CSCI 4152/6509
Natural Language Processing

Lab 3:

Perl Tutorial 3

Lab Instructor: Sigma Jahan and Tymon Wranik-Lohrenz

Faculty of Computer Science

Dalhousie University

Lab Overview

- We will finish Perl Tutorial with this Perl Tutorial 3
- In this lab you will learn more about
  - arrays
  - hashes
  - references
  - modules

Step 1. Logging in to server timberlea

1-a) Login to the server `timberlea`

1-b) Change directory to `csci4152` or `csci6509`

1-c) `mkdir lab3`

1-d) `cd lab3`

Arrays

- An array is an ordered list of scalar values
- Array variables start with `@` when referred in their entirety; examples:

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

- When referring to individual elements, use notation such as:

```
$animals[0] = 'Camel';
$numbers[4] = 70;
$mixed[1]++;
```

Arrays or Lists

- Perl arrays are dynamic, also called lists
- Some examples:

```
my @a = (); # creating an empty array
$a[5] = 10; # array extended to: ’’,’’,’’,’’,’’,10
$a[-2] = 9; # use of negative index, array is now
            # ’’,’’,’’,’’,9,10
print $#a;           # 5, index of the last element
print scalar(@a);    # 6, length of the array
for my $i (0..$#a) { # printing all elements
  print $a[$i]."\n"; }
```

Iterating over Arrays

- The loop `foreach` (or its synonym `for`)

```
my @a = ("a", "b", "c");
foreach my $element (@a)
{ print $element; }
```

- the default variable `$_` can be used in the `foreach` loop

```
foreach (@a)
{ print; }
```

- or using `for` and the index

```
for (my $i=0; $i<=$#a; $i++) {
   print $a[$i]; }
```

More about Array Functions (Operators)

- **push** `@a,` *elements*`;` or **push(**`@a,` *elements***)**`;`
- Example:

```
@a = (1,2,3);                # @a = (1, 2, 3)
push @a, 4;                  # @a = (1, 2, 3, 4)
```

- Built-in function **push** adds elements at the right end of an array
- Built-in functions generally do not require parentheses, but they are allowed, and sometimes needed to resolve ambiguities
- **pop** `@a;` removes and returns the rightmost element

```
$b = pop @a;          # $b=4, $a = (1, 2, 3)
```

Functions: shift, unshift, scalar; Array Flattening

- **shift** `@a;` removes leftmost element

```
@a = (3, 1, 2);
$b = shift @a;                # $b=3, $a = (1, 2)
```

- **unshift** `@a,` *elements*`;` adds at the left end

```
unshift @a, 5;                # @a = (5, 1, 2)
```

- Array flattening in Perl:

```
@a = ((1,2,3),4,(5));         # @a = (1, 2, 3, 4, 5)
@a = (0,@a,6);                # @a = (0, 1, 2, 3, 4, 5, 6)
```

- **scalar** `@a;` returns length of an array
- Remember that you can always used parentheses if you prefer, e.g.:

```
$n = scalar(@a);
unshift(@a,5);
push(@a,1,2,3);    # etc.
```

Revisiting Function print

- **print**, like many other functions, can take a list of arguments; e.g.:

  ```
  print 'Print ', 'a ', 'list', "\n";
  # prints: Print a list   (with a newline at the end)
  ```

  The arguments are printed sequentially without any spaces

- This is why we have:

  ```
  @a = (1, 2, 3);
  print @a;       # prints: 123
  print "@a";     # prints: 1 2 3
  ```

- The second example uses string interpolation, where an array is expanded by inserting a space between elements.
- Note that we do not use comma when printing to a file handle:

  ```
  print STDERR "print can use a file handle\n";
  print $fh    "printing to a file";
  ```

Function sort

- **sort** `@a;` sorts an array

  ```
  @a = sort (5, 1, 2);  # @a = (1, 2, 5)
  ```

- **sort** `BLOCK @a;` can be used to modify sorting criterion `BLOCK` decides how to compare two variables `$a` and `$b` by returning a result similar to `$a-$b`; i.e., negative means `$a<$b`, zero means `$a=$b`, and positive means `$a>$b`, with the final increasing order
- predefined operators `<=>` (numeric) and `cmp` (string) are usually useful; some simple examples:

  ```
  @a = (2, 1, 10, ’b’, ’a’)
  @a = sort @a; # string order: 1, 10, 2, a, b
  @a = sort {$a<=>$b} @a; # numeric order: a b 1 2 10
                          # (a, b treated as zeros)
  @a = sort {$b cmp $a} @a; # inverse string order:
                            # b a 2 10 1
  ```

