CS3157: Advanced Programming

Lecture #2

Sept 12

Shlomo Hershkop

shlomo@cs.columbia.edu

Outline

- Feedback
- Introduction to Perl review and continued
- Intro to Regular expressions

- Reading
  - Programming Perl pg 1-45

Feedback from last class

- More computer science background
- Better board presentation
  - Will move examples to laptop screen easier to follow and illustrate.
  - You will need to let me know if you need more time to read something presented.
- Very varied skill set, a lot of programming experience and backgrounds
  - Hardware
  - Software
  - educational

Last plug

- One of the points of computer science is to teach you how to think, learn, and analyze computational related information.
- Each course is a tool which you will collect for later use.
- Lots of tools in this course, since we will be covering many different topics and subjects.

Welcome again

- Perl
  - History
  - Version 5.6+
- What is it?
  - Scripting language
  - Aims to be a USEFUL language
  - Base + tons of libraries
  - Both a compiler and byte code executable
- Where to get it?
  - cpan.org
  - www.activestate.com/Products/ActivePerl/

Conventions

- Something.pl
  - version: >perl –v
  - Location: >which perl
- First line of script
  - Linux: #!/usr/bin/perl
  - Windows: #!c:\perl\bin
- comment lines
  - Hash (#) to the end of the line
- Can make the perl script executable (chmod +x command).

Structure

- Whitespace
  - only needed to separate terms
  - all whitespace (spaces, tabs, newlines) are treated the same
  - Use them to make the code look nice, easier to look over
- Semicolons
  - every simple statement must end with one
  - except compound statements enclosed in braces (i.e., no semicolon needed after the brace)
  - except final statements within braces
- Declarations
  - only subroutines and report formats need explicit declarations
  - otherwise, variables in perl are like in shell scripts — they are declared and initialized all at once

Variables

- Variables
  - Data dependant
  - No space
  - names consist of letters, digits, underscores; up to 255 chars
  - CASE SENSITIVE
  - Should start with letter or underscore
  - Initialized variables have the value of **`undef`**

Data types

- scalars ($)
- arrays (@)
- hashes (%)
- subroutine(&)
- typeglob(*)

Scalars

- Starts with $
  – $first
  – $course
- int, real, string
    - 234
    - -89
    - 36.34
    - “hello world”
- Context dependant
  – $name = “shlomo”;
  – $name = 123;

Arrays

- Starts with @
- Order list of scalars
- `@class3157 = (“shlomo”,”weijen”,”edward”);`
- To reference elements, use the variable name with a dollar in front and subscript
- **`$class3157[0]; #is shlomo`**
- What is
  1) `$class3157[-1];`
  2) `$a = @class3157;`

Hashes

- name/values pairs
- `%phonelist = {shlomo=>718, barry=>345};`
  or
  `%phonelist = {“shlomo”,718,”barry”,345};`
- Use the name to find the value
  `$phonelist{“shlomo”}  #is 718`
- Any other ideas for this?

Variables II

- Local
- Global
- Special

Programming statements

- simple statements are expressions that get evaluated
- they end with a semicolon (;)
- a sequence of statements can be contained in a block, delimited by braces ({ and })
- the last statement in a block does not need a semicolon
- blocks can be given labels:

```
myblock: {
print "hello class\n";
}
```

Conditional Statements

1. simple if

   if (expression) {block} else {block}

2. unless

   unless (expression) {block} else {block}

3. compound if

if (expression1) {block}
elsif (expression2) {block}
...
elsif (expressionN) {block}
else {block}

