

CS306: Introduction to Perl

Section #11: Stocking Your Perl Toolbox

U. of Alabama at Birmingham
Dept. of Computer & Information Sciences

Slide 1

Section #11: Stocking Your Perl Toolbox

Techniques To Expand Your Perl-Fu

Slide 2

grep() - Select Items From A List

- You can use the `grep()` function to define a rule and select only those items from a list that match the rule
- # grab only the odd numbers in @numbers

```
my @odds = grep { $_ % 2 } @numbers;
```
- # grab only the Subject: lines from a file

```
my @subjects = grep /^Subject:/, <IN>;
```

Slide 3

map() - Transform Each List Element

- Use the `map()` function when you want to make a change to each element of a list, but preserve the initial data.
- @names = qw/Fran Katie Matt Caroline/;
shout out the names to the screen

```
map {printf "%s!\n", uc($_)} @names;
```



```
FRAN!  
KATIE!  
MATT!  
CAROLINE!
```

Slide 4

More grep() and map() examples

- @years = grep {
 (\$_ % 4 == 0) &&
 !((\$_ % 100 == 0 && \$_ % 400 != 0)
} 1900..3000;
- @instock = grep {checkstock(\$_)} @partnumbers;
- @spanish = map { translate(\$_) } @english;
- # not always 1-to-1 mapping
 @words = map {split} @lines;

Slide 5

Finding substrings

- Use index() to find a string within a larger string
- index() returns the position as an integer. You can think of it as “the number of characters to skip over before the smaller string starts”
- \$where = index(\$big, \$small);
- my \$str = “Hello World”;
 my \$where = index(\$str, “orl”); # returns 7
- my \$str = “is it, or is it not?”;
 my \$wherelast = rindex(\$str, “it”); # returns 13

Slide 6

Grabbing Substrings

- \$sub = substr(\$string, \$initial_position, \$length);
- zero-based initial position
- \$str = “Firewall”;
 \$heat = substr(\$str,0,4);
 \$brick = substr(\$str,4,4);
- Omit last parameter to grab to end of string
- \$brick = substr(\$str,4);
- You can modify the string if it's a variable
- substr(\$str,4) = “ engine”; # “Fire engine”

Slide 7

Commenting Patterns

- Use /x modifier to tell Perl to ignore most whitespace and consider # the start of a comment
- ```
/^Subject: # Line starts Subject:
\s+ # then 1+ whitespace
([Rr][Ee]:\s+)+ # then 1+ 'RE: '
 # allowing 1+ ws after :
(.*) # then grab rest of line $2
/x # x for nice formatting
```
- You have to escape literal ' ' and '#' characters

Slide 8

## Greedy Matching

- We've already seen these quantifiers
  - \*
  - +
- We know that these will “eat up” as much as the string as possible while still keeping the match valid - we call this greedy matching
- It gobbles up everything, then grudgingly gives back a character at a time, to see if that helps

Slide 9

## Greedy Matching, Continued

- “Campbell Hall is on University Blvd”
- `/Camp(.*)Hall/` - the `.*` prefers to match as much as possible
- First we match the `Camp`. Then the `.*` eats the rest of the string, and then it gives one character at a time to see if the `Hall` can match.
- This backtracking can be expensive, and also can lead to surprising results

Slide 10

## Greedy Matching, Continued

- “Campbell Hall is the best Hall of all!”
- `/Camp(.*)Hall/`
- `$1` would have a value of “bell Hall is the best “ which is probably not your intent
- As strings come longer, it becomes more likely that `.+` and `.*` won't backtrack as far as you expect

Slide 11

## Non-Greedy Matching

- Each of the quantifiers has a non-greedy form as well. You request it by adding a `?` at the end.
- `.*` - greedy    `.*?` - non-greedy
- `.+` and `.+?`
- If the match is non-greedy, it prefers to match as few characters as possible

Slide 12

## Non-Greedy Matching, Continued

- “Campbell Hall is the best Hall of all!”
- `/Camp(?:)Hall/`
- Now `$1` is “bell “, which is probably what you intended in the first place

Slide 13

## Non-Greedy Matching, Continued

- `.*?` - zero or more characters, as few as possible
- `.+?` - one or more characters, as few as possible
- `{3,5}` - 3 to 5 characters, as few as possible
- `??` - zero or one characters, preferably zero

Slide 14

## Here-Docs

- Perl has a construct known as a *here-document*
- If you find yourself using lots of print statements with `\n`'s to format simple reports for output, you may want to use a here document
- They start at the current line and terminate on a line that contains nothing other than the terminating string

