Introduction to Perl for Statisticians

Phil Spector

Statistical Computing Facility
Department of Statistics
University of California, Berkeley

What is Perl?

- Practical Extraction and Report Language
- Developed by Larry Wall in the late 1980s
- Combines features of `awk`, `sed`, `grep` and shell scripts
- Originally developed for UNIX; now available widely
- No inherent limitations to the size of problems it can handle

Some Typical Uses of Perl

- **NOT** for statistical algorithms
- preprocessing input for other programs
- postprocessing output from other programs
- repetitive editing, searching or processing of a series of files
- simplified access to system calls
- managing tasks requiring coordination among programs
- prototyping more ambitious projects

Accessing Perl

Under UNIX, create an executable file whose first line is

`#!/usr/bin/perl` *`optional_flags`*

and whose remaining lines contain your perl program

Alternatively, invoke perl from the command line:

`perl` *`optional_flags perl_program ...`*

Some of the flags available include:

`-e` - invoke perl with a one line program
`-w` - invoke perl in warning mode
`-n` - invoke perl with automatic input loop
`-d` - invoke perl with symbolic debugger

Flags can be combined. A useful example is

```
perl -de 0
```

which will run the debugger with no program.

Variables

- All variables begin with a special symbol:
  `$` - scalar `@` - array `%` - associative array
- Each type of variable has its own namespace
- By default, all variables are global
- Perl determines type (numeric or character) by context
- Numbers initialize to 0, characters initialize to ""
- Eliminate variables with `undef`

Arrays

- Regular arrays (`@`) are indexed by number
- Associative arrays (`%`) are indexed by strings
- Subscripting can be used to create scalars or arrays
  Suppose `@T` is an array with 10 elements

```
@part = @T[0,1,2]  # array with three elements
$part = $T[0]      # scalar
```

- By default, indexing of arrays starts at 0
  Can be changed with automatic variable `$[`
- The highest index of an array can be found using `$#array`
- Associative arrays use curly braces (like `$ary{$value}`)

Operators

- All the standard binary operators from C
  `+` - addition `-` - subtraction `/` - division `*` - multiplication
- Additional operators
  `**` - exponentiation `.` - string concatenation
  `x` - string repetition `..` - range operator (`1..3`⇒`1,2,3`)
- All operators have a corresponding assignment operator
  For example `$string .= $add` concatenates `$string` and `$add` and puts the result into `$string`
- Increment and decrement operators
  `++` - increments `--` - decrements
  prefix ⇒ before evaluation, postfix ⇒ after evaluation
- `||` is logical or, `&&` is logical and

Comparison Operators

Perl provides two sets of comparison operators, one for numbers and one for strings
It's your responsibility to use the right one (the `-w` flag helps)

Function	Numerical	Character
Equality	`==`	`eq`
Non-equality	`!=`	`ne`
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less than or equal	`<=`	`le`
Greater than or equal	`>=`	`ge`
Comparison	`<=>`	`cmp`

Simple Examples

- Print only lines with exactly five fields
  `perl -ne 'print if split == 5'` *filename*
- Print lines that are longer than 80 characters
  `perl -ne 'print if length > 80'` *filename*
- Print the 10th through 20th lines of a file
  `perl -ne 'print if $. >= 10 && $. <= 20'` *filename*
- Print lines where the third field is 0
  `perl -ne 'print if (split)[2] == 0'` *filename*

Perl Functions 1

Note: Most functions operate on `$_` if they have no arguments

- `split` - break a line into words

```
@T = split;              # breaks $_ into words,
@T = split(",");         # uses comma as separator
@w = split(/[ ,]+/,$in)  # uses comma or space
```

- `chop` - removes last character of its argument
  Useful for removing newlines:

```
while(<>){
    chop;
     ...   # $_ no longer contains a newline
   }
```

Perl Functions 2

- `join` - combine an array or list

```
print join(" ",@T);
$fn = join("/",@parts);
```