Functions split and join

- **split** /*regex*/, *string*;   splits a string into array using a breaking regex pattern

```
$s = "This is a sentence.";
@a = split /[ .]+/, $s; # @a=('This','is','a',
                        #     'sentence')
```

- **join** *string*, *array*;   joins array elements into a string by inserting given string between elements

```
@a = (1, 2, 3);
$s = join ' <> ', @a; # $s = '1 <> 2 <> 3'
```

- We can also use parentheses: `join(`*string, array*`)`
- Array arguments can be given directly; e.g., `join '<>', 1, 2, 3;`
- And several arrays are flatten in one array; e.g., `join '<>', @a, @b;`

Functions grep and map

- **grep** `expr`, `@list` or **grep** `BLOCK @list` returns all elements of a list for which expression or block evaluates to true, when `$_` is given as the element of the list; example

  ```
  @a = grep {$_>0} @b; # @a gets positive elements
                       # of @b
  @foo = grep {!/^#/} @bar; # @foo gets all elements
      # (strings) from @bar that do not start with #
  ```

- **map** `expr`, `@list` or **map** `BLOCK @list` returns results of expression or block when applied on each element of the list, where `$_` given as element; example

  ```
  @a = map {lc $_} @b; # @a gets elements of @b in
                       # lowercase letters
  ```

- See `man perlfunc` for more about Perl functions

Step 2: Example with Arrays

- Type and test the following program in a file named ‘`lab3-array-examples.pl`’

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);

print "animals are @animals
that is: $animals[0] $animals[1] $animals[2]\n";
print "There is a total of ",$#animals+1," animals\n";
print "There is a total of ",scalar(@animals),
      " animals\n";

$animals[5] = ’lion’;
print "animals are @animals\n";
```

Submit: lab3-array-examples.pl

- Submit the file '`lab3-array-examples.pl`'
- This submission will be marked as a part of an Assignment

Associative Arrays (Hashes)

- Similar to array; associates keys with values
- Examples of use: phonebook, translation dictionary
- English-French translation example:

```
$fr{’one’}   = ’un’; $fr{’two’} = ’deux’;
$fr{’three’} = ’trois’;
$w = ’two’;
print $w." in French is ".$fr{$w};
# Output: two in French is deux
$w = ’four’;
print $w." in French is ".$fr{$w};
# Output: four in French is
# because $fr{’four’} is not assigned
```

Hashes: Declarations, Functions

- Hashes use `%` similarly like arrays use `@`, example

```
my %p;
my %p = (); # start with empty hash
my %p =
   ('one' => 'first', 'two' => 'second');
```

- The previous example shows how we can initialize a hash
- **exists** `$hash{$key}` checks if a hash element exists
- **delete** `$hash{$key}` deletes a hash element
- **keys** `$hash` gives an array of all keys
- **values** `$hash` gives and array of all values
- Keys and values are not returned in any particular order

Counting Words, Characters, and similar

- Hashes are convenient for counting words or other similar strings
- For example, if whenever we find a word `$w` in text

  ```
  $count{$w} += 1;  # add 1 to word count
  ```

  collects word counts in the hash `%count`
- If a word is not in the hash, the above expression adds it to the hash, and adding 1 makes the initial count 1, because an undef value is treated as a zero
- A sorted list of words can be obtained with

  ```
  @words = sort keys %count;
  ```

Iterating over a Hash

- Example of iterating over all key-value pairs in a hash:

```
my %p=(’one’=>’first’, ’two’ => ’second’);
foreach my $k (sort keys(%p)) {
    my $v = $p{$k};
    print "value for $k is $v\n";
}
```

- If we do not care about sorting keys, we can also use the special function **each**:

```
while (my ($k,$v) = each %p) {
  print "$k => $v\n";
}
```

‘Barewords’ in Keys

- For more convenience, words without quotes, or so-called *barewords,* are allowed to be used in hashes like this:

  ```
  %p = (one => first, two => second);
  $p{three} = ’third’;
  ```

- Even a starting minus sign is allowed, and used sometimes:

  ```
  %p = (-one => first, -two => second);
  $p{-three} = ’third’;
  ```

- Even the following would work:

