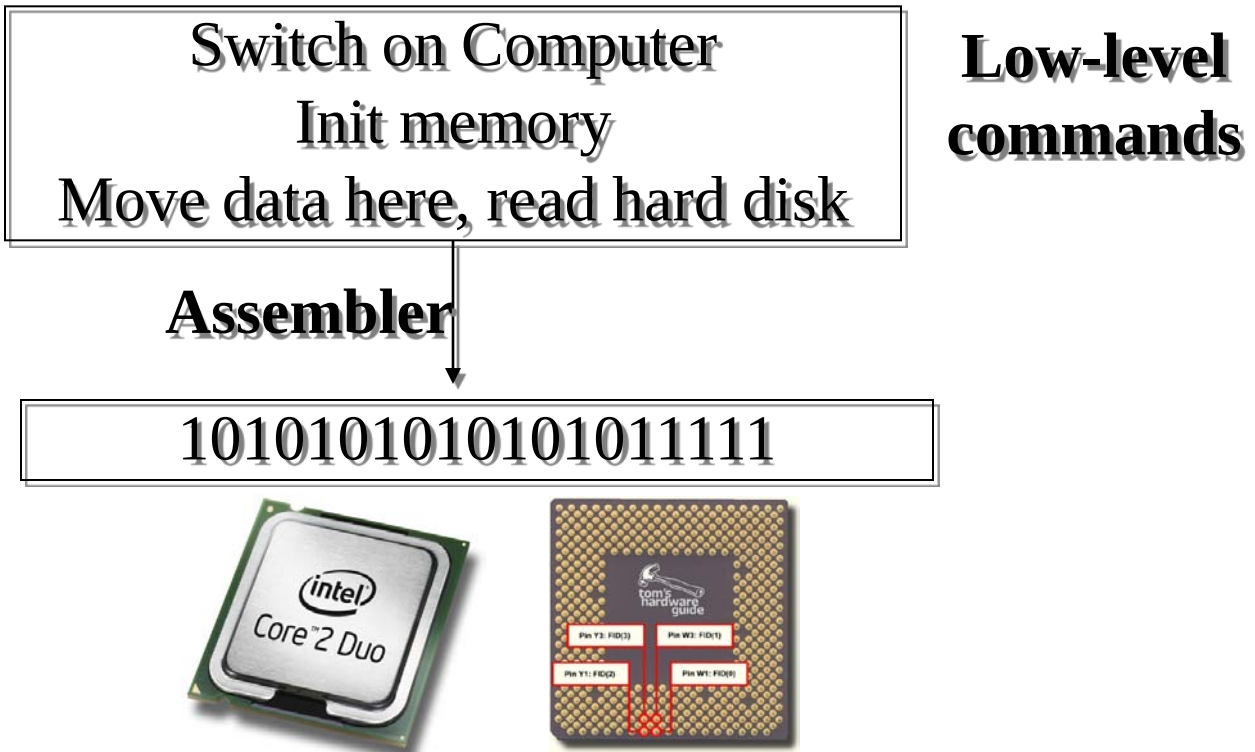
  http://en.wikipedia.org/wiki/Agile_software_development

