CS 3101-4 - Programming Languages: Perl

Lecture 2: The basics II

Vinodkumar Prabhakaran
vinod@cs.columbia.edu

March 30, 2012

Contents

Homework

- to be posted tonight
- email vinod@cs.columbia.edu
- by Wednesday, April 4th, night 11.59 EST
- detailed submission instructions in homework posting
- document your code with meaningful comments
- academic honesty policy
  - all problems must be solved individually.
  - do not copy literal code and solutions from online.
  - is ok to discuss problems with other students, but you have to do the write-up and implementation yourself.

Project

- proposal due on April 9th. (submit early to start early)
- proposal review and finalize by April 12th.
- project submission and report due on May 4th.
- project ideas (partially from COMS 3101-3 (Python))
  - a tool that harvests and processes information from the web (news headline aggregator, monitoring flight ticket prices...).
  - some web-app (simple blogging platform, meeting scheduling site...)
  - implementation of some interesting NLP application
  - implementation of some interesting algorithm(s) (machine learning, NLP, graphs, ...)
  - a tool for analyzing a specific scientific data set.

Review of lecture 1

- scalars ($), arrays (@), hashes (%)
- context - scalar vs list
- quotes and interpolation
- operators
  - numeric vs string operators
- control structures
  - if-elsif-else, unless
  - while, until
  - for, foreach
- statement modifiers

ClassAverage.pl (The C'ish version)

```
1  # call as 'perl ClassAverage.pl'
2  open (IN,"marks.txt"); # open input file
3  $sum = 0; # $ denotes scalar
4  $count = 0;
5  $line = <IN>; # read one line from file
6  while($line) {  # while a valid line a read
7          @fields = split(' ',$line);
8          $mark = $fields[1];
9          $sum = $sum + $mark;
10         $count++;
11         $line = <IN>; # read another line
12 }
13 $avg = $sum/$count; # floating-point division!
14 print "The average marks of $count students: $avg\n" ;
15 close(IN);
```

Execution

Working with files

Opening and closing a file

```
1 open(INFILE, "<filename"); # "<" is optional;
2 open(OUTFILE, ">filename"); # Write to a new file
3 open(OUTFILE, ">>filename"); # Append to existing file
4 close(INFILE); # Close the file
```

Writing to a file

```
1 print OUTFILE "this goes to a file";
2 print STDOUT "this goes to standard output";
3 print  "this goes to standard output too";
```

Reading from files - line input operator (< ... >)

```
1 open(INFILE , "marks.txt");
2 open(OUTFILE , ">out.txt");
3 while ($line = <INFILE>) { # yields next line
4     print OUTFILE $i++, ": $line";
5 }
6 close(INFILE, OUTFILE);
```

```
1 open(INFILE, "filename");
2 @alllines = <INFILE>;  # returns all lines
3 close(INFILE);
```

```
1 open(INFILE, "filename");
2 $line1 = <INFILE>; # yields first line.
3 $line2 = <INFILE>; # yields second line.
4 @restlines = <INFILE>;  # returns remaining lines
5 close(INFILE);
```

More on reading from files

Auto assignment to $_ in while loop

```
1 while(<INFILE>) { # same as while($_ = <INFILE>)
2     print;        # same as print $_;
3 }
```

only for while loop (not for if, unless etc.)

```
1 if (<INFILE>) { # $_ don't get line from INFILE
2     print;      # prints whatever was in $_ before
3 }
```

only when <> is the one and only thing in the while ()

```
1 while (<INFILE1> && <INFILE2>) {
2 # throws away both lines
3     print;      # prints whatever was in $_ before
4 }
```

Command line arguments

- can pass arguments into a program from command line
- could be file names, parameters etc.

```
1 $ perl test.pl infile.txt outfile.txt
```

test.pl

```
1 if ($#ARGV != 1 ) {
2     print "wrong number of arguments passed";
3     exit;
4 }
5 $infile=$ARGV[0];
6 $outfile=$ARGV[1];
```

