CSCI: 4500/6500 Programming Languages

Scripting Languages

Chapter 13

What is Scripting?

- Yes! The name comes from written script such as screenplay, where dialog is repeated verbatim for every performance

Origin of Scripting Languages

- Scripting languages originated as *job control languages*
  - » 1960s: IBM System 360 had the Job Control Language (JCL)
  - » Scripts used to control other programs
    - – Launch compilation, execution
    - – Check return codes
- Scripting languages evolved in the UNIX world
  - » Shell programming: AWK, Tcl/Tk, Perl
  - » Scripts used to combine component ("programming in the large")
    - – *Gluing* applications [Ousterhout 97]

Glue that puts components together

Higher-level Programming

- Scripting languages provide an even higher-level of abstraction than languages we have seen previously
  - » The main goal is programming productivity
    - – Performance is a secondary consideration
  - » Modern SL provide primitive operations with greater functionality
- Scripting languages are *usually* interpreted
  - » Interpretation increases speed of development
    - – Immediate feedback
  - » Compilation to an intermediate format is common (e.g., Perl).

Contemporary Scripting Languages

- Unix shells: sh, ksh, bash
  - » job control

- Perl
  - » Slashdot, bioinformatics, financial data processing, CGI
- Python
  - » System administration at Google

- Ruby
  - » Various blogs, data processing applications

- PHP
  - » Yahoo web site

- JavaScript
  - » Google maps

What is Scripting Language Again?

- Favor rapid development over efficiency of execution
  - » Code can be *developed* 5-10 times faster in a scripting language but will *run slower* at a 10th/20th of the speed of a systems language such as C, C++ [Ousterhout]
- Coordinates multiple programs
  - » Strong at communicating with program components written in other languages
- Hard to put a finger on -- difficult to define *exactly* what makes language a scripting language

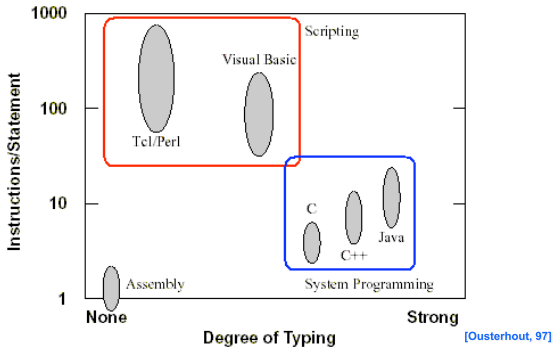
Designed to support "quick programming"

Common Characteristics

- Batch and interactive use
- Economy of expression (readability?)
- *Weakly* typed
  - » meaning is *inferred* (no declaration required)
  - » => less error checking at compile time
    - – run time checking is less efficient (strict run type checking by Python, Ruby, Scheme).
  - » Increases speed of *development*
    - – more flexible
    - – fewer lines of code
- High-level model of underlying machines
- Easy access to other programs
- Sophisticated pattern matching and string manipulation
- High-level data types (sets, bags, lists, dictionaries and tuples)

We will talk more about types later…

"Typing" and Productivity



[Ousterhout, 97]

Design Philosophy

Often people, especially computer engineers, focus on the machines. They think:

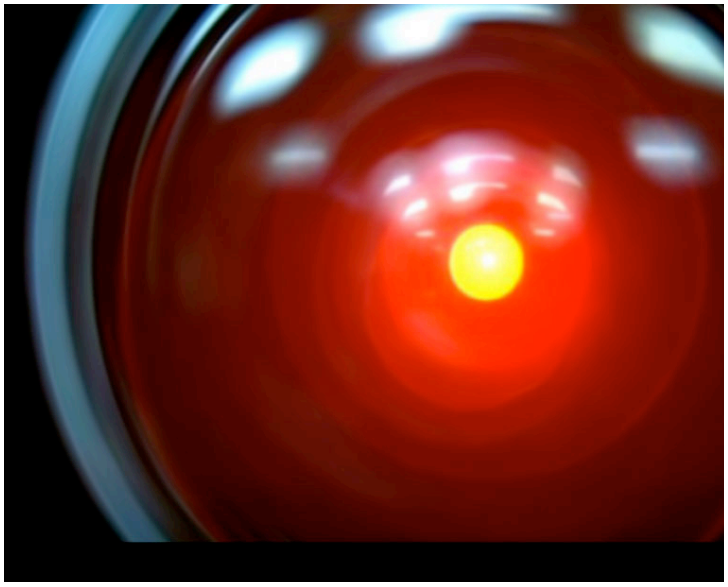
"By doing this, the machine will run faster.