# Compiled Vs. Interpreted Languages?


# Compiled Vs. Interpreted Languages?

1010101010101011111


# Compiled Vs. Interpreted Languages?



# Compiled Vs. Interpreted Languages?

**Program In High-level Language**
***myprogram.java***

java

```
import java.util.*;
Public static void main(…)
{
}
```

**Compiler**

**Same program in Assembly Language**
***myprogram.asm***

assembly

```
Switch on Computer
Init memory
Move data here, read hard disk
```

**Low-level commands**

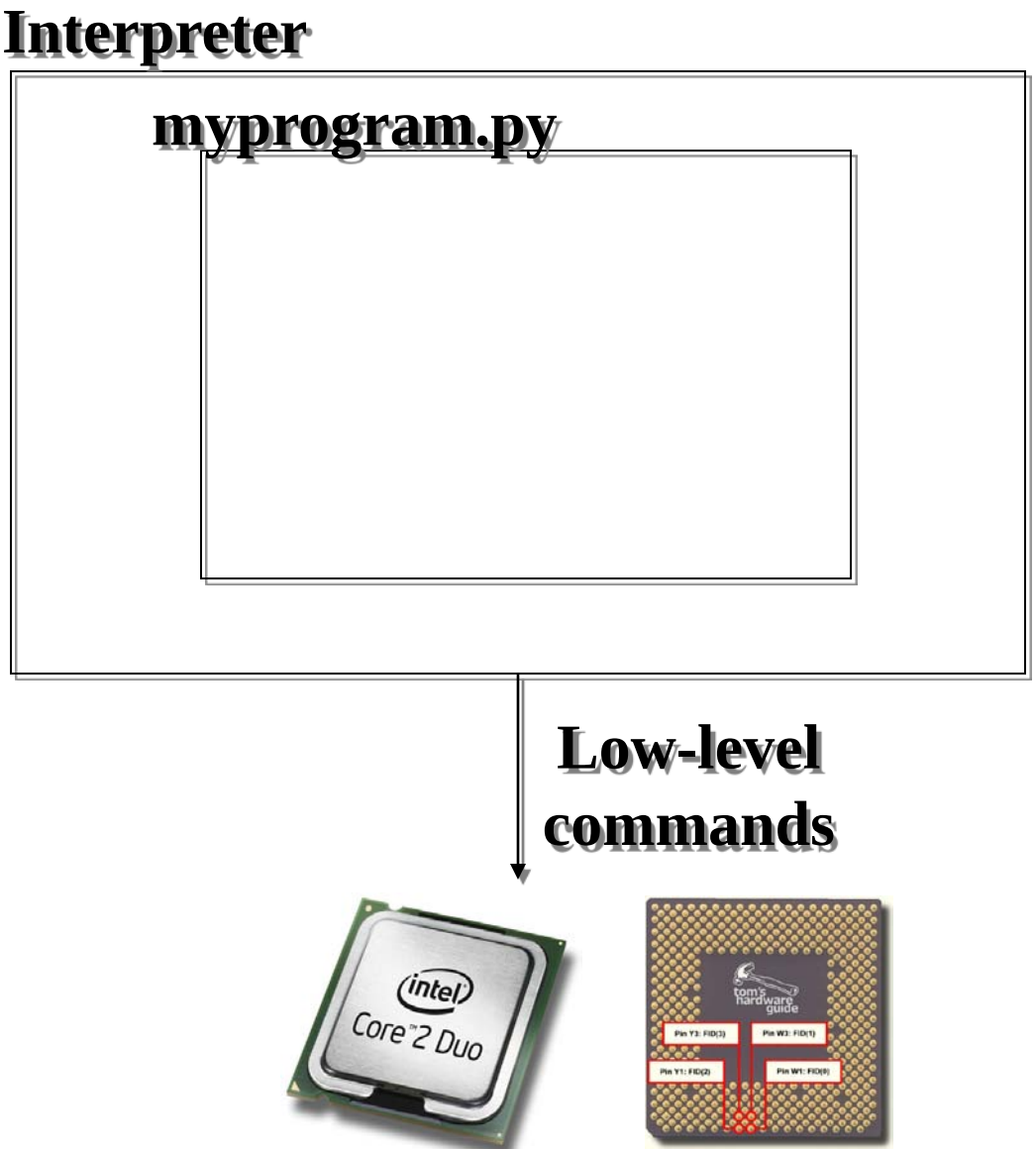
**Assembler**

**Same program in binary**
***myprogram.exe***

1s and 0s

```
1010101010101011111
```


# Compiled Vs. Interpreted Languages?



# Python 101

# What Is Python?


- Created in 1990 by Guido van Rossum
  - While at CWI, Amsterdam
  - Now hosted by centre for national research initiatives, Reston, VA, USA
- Free, open source
  - And with an amazing community
- Object oriented language
  - “Everything is an object”

# Why Python?

- Designed to be easy to learn and master
  - Clean, clear syntax
  - Very few keywords
- Highly portable
  - Runs almost anywhere - high end servers and workstations, down to windows CE
  - Uses machine independent byte-codes
- Extensible
  - Designed to be extensible using C/C++, allowing access to many external libraries
- Agile language

# Example program

- # Run with python helloworld.py
- # Note: no 'main()', no declarations.

- # Assign a string to a variable.
- str = "Hello"

- # Print strings, plus newline.
- print str, "world! "

# Running Python Interperter

- Several options:
  - From command line (good for debugging)
    - python
  - By putting code in a file, and loading it
    - python helloworld.py
  - By making a native executable
    - Unix: Put this at top of file
      - #!/usr/local/bin/python
    - Windows: File association
  - By compiling to a self-contained program
    - Instructions are still actually interpreted

# Python variables

- Variables are just names for values
- Created by use
  - no declarations
- Variables don't have types, but values do
  - x = 123
  - y = "one two three"
  - z = x + y
  - TypeError: unsupported operand types for +

# Define variables before use

- Must give a variable a value before using it
  - Python doesn't try to guess a sensible value

