

CS306: Introduction to Perl

Stocking Your Perl Toolbox

Stocking Your Perl Toolbox

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

Techniques To Expand Your Perl-Fu

Slide 1

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

## 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 9

## 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 10

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 11

## 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 12

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 13

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 14

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 15

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 16

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 17

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 18

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 19

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 20

## 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 21

## 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 22

## 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 23

## 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 24

## 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 25

## 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>;`
- `|` in front, you are opening for writing to it. `|` 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 26

## 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 27

## How is Perl Organized?

- Core functions
  - This is what you get for “free” just by saying ‘perl’ on the command line or `#!/usr/bin/perl` at the top of your program
- Standard modules
  - Modules that are so useful they ship with Perl. You use them by saying things like ‘use Data::Dumper’ in your program
- Optional modules that don't ship with Perl

## What is the CPAN?

- The Comprehensive Perl Archive Network
- <http://www.cpan.org/>
- A large collection of Perl software and documentation
- The best place to get perl modules that are not part of the core distribution

## How do I use the CPAN?

- Two ways
  - Manual method – browse to CPAN, find the module you want, download the .gz file and install it
    - perl Makefile.PL, make, make test, make install
    - many require a C compiler
  - Using a package manager
    - CPAN.pm
    - PPM for ActiveState Perl (Windows)

## What's available on the CPAN?

- Everything
- If you can think of a piece of functionality that would be useful, there's a 99.9% chance that it is already there
- For example, there are:
  - 85 modules for working with Email
  - 220 modules for working with Dates and Calendars
  - Hundreds for working with the Internet and WWW

## Using CPAN.pm

- CPAN.pm is a standard module (comes with Perl) that is used to simplify the download and install process from CPAN
- Generally used with command-line perl, like this:
  - `perl -MCPAN -e 'shell'`
  - This says I want to load the CPAN module (-M) and call the 'shell' function (-e).
  - 'shell' is a function in the CPAN module that provides a command-line interface to CPAN

## CPAN.pm Example

- `perl -MCPAN -e shell`

```
perl -MCPAN -e shell
cpan shell -- CPAN exploration and modules installation
(v1.7601)
cpan> install Email::Simple
Running install for module Email::Simple
Running make for C/CW/CWEST/Email-Simple-1.92.tar.gz
Fetching with LWP:
ftp://archive.progeny.com/CPAN/authors/id/C/CW/CWEST
(Continued...)
```

## Installing Modules Without root

- If you don't have root access to the machine, CPAN.pm will not be able to install modules into the system-wide perl module directories (this is something like `/usr/lib/perl5/5.8.3/...`)
- You can still install modules to your home directory for your own use

## CPAN.pm Example

```
CPAN.pm: Going to build C/CW/CWEST/Email-
Simple-1.92.tar.gz
Checking if your kit is complete...
Looks good
[SNIP]
Running make test
[SNIP]
All tests successful.
Running make install
[SNIP]
/usr/bin/make install -- OK
cpan> quit
```

## Installing Modules Without root

- The manual way

```
perl Makefile.PL PREFIX=/home/fran/perl
LIB=/home/fran/perl/lib
and then in your program...
use lib '/home/fran/perl/lib';
use WhateverModuleYouInstalled;
or set the PERL5LIB environment variable...
PERL5LIB=/home/fran/perl/lib
```

## Installing Modules Without root

- The CPAN.pm way

```
cpan> o conf makepl_args 'PREFIX=/home/fran/perl
LIB=/home/fran/perl/lib'
makepl_args PREFIX=/home/fran/perl
LIB=/home/fran/perl/lib
```

```
cpan> install Email::Simple
```

## Networking

- Perl provides all of the same low-level functions and functionality that C does for working with sockets
- Perl also provides IO::Socket modules to make the process a little more user-friendly

## What if I Don't Know What I Need?

- Then search the CPAN
  - <http://search.cpan.org/>
  - <http://cpan.uwinnipeg.ca/htdocs/faqs/cpan-search.html>
- Often there are too many choices! Which one is best? Which one is most popular? Sometimes Google searches help. Sometimes you just learn over time. TIMTOWTDI is a double-edged sword.