Loops

- while
- for
- foreach

while

syntax:
while (expression) {block}

example

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;
$i=0;

while ( $i < $a ) {
      print "i=",$i," b[i]=",$b[$i],"\n";
      $i++;
}
```

for

syntax:
for (expression1; expression2; expression3) {block}

example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;
for ( $i=0; $i<$a; $i++ ) {
      print "i=",$i," b[i]=",$b[$i],"\n";
}
```

foreach

syntax:
  foreach var (list) {block}

example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;

foreach $e (@b) {
     print "e=",$e,"\n";
}
```

Controlling loops

- next
  within a loop allows you to skip the current loop iteration
- last
  allows you to end the loop

- test3.pl

Modifiers

- you can follow a simple statement by an if, unless, while or until modifier:

statement *if* expression;
statement *unless* expression;
statement ***while*** expression;
statement ***until*** expression;

- example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;

print "hello world!\n" if ($a < 10);
print "hello world!\n" unless ($a < 10);
#print "hello world!\n" while ($a < 10);
print "hello world!\n" until ($a < 10);
```

Operators

you can follow a simple statement by an if, unless, while or until modifier:

- statement if expression;
- statement unless expression;
- statement while expression;
- statement until expression;

example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;

print "hello world!\n" if ($a < 10);
print "hello world!\n" unless ($a < 10);
print "hello world!\n" until ($a < 10);

#print "hello world!\n" while ($a < 10);
```

Sample #1

```
#!c:\perl\bin
($first,$last) = &getname();
print "First is $first";


#return the fill name as a string
sub getname(){
return "shlomo hershkop";
}

#return name split
sub getname(){
    return ("shlomo","hershkop");
}
```

Reserved variables

there’s a (long) list of global special variables...
a few important ones:

$_ = default input and pattern-searching string

example:

```
#!/usr/bin/perl
@b = (2,4,6,8);

foreach (@b) {
        print $_,"\n";
}
```

Reserved II

- $/ = input record separator (default is newline)
- $$ = process id of the perl process running the script
- $< = real user id of the process running the script
- $0 = (0=zero) name of the perl script
- @ARGV = list of command-line arguments
- %ENV = hash containing current environment
- STDIN = standard input
- STDOUT = standard output
- STDERR = standard error

Operators

- unary:
  1. ! : logical negation
  2. - : arithmetic negation
  3. ˜ : bitwise negation
- arithmetic
  1. +,-,*,/,% : as you would expect
  2. ** : exponentiation
- relational
  1. >, <=, <=, <= : as you would expect
- equality
  1. ==, != : as you would expect
  2. <=> : comparison, with signed result:
  3. returns -1 if the left operand is less than the right;
  4. returns 0 if they are equal;
  5. returns +1 if the left operand is greater than the right

Operators II

assignment, increment, decrement

- =
- +=, ++
- -=, --
- *=, **=, /=, %=
- &&=, ||=

just like in C

Working with files

- open(FILEHANDLE, filename); : to open a file for reading
- open(FILEHANDLE, >filename); : to open a file for writing
- open(FILEHANDLE, >>filename); : to open a file for appending
- use || warn print "message"; or || die print "message"; for error checking
- print FILEHANDLE, ...;
- close(FILEHANDLE);

example:

```
#!/usr/bin/perl
open( MYFILE,">a.dat" );
print MYFILE "hi there!\n";
print MYFILE "bye-bye\n";
close( MYFILE );
```

Example II

```
#!/usr/bin/perl
open( MYFILE2,"b.dat" ) || warn "file not
  found!";

open( MYFILE2,"a.dat" ) || die "file not
  found!";