Angle with null file handle <>

- Opens file specified in command line arg, by default

```
1 # execute with input file as command line argument
2 # perl sample.pl infile.txt
3 while(<>) {      print;  }
4 # equivalent to
5 open (IN, $ARGV[0]);
6 while(<IN>) {     print;  }
```

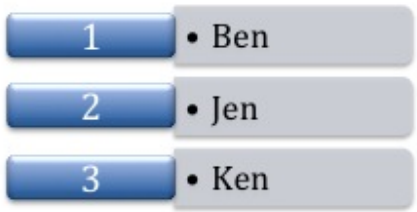
- loops through @ARGV as if they were 'one big happy file'
- you can manipulate @ARGV before using <>
- if no arguments passed, lines read from STDIN

```
1 $inline = <>; # default STDIN
```

- more detailed pseudocode in the camel book.

Adding and removing array elements

- push - adds at the end
- pop - removes from the end
- shift - removes from the beginning
- unshift - adds at the beginning
- delete - delete an array element




```
1 @names = (Ben,Jen,Ken);
2 push(@names,"Lin"); # (Ben,Jen,Ken,Lin)
3 $popped = pop(@names); # (Ben,Jen,Ken)
4 $shifted = shift(@names); # (Jen, Ken)
5 unshift(@names,"Sen"); # (Sen,Jen,Ken)
6 delete $names[1]; # deletes Jen (Sen,Ken)
```

More on arrays

- Concatenating arrays

```
1 @boys = ("Alex","Ben","Chad");
2 @girls = ("April","Becky","Cindy");
3 @children = (@boys, @girls);
4 # ("Alex","Ben","Chad", "April","Becky","Cindy")
```

- join - join an array to form a string
- split - split a string into array

```
1 print join(",",(3..6)); # prints 3,4,5,6.
2 @words = qw(this is a sentence);
3 $sentence = join("_", @words); # "this_is_a_sentence"
4 @newwords = split(/_/,$sentence);
5 @newwords = split(/s/,$sentence);
6  # ("thi","_i","_a_","entence")
```

More on arrays

- Size of arrays

```
1 $num = @children; # array used in scalar context
2 $num = scalar(@children); # explicit scalar conversion
3 $num = $#children + 1; # highest index + 1
```

- `length` function != size of array
- number of characters in the string
- expects scalar argument

```
1 print length("Columbia"); # 8
2 @nums = (5..9);
3 print length(@nums); # 1 which is length("5")
4 @nums = (5..150);
5 print length(@nums); # 3 which is length("146")
```

Sort arrays

Alphabetical ascending sort by default

```
1 @names = (Ken, Ben, Jen);
2 @ages = (4,37,34);
3 @sorted = sort (@names); # (Ben,Jen,Ken)
```

Numeric vs alphabetical sort

```
1 @sorted = sort (@ages); # (34,37,4) WRONG
2 @sorted = sort {$a <=> $b} (@ages); # (4,34,37) YAY!
```

Descending sort

```
1 @sorted = sort {$b cmp $a} (@names); # (Ken,Jen,Ben)
2 @sorted = sort {$b <=> $a} (@ages); # (37,34,4)
```

Reverse an array

```
1 @reversed = reverse(@sorted); # (4,34,37)
```

Looping through arrays and hashes

```
1 foreach $item (@items) {
2     # some processing on the $item
3     # any changes to $item carries over to the array
4 }
```

```
1 while ( ($key, $value) = each (%agehash)) {
2     # use $key and $value, but don't change %agehash
3     print "Age of $key is $value\n"; # OK
4     $agehash{$key} = $value + 1;   # NOT OK!
5  }
```

```
1 foreach $key (keys %agehash) {
2     # processing which may or may not alter %agehash
3     $agehash{$key} = $agehash{$key} + 1;
4 }
```

Processing hashes in sorted order

Ordered on keys

```
1 foreach $key (sort keys %hash) {
2     print "$key: $value\n";
3 }
```

