# COMS 3101
# Programming Languages: Perl

## Lecture 4

Fall 2013
Instructor: Ilia Vovsha

http://www.cs.columbia.edu/~vovsha/coms3101/perl

## Lecture Outline

- Pattern Matching (continued)
- Concepts:
  - More on files
  - Scoped variables
  - Special variables
- References
- Data Structures
- Next: Packages & Modules

## Remarks

- Extract substring: start at “offset”, return “length” characters:
  - $sub_name = substr($name, 4, 5);                # start at 4, return 5
  - $sub_name = substr($name, 4, 5, “AA”);      # same but replace
- Formatted string:
  - $FSTR  =  sprintf(“%.3f”, $number);           # %s,%d,%f
  - printf  “%.3f”,   $number;                          # print FH sprintf(FORMAT, LIST)
- Get key/value pair of a hash:
  - while ( ($key, $value) = each %hash) {              # Random order

# Pattern Matching

- Regular Expressions (REGEX)
- Operators (cages): match, substitute
- Elements:
  - Metacharacters
  - Pattern modifiers
  - Character classes (classic, Unicode)
  - Quantifiers
  - Assertions
  - Grouping & capturing
  - Alternation
  - Magic dot
  - Return values
- General advice

# PM (grouping & capturing)

- Group and remember sub-patterns using ( )
- After the pattern: $N denotes the Nth group from start
- Within the pattern: \N denotes the Nth group from start
- Dynamic scope until end of block or next match
- Can use them with s/// as replacement
- Nested (()): counting by location of left parenthesis '('
- Special variables:

| | |
|---|---|
| $` | all to left of match |
| $& | entire matched string |
| $' | all to right of match |

# Grouping & Capturing (examples)

- Examples:

```
1.    if  ($line =~ /<(.*?) >.*?<\/\1>/)
            # matches any tags <xyz>.....</xyz>

1.    if  ($line =~ /<(.*?) >.*?<\/\1>/) {
2.              print  "$1  :  $2";                    }

1.    $line  =~  s/^(\w+) , (\w+)/$2 , $1/;        # swap two words
2.    $line  =~  s/((\w+) , (\w+))/;               # nested (())
3.    $line   =  "perl is fun , or is it";
4.    $line  =~ m/((\w+) , (\w+))/;
5.    print  "pre - match – post: $`\t$&\t$'\n";
      # prints "….perl is    fun , or         is it"
```

# PM (alternation)

- Specify a set of possibilities ( | )
- For overlapping matches, place longer string first
- Extends only to the innermost enclosing ()
- Examples:
  1. `if ($line =~ /bank|banker/)      # always matches bank!`
  2. `if ($line =~ /(banker|bank)/)  # banker can match`
  3. `if ($line =~ /ban(k|b)er/)       # banker or banber`

## PM (magic . )

- The dot . matches any character (wildcard)
- Turn maximal (greedy) matching into minimal matching by adding a ‘?’
- Examples:

```
1.  if  ($line  =~  /<xml>(.*)<\/xml>/ )
    # match any text between two xml tags
```

```
1.  $line = "Want this <bold>content's tag<bold> matched ";
2.  if  ($line  =~ /<(.*)>/ )         # matches "bold>content’s tag <bold"
3.  if  ($line  =~ /<(.*?)>/ )        # matches "bold“
4.  if  ($line  =~ /<(.+?)>/ )        # matches only if its non empty
```

# PM (return values)

- Match (m//) :
  - scalar context - returns true (1) if successful, else false (“”)
  - list context - returns a list of matched groups
- Substitute (s///):
  - returns number of times it succeeded (scalar & list context)

```
$line   =  "<text>this is the text</text>";
($tag, $content)  =   $line   =~  /^<(.*?)>(.*?)<\/\1>$/;

if  (@perls  =  $text   =~  /perl/gi)  {
   print  "Number of times Perl mentioned : ",  scalar(@perls);  }

$string  =  "name=xyzzy id=9 score=0";
%hash  =  $string  =~  /(\w+)=(\w+)/g;

$num  =  $text  =~  s/perl/PERL/g;
```

## PM (general advice)

- When matching multiple regex, list common case first
- When writing a long regex, simplify with variables (interpolation)
- Consider using || (logical) instead of | (regex) to be more efficient
- Avoid $& $` $' if you can (slows down execution). However, if used once, use all the time without penalty
- Not every problem should be solved with regex: consider functions to manipulate strings (substr) instead
- Start by writing down all the patterns you need to identify, then proceed to contrive the regex