## Creating a TCP client

- ```
use IO::Socket;  
$socket = IO::Socket::INET->new(  
    PeerAddr => $remote_host,  
    PeerPort => $remote_port,  
    Type     => SOCK_STREAM, );  
  
print $socket "E.T. is phoning home.\n";  
$answer = <$socket>;  
print $answer;  
close $socket;
```
- ```
$client =
 IO::Socket::INET->new('www.cis.uab.edu:80')
 or die();
```

## Creating a TCP Server

```
• use IO::Socket;
 $server = IO::Socket::INET->new(
 LocalPort => $server_port,
 Type => SOCK_STREAM,
 Reuse => 1,
 Listen => 10,) or die();

while ($client = $server->accept()) {
 $request = <$client>;
 print $client "Hello, E.T. Enjoy Earth\n";
}
close $server;
```

## Learn More About Networking

- Read the Sockets section of Chapter 16 in your textbook (page 437)
- I recommend using the higher-level IO::Socket module instead of the Socket module
- There's a chapter of recipes in the Perl Cookbook
- There are other books written on the topic, as well

## A Forking Server

```
use IO::Socket;
$server = IO::Socket::INET->new(
 LocalPort => 1234, Type => SOCK_STREAM,
 Reuse => 1, Listen => 10)
 or die;
while ($client = $server->accept()) {
 if ($kidpid = fork) {
 close $client;
 next;
 }
 defined $kidpid or die("Can't fork");
 close $server;
 $request = <$client>;
 print $client "Got it.\n";
 exit;
}
close $server;
```

## Dynamic Web Apps With CGI.pm

- CGI is the Common Gateway Interface
- The simplest way to provide interactive applications on the web
- Stateless - works as http requests. Request received from browser, response sent to browser, connection closed, program terminated.
- Perl provides CGI.pm to create CGI scripts

## The CGI Model

1. The web server gets a request for a Perl CGI script.
2. The script is executed, meaning that the Perl interpreter is launched, and the web server hands over the request and data based on the CGI protocol.
3. STDOUT is redirected back through the CGI gateway. The script produces its output and exits. The Perl interpreter terminates.

## CGI.pm Features

- CGI.pm provides a lot of helper functions that produce HTML as output. Many take arguments... `header()`, `h1('heading')`, `p`, `start_form`, `start_html('title')`, etc.... many correspond to the equivalent HTML tag.
- CGI.pm gathers up all the CGI input data (passed by either the GET or POST method) and exposes it through the `param()` function for you to get at it.

## CGI.pm - Simple Example

```
• use CGI qw/:standard/;

print header, start_html('Custom Doc Quoter'),
 h1('Doc Quoter'),
 start_form,
 "Enter a number: ", textfield('num'),
 p,
 submit, end_form, hr;

if (param()) {
 print param(num) . " gigawatts!!! ",
 "The only place to get " . param(num) .
 " gigawatts is a bolt of lightning!!";
}
```

## A General CGI Framework

```
• use CGI qw/:standard/;

Do stuff for ALL HTML pages here...
Setup header, page title, etc...

if (param()) {
 # User just submitted the form
 # process the data and put up a response page
} else {
 # User is surfing to the page for the
 # first time
 # show them a fill-in form/options menu/etc...
}
```

## Sticky Forms

- Sticky forms mean that the form fields are pre-populated with their previous values if you revisit them.
- The CGI.pm html shortcuts like `textfield()` provide this for free. Can be especially useful if you are going back to a form to ask a user to correct a mistake - they don't have to type everything again.

## Client-Side Data Persistence

- Generate hidden form fields with CGI.pm's `hidden()` function. Read the perldoc.
- Safe and easy because each client then hands the data back in on the next request, so you know exactly who it is coming from.

## Passing Data Between Pages

- Since HTTP is stateless and the Perl interpreter ends, you can't keep variables in memory across multiple page submissions.
- Two strategies:
  - Embed the data back into the page in hidden form fields. Simpler, if a bit clunky.
  - Store the data server-side in some persistent form. Harder, because you have to differentiate between multiple clients with a session key or similar.

