Introduction to Perl

Andy Lester

Perl's strengths

- Easy stuff easy, hard stuff possible
- Text manipulation
- CPAN
- Internet plumbing
- Mature language
- Thriving community

Perl thinking

- TMTOWTDI
  - There's More Than One Way To Do It
- Laziness, Impatience and Hubris
- Larry is a linguist: Your programs may read like English

Please stop me!

Rule #1 for beginners

- Always start your program like this

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

use warnings;
use strict;
```

- warnings tell you when you misuse variables.
- strict tells you if you misuse references.
- Without strict & warnings, you can write just about anything without an error messages.

Data types

- Scalars
- Arrays & lists
- Hashes
- References
- Filehandles

Scalars

- Single value
- Arbitrarily large
- Can contain binary data, even nulls
- Can be numeric, depending on context

Single-quoted strings

- Define a string

```
my $name = 'Inigo Montoya';
```

- Single quotes have to be escaped

```
my $song = 'Ain\'t Misbehavin\'';
```

- Can even contain newlines:

```
my $signature =
'xoxo,
 Andy';
```

Double-quoted strings

- Define a string

```
my $name = "Inigo Montoya";
```

- Can interpolate other variables

```
my $nametag = "My name is $name."
```

- Handles embedded control characters

```
my $signoff = "xoxo,\nAndy";
```

- Can get dangerous on Windows

```
my $file = "c:\temp";  # WRONG
my $file = 'c:\temp';  # right
my $file = "c:\\temp"; # right
```

Operators

- Standard math operators

```
my $area = $pi * ($radius ** 2);
my $average = ($hi + $lo) / 2;
```

- Join strings with .

```
my $name = $first ." " . $last;
my $name = "$first $last";
```

- Repeat strings with x

```
my $divider = '-' x 40;
```

Comparisons

	Numbers	Strings
Equal	==	eq
Not equal	!=	ne
Greater than	>	gt
Greater or equal	>=	ge
Less than	<	lt
Less or equal	<=	le

Comparison dangers!

- Perl switches between numeric & string as needed
  - 30 > 7 is false
  - '30' lt '7 is true

What is truth?

- Any expression can be evaluated as Boolean
- There are three false values

  0

  ""

  undef
- Everything else is true

  "0000"

  "0 but true"

  "A"

Arrays

- Ordered groups of scalars
- Arbitrary size
- Can contain any scalar
- Used when you need to preserve order

Defining arrays

- Denoted by parentheses

```
my @empty = ();
my @primes = ( 2, 3, 5, 7 );
my @stooges = ( "Moe", "Larry", "Curly" );
```

- Indexed with [] from 0

```
print $stooge[0]; # Larry
print $stooge[-1]; # Curly, from the end
my @first_two = @stooges[0..1];
```

qw() list constructor

- qw() is a list constructor

```
my @stooges = qw( Larry Moe Curly );
```

- Don't use commas

```
my @stooges = qw( Moe, Larry, Curly );
# $stooges[0] = "Moe,";
```

- Any whitespace is a separator

```
my @stooges = qw(
    Moe Larry Curly
    Iggy
    Shemp
);
```

Using arrays

- push/pop/shift/unshift

```
my @stooges = qw( Moe Larry Curly );
pop( @stooges );
push( @stooges, "Shemp" );
# @stooges = ("Moe","Larry","Shemp")
```

- Lists always flatten, not nesting (unlike PHP)

```
my @new_states = qw( Alaska Hawaii );
my @all = (@lower_48, @new_states);
# list ---^^^^^^^^^^^^^^^^^^^^^^^
```

- Scalar context gives a count

```
my $nstates = @all;  # 50
```

List functions

- sorting & reversing

```
@stooges = sort qw( Moe Larry Curly );
# Curly Larry Moe
@stooges = reverse (@stooges, "Iggy");
# Iggy Moe Larry Curly
```

- join: makes a scalar out of a list

```
my $str = join( ", ", @stooges );
# "Iggy, Moe, Larry, Curly"
```

- split: Makes a list out of a scalar

```
my $date = "12/7/1941";
my @parts = split( "/", $date );
my ($m,$d,$y) = split( "/", $date );
```

Control structures

- if/else/elsif

```
if ( $name eq "Quinn" ) {
    print "Hi, sweetie!  How was school?\n";
} elsif ( $name eq "Amy" ) {
    print "Hi, honey!\n";
} else {
    print "You are not my girls!\n";
}
```

Conversational control

- Test can come after action

```
print "How was school?\n" if $name eq "Quinn";
```

- unless is opposite of if

```
unless ( $amount > 0 ) {
    print "Must enter positive amount\n";
}
```

- Combine them both!