Ordered on values

```
1 # Prints on descending order of ages
2 foreach $key (sort {$hash{$b}<=>$hash{$a}} keys %hash)
3 {
4     print "$key: $value\n";
5 }
```

More on hashes

- Size of hashes

```
1 $num = scalar (keys %agehash);
2 $num = keys %agehash;   # implicit scalar conversion
```

- Remove a hash entry

```
1 delete $agehash{"Ben"}; # deletes Ben's entry from
    hash
```

- Reverse a hash - swap keys and values
- DO NOT use if values are not unique

```
1 %code2state = ("OH","Ohio","TX","Texas","IA","Iowa");
2 %state2code = reverse(%code2state);
```

Checking for keys

- `defined` - checks if the value is undef
- `exists` - checks if the key exist at all

```
1  %test = (a,b,c); # $test{a} = b, $test{c} = undef
2  if (exists $test{c}) {
3      print "C exists\n";
4  } else {
5      print "C does not exist\n";
6  }
7  if (defined $test{c}) {
8      print "C is defined\n";
9  } else {
10     print "C is not defined\n";
11 }
```

Output

```
1  C exists
2  C is not defined
```

map and grep

map - change elements of a list to return a new list

```
1  @squares = map { $_ * $_ } @numbers;
2  # equivalent to
3  @squares = ();
4  foreach (@numbers) {
5      push (@squares, $_*$_);
6  }
7  # Can result in different number of elements
8  @squares = map { $_ > 5 ? ($_ * $_) : () } @numbers;
9  @lowers = map { lc } @names; # ("ben","jen","ken")
10 @lowers = lc @names; # (3) lc is a scalar operator!
```

grep - locate elements in a list test true against a given criterion

```
1  my @squares = map {$_ * $_} grep {$_ > 5} @numbers;
```

ClassAverage.pl (The C'ish version)

```
1  # call as 'perl ClassAverage.pl'
2  open (IN,"marks.txt"); # open input file
3  $sum = 0; # $ denotes scalar
4  $count = 0;
5  $line = <IN>; # read one line from file
6  while($line) {  # while a valid line a read
7           @fields = split(' ',$line);
8           $mark = $fields[1];
9           $sum = $sum + $mark;
10          $count++;
11          $line = <IN>; # read another line
12 }
13 $avg = $sum/$count; # floating-point division!
14 print "The average marks of $count students: $avg\n" ;
15 close(IN);
```

ClassAverage.pl (A readable Perl version)

```
1  #!/usr/bin/perl
2  # call as './ClassAverage.pl marks.txt'
3
4  open (IN,$ARGV[0]);
5  @inlines = <IN>;
6
7  $sum = 0;
8  $count = 0;
9
10 foreach $line (@inlines) {
11         ($student,$mark) = split(/ /,$line);
12         $sum = $sum + $mark;
13         $count++;
14 }
15 $avg = $sum/$count;
16 print "The average marks of $count students: $avg\n" ;
17 close(IN);
```

ClassAverage.pl (A lazy Perl programmer)

```
1  #!/usr/bin/perl
2  # call as './ClassAverage.pl marks.txt'
3  $sum = 0; # not required
4  while(<>) {
5          chomp;
6          split;
7          $sum += pop(@_);
8  }
9  $count = $.; # not required, can directly use $.
10 print "The average marks of $count students: ", $sum/
       $count, "\n" ;
```

Defining a subroutine (can be defined anywhere in the file)

```
1 sub printnextnum {
2   $n += 2;   # global variable $n
3   print "Next number $n\n";
4 }
```

Invoking the subroutine

```
1 &printnextnum;
2 &printnextnum;
3 &printnextnum;
```

Output

```
1 Next number 2
2 Next number 4
3 Next number 6
```

Return values - last evaluated expression

```
1 sub nextnum {
2   $n += 2;   # global variable $n
3 }
4 ...
5 print "Next number ", &nextnum, "\n";
6 print "Next number ", &nextnum, "\n";
7 print "Next number ", &nextnum, "\n";
```