  - # This is the whole program
  - print y
  - NameError: name 'y' is not defined

# Strings

- Use either single or double quotes
  - print 'a', "b", '"c"', '"d"'

    a b "c" 'd’
- Back-quoting converts value to string
  - print "carbon-" + 14

    TypeError
  - print "carbon-" + `14`

    carbon-14

# Numbers and arithmetic

- Numeric types
  - 14 is an integer (32-bit on most machines)
  - 14.0 is a floating point (double – 64 bit)
  - 1+4j is complex (2x64 bit)
- Python borrows C's numeric operators (very like Java's)
  - x = 5 * 4 + 3 # x now 23
  - x -= 10 # x now 13
  - y = x % 3 # remainder is 1

# Booleans

- Like C, so much looser than in Java
- True and False are true and false
- Empty string, 0, and None are false
- (Almost) everything else is true
- Usual Boolean operators (and, or, not)
  - short-circuit
  - return the last thing evaluated rather than 1 or 0
  - “a” or “b” # returns “a”
  - 0 or “b” # returns “b”
  - “a” and “b” # returns “b”
  - “a” and 0 and (1/0) # returns 0

# Comparisons

- Python borrows C comparisons
  - results are always True or False
- Comparisons can be chained together, as in mathematics
  - print -1 < 0 < 1
  - print -1 < 3 < 2

# String operators

- Use + for concatenation and * for multiplication
  - greet = "Hi " + "there"
  - # greet is Hi there
  - jolly = "ho" * 3
  - # jolly is “hohoho”

# Nested statements: if and while

- Use colon and indentation to show nesting

```
a = 3
if a < 0:
    print "less"
elif a == 0:
    print "equal"
else:
    print "greater"
```

```
a = 3
while a > 0:
    print a
    a -= 1
```

# Files

- Use built-in function open() to open a file
  - first argument is a path
  - second is "r" for read, or "w" for write
- Result is a file object
  - input = open("file.txt", "r")
  - output = open("copy.txt", "w")
  - line = input.readline()
  - while line:
    - output.write(line)
    - line = input.readline()
  - input.close()
  - output.close()

# I/O, alternatively

- input = open("file.txt", "r")
- output = open("copy.txt", "w")
- for line in input:
- output.write(line)
- input.close()
- output.close()

# I/O, alternatively alternatively

- input = open("file.txt", "r")
- contents = input.readlines()
- input.close()
- output = open("copy.txt", "w")
- output.writelines(contents)
- output.close()

# Functions: a first pass

- def outside of a class defines a *function*
  - These are classless and objectless methods
- Example: a simple function:
  - # define function
  - def average(x, y):
    return (x + y) / 2.0
  - # use function
  - print average(20, 30)
  - **Output: 25.0**

# The rules for functions

- Define a new function using def
  - Weird: the actually creates a function object, then assigns it to a variable
- Argument names follow in parentheses
  - No types for either return or parameters
- Finish at any time with return
  - Functions without return statements return None

# Scope of variables

- Variables created in functions are local to the function
- x = 123
- def f(arg):
- x = arg
- print "x in f is", x

- f(999)
- print x
- **x in f is 999**
- **123**

# Recursion example

- def fac(n):
  - if (n == 1):
    - return 1
  - else:
    - return n*(fac(n-1))

- Note that this will get ugly if n is not an integer - any suggestions?

# Object-oriented definitions

- Object: instance of a class
- Class: defines possible operations and states
  - Each instance is independent of all others
- Object-oriented languages support:
  - Encapsulation: each instance manages its own state
  - Polymorphism: the ability of the same method call to invoke one of several different methods depending on an object's type
  - Inheritance: define new classes by extending existing ones
  - Reflection: Programs can inspect themselves
    - not part of official OO definition, but useful

# Creating a class in Python

- class Glorp:
  - *... stuff ...*
  - def getValue(self):
    - return self.value
  - *... more stuff ...*

# Creating a class in Python

- A source file may define any number of classes
  - Start definition using the `class` keyword followed by name of class
  - Contents of class are indented

# Methods in Python

- Methods
  - Define using def; parameter list follows in parentheses
  - Indent body of method
  - No return type, no types for parameters
  - Finish at any time with return
    - Methods without return statement return None
  - Instance methods must have at least one parameter
    - The first parameter represents the particular instance of the class (i.e. this object)
    - Convention: call this parameter self (sort of like this in Java)
    - self is not given as an argument when this method is called (except within the class itself)

# Creating a class in Python

- Class members (cont'd):
  - Methods (cont'd):
    - methods can be called anything
    - but method names beginning and ending with double underscore mean special things
      - For example, __init__ is the class's constructor
  - Instance Variables:
    - Created by assignment to self.varname within a method
    - No declaration, just use!