# More Examples

```
while (<>)  {
  next  if  $line  =~   /^#/;
  …
}
```

```
from:       vp2198@columbia.edu
to:    vp2198@columbia.edu
date:       Sun, Apr 1, 2012 at 11:08 PM
subject:    COMS W3101.004 Office Hours Change (this week only)
mailed -by:      columbia.edu

while (<>)  {
      $_   =~   /^(.*?):\s*(.*)$/;
      $hash{$1}  =  $2;
}
```

## Concepts

- More on files
- Scoped variables
- Special variables

## < … > Rules

- "Angle operator": apply to a filehandle to read the next line
- Auto assignment to $_ **only** in while loop! (not if, unless,…)
- Examples:

1.
```
while ( < INFILE > ) {                # next line
        print  $_;
```

2.
```
if ( < INFILE > ) {                   # WRONG
         print  $_;                   # prints whatever in $_ before
```

3.
```
while ( <IN1> && <IN2> ) {            # WRONG (throw away lines)
        print  $_;                    # prints whatever in $_ before
```

## Scoped Variables

- my
  - creates private variable visible only within block
  - hidden from outside of enclosing scope, and hides previously declared variables with identical name
  - confines *name & value* to scope
  - suitable for scalar/array/hash variables
- our
  - confines *name* only to scope (no effect on visibility)
  - suitable for scalar/array/hash variables
  - used to access global variables, their initial value inside block unchanged
  - effects or assignment persist after the scope of declaration (block)
- local
  - confines *value* only to scope
  - suitable for scalar/array/hash + more variables
  - initial value for variable is () or undef
  - value of variable is restored no matter how you exit the block (changes thrown away)
  - “dynamic” scope: value of variable depends on scope & changes during run-time
  - ‘my’ is preferable over ‘local’

# Scoped Variables (example)

```
$office  =  "global";                # Global $office
&say();                              # prints "global"
&barney();                           # prints "barney global", lexical scope;
&fred();                             # prints "fred fred", dynamic scope,
&say();                              # prints "global", restored after &fred()

sub  say  { print  "$office\n";  }   # print the $office

sub  barney  {
     my  $office  =  "barney";
     print  "$office ";  &say();   }

sub  fred  {
     local $office = "fred";
     print "$office ";  &say();   }
```

## Special Variables

- Predefined variables with a special meaning:
  - $_ @_ @ARGV $a $b
  - $1,$2,… : matched groups in pattern (outside pattern)
  - \1,\2,… : matched groups in pattern (in pattern)
  - $&, $`, $’ : match, pre-match, post-match pattern
  - $0 : program/script file name

## References

- Motivation
- Definition
  - Hard references
  - Symbolic references
- Creating (initialization)
  - Backslash \
  - Anonymous  [], {}
- Using references (de-referencing / access)
  - Braces {}
  - Arrows ->

## References (motivation)

- City – State listing

```
%hash  = (
      “Albany”  => “NY”,
      “Boston”  => “MA”,
      “Buffalo”  => “NY”,
      “Salem”  => “MA”,
      “Dallas”  => “TX”,
);
```

- Problem: how do we represent State – City listing?
- Solution: merge all cities for each state into one string?

```
%hash  = (
      “NY”  => “Albany,Buffalo”,
      “MA”  => “Boston,Salem”,
      “TX”  => “Dallas”,
);
```

# References (motivation)

- Solution: merge all cities for each state into one string?

```
%hash  = (
      “NY”  => “Albany,Buffalo”,
      “MA”  => “Boston,Salem”,
      “TX”  => “Dallas”,
);
```

- Why? Arrays & hashes can only hold scalars
- Why not? Cumbersome to maintain/access elements
- Better approach: hashes of arrays

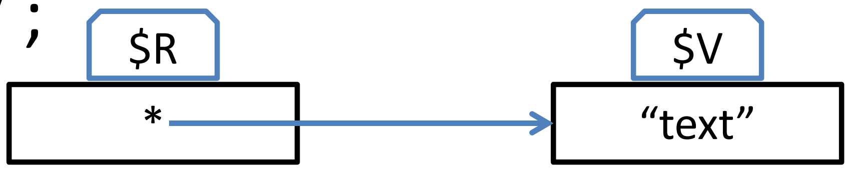
```
foreach $state (keys %hash) {
   print  “\n Cities in $state";
   @cities  = split (/,/, $state);
   print  “$_\t”  foreach (@cities);
}
```

## References (definition)

- Scalar types: strings, numbers, references
- Pointer to location of variable (scalar, array, hash)
- Unlike pointers in C:
  - Perl reference can refer to data and functions (subs)
  - Can’t access raw memory location
- Can access entire array/hash by dereferencing the reference to the structure
- Hash of arrays is a hash where each value is a reference (pointer) to an array