```
print "Upgrade your CMS!"
    unless $cms eq "WebGUI"
```

Control -- for/foreach

- foreach -- loops over a list

```
foreach my $stooge ( @stooges ) {
    print "Hello from $stooge\n";
}
```

- No variable uses $_ by default

```
foreach ( @stooge ) {
    $_ = uc;
}
```

- Reverse the loop, and modify the contents

```
$_ = uc foreach @stooge;
```

Control structures

- while

```
while ( condition ) {
    # do something
}
```

Hashes

- Key-based collection of scalars
- Unordered
- Used for lookups, existence or has-a
- Think of a dictionary

Using hashes

- Set the key and value for lookup

```
my %books;
$books{"0-596-00577-6"} = "Spidering Hacks";
# Using the element creates it
```

- Print a value from the hash

```
print $books{ $isbn };
```

- Define properties

```
my %daughter;
$daughter{ age } = 2;
$daughter{ name } = "Quinn";
```

Using hashes

- Get key & value lists

```
my @isbns = keys %books;
my @titles = values %books;
```

- Define properties

```
$daughter{ age } = 2;
$daughter{ name } = "Quinn";
```

- Hash elements must be scalars

```
my @friends = qw( Collin Aidan Grant );
$daughter{ friends } = @friends; # wrong!
```

- Elements autovivify

```
++$daughter{ nbooks };
```

Using hashes

- Non-existent keys return undef

```
print "Quinn married ", $quinn{husband}, ".";
# Quinn's husband is.
```

- Use exists() to see if a key exists

```
print "Quinn married ", $quinn{husband}
  if exists( $quinn{husband} );
```

- Use delete() to delete a key

```
delete $quinn{crib} if $quinn{age} >= 3;
```

Hash ordering

Hash ordering

- Hashes are never in order
- You can't rely on the order between runs
- keys and values always match

Counting words

- First go through the lines and count the words

```
my %count;

for my $line ( @lines ) {
    my @words = split( " ", $line );
    for my $word ( @words ) {
        ++$count{ $word };
    } # for words
} # for lines
```

Reporting counts

- Print words in alphabetical order

```
my @words = keys %count;
@words = sort @words;
for my $word ( @words ) {
    print "$word --> $count{$word}\n";
}
```

```
Address --> 1
Administration --> 2
Advanced --> 3
All --> 1
Any --> 1
Apache --> 5
```

File I/O

- Text files are Perl's specialty
- Line-oriented
- Binary data is OK too, but not in this class
- Current filehandles are typically $fh, but old ones could be like FH.

open

- open()

  open(my $fh, ">", $filename); # create

  open(my $fh, ">>", $filename); # append

  open(my $fh, "<", $filename); # input

- returns false on error
- Error code in $!

  open(...) or die "Can't open: $!";

print

- print can take a filehandle

```
print $fh "Name: ", $name, "\n";
```

- No comma after the filehandle
- Have to be opened for output, of course.

Reading from the file

- Use the diamond operator.

```
my $line = <$fh>;
```

- Reads the next line of the file.
- Has the "\n" at the end of the line

```
chomp $line; # removes the "\n"
```

- Has to be opened for input.
- Returns undef at end of file.

$/ -- input separator

- $/ defines the end-of-line field
- Defaults to "\n"
- Set to "\r\n" for DOS/Windows.

```
my $line = <$fh>;
```

$/ -- paragraph mode

- Set $/ to "" for paragraph mode
- Paragraph is lines of text separated by a blank line.

  This is the first line of a paragraph. It continues to here.

  Second paragraph is here....
- Paragraph is lines of text separated by a blank line.

$/ -- slurp mode

- Set $/ to undef for slurp mode
- Reads the entire rest of the file as a single block.

