CPS160
Perl Tutorial 2

Abrita Chakravarty
Department of Computer Science
Duke University

Material adapted after Pallavi Pratapa, Jason Stajich, and Raluca Gordan
(former CPS160 TAs)

Outline

- Review
- Subroutines
- Special Variables
- Regular Expressions
- Debugging

Review

- Scalars ($): strings, numbers, references
- Arrays (@)
- Arrays ⇔ Strings (split, join, qw)
- Hashes (%)
- Conditionals and Loops
- File I/O
- Referencing and De-referencing

Subroutine

- Function, method
- Encapsulates a piece of code (used repeatedly), has defined input and output

```
sub subroutine_name {
  some Perl code here
}
```

- The input parameters are in a special array variable `@_`
- Return value: a scalar or an array (can be references)
- Calling a subroutine:

```
$var = subroutine_that_returns_something()
subroutine_with_no_return()
```

Input Arguments

- `@_` is the array of input arguments
- Individual input arguments accessible inside the function as `$_[0]` and `$_[1]`

```
$s = sum($op1, $op2);
sub sum  {
   my($a, $b) = @_;
   return $a + $b;
}
```

$a now has the value of $op1
$b now has the value of $op2

- Values in `@_` are aliases to the original input parameters. Modifying these will modify the actual parameters (Be careful!)

Output

- **`return`** will return a scalar or a list of items

```
sub sum {
  my ($a, $b) = @_;
  return $a + $b;
}
```

Result of $a + $b returned

- If no **`return`** statement added, then result of the last expression evaluated will be returned

```
sub log_10 {
  my ($a) = @_;
  log($a)/log(10);
}
```

Result of log($a)/log(10) returned

Returning a Scalar or an Array

```
sub sum {
   my ($a, $b) = @_;
   return $a + $b;
}

sub sum1 {
   my ($a1, $b1) = @_;
   $a1++;
   $b1++;
   return ($a1, $b1);
 }
```

$a = 2
$b = 3
$a + $b = 5
Scalar 5 returned

$a1 = 2
$b1 = 8
$a1++ makes $a1=3
$b1++ makes $b1 = 9
Array (3,9) returned

Returning a Hash

- Subroutine can return a scalar or a list of scalars (references are scalars!)
- Any arrays or hashes passed in this list are flattened into one array
  - (this is also true for the list of input arguments)

```
sub just_return {
  my %hash = ("A"=>1, "B"=>2, "C"=>3);
  return %hash;
}
```

- **ex7.pl, ex8a.pl, ex8b.pl**

What is actually returned:
("A", 1, "B", 2, "C", 3)

Special Variables

- `@_, $!, $_` implicit variables

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

- `@ARGV` command line arguments
- `$,` list separator for printing arrays
- In regular expressions
  - `$`` (prematch) string preceding last successful match
  - `$&` (match) string matched by the last pattern match
  - `$'` (postmatch) string following last successful match

Regular Expressions

- Make Perl “powerful”
- Allow matching of patterns
- Syntax can be tricky
- Initially requires effort but pays off dividends
- For long regular expressions, always give a comment so that you can remember what pattern you coded in.

A Simple Example

- Option 1

```
if( $seq eq 'DNA' || $seq eq 'RNA'
                  || $seq eq 'Protein')
{
  print "got a sequence of type  $seq\n"
}
```

- Option 2

```
if( $seq =~ /[DR]NA|protein/ )
{
  print "matched sequence type $seq";
}
```

Regular Expression Syntax

- use the **=~** operator to match

  ```
  if ($var =~ /pattern/) {}
  ```

- `($var =~ /pattern/)` is true iff there is at least one substring in `$var` that matches the pattern
  - `my $var = "a red apple";`
  - `$var =~ /apple/` will be true
  - `$var =~ /[Aa]pple|pear/` will also be true
- `!~` is just like `=~` except that the return value is negated logically

Metacharacters

`|`	means or `/apple|pear/`
`(...)`	allows for grouping
`[...]`	means any of the characters between `[]`
`.`	means any character
`^`	requires an item to be at the beginning of a string `/^start/` matches if the string starts with the word “start”
`$`	requires an item to be at the end of a string `/end$/` matches a string that ends with the word “end”