while ( <MYFILE2> ) { print "$_\n" }

close( MYFILE2 );
```

Subroutine

- syntax for defining:

sub name {block}
sub name (proto) {block}

- where proto is like a prototype, where you put in sample arguments

- syntax for calling:

name(args);
name args;

- any arguments passed to a subroutine come in as the array @_

Built in functions

- chomp $var
- chomp @list

removes any line-ending characters

- chop $var
- chop @list

removes last character

- chr number

returns the character represented by the ASCII value number

- eof filehandle

returns true if next read on filehandle will return end-of-file

- exists $hash{$key}

returns true if specified hash key exists, even if its value is undefined

- exit

exits the perl process immediately

More built in

- getc filehandle

reads next byte from filehandle

- index string, substr [, start]

returns position of first occurrence of substr in string, with optional starting position; also

- rindex which is index in reverse

- opendir dirhandle, dirname

opens a directory for processing, kind of like a file; use readdir and closedir to process

- split /pattern/, string [, limit]

splits string into a list of substrings, by finding delimiters that match pattern;

example: split /([-,])/,"1-10,20"; returns (1, '-', 10, ',', 20)

- substr string, pos [, n, replacement]

returns substring in string starting with position pos, for n characters

Strict mode

- This isn't about the midterm
- Tells perl to only allow variable you explicitly create in your programs
  - Prevents typos
  - Easier to maintain
  - Less work for interpreter

Perl References

- there are lots and lots of advanced and funky things you can do in perl; this is just a start!

here's a quick start reference:

- http://www.comp.leeds.ac.uk/Perl/
- http://www.perl.com
- http://www.perl.com

function reference list is here:

- http://www.perldoc.com/perl5.6/pod/perlfunc.html

Regular Expressions

- simplest regular expression is a literal string
- complex regular expressions use *metacharacters* to describe various options in building a pattern.

\	escapes the character immediately following it
.	matches any single character except newline
^	matches at the beginning of a string
$	matches at the end of a string
*	matches the preceding element 0 or more times
+	matches the preceding element 1 or more times
?	matches the preceding element 0 or 1 times
{ ... }	specifies a range of occurrences for the element preceding it
[...]	matches any one of the class of characters in the brackets
(...)	groups expressions
|	(pipe) matches either the expression before or after it

Basic

- The most basic match is:
  - $string =~ m/sought_text/;
  - Will return true if sought_text is part of string, false otherwise
  - Perl assume m/???/ when use /???/

```
#!c:\perl\bin

$name = "shlomo hershkop";

if($name =~ /lom/){
    print "have found match\n";
}
else{
    print "no match found\n";
}
```

What about?

```
$name = "shlomo hershkop";

if($name =~ m/^her/){
   print "have found match\n";
}
else{
   print "no match found\n";
}
```

Basic II

- Will match case sensitive unless told not to by matching operators

```
If($name =~ /shlomo/i ){
      something
}
```

Pattern attributes

- =˜ binds a scalar to a pattern match, substitution or translation
- !˜ just like above, except that the return value is negated in the logical sense
- operators:
  - m/pattern/gimosx : match
- g = match globally (all instances)
- i = do case insensitive matching
- note that first m is optional
  - s/pattern/replacement/egimosx : search
- e = evaluate right side as an expression
- g = match globally (all instances)
- i = do case insensitive matching
  - y/pattern1/pattern2/cds : translate
- c = complement pattern1
- d = delete found but unreplaced characters
- s = squash duplicate replaced characters

Example

```
#!/usr/bin/perl
$s = "hello world";
print '$s=[',$s,"]\n";
if ($s =˜ m/x/)
    { print "there's an x in ",$s,"\n" }
else
    { print "there isn't\n" }

if ($s =˜ m/L/i)
    { print "there's an l in ",$s,"\n" }
else
    { print "there isn't\n" }
```

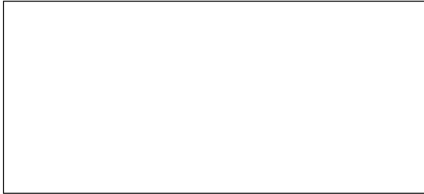
Example 2

```
#!/usr/bin/perl
$s = "hello world";

print '$s=[',$s,"]\n";

$t = ($s =~ s/l/x/g);

print '$t=[',$t,"]\n";
print '$s=[',$s,"]\n";
```



Example 3

```
#!/usr/bin/perl
$s = "hello world";
print '$s=[',$s,"]\n";
$u = ($s =~ y/l/o/c);
print '$u=[',$u,"]\n";
print '$s=[',$s,"]\n";
```

Next time

- Read up on regular expressions