```
open( my $fh, "<", $filename ) or die;
local $/ = undef;
my $everything = <$fh>;
close( $fh );
```

print

- print can take a filehandle

```
print $fh "Name: ", $name, "\n";
```

- No comma after the filehandle

Input examples

- Print a file without any comments (#....)

```
open( my $in, "<", "myprog.pl" ) or die $!;
while ( my $line = <$in> ) {
    print $line
        unless substr( $line, 0, 1 ) eq "#";
}
close $in;
```

Subroutines

- Encapsulate functionality
- Localize variables
- Reduces repeated code
- Totally (overly?) flexible
- DRY!

Subroutines

- Simplest subroutines use no data

```
sub print_hello {
    print "Hello, world!\n";
}


print_hello();
```

Subroutine parameters

- Parameters passed in special array @_

```
sub square_of {
    my $n = shift;
  # my $n = $_[0]; # Alternative

    return $n*$n;
}


print "8 squared is ", square_of(8), "\n";
```

Subroutine parameters

- Pass as many parms as you want!

```
sub square_of {
    my $n = shift;
    return $n*$n;
}


print "8 squared is ",
    square_of(8, "Bob", \%stooges ), "\n";
# 2nd and 3rd parms ignored

print "9 squared is ", square_of(), "\n";
# Multiplies undef * undef = 0
```

Variable parameters

- Variable parms can be a good thing!

```
sub longest {
    my $long = "";
    for my $maybe ( @_ ) {
        $long = $maybe
            if length($maybe)>length($long);
    }
    return $long;
}

print "The longest stooge name is ",
    longest( qw( Larry Moe Curly Iggy Shemp ) );
```

Dangerous returns

- Last evaluated value is returned

```
sub square_of { $_[0] * $_[0] }

print "8 squared is ", square_of(8), "\n";
```

- Don't do it! Use the explicit return.

Returning lists

- Subroutines can return lists

```
sub favorite_stooges {
    return qw( Iggy Curly );
}


my @faves = favorite_stooges();
```

- Don't read a list into a scalar

```
my $fave = favorite_stooges();
# $fave is now 2
```

Regular Expressions

- A mini-language for matching patterns
- Sort of like filename wildcards
- Matches against $_ by default
- Use the =~ operator

Simplest regex

- Most characters match themselves

```
my $name = "Mr. Bobby Smith";
if ( $name =~ /Bob/ ) {
    print "Hi, Bob!\n";
}
```

- The /i flag goes case-insensitive

```
if ( $name =~ /bob/i ) {
    print "Hi, Bob!\n";
}
```

Metacharacters

.	Matches any single character
*	Zero or more
+	One or more
?	Zero or one
[aeiou]	Character class (here, the vowels)
^	Beginning of the line
$	End of the line
\b	Word boundary
\d \D	Matches a digit/non-digit
\s \S	Matches a space/non-space
\w \W	Matches a word character/non-word character
|	Separates subexpressions to match

Characters

- "." matches any single character

  /c.t/

  # matches cat, cut, cot, cxt

- Character classes define groups

  /c[auo]t/

  # matches cat, cut, cot, but not cxt

- Ranges specified with a hyphen

  /200[0-4]/ # matches 2000 thru 2004

- Caret negates

  /[^abc]/ # any char except a, b or c

Character class codes

- Digits

  /[0-9]/ # Any digit

  /\d/ # Same thing

  /\d\d\d-\d\d\d\d/ # Simple phone number

- Word characters

  /\w/ # letters, numbers or underscore

- Whitespace

  /\s/ # Space, tab, line feed, carriage return

- Capital means opposite

  /\D/ # Anything except a digit

Repetition

- {*x*,*y*} means at least *x*, no more than *y*

  /\d{3}-\d{4}/ # Same phone number

- * means {0,}

  /ABC\d*/ # ABC, ABC9, ABC9423472

- + means {1,}

  /go+al!/ # goal! gooooooooal!

- ? means {0,1}

  /dogs?/   # dog or dogs only

Grouping

- () can group alternatives