# Simple Counter class

```
class Counter:
 def __init__(self):
     self.value = 0
 def step(self):
     self.value += 1
 def current(self):
     return self.value

# Testing the class definition:
c = Counter()
print "initial value", c.current()
c.step()
print "after one step", c.current()
c.nonExistentMethod()
```

**Output:**

```
initial value 0
after one step 1
AttributeError Counter instance
has no attribute
'nonExistentMethod'
```

# Encapsulation

- Python does not enforce encapsulation
  - No equivalent of *protected* or *private*
  - Anyone can happily execute
    - obj.value = "abc"
- Generally a bad idea
- Remember: the things that make it easy to write code quickly in Python make it harder to maintain.

# Inheritance

- Extend a parent class to create a child class
  - put parent's name in parentheses after child's
- Must invoke parent's constructor explicitly
  - Unlike Java, it can be called like any other method

```
- from counter import Counter
- class Stepper(Counter):
-   def __init__(self):
-     Counter.__init__(self)
-   def reset(self):
-     self.value = 0
```

# Example: overriding Counter

- Methods defined in child take precedence over those defined in parent

# Example: overriding Counter

```
class Incrementer(Counter):
  def __init__(self, increment=1):
    Counter.__init__(self)
    self.increment = increment
  def step(self):
    self.value += self.increment

# Test the class:
obj = Counter()
for i in range(2):
  obj.step()
  print "Counter (parent) ", i, ":",obj.current()
obj = Incrementer(3)
for i in range(2):
  obj.step()
  print "Incrementer (child) ", i, ":",obj.current()
```

***Output:***
***Ctr (parent) 0 : 1***
***Ctr (parent) 1 : 2***
***Incrnter (child) 0 : 3***
***Incrnter (child) 1 : 6***

# Class members

- Variables defined directly in the class belong to the class
  - Not related to any self instance
  - Like static in Java
- Nothing equivalent for methods
  - Concept is easy
  - Coming up with a simple syntax has proven difficult
  - We'll see later that it is possible to have methods that are independent of classes: *functions*

# Example

- A class variable:
- class Tracker:
- numCreated = 0
- def __init__(self):
- Tracker.numCreated += 1

- t1 = Tracker()
- t2 = Tracker()
- print Tracker.numCreated

- ***Output: 2***

# Example: __add__

- Specially-named methods associated with every arithmetic operator
  - __add__ for +
  - __mul__ for *
  - If x is an object, x+2 is really x.__add__(2)
- Operators also have right-hand methods
  - *E.g.* __radd__, __rmul__
  - So 2+x is x.__radd__(2)
- Execution order for a+b is:
  - If a has a method __add__, call a.__add__(b)
  - If b has a method __radd__, call b.__add__(a)
  - Else use Python's built-in default

# Example: __add__

```
# modInt: only has values in the range 0..base-1
class modInt:
    def __init__(self, base):
        self.base = base
        self.value = 0
    def __add__(self, other):
        self.value += other
        self.value %= self.base
        return self
    def val(self):
        return self.value

if __name__ == "__main__":
    a = modInt(3)
    for i in range(5):
        a = a + 1
        print a.val(),
```

***1 2 0 1 2***

# Some other __special__ methods

| | |
|---|---|
| __str__(self) | Convert to string |
| __getitem__(self, index) | Indexing([]) |
| __contains__(self, item) | Membership test (in) |
| __len__(self) | Length (len) |
| __int__(self) | Convert to integer (int) |

# Python and Java: differences

- Java:
  - each file is a class
  - execution starts with the main method of the class that is loaded first
- Python:
  - no need for classes in a file
  - execution starts with the first executable statement in a file
  - The execution of a class indentation block is storing the set of statements that define the class
  - Similarly, a def indentation block inside a class stores the definition of a method

# Creating and loading modules

- Any Python file can be loaded as a module using import module
  - File called xyz.py becomes module xyz
- Statements are executed as module loads
  - Libraries typically just define constants and functions
- Module contents referred to as module.content
  - E.g. sys.argv
- Can also use
  - from module import name1, name2
  - from module import *

# Module: example

- # stuff.py
- value = 123
- def printVersion():
- print "Stuff Version 2.2”

---

- # loader.py
- import stuff
- print stuff.value
- stuff.printVersion()

- *$ python stuff.py*
- *$ python loader.py*
- ***123***
- ***Stuff Version 2.2***

# Modules: loading versus running

- Special variable __name__ is module's name
  - Set to "__main__" when run from the command line
  - Set to the module's name when loaded by something else
- Often used to include self-tests in module
  - Tests use assert when module run directly

# Module: self-test

```
class C:
def double(self,val):
    return val * 2


if __name__ == '__main__':
    print "testing C.double"
    c = C()
    assert c.double(0) == 0
    assert c.double('a') == 'aa'
    assert c.double([1]) == [1, 1]
    print "tests passed"
```