Intro to Perl Programming

Greg Book
Olin Neuropsychiatry Research Center
2006

What is Perl?

Perl stands for **P**ractical **E**xtraction and **R**eporting **L**anguage

Developed by Larry Wall in 1987 as a tool to parse large text files

Primarily used now as a scripting language for batch processing of files

How to Get Perl

- Perl is available for most operating systems. It is generally part of most UNIX and Linux systems, and can be downloaded for Windows.
- http://www.perl.com

How to Get Help on Perl

- Comprehensive Perl Archive Network: http://www.cpan.org
- “Programming Perl” published by O’Reilly (The Camel Book)
- **Google**! Search for “`perl` *`topic`*”

Perl Basics – the program

- Perl programs are text files, that are run through an interpreter
- If run in UNIX/Linux, all perl files must contain the location of the perl interpreter on the first line. This location is specified in the shebang line:
- `#!/usr/bin/perl`

Perl Basics – a sample program

myfirstprogram.pl

```
#!/usr/bin/perl
# my first perl program

#setup variables
$var_num = 1;
$var_string = “How you doing! ”;

# print the variables
print $var_string . $var_num;
```

Programming Perl

- Datatypes
- Variables
- Control structures
- Statements
- Functions (user-defined)

Datatypes

- Most programming languages have a few main datatypes
- **character** ‘c’
- **integer** 1253
- **float** 4.5678
- **string** “Perl is cool”
- **other** $db_conn

Variables

- Variable?? Because it can change value

- Three kinds of variables
- **Scalar** ($) – ex: `1`, `3.1415`, `"strings"`
- **Arrays** (@) – Lists of scalars, indexed by number
- **Hashes** (%) – Arrays that use names instead of numbers to index the elements

Scalar variables

- Scalars can contain any single item: Integers, reals, strings, characters, or pointers to objects
- When **accessing** the contents of any variable, it is prepended with a **$**
- Imagine you want to have a variable with a value of Pi... What to do?

Assigning Scalar Variables

- We know Pi ~ 3.14...
- So... **`$pie1 = 3.14`**
- **`$pie2`** can also equal **`“3.14”`**
- **`$pie3`** can also equal **`“ThreePointOneFour”`**
- **`$pie1`** is a number, but **`$pie2`** is a string, and so is **`$pie3`**
- You can add numbers, but you cannot add strings... Or can you?

Assigning Scalar Variables

- If it looks like a number, then Perl can convert it to a number. So **`$pie1 + $pie2`** will equal **`6.28`**
- However, **`$pie2 + $pie3`** will equal **`“3.14ThreePointOneFour”`**

- Perl variables do not have to be declared at the beginning of the program… But its a good to do so

Arrays (lists)

- Arrays contain multiple variables
- It can contain a list of random numbers, a grocery list, or a list of serial numbers

- Array elements are accessed using indexes... Indexing starts at zero

Accessing Arrays

- To assign or retreive array elements, use the [] bracket operator

```
# assignment
$array[0] = 10;
$array[1] = “20”;
$array[2] = 30.56;
# accessing elements
$var1 = $array[0];
$var2 = $array[1];
```

Hashes

- Hashes are like arrays, but use strings as indexes

```
$defs{‘nit’} = ‘louse egg’;
$defs{“dozen”} = 12;
```

Advanced data structures

- **Hashes of Arrays**

```
$HoA{‘one’}[0] = 1;
$HoA{‘one’}[1] = 2;
```

- **Hashes of Hashes**

```
$HoH{‘one’}{‘zero’} = 1;
$HoH{‘one’}{‘one’} = 2;
```

- **Arrays of Hashes**

```
$AoH[0]{‘one’} = 1;
$AoH[1]{‘one’} = 2;
```

- **Arrays of Arrays**

```
$AoA[0][0] = 1;
$AoA[0][1] = 2;
```

Operators

- Operators are used to operate on variables (it does stuff)
- The important operators are

Assignment

Modifier

Comparison

Logical

Assignment Operators

- These operators assign a value (number, string, etc) to a variable
- `=`
- `+=  -=  *=`

Modifier Operators

- Adding, multiplying, incrementing, etc

- `+ - / *`
- `++ -- **`

Comparison Operators

- Does something equal something else? Is it greater or less than something? Is it not equal...

- Number comparisons: `< > <= >= == != <=>`
- String comparisons: `lt gt le ge eq ne cmp`

Logical Operators

- As opposed to illogical ones?

- Logical operators allow you to do multiple comparisons.
- **Example:** Are these two things equal AND is this third thing not equal to 1

```
($var1 == $var2) && ($var3 != 1)
```

Control Structures

- Control structures change program flow... Useful for making decisions and doing things multiple times
- Basic types of structures:

Statements

Loops

Decisions

Functions

Control Structures - Statements

- Anything in which something is done
- Generally assignments and operations

- `$var = “word”;`

- Statements always end with a semicolon. You can have multiple statements per line, but its best to only have one per line

Control Structures - Loops

- I want this thing done for each item in a list and printed out.