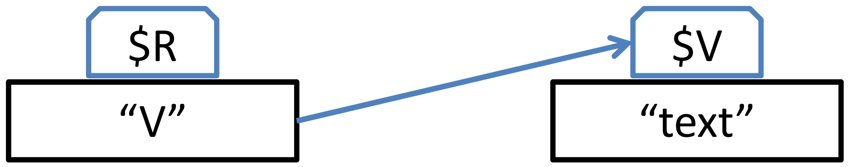
# References (definition)

- Hard & Symbolic references:
  - Hard (real): refers to location (actual value) of variable
  - Symbolic: refers to the name of variable
- Create hard ref: $R = \$V ;



- Create symbolic ref: $R = "V";

=> If we try to dereference $R we get access to variable $V!



## Creating References (backslash)

```
$a_scalar   =  “boom”;
@an_array   =  (“A1”, “B2” , “C3”);
% a_hash    =  (“Albany” => “NY”, “Boston“ => “MA”);

$scalar_ref   =  \$a_scalar ;
$array_ref   =  \@an_array ;
$hash_ref   =  \% a_hash ;
                                                  # Refs can be used like any other scalar
$one_ref   =  $another_ref ;
$arr [3]  =  $a_ref ;
$h{$i}  =  $a_ref ;
                                                  # Use backslash on a list of refs
@refs   =  \($one,$two,$three);
@refs   =  (\$one,\$two,\$three);                 # Same as previous line
@refs   =  \($a_scalar, @an_array, %a_hash);
@refs   =  (\$a_scalar, \@an_array, \%a_hash);     # Same as previous line
```

## Creating References (anonymous)

```
$sc_ref   = \“word”;
$sc_ref2   = \123;
                                                # Arrays:
$ar_ref   = \(“A1”, “B2” , “C3”);               # WRONG!
$ar_ref   = [ “A1”,  “B2”, “C3” ];              # Square brackets, different from $a[1]

                                                # Hashes:
$hs_ref   = \(APR => 4, MAY => 5);              # WRONG!
$hs_ref   = { “April” => 4,  “May” => 5 } ;     # Braces, different from $h{$key}

% a_hash  =  ( “April” => 4,  “May” => 5 );
$hs_ref   =  \% a_hash;                         # Backslash instead of anonymous

@AoA  =  ( [ 1, 2], [ “A1”,  “B2”, “C3” ] );    # Array of arrays
$AoA_ref  = [ [1, 2], [ “A1”,  “B2”, “C3” ] ];  # Reference to array of arrays
```

# Using References (braces)

```
$sc_ref   = \“word”;
$ar_ref   = [ “A1”,  “B2”, “C3” ];
$hs_ref   = { “april” => 4,  “may” => 5 } ;

print  ${ $sc_ref };                          # prints “word”
${ $sc_ref }  =  “Number";                    # prints “number”,  if we print
@ar2  =  @{ $ar_ref };                        # ar2 now has (A1,B2,C3) ;
${ $ar_ref }[2]  =  “D4”;                     # $ar_ref now refers to array (A1,B2,D4)
%hs2  =  %{ $hs_ref };                        # hs2 now has (april ,4,may ,5);
${ $hs_ref }{ may }  =  99;                 # $hs_ref now refers to hash with may => 99

$$sc_ref  =  “Number";                        # equivalent to above
@ar2  =  @$ar_ref ;                           .
$$ar_ref [2]  =  “D4”;                        .
%hs2  =  %$hs_ref ;                           .
$$hs_ref { may }  =  99;                      .
${ $ar_ref[2] }          NOT THE SAME AS       ${ $ar_ref }[2]
```

## Using References (arrows)

- Arrows are syntactic sugar to simplify access
- Type of dereference is determined by what follows the arrow:
  - [ or { or (                    array or hash or subroutine

```
# equivalent statements:
               ${ $ar_ref }[2] =  “D4”;
               $$ar_ref [2]  =  “D4”;
               $ar_ref ->[2]    =  “D4”;

# equivalent statements:
               ${ $hs_ref }{may}  =  99;
               $$hs_ref {may}  =  99;
               $hs_ref ->{may}  =  99;

               $ar_ref ->[2]          NOT THE SAME AS          $ar_ref [2]
```

# Data Structures

- Data Structures (DS):
  - Array of Arrays (AoA)
  - Hash of Arrays (HoA)
  - Array of Hashes (AoH)
  - Hash of Hashes (HoH)
- Functionality:
  a) Initialization (composition)
  b) Adding elements (generation)
  c) Printing elements (access)
  d) Slice (arrays), sort (hashes)