Output

```
1 Next number 2
2 Next number 4
3 Next number 6
```

Return values - last evaluated expression

```
1  sub nextnum {
2    $n += 2;   # global variable $n
3    print "just incremented the value of n";
4  }
5  ...
6  print "Next number ", &nextnum, "\n";
7  print "Next number ", &nextnum, "\n";
8  print "Next number ", &nextnum, "\n";
```

Output

```
1  just incremented the value of n
2  Next number 1
3  just incremented the value of n
4  Next number 1
5  just incremented the value of n
6  Next number 1
```

Return values - last **evaluated** expression

```
1  sub nextnum {
2    if ($n < 2) {
3      $n += 1;   # global variable $n
4    }
5    else {
6      -1;
7    }
8  }
9  ...
10 print "Next number ", &nextnum, "\n";
11 print "Next number ", &nextnum, "\n";
12 print "Next number ", &nextnum, "\n";
```

Output

```
1  Next number 1
2  Next number 2
3  Next number -1
```

Return list of values

```
1  sub nextnums {
2      $n++;
3      ($n..$n+2);
4  }
5  @nums = &nextnums;
6  print "@nums\n"; # 1 2 3
7  @nums = &nextnums;
8  print "@nums\n"; # 2 3 4
```

Explicit return statement

```
1  sub nextnums {
2      $n++;
3      return ($n..$n+2);
4  }
5  sub nextnum {
6      return $n++;
7  }
```

Passing arguments through magic @_

```
1  sub max {
2    # @_ holds all values passed
3    if ($_[0] > $_[1]) {
4      $_[0];
5    } else {
6      $_[1];
7    }
8  }
9
10 print &max(34,12); # 34
```

- pass arbitrary number of arguments
- throws away those which are not used (e.g. max(12,4,23))
- `undef` for those which are not passed (e.g. max(32))

Private variables

```
1  sub max {
2    my ($first,$second) = @_; # what's this my?
3    if ($first > $second) {
4      $first;
5    } else {
6      $second;
7    }
8  }
```

```
1  sub max {
2    my $first = shift;
3    my $second = shift;
4    if ($first > $second) {
5      $first;
6    } else {
7      $second;
8    }
9  }
```

A better max

Works for arbitrary number of arguments!

```
1  sub max {
2    my $max_so_far = shift;   # first is the max so far
3    foreach (@_) {            # look at rest
4      if ($_ > $max_so_far) {
5        $max_so_far = $_;
6      }
7    }
8    $max_so_far;
9  }
10 print &max (12,231,34,245,6,46); # 245
11 print &max (34,142,15); # 142
```

When is & optional?

Use neat paranthesis

```
1 my @cards = shuffle(@deck_of_cards);  # Calls &shuffle
```

Define the subroutine before invocation

```
1 sub division {
2   $_[0] / $_[1];
3 }
4 ...
5 my $quotient = division 355, 113;  # Calls &division
```

Except if its a perl built-in function

```
1 sub chomp {
2   print "Munch, munch!";
3 }
4 ...
5 &chomp;  # That ampersand is not optional!
```

Summary

- Input/Output
  - open, close, read (< ... >), write (print)
  - while(<>) ...
- More on arrays, hashes, lists
  - arrays: push, pop, shift, unshift, delete
  - arrays: split, join, sort, reverse
  - hashes: sorted order
  - hashes: keys, values, each, delete, reverse
  - hashes: exists, defined

Summary

- Subroutines
  - defining (anywhere) and invoking
  - return values, implicit and explicit
  - passing args @_, arbitrary number of args

Special variables

- $_ - default input (and few other things)
- $. - input line number
- $/ - input record delimiter
- $\- output record separator
- $, - output field separator
- $" - similar to $,. in array interpolation
- $! - error number or message
- ...