- `keys` - produce array containing the indices of an associative array

```
@vals = keys(%array);
```

- `unpack` - breaks a string into an array using format codes
  Can be used for fixed format records or binary data

```
@T = unpack("x3A4A5",$in); # like (3x,a4,a5)
@N = unpack("d4",$vals);   # reads 4 doubles (binary)
```

- `pack` does the opposite of unpack

Perl Functions 3

- `eof` - returns 1 if the next input (<>) would return end of file.
- `eval` - evaluate a string as if it were a Perl program
- `push` - add an element to the end of an array
  `push(@array,$newval);`
- `pop` - remove and return the last value of an array
  `$lastvalue = pop(@array);`
- `shift` - remove and return the first value of an array
  `$firstvalue = shift(@array);`

Programming in Perl

- All statements must end in a semicolon (`;`)
- Blocks of statements are surrounded by curly braces (`{ }`)
- Control statements from C: `for`, `while`, `goto`
- Additional statements: `unless` and `until`
- Inside loops, use `next` and `last`
- Compound statements can eliminate the need for brackets
  Examples: `$x = 3 if $a < $b;`
  `$new = &sub($new) until ($new > $old);`
- `foreach` statement makes processing arrays easy
  Example: `foreach $value (@array){ ... }`

Reading Input

- Basic input operator is angle brackets: `< >`
- Perl is clever about figuring out where input comes from
  - Reads from files if their names appear on the command line
  - Reads from standard input otherwise
- `open` function creates a filehandle

```
open(FI,"filename") || die "Couldn't open filename";
```

  Then read from `filename` with

```
$in = <FI>;
```

Two Special Cases

- With a `while` statement, input goes to `$_`

```
while(<>){  # or while(<FILEHANDLE>)
     ...    # each line is read into $_
   }
```

  This is so useful, it's implemented as the `-n` flag.

- If a wildcard is in the brackets, it is expanded and `$_` set to each filename

```
while(<*.c>){
    open(FI,$_);
        . . .
    }
```


Filehandles

- Special cases:

```
open(FH,"<filename");  # open for input only
open(FH,">filename");  # open for output only
open(FH,">$filevar");  # same using variable
open(FH,">>filename"); # open for append
open(PI,"command|");   # open pipe from 'command'
open(PI,"|command");   # open pipe to 'command'
```

- Automatic Filehandles: `STDIN`, `STDOUT`, `STDERR`

```
$response = chop(<STDIN>);
```

- Input record separator defined by automatic variable `$/`
- Setting `$/ = 0777` causes the whole file to be read into a scalar
- Reading a filehandle into an array reads the whole file

Printing

Remember that variables are expanded inside of quotes, making many printing tasks simple.

- `print` function
  - `$,` is field separator
  - `$\` is record separator
- `printf` function behaves like its counterpart in C
- You can specify a filehandle as an alternative to `STDOUT`

```
print STDERR "Error in input line $.\n";
```

- Simplified printing of errors and warnings
  `warn` - prints message (with newline) to standard error
  `die` - like warn, but exits after printing

More on Printing

- Use escape characters to literally print special characters

```
$cost = 100;
print "The cost is \$$cost\n";
```

  results in

```
The cost is $100
```

- Include large blocks of text using perl's "here-is" syntax:
  `print<<`*`identifier`*;

```
      ...
 text to be printed
     ...
identifier
```

- Extensive formatted printing can be done with `format`/`write`

Example: Reading a Ragged Array into S

Suppose we have a data file where the rows of a ragged array are separated by blank lines:

```
12 19 7 15 23 12                # first row

1 17 18 22 15 13 18 24 92       # second row

14 3 18 12                      # third row
7 19 3

8 7 5 4 2 11 21                 # fourth row
```

We can write a perl program to write an S program to read the data.