# AoA & HoA (initialization)

```
@AoA  =  ( [ 1, 2], [ “A1”,  “B2”, “C3” ] );        # Array of arrays
$AoA_ref  = [ [1, 2], [ “A1”,  “B2”, “C3” ] ];      # Reference to array of arrays

print  $AoA[1] -> [1];                              # prints “B2”
print  $AoA[1] [1];                                 # Arrows not required between [], {}
print  $AoA_ref -> [1] -> [1];                      # Same thing, using ref
print  $AoA_ref -> [1][-2];                         # Using negative indices
```

```
%HoA   =  (
        “NY”  => [Albany,Buffalo],                        # instead of “Albany,Buffalo”
        “MA”  => [Boston,Salem],
        “TX”  => [Dallas],
);

print  $HoA{NY} -> [1];                             # prints “Buffalo”
print  $HoA{NY} [1];                                # Arrows not required between [], {}
```

# AoA & HoA (adding elements)

```
@AoA  =  ( [ 1, 2], [ “A1”,  “B2”, “C3” ] );          # Array of arrays
push  @AoA,  [3,4];                                   # Add row (anon array ref)
push  $AoA[1],  “D4”, “E5”;                           # WRONG!  (append to row)
push  @{ $AoA[1] },  “D4”, “E5”;                      # Append to row

for  $x   (0..2)  {                                   # For each row
    $AoA[$x] [2]  =  5;                               # Set 3rd column
}
```

```
%HoA   =  ( “NY”  => [A,B], “MA”  => [B,S], “TX”  => [D], );
$HoA{$state}   =  [C,D];                              # Add array (anon array ref)
push  @{ $HoA{TX} },  “C”, “D”;                       # Append to array for key “TX”

while ( <> ) {
    next  unless  m/^(.*?),\s*//;                     # skip other lines
    $hash{$1}  =  [ split ];                          # split $_ on / /
}
```

# AoH & HoH (initialization)

```
@AoH   =  (
        { “NY”  => 1, “MA”  => 2, “TX”  => 40,  },
        { “NY”  => 40, “MA”  => 45, “TX”  => 1,  },
);

print  $AoA[1] -> {NY};                         # prints “40”
print  $AoA[1] {NY};                            # Arrows not required between [], {}
```

```
% HoH  =  (
        python  =>  {instructor => “NA”, room => “TBA”},
        java =>  {instructor => NA2, room => TBA2},
);

print  $HoH{ python }{ instructor };            # Prints NA
print  $HoH{ java }{ room };                    # Prints TBA2
```

## AoH & HoH (adding elements)

```
$rec   =   {};                                  # Ref to anon hash
$rec->{$key}  =  $value                         # Populate hash
push  @AoH,  $rec;                              # Add hash-ref to array

push  @AoH,  { @fields };                       # Add anon hash-ref to array
```

```
$rec   =   {};                                  # Ref to anon hash
$HoH{$key1}   =  $rec                           # Add ref as element for $key1
$rec->{$key2}  =  $value                        # Populate hash for $key1

$HoH{$key1}{$key2}   =  $value                  # Populate hash from scratch
$HoH{$key}   =  { @fields };                    # Add anon hash-ref to hash
```

## Common Mistakes

- Printing values without dereferencing (get “stringifed” references!)
- With AoA: not composing new references for sub-arrays
- With loops: taking references to the same memory location

```
@AoA  =  ( [ 1, 2], [ “A1”,  “B2”, “C3” ] );              # Array of arrays
print   “@AoA”;                                           # WRONG! “ARRAY(#) ARRAY(#)…”

for  $i   (0..2)  {                  for  $i   (0..2)  {
    @A  =  somefunc($i);                 @A  =  somefunc($i);
    $AoA[$i]  =  @A;        }            $AoA[$i]  =  \@A;       }     # BOTH WRONG!

for  $i   (0..2)  {                  for  $i   (0..2)  {
    @A  =  somefunc($i);                 my @A  =  somefunc($i);
    $AoA[$i]  =  [ @A ];        }        $AoA[$i]  =  \@A;       }     # BOTH CORRECT
```

# HoA & HoH (sorting)

```
# Sort arrays (hash elements) by # of elements in each (decreasing)

foreach  $key  ( sort  { @{ $HoA{$b} } < = >  @{ $HoA{$a} } }  keys %HoA)  {
```

```
# Sort hashes of hashes first by keys of outer hash), then by keys of inner hash

foreach  $key1  ( sort  keys  %HoH )  {
        foreach  $key2  ( sort  keys  %{ $HoH{$key1} } )  {
```