```
/^(800|877|866)-\d\d\d-\d\d\d\d$/ # Toll-free?
```

- () captures results into $1, $2 and so on.
- $1, $2 etc are only set if the match succeeded

```
if ( /(w+)\s+\$(\d+\.\d+)/ ) {
    $item = $1;
    $amount = $2;
} else {
    print "Couldn't find item & amount";
}
```

String matching

- Find appearances of Romeo & Juliet

```
open( my $fh, "rj.txt" ) or die;
while (my $line = <$fh>) {
    print "$.: $line"
        if ( $line =~ /\b(RomeoIJuliet)\b/ );
}
```

```
223:   M. Wife. O, where is Romeo? Saw you him to-day?
264:                       Enter Romeo.
276:   Ben. It was. What sadness lengthens Romeo's hours?
314:     This is not Romeo, he's some other where.
```

String counting

- Count appearances of many R&J characters

```
my %counts;
while (<$fh>) {
    if (/(Romeo|Juliet|Friar \w+|Mercutio|Sparky)/) {
        ++$counts{ $1 };
    }
}
for my $char ( sort keys %counts ) {
    print "$char seen ", $counts{$char}, " times.\n";
}
```

```
Friar John seen 5 times.
Friar Laurence seen 11 times.
Juliet seen 65 times.
Romeo is mentioned 155 times.
Tybalt is mentioned 66 times.
```

Regex Replacing

- Instead of m// , use s///

```
s/perl/Perl/ig
# /i = case-insensitive
# /g = global
```

- Normalize spelling of JT's name

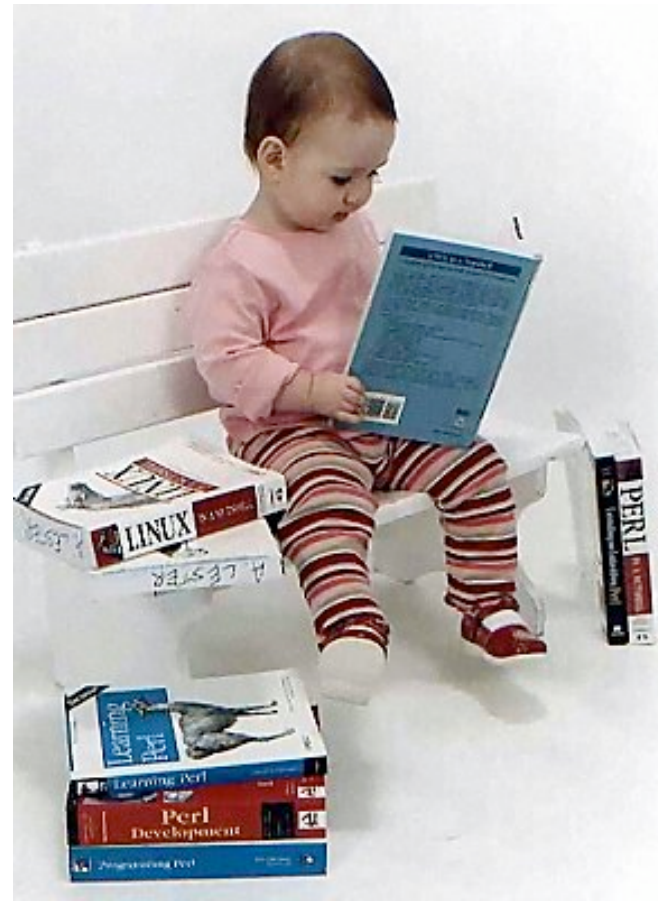
```
s/J\.?T\.?/JT/ig
```

- Replacement part of a regex can contain references to the search part.

```
s/(\d\d)-(\d\d)-(\d\d\d\d)/$3$1$2/;
# Changes "mm-dd-yyyy" to "yyyymmdd"
```

Where to Get More Information

- *Learning Perl Objects, References & Modules*, by Randal Schwartz
- *Object-Oriented Perl*, by Damian Conway
- *Mastering Regular Expressions, 2nd ed.*, by Jeffrey Friedl



Bonus topics if time allows

- map/grep
- Anonymous subroutines
- Command-line wizardry
- Handy magic variables