Quantifiers

- `*` indicates zero, one, or more of the preceding item
- `+` indicates one or more of the preceding item
- `?` indicates zero or one of the preceding item

`/(.*)9*(.*)/` matches 129974 and 12743

`/(.*)9+(.*)/` matches 29974, but not 12743

`/(.*)2(9?)7(.*)/` matches 12743, 12974 but not 129974

Useful Characters

`\s`	whitespace (tab, space, newline, etc)
`\S`	NOT whitespace
`\d`	digits (`[0-9]`)
`\D`	NOT digits
`\t`	tab
`\n`	newline

- Starts with one or more non-whitespace
- Followed by one or more space
- Followed by one or more digits
- Ends with a newline character

```
$var=~ /^(\S+)\s+(\d+)\n$/
```

Match, Substitute or Translate

- Match /REGEXP/
- Match m/REGEXP/

  if m is specified, then / can be replaced with anything (e.g. m##, m[], m!!)
- Substitute s/REGEXP /REPLACE /
- Translate tr/VALUES /NEWVALUES /

Regular Expression Modifiers

- //i case **insensitive**
- //g **global** match (more than one)

```
while( $pattern =~ /(ATG)/g ) {
        print "Found a start codon.\n";
}
```

- //x **extended** regexps (allows comments and white spaces)
- //s treat string variable as **single** long line ('.' matches \n)
- //m treat string variable as **multiple** lines ('.' does not match \n)

Saving what you match…

- Things in parentheses can be retrieved via variables $1, $2, $3, etc for 1st, 2nd, 3rd matches

```
if ( $var =~ /(\S+)\s+([\d\.\+\-]+)/ )
{
  print "$1 --> $2\n";
}
```

$var = "Bob 98.2"
$1 will contain Bob
$2 will contain 98.2
$name will contain Bob
$score will contain 98.2

- Save into variables

```
my ($name,$score) =
  ($var =~ /(\S+)\s+([\d\.\+\-]+)/);
```

Substitute (s///)

- Same as m// but will allow you to
  - substitute whatever is matched in the first section
  - with the value in the second section

```
$sport =~ s/soccer/football/;
$addto=~ s/(Gene)/$1-$genenum/;
```

- Modifiers: /i (ignore case), /g (replace globally), …

Translate (tr///)

- Match and replace
  - what is in the first section, in order,
  - with what is in the second section.
- Lowercase: `tr/[A-Z]/[a-z]/`
- Shift cipher: `tr/[A-Z]/[B-ZA]/`
- Complement a DNA sequence: `tr/ACTG/TGAC/`
- **Example: ex9.pl**

Good Coding Practices

- use strict; use warnings;
- Comment as you go
- Start small; write in blocks/modules; test.
- Use PerlDoc

Debugging your Program

- Simplest debugging technique: `print` statements
  - Try the `debug` function from cps160lib
- Use the EPIC debugger (the ***bug*** icon)
  - Error signals
  - Breakpoints
  - Resume/Terminate
  - Step Into, Step Over, Step Return.

PadWalker Module

- Windows
  - Open `cmd`
  - Type `ppm` to open the Perl Package Manager gui
  - Under Edit→Preferences→Repositories→Add site http://www.bribes.org/perl/ppm/ as a site to browse for Perl packages
  - Once the packages get loaded up in the PPM, search and find PadWalker
  - Select it, mark for installation and execute.

  - (Instructions from http://cpanforum.com/posts/7273)

PadWalker Module

- Mac
  - Open a terminal. Change to `su`
    - (To enable and use root user http://support.apple.com/kb/ht1528)
  - Type `perl -MCPAN -e shell`
  - Answer "`no`" for manual configuration
  - At `CPAN>` type `install PadWalker`
  - Type `q` to quit CPAN prompt
  - Type `exit` to quit su mode

Further Reading

- Course Website
  - http://www.cs.duke.edu/courses/fall10/cps160/resources.html
- Online resources
  - http://perldoc.perl.org/
  - http://www.perl.org/books/beginning-perl/
- Textbook
  - Beginning Perl, Second Edition, James Lee