Example: Reading a Ragged Array into S (cont'd)

```
#!/usr/bin/perl
open(TOS,"|S") || die("Couldn't pipe to S");
print(TOS "value <- list(\n");
while(<>){
        @T = split(" ");
        if($#T == -1){    # blank line
                printf(TOS "c(%s),\n",join(",",@line));
                undef @line;
                next;
                }
        push(@line,@T);
        }
printf(TOS ")\nprint(value)\n");
```

Example: Reading a Ragged Array into S (cont'd)

The previous program would send the following lines to S:

```
value <- list(
c(12,19,7,15,23,12),
c(1,17,18,22,15,13,18,24,92),
c(3,7,10,4),
c(14,3,18,12,7,19,3),
)
print(value)
```

resulting in the following output from S:

```
[[1]]:
[1] 12 19  7 15 23 12
[[2]]:
[1]  1 17 18 22 15 13 18 24 92
```

...

Sorting

The `sort` function by default performs a lexigraphical sort.
Alternatively, a subroutine or block of statements can be provided.

```
# plain sort gives lexigraphical sort
  @snames = sort(@names);
# using <=> gives numerical sort (note no comma)
  @svals  = sort({$a <=> $b} @values);
# sorting an associative array based on keys
  @skeys = sort(keys(%array));
  foreach $key (@skeys){
       push(@sarray,$array{$key});
       }
# sorting keys based on the values of an associative array
  @skeys = sort({$array{$a} <=> $array{$b}} keys(%array));
```

Functions which return Arrays

- You can use a list with individual members instead of an array

```
($first, $last) = split(" ",$name);
```

- You can extract just part of a returned array

```
# first word from $_ is stored in $first
   $first = (split)[0];
# second through fifth words in $line
   @some = (split(" ",$line))[2..5];
```

- Setting a scalar to an array results in the number of elements

```
# $num is number of comma-separated items in $line
   $num = split(",",$line);
```


Regular Expressions

Special functions and operators for using regular expressions

- `=~` operator allows testing or substitution
  `$in =~ /r_e/` `# returns 1 if $in contains r_e`
  After a match, automatic variables hold useful information
  - `$&` the matched string
  - `$‘` everything before the match
  - `$’` everything after the match
- `$str =~ s/old/new/; # changes old to new in $str`
  `($new = $save) =~ s/old/new/; #doesn’t change save`
- Variables are expanded inside of regular expressions
- Regular expressions can be arguments to `split` or to set `$/`

Constructing Regular Expressions

Regular Expressions can contain any combination of:

- Literal Characters
  Note: Always escape `. ^ $ + ? * ( ) [ ] { } | \`
- Character Classes
  Characters surrounded by square brackets (`[ ]`)
- Negated Character Classes
  First character in brackets is a caret (`^`)
- Escape Characters

`\n`	newline	`\f`	formfeed	`\t`	tab
`\b`	word boundary	`\d`	digit	`\w`	alphanumeric
`\B`	non-boundary	`\D`	non-digit	`\W`	non-alphanumeric

Constructing Regular Expressions

- Operators within Regular Expressions
  - `.` matches any single character except newline
  - `*` matches 0 or more occurrences of preceding character
  - `+` matches 1 or more occurrences of preceding character
  - `^` anchors expression to beginning of target
  - `$` anchors expression to end of target
  - `|` separates alternative patterns
  - `()` tags expressions to be referred to as `$1`, `$2`, etc.
- Changing Case (on right side of `s` command)
  `\U` convert to uppercase `\L` convert to lowercase
  `\E` end conversion

Examples of Regular Expressions 1

Change NA to . (period)

```
s/NA/./
```

Problem: NAME gets changed to .ME

Solution: Use \B to specify word boundaries

```
s/\BNA\B/ . /
```

The s command returns 1 if a substitution is made, 0 otherwise

```
while(<>){
   $i = 0;
   $i++ while s/\BNA\B/ . /;
   print "Too many missing in line $.\n" if $i > 5;
   }
```

Note: By default s operates on only the first occurence
A trailing g makes it global: s/NA/./g;
A trailing i makes the test for a match case-insensitive.