Slide 15

## Here Documents, Continued

- ```
sub printReport{
    my ($var2, $var2) = @_;
    print <<END_REPORT;
This is a simple report.
The first value is: $var1
The second value is: $var2
END_REPORT
}
```
- The terminating string must start in column 1.
- Any whitespace is treated literally inside of a here doc (that's why there's no indentation)

Slide 16

Indenting Here Documents

- You can get indenting with a trick
- ```
sub printReport{
 my ($var2, $var2) = @_;
 (my $hd = <<END_REPORT) =~ s/^\s+//gm;
 This is a simple report.
 The first value is: $var1
 The second value is: $var2
END_REPORT
 print $hd;
}
```

Slide 17

## Indenting Here Documents, Con't

- ```
sub printReport{
  my ($var2, $var2) = @_;
  print fix (<<END_REPORT);
  This is a simple report.
  The first value is: $var1
  The second value is: $var2
END_REPORT
}

sub fix {
  my $string = shift;
  $string =~ s/^\s+//gm;
  return $string;
}
```

Slide 18

Random Numbers

- `my $random = int(rand(51)) + 25;`
- `rand(51)` generates a random number between 0 and 51 (inclusive of 0, exclusive of 52)
- `int()` returns the integer portion of the passed-in value
- `+ 25` shifts this from 0-50 to 25-75
- The seed is set at program start

Slide 19

Dates and Times

- Getting today's date
`($DAY, $MONTH, $YEAR) = (localtime)[3,4,5];`
- What's that thing called right...^^^^^ here?
- The `localtime` function returns a 9-element list:

- [0]	seconds	0-60
[1]	minutes	0-59
[2]	hours	0-23
[3]	day of month	1-31
[4]	month of year	0-11
[5]	years since 1900	1-
[6]	day of week	0-6 (0 = sunday)
[7]	day of year	1-366
[8]	daylight svngs	0 or 1

Slide 20

Dates and Times, Continued

- localtime returns time in the local timezone. gmtime() returns GMT, same 9 fields
- Number of seconds since the epoch: time()
- Converting from day/month/year to epoch timelocal(), timegm() - in the standard Time::Local module - see docs for usage

Slide 21

Dates and Times, Continued

- Converting from epoch to day/month/year
- Use localtime(\$epochtime) or gmtime(\$epochtime)
- Remember to add 1900 to your year values
- Remember to add 1 to your month values

Slide 22

Date and Time, Continued

- Adding to a Date
- Convert to epoch seconds, and add or subtract a number of seconds as appropriate
- Or, if you have DMYHMS values, you can use the Date::Calc module (more on that later in the week)

Slide 23

Printing Dates and Times

- Use localtime() or gmtime() in scalar context
- \$string = localtime(\$epochseconds);
- \$string gets a string like 'Tue May 26 05:15:20 1998'

Slide 24

Parsing Dates

- You may want to look at the Date::Manip module on CPAN (we'll talk CPAN in the optional lecture this week)

Slide 25

Fine-Grained Timers

- If all the granularity you need is seconds, just use the time() function
- ```
$start = time();
do something here
$end = time();
$duration = $end - $start; # in seconds
```
- For more fine-grained control, use Time::Local from CPAN

Slide 26

## Sleeping

- sleep(\$num\_seconds)
- select(undef,undef,undef,\$timetosleep)  
# here \$timetosleep can be fractional
- select() not supported everywhere. Time::HiRes has a sleep() that can also take fractional values.

Slide 27

## Grabbing Files Like The Unix Shell

- glob() - Grab list of matching filenames from a pattern
- my @allfiles = glob "\*";
- my @txtfiles = glob "\*.txt";
- my @dotfiles = glob ".\*";
  - Note that in UNIX, \* doesn't return files beginning with . because those are considered hidden files

Slide 28

## Another Way To Glob

- Can also use the < > syntax
- `my @allfiles = <*>;`
- `my $dir = "/etc";`  
`my @dirfiles = <$dir/* $dir/.*>;`
- Use globbing as an alternative to getting directory handles with `opendir()`

Slide 29

## Recursive Directory Listing

- If you ever find yourself needing to recursive directory operations, look into Perl's standard module, `File::Find`
- `File::Find` does lots of other nifty stuff, look into it anyway

Slide 30

## Removing Files - `unlink()`

- `unlink()` - just like Unix's `rm` command
- `unlink list`
- `unlink glob "*.o"` - remove all `.o` files in the current directory
- `unlink` returns number of files removed
- Don't get careless with this while logged in as root/Administrator!