- There are several types of loops, depending on how you want to go about it

`foreach`

`for`

`while`

`until`

Loops - foreach

- **foreach** does something for each item in an array

```
foreach $item (@list) {
   $item += 1;
   print $item;
}
```

Loops - for

- If you know ahead of time exactly how many times you want to do something...

```
for (start; condition; increment) {
   #do stuff
}
```

Loops - while

- Used when you want to do something while some condition is true

```
while (condition) {
   #do some stuff
}
```

Loops - until

- Used for when you want to do something until something is true

```
until (condition) {
   #do stuff
}
```

Decisions – if

- Used to test if a **condition** is true

- A condition is a test (x = 5)
  - a <= 5 true
  - a == 6 false
  - a != 7 true

```
if (condition) {
   #do stuff
}
```

Descisions – if-else

- You may want to do something if something wasn't true... **Example:** If apples are red, eat them, else discard them.

```
if ($apple eq 'red') {
   $eat = true;
}
else {
   $discard = true;
}
```

Decisions – if-elsif-else

- You may have more than one condition to test at once

```
if ($apple eq ‘red’) {
   $eat = true;
}
elsif ($apple eq ‘green’) {
   $eat = true;
   $discard_if_sour = true;
}
else {
   $discard = true;
}
```

Functions (sub routines)

- Functions allow you to input parameters and return parameters. These can be placed anywhere in your Perl program.

```
sub FunctionName {
    my ($invar1, $invar2) = @_;

    #do stuff...

    return $ret;
}
```

Part 2 – the fun stuff

Regular expressions, working with text files

There's More Than One Way To Do It

Regular Expressions

- Imagine you have a list of imformation, and you are looking for something in that list.
- You know what it looks like... It has a pattern to it. It starts with a “W”... and ends in “ter”
- Imagine your list is a dictionary. You might match words like “water” or “waiter”

Regular Expressions

- Perl can find patterns in strings. You can either find a pattern or find and replace a pattern

- An example is a list of ID numbers... They are all *YearMonthDay_Time_ID*
- Example: 20060625_124520_06032883

Regular Expressions

- If you have a list of these IDs, and you want to find all that occured in 2006 and with an ID starting with 06032

```
m/2006.+_.+_06032.+/;
```

Regular Expressions

- The important characters
- \ | ( ) [ ] { } ^ $ * + ? .
- What the #%$@!!!

Regular Expressions - metasymbols

\	beginning of a control character
|	OR – match one group or another
( )	group items together
[ ]	Character class, match characters in the list
^	match at beginning of string
.	match one character
$	match at end of string
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times
{*n*}	match *n* times

More metasymbols – the common ones

\n	Match newline
\d	Match any digit
\D	Match any non-digit
\f	Form-feed
\r	Match carriage-return
\s	Match any whitespace character
\S	Match any non-whitespace character
\t	Tab
\w	Match ‘word’ character (alphanumerics plus ‘_’)
\W	Match any nonword character
\a	Match alarm character

Regular Expression Functions

- Regular expressions are neat little devices, but what do you do with them?
- The regular expression is always defined within these: //
- Two slashes on either side delimit the regular expression. You use functions to use that expression.

m//

- m means match... If your expression is found within a string, this will return true. (This is useful in an if statement)
- So a regular expression for matching may look like this:

```
m/[abc][123]/
```

Yes, it does look like that, but how do I use it?

Regular Expressions - Matching

- m// only works with the pattern *binding operator* (=~)

```
if ($variable =~ m/[abc][123]/)
{
   statements;
   # will only be done if the above
   # expression is true
}
```

s///

- s/// is the substitution function. The regular expression you are searching for goes between the first set of /'s. What you want to replace it with goes in the second set of /'s.
- Imagine you want to replace all occurences of any number with an underscore _

```
$variable =~ s/\d/_/g;
```

Text files

- Perl is excellent at working with text files
- There are a few basic things you will do with text files:

open

read or write

close

Opening text files

- Open a text file using the open() function. You also need to know what you plan to do with the file: read/write/append?

```
open(HANDLE, “< filename.txt”) or die
   “Sorry, it didn’t work out.”;
```

Notice the **HANDLE** ...

open() parameters

mode	read	write	append	create	clobber
<	✓				
>		✓		✓	✓
>>		✓	✓	✓	
+<	✓	✓			
+>	✓	✓		✓	✓
+>>	✓	✓	✓	✓	

Reading from a file - #1

- As always with Perl, there's more than one way to do it. Here is the most logical way:

```
open(FILE, "< list.txt") or die "Oy!";
while ($line = <FILE>) {
   print $line;
}
close(FILE);
```

Reading from a file - #2

- Here's a shorter way to do it, but not as intuitive:

```
open(FILE, "< list.txt") or die "Oy!";
@filecontents = <FILE>;
print @filecontents;
close(FILE);
```

Writing to a file

- Writing to a file is the same as printing to the screen
- To print to a file, specify the file **handle** in the print function.

```
print FILE “One Two Three Four.”;
print FILE “Ay Bee Cee Dee Eee.”;
```

What does the file look like??

Doh!

list.txt:

```
One Two Three Four.Ay Bee Cee Dee Eee.
```

- So, what happened?... It's missing the carriage return "**\n**"
- It should look like this:

```
print FILE "One Two Three Four.\n";
print FILE "Ay Bee Cee Dee Eee.";
```

Recap

There's more than one way to do it.