By doing this, the machine will run more effectively.

By doing this, the machine will something something something."

They are focusing on the machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.

Yukihiro "Matz" Matsumoto
Creator of Ruby

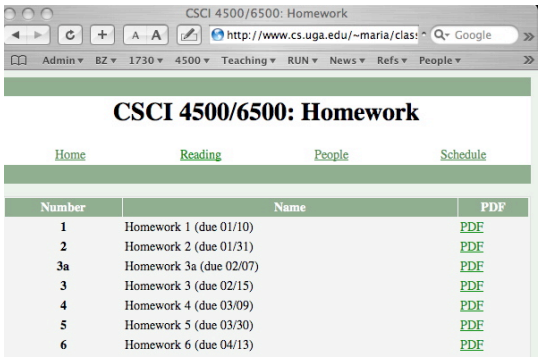


Application Domains

- Shell scripts
- Macro
- Application specific
- Web programming
- Text processing
- Extension/Embedded
- Others

http://www.cs.uga.edu/~maria/classes/4500-Spring-2006/4500-hw.html

We will use Ruby here, but easy (and similar) in most scripting languages



Demo: Getting due dates of homework

- What if I don't want to go to the web site to see if I have CSCI 4500/6500 homework?
- What if I don't want to launch a heavy duty web browser?
- Write a script to check for me!

```
{saffron} check http://www.cs.uga.edu/~maria/classes/4500-Spring-2006/4500-hw.html
Hwk 6 is due on Thursday, April 13.
Hwk 1 was due on Tuesday, January 10.
Hwk 2 was due on Tuesday, January 31.
Hwk 3 was due on Wednesday, February 15.
Hwk 4 is due on Thursday, March 09.
Hwk 5 is due on Thursday, March 30.
```



```ruby
#!/usr/bin/ruby
require 'uri'; require  'net/http'

uri= URI.parse( ARGV[0] )
h=Net::HTTP.new(uri.host,80)

resp,data = h.get(uri.path)
hwk = {}
if resp.message == "OK"
   data.scan(/Homework (\d*) \(due (\d*)\/(\d*)\)/)\
                              {|x,y,z| hwk[x] = Time.local(2006,y,z)}
end

hwk.each{| assignment, duedate|
  if duedate < (Time.now - 60 * 60 * 24)
     puts "Hwk #{assignment} was due on #{duedate.strftime("%A, %B %d")}."
  else
     puts "Hwk #{assignment} is due on #{duedate.strftime("%A, %B %d")}."
  end
 }
```

"Shebang"

useful libraries

Powerful regular expression support

Associative arrays: Keys & Values

```
#!/usr/bin/ruby
require 'uri'; require  'net/http'

uri = URI.parse( ARGV[0] )
h = Net::HTTP.new(uri.host,80)

resp,data = h.get(uri.path)
hwk = {}
if resp.message == "OK"
   data.scan(/Homework (\d*) \(due (\d*)\/(\d*)\)/)\
                              {|x,y,z| hwk[x] = Time.local(2006,y,z)}
end

hwk.each{| assignment, duedate|
  if duedate < (Time.now - 60 * 60 * 24)
     puts "Hwk #{assignment} was due on #{duedate.strftime("%A, %B %d")}."
  else
     puts "Hwk #{assignment} is due on #{duedate.strftime("%A, %B %d")}."
  end
 }
```

String processing

"Shebang"

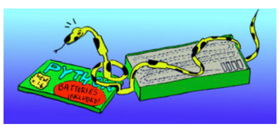
- In Unix systems, shebang tells the OS how to evaluate an executable text file.
  - Shebang: sharp bang or haSH bang, referring to the two typical UNIX names of the two characters.

doit:
```
#! interp-path
prog-text
```
```
> ./doit args
```
→
```
> interp-path doit  args
```

- Advantages: Don't need file extensions, program looks built-in, and can change implementation transparently.