Examples of Regular Expressions 2

Reverse the order of two words

```
s/(\w+) (\w+)/$2 $1/;
```

```
$line = "one two three four";
$line =~ s/(\w+) (\w+)/$2 $1/; ⇒ two one three four
$line =~ s/(\w+) (\w+)/$2 $1/g; ⇒ two one four three
```

Find a word followed by a number

```
if(/(\w+) (\d+)/){
     print "word is $1, number is $2\n";
     }
```

Note that $1 and $2 retain their values after the regular expression.

More Examples of Regular Expressions

- Remove trailing newline (alternative to `chop`)
  ```
  s/\n$//;
  ```
- Eliminate leading 0s in a line of numbers
  ```
  s/\b0+(\d)/$1/g;
  ```
- Removing the leading portion of a pathname
  ```
  ($tail = $path) =~ s#.+/(.*)#$1#;
  ```
- Capitalize the first letter of first word in a line
  ```
  s/^([a-z])/\U$1\E/;
  ```
- Remove comments (`#`) from a line
  ```
  $line =~ s/(.*)#.*/$1/;
  ```


Using Perl to Edit Multiple Files

Two flags are especially useful when using perl as a stream editor:

- `-p` sets up input loop and automatically prints `$_`
- `-i` allows inplace editing (an extension ⇒ backup copy)

For example,

```
perl -pe 's/\BNA\B/ . /g' old.dat > new.dat
```

would change all `NA`s to periods in old.dat, and write to new.dat

```
perl -i.bak -pe 's/\BNA\B/ . /g' *.dat
```

would do the same in place for all files ending with `.dat`, creating copies with `.bak` appended to the names

Examples

- Finding Unique Lines in a File

```
#!/usr/bin/perl -n
# -n puts the perl program in a while(<>) loop
  print if($x{$_}++ == 0);
```

- Finding the Longest Line in a File

```
#!/usr/bin/perl -n
chop;               # remove the newline
$ll = length;
$maxl = $ll if $ll > $maxl;
print "$maxl\n" if eof;
```


Example: Keeping Track of Number of Fields

```
#!/usr/bin/perl
while(<>){
        $n = split;
        $ct{$n}++;
        }

@skey = sort({$a <=> $b} keys(%ct));

foreach $key (@skey){
        print "$key:\t$ct{$key}\n";
        }
```

Example: Generating Random Data

```
#!/usr/bin/perl

# generate 500 sequences of 20 letters

 @alpha = ("A","B","C","D","E","F","G","H","I","J");

 foreach $i (1..500){
        foreach $j (1..20){
                printf("%s",$alpha[int(rand(10))]);
                }
        printf("\n");
        }
```


Interacting with the Operating System

- `system` lets you execute an operating system command
- backquotes (` `) let you capture output into a perl variable
  - retains newlines
  - multiline output can be captured in an array
  - variable expansion takes place inside of backquotes
- Perl provides functions for many operating system calls `chdir`, `kill`, `sleep`, `wait`, `exec`, `fork`, `unlink`, `mkdir`, etc.
- The associative array `%ENV` contains environmental variables `$ENV{"USER"}`, `$ENV{"EDITOR"}`, etc.

File System Operators

As in many shells, perl offers operators for simplified file queries

Op	Tests	Op	Tests	Op	Tests
`-r`	Readable	`-w`	Writable	`-x`	Executable
`-l`	Link	`-e`	Existence	`-f`	Plain File
`-d`	Directory	`-T`	Text	`-B`	Binary

Example:

```
$prefix = "/usr/local/help/";
$filename = $prefix.$topic;
warn("No help file for $topic") unless -f $filename;
```