Slide 31

## Renaming Files - `rename()`

- `rename old, new`
- `rename "foo.txt", "bar.txt";`
- `rename "/tmp/foo.txt", "foo.txt"; # move file`
- ```
foreach my $file (glob "*.old") {
  my $newfile = $file;
  $newfile =~ s/old$/new/;
  if (-e $newfile) {print "Already exists";}
  rename $file, $newfile
    or die ('rename failed for $file');
}
```

Slide 32

Making and Removing Directories

- `mkdir dirname, mode`
- `mkdir "tempdir", 0755;`
- *mode* is octal - easy to represent groups of 3 bits

- `rmdir list`
- `rmdir glob "junk/*"`
- `rmdir` returns number of dirs removed

Slide 33

Changing Permissions and Owners

- `chmod mode, list`
- `chmod 0600, qw/secret1.txt secret2.txt/;`
- The Unix shortcuts `+x`, `g+w`, etc... don't work
- returns the number of items successfully changed
- `chown user, group, list` to change ownership of a file or files

Slide 34

Getting a File's Basename

- Say you have `/usr/local/bin/foo` and you just need the filename, `foo`
- `my ($fname) = $name =~ s#.*###;`
 - Problem: Assumes Unix-like /
- Use the portable `File::Basename` module instead
- use `File::Basename`;
`my $fname = basename $name;`
- See `File::Spec` to go the other way

Slide 35

Expression Modifiers

- `print "$n is an odd number.\n" if n % 2 == 1;`
- `$i*2 until $i > 1000;`
- `print "Processing data..." unless $quiet;`
- `print "$_ loves Perl!\n" foreach @names;`
- They work just like their traditional counterparts, but no parentheses or braces required.
- They read like English, so can make your code very easy to read.

Slide 36

Naked Blocks

- Sometimes it's useful to declare a naked block.
- The classic example:

```
{  
  local $/; # undef the separator  
  $file = <IN>; #entire file into one scalar  
} # now $/ is back to its previous value
```
- Also good for limiting the scope of lexical variables declared with my.

Slide 37

Logical Operators

- && - and
- || - or
- short-circuit operators
- if (\$n != 0 && \$total / \$n < 5) - right side evaluated only if left is true, and the return value is the return value of the last expression evaluated. Use this to your advantage:
- my \$option = \$ARGV[0] || \$default;

Slide 38

System Calls

- Three basic ways to make system calls
- system()
- exec()
- Backticks

Slide 39

system()

- system "date";
- A child process is launched, which inherits Perl's STDIN, STDOUT and STDERR. So if your system call produces output, that's where it'll go.
- system 'ls -l \$HOME'
 - Notice the switch to ' from " - why?
- Perl blocks and waits for the command to end or be backgrounded with a & (standard Unix shell stuff)

Slide 40

system() Continued

- Multi-parameter mode avoids the shell
- `@args = qw/-cf $filename/;`
`@dirs = qw/hw1 hw2 hw3/;`
`system "tar", @args, @dirs;`
- Now `@args` and `@dirs` are fed directly to the tar program, the shell doesn't get a chance to mangle them.
- More secure: `$filename` could have shell

Slide 41

system()

- The return value of `system()` is the return value of the child process. In Unix, 0 typically means everything was ok.
- This is backwards from normal logic, so this becomes common:
- `!system "rm -rf dir1" or die "Cmd failed.\n";`

Slide 42

exec()

- Similar to `system()`, except `exec()` does not spawn a child. Instead the Perl process itself runs the command.
- The Perl program stops execution
- Any code other than a `die()` or `exit()` after an `exec()` is considered a warning
- Rare to use - if you are picking between this and `system()`, use `system()`

Slide 43

Backquotes

- Use the backquotes when you want to capture and operate on the output of the command
- `my $diskusage = `du -sh --max=1 /`;`
- Backslash escapes and variables behave the same way within `` as they do with ""
- `@diskusage = `du -sh --max=1 /`;`
- Now, `@diskusage` has one line of the output per array element

Slide 44

Processes as Filehandles

- open MAIL, “lmail fran” or die “Can't mail.\n”;
print mail “To: fran@cis.uab.edu\n”;
- open DISK, “du -sh --max=1l” or die;
my \$onedir = <DISK>;
- l in front, you are opening for writing to it. l in back, you are wanting to read from it. Think of where it would be in a Unix command line:
program.pl | mail fran
du -sh --max=1 | program.pl

Slide 45

Command-Line Options

- Getopt::Long – CLI argument processing
MyPerlProg.pl --infile foo.txt --outfile results.txt

```
use Getopt::Long;
my $infile = 'input.txt';
my $outfile = 'output.txt';
GetOptions( "infile=s" => \$infile,
           "outfile=s" => \$outfile);
```

Slide 46