Large Standard Library

- Date, ParseDate
- File
- GetoptLong: processing command line switches
- profile: automatic performance profiling
- BasicSocket, IPSocket, TCPSocket, TCPServer, UDPSocket, Socket
- Net::FTP, Net::HTTP, Net::HTTPResponse, Net::POPMail, Net::SMTP, Net::Telnet
- CGI: cookies, session management
- Threads
- Matrix



Contributing users

- Ruby Application Archive (RAA)
  - http://raa.ruby-lang.org/
  - 144 library categories, 833 libraries available
  - eg: URI library, database access
- Comprehensive Perl Archive Network (CPAN)
  - http://www.cpan.org/
  - 8853 Perl modules from 4655 authors
  - "With Perl, you usually don't have to write much code: just find the code that somebody else has already written to solve your problem."

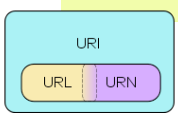
Example: URI and HTTP Libraries

```
require 'uri'; require 'net/http'

uri = URI.parse(ARGV[0])
h = Net::HTTP.new(uri.host,80)

resp,data = h.get(uri.path)
```

Require clauses cause Ruby to load named libraries.



Street address & Name (identity)

```
URI Syntax (Uniform Resource Identifier): URL & URN ( both )
 http, ftp, mailto.
URL: <protocol>:// <host> [:<port>] [<path> [? <query>]]
http://user:pass@example.com:992/animal/bird?species=seagull#wings
\__/  \________/ \_________/\__/\__________/\______________/\____/
|          |            |        |        |                |              |
protocol  login    hosts    port    path          query       anchor/fragment

URN: urn:<namespace>:<string>    if the books is a file (URL: file path file://home/maria/book.pdf)
urn:isbn:nnnn-nnn-nnn
```

Example: URI and HTTP Libraries

```
require 'uri'; require 'net/http'

uri = URI.parse( ARGV[0])
h = Net::HTTP.new(uri.host,80)

resp,data = h.get(uri.path)
```

URI.parse converts argument string into uri object, with host and path component (and more)

Example: URI and HTTP Libraries

```
require 'uri'; require 'net/http'

uri = URI.parse( ARGV[0] )
h = Net::HTTP.new(uri.host,80)

resp,data = h.get(uri.path)
```

Net::HTTP.new creates an http connection object, ready to converse with the specified host on the indicated port.

Example: URI and HTTP Libraries

```
require 'uri'; require 'net/http'

uri = URI.parse( ARGV[0] )
h = Net::HTTP.new( uri.host,80 )

resp,data = h.get( uri.path )
```

h.get asks to retrieve the headers and content of the given path from the site associated with h. It returns response code and the payload data

Strings

- Strings are just objects:

  `"simon".length` yields `5`

- Strings can include expressions with # operator:

  `"3 + 4 = #{3 + 4}"` yields `"3 + 4 = 7"`

- Plus operator concatenates strings:

  `"Simon" + "Cowell"` yields `"Simon Cowell"`

- Many more operations (more than 75!).

Powerful regular expressions

- Regular expressions are patterns that match against strings, possibly creating bindings in the process. Uses greedy matching.
- In Ruby, regular expressions are objects created with special literal forms:

  `/reg-exp/` or `%r{reg-exp}`

- Examples:

  `/arr/` matches strings containing `arr`
  `/\s*\|\s*/` matches a `|` with optional white space

Simple Matches

All characters except `. | () [ \ ^ { + $ * ?` match themselves	
`. | () [ \ ^ { + $ * ?`	Precede by `\` to match directly
`.`	Matches any character
`[characters]`	Matches any single character in [...] May include ranges; Initial `^` negates
`\d`	Matches any digit
`\w`	Matches any "word" character
`\s`	Matches any whitespace
`^`	Matches the beginning of a line
`$`	Matches the end of a line

Compound matches

`re*`	Matches 0 or more occurrences of re.
`re+`	Matches 1 or more occurrences of re.
`re{m,n}`	Matches at least m and no more than n occurrences of re.
`re?`	Matches zero or one occurrence of re.
`re1 | re2`	Matches either re1 or re2
`(...)`	Groups regular expressions and directs interpreter to introduce bindings for intermediate results.

Bindings

Matching a string against a regular expression causes interpreter to introduce bindings:

`$``	Portion of string that preceded match.
`$&`	Portion of string that matched.
`$'`	Portion of string after match.
`$1, $2,…`	Portion of match within *i* th set of parentheses.