## Server-Side CGI Data Persistence

- One way to do server-side persistence is to use CGI.pm's ability to set a cookie in the browser.
- Generate a unique ID, store it in a cookie, and then check for that cookie every time the script is hit.
- If cookie found, can then retrieve data previously stored on disk tagged with the unique key
- Requires user to accept cookies

## Some Notes on CGI

- The CGI script is called with a URL like `http://www.cis.uab.edu/fran/sample.cgi`
- The web server has to allow the `fran/` web directory to run CGI scripts
- The web server must be configured to recognize the `.cgi` extension as a CGI script
- `sample.cgi` must be set to be executable by the user running the web server (mode 755)

## Debugging CGI.pm Scripts

- If you run them from the command-line, they allow you to pass `name=value` pairs to represent the incoming data.
- Hit `Ctrl-D` when you are done setting input data, and the script will run. You'll see what it would have sent to the browser.

## More CGI.pm Usage Notes

- Make sure you speak HTML. Browsers don't understand `\n`, they understand `<br>`
- Always print out a proper header (and only one). It's generally appropriate to do this near the top of your script, not inside each case.

## More Debugging

- use `CGI::Carp qw(fatalsToBrowser);`
- This will redirect errors to the browser for easier debugging if you do not have access to the web server's error log.
- Read the `CGI::Carp` documentation for ways to redirect the errors to a file instead.

## Learn More About CGI

- perldoc CGI for the CGI.pm documentation
- Lincoln Stein's CGI.pm book - only of the only books published specifically for one Perl module.
- The Perl Cookbook has a chapter on CGI programming

## DBM

- DBM actually refers to a variety of file-based database formats (NDBM, DB, GDBM, SDBM, ODBM)
- Different platforms had different DBM implementations so Perl needed multiple interfaces to support all of them.
- Perl ships with SDBM so that's guaranteed to be everywhere, so there is always some form of DBM available.

## Simple Databases

- Three levels of databases...
  - Plain text files in some user-defined format
  - File-based databases like BerkeleyDB (DB)
  - SQL engines like MySQL, Oracle, MS-SQL, etc...
- You already know how to implement the first
- We will talk about the 2<sup>nd</sup> and 3<sup>rd</sup> methods

## DBM Abstraction

- All of these different implementations made it difficult to write portable programs
- Perl now has AnyDBM\_File, which will use whatever is available locally.
- The general philosophy of DBM files is that you open them, work on them just like a hash, and then save them to disk when done. They can then be retrieved during a later run.

## DBM Example

- ```
use AnyDBM_File;
dbmopen(my %stocks,"ticker",0666);
print "Enter a stock ticker symbol: ";
chomp(my $sym = <STDIN>);
print "Enter a price for $sym: ";
chomp(my $price = <STDIN>);
$stocks{$sym} = $price;
for (keys %stocks) {
    print "$_ is $stocks{$_}.\n";
}
dbmclose(%stocks);
```
- Each run through this will add another stock symbol and price to the DBM

Emptying a DBM File

- Quite easy....
- ```
dbmopen (my %stocks, 'ticker', 0666);
%stocks = ();
dbmclose(%stocks);
```

## DBM Limitations

- DBMs are designed to store simple key=value pairs. Most can't store references to data more complex than scalars (they'll actually store the string "HASH{0x2bbc4a}" instead)
- Solution: make multiple DBMs, or use MLDBM. It uses Data::Dumper to generate storable strings from complex data. It's not standard, though.
- DBMs also have record size limits (1k is safe)

## Sorting Large DBM Files

- It's worth noting that DB\_File (BerkeleyDB) has a BTREE mode which will store your data as a binary tree. This is much more efficient for large data sets.
- Now, calls to keys(), values() and each() come back ordered automatically.
- You can provide a comparison function to tell DB\_File how to sort the data.

## SQL Databases - DBI

- Perl provides an extremely robust abstraction layer called the DBI which allows you to write database-independent code to execute SQL queries.
- With the DBI, it is very easy to port applications between databases, and the syntax remains the same no matter which database you are using.

## DBI Example

```
• use DBI;
 my $dbh = DBI->connect('DBI:driver:database',
 'username',
 'auth',
 {RaiseError => 1,
 {AutoCommit => 1, }});

 $dbh->do($sql);
 my $sth = $dbh->prepare($sql);
 $sth->execute();
 while (my @row = $sth->fetchrow_array) {
 # do something
 }
 $sth->finish();
 $dbh->disconnect();
```

## DBD

- The DBI works by sitting on top of a database-specific DBD driver library. There are DBD modules available for just about all popular databases, including MySQL, PostgreSQL, Oracle, and ODBC (for MS and other databases).
- Specify in the connect string:  
my \$dbh = DBI->connect('DBI:mysql:dbname'...