Example: Running Simulations

```
#!/usr/bin/perl

 @vals = ("1000 10","1000 20","1000 50");

 foreach $val (@vals){
        ($n,$m) = split(" ",$val);
        $outfile = join(".","out",$n,$m);
        die("$outfile already exists") if -e outfile;
        $cmd = "runsim -n $n -m $m > $outfile";
        system($cmd);
        }
```

Formatted Output

- The format statement allows you to define a template for output produced by the `write` function. Substitutions are made as follows:

`@<<<`$\cdots$	left justify as many columns as there are `<`s
`@>>>`$\cdots$	right justify as many columns as there are `>`s
`@|||`$\cdots$	center as many columns as there are `|`s

- Specify the variables to be printed below the format lines, separated by commas
- Final line of the format should have a single period
- You can specify a header to appear on top of each page (`$=` is the page size)

Example of Format Statement

```
format STDOUT_TOP =
Name                            Office   Extension
.
format STDOUT =
@<<<<<<<<<<<<<<<<<<<<<<<<<<<< @>>>>>>>     @>>>>>>
$name,$office,$ext
.
while(<>){
     ($name,$office,$ext) = split(",");
     write;
     }
```

Example of Format Statement (cont'd)

The previous example would convert

```
Fred Smith,425,x7743
John Jones,372,x4450
Harold Johnston,421,x4622
```

to:

```
Name                          Office   Extension
Fred Smith                       425        x7743
John Jones                       372        x4450
Harold Johnston                  421        x4622
```


Creating Subroutines

- Define a subroutine using
  `sub` *`subroutine_name`*`{`
  `statements`
  `}`
- The last expression evaluated is returned, or use `return`
- Arguments to subroutines are passed through `@_`
  First arg is `$_[0]`, second is `$_[1]`, etc.
- Recall by default perl variables are global
  The `local` function can create variables within the subroutine.
- Perl subroutines may be recursive

Using Subroutines

- Call with either `do` *`sub`*`(`*`args...`*`)` or `&`*`sub`*`(`*`args...`*`)`
  Note that parentheses and arguments are optional
- Include subroutines using the `require` statement

```
require 'file.pl';
```

- The `require` statement searches the directories in `@INC`, which will always contain the current directory.
- To read in a subroutine using require, the last line must be

```
1;
```


Example: Prompting for a Password

```
sub getpass{
   local($prompt) = $#_ < 0 ? "Password:" : $_[0];
   system("stty -echo");
   print(STDERR $prompt);
   $pass = chop(<STDIN>);
   system("stty echo");
   printf(STDERR "\n");
   return $pass;
}
1;
```

If the subroutine was in a file called `getpass.pl`, use:

```
require 'getpass.pl';
$secret = &getpass;   # or &getpass("Prompt:");
```

Command Line Arguments

- Arguments are stored in the array `@ARGV`
  First argument is `$ARGV[0]`, second is `$ARGV[1]`, etc.
- `&Getopts` subroutine
  - Argument to `Getopts` is a string of characters
  - Options with arguments are followed by `:` in the string
  - Variables of the form `$opt_`*`letter`* are set appropriately
  - Options and arguments are removed from `@ARGV`

```
require 'getopts.pl';
&Getopts("an:x");
```

Running
`program -a -n 100`
will set `$opt_a` to `1`, `$opt_n` to `100`, and `$opt_x` to `""`

Resources

- Manual page (`man perl`) - around 80 pages
- Books:
  Learning Perl by Randal L. Schwartz
  Programming Perl by Larry Wall & Randal L. Schwartz
  both published by O'Reilly and Associates
- Internet Newsgroup: `comp.lang.perl`
  FAQ from convex.com (pub/perl/info/faq) (anon. ftp)
- Reference Card (part of *Programming Perl*)
  from ftp.cs.ruu.nl (pub/DOC/perlref-4.035.tar.Z) (anon. ftp)
- Source is available from any `comp.sources.misc` archive
- Gopher Server at `feenix.metronet.com`: Path=`1/perlinfo/`