  ```
  $p{-three} = third;
  ```

  but not if we defined a subroutine called ‘`third`’, so we should not use it.

Barewords in Perl

- Since Perl uses so much special symbols (sigils) as starting symbols in variables, it is interesting to see how are identifiers without such sigils treated
- *Barewords* are words (i.e., identifiers) that appear without initial special symbol (sigil) such as `$`, `@`, `%`, `&`, and `*`
- If they are not reserved words, or pre-defined function names, they are treated as literal strings; or forbidden if `use strict` is in place
- For example, the following code produces output shown:

```
$s = one.two; print "$s\n"; # out: onetwo
sub one { return "(1)" }
$s = one.two; print "$s\n"; # out:(1)two
sub two { return "(2)" }
$s = one.two; print "$s\n"; # out:(1)(2)
```

Barewords in Keys (2)

- Back to barewords in keys, hash keys have a stronger preference for barewords and are allowed even with ‘`use strict;`’. For example:

```
use strict;

my %a;
sub p { print join(’,’,%a), "\n"; }

%a = (one => ’one’);   p; # out: one,one
sub one { return "(1)" }
%a = ( one => one );   p; # out: one,(1)
%a = ( &one => one );  p; # out: (1),(1)
%a = ( (one) => one ); p; # out: (1),(1)
%a=(); $a{one} = one;  p; # out: one,(1)
%a=(); $a{&one} = one; p; # out: (1),(1)
```

Step 3: Example with Associative Array

- Write, test, and submit the following program in a file called `lab3-test-hash.pl`

```
#!/usr/bin/perl
# File: lab3-test-hash.pl

sub four { return ’sub4’ }
sub fourth { return ’sub4th’ }

%p = (one => first, -two => second);
$p{-three} = third;
$p{four} = fourth;
$p{four2} = ’fourth’;

for my $k ( sort keys %p ) { print "$k => $p{$k}\n" }
```

Step 4: `lab3-letter_counter_blanks.pl`

4-a) Copy the following files to your `lab4` directory:

```
 ~prof6509/public/TomSawyer.txt
 ~prof6509/public/lab3-letter_counter_blanks.pl
```

4-b) Open the file `lab3-letter_counter_blanks.pl` and fill in three blanks.

4-c) Run the command:

```
./lab3-letter_counter_blanks.pl TomSawyer.txt >
                                 lab3-out_letters.txt
```

4-d) Submit `lab3-letter_counter_blanks.pl` and

`lab3-out_letters.txt`

Step 5: `lab3-word_counter.pl`

- Write a Perl program `lab3-word_counter.pl` that counts words (all words must be translated into lowercase letters)
- Word is defined by regular expression `\w+`
- You may want to start with a copy of `lab3-letter_counter_blanks.pl`
- The program should print 10 most common words, and the number of hapax legomena
- Follow the rest of the specifications in the lab notes
- Submit the files: `lab3-word_counter.pl` and `lab3-out_word_counter.txt`

References to Arrays and Hashes

A reference is a scalar pointing to another data structure, usually an array or a hash:

```
my @a=(’Mon’,’Tue’,’Wed’); # an array
my %h = (’one’ => ’first’, ’two’ => ’second’); # a hash

my $ref_a = \@a; # reference to an array
my $ref_h = \%h; # reference to a hash
```

- References are necessary in Perl to make more complex data structures, such as creating arrays of arrays

Using References (1): Scalar Variables as References

Method 1: If your reference is a simple scalar variable, then wherever the identifier of an array or hash would be used as a part of an expression, one can use the variable that is the reference to the array or the hash, as in following examples:

```
@array=@a;      #using an array
@array=@$ref_a; #using a reference to an array

$element=$a[0];      #using an array
$element=$$ref_a[0]; #using a reference
$$ref_a[0]='xxx';    #using a reference

%hash=%h;      #using a hash
%hash=%$ref_h; #using a reference

$value=$h{'one'};      #using a hash
$value=$$ref_h{'one'}; #using a reference
$$ref_h{'one'}='f';    #using a reference
```

Using References (2): Using Braces { }

Method 2: Regardless whether your reference is a simple scalar or not. As Method 1, but enclose the reference in `{ }`

```
@array=@a;         #using an array
@array=@{$ref_a}; #using a reference

$element=$a[0];         #using an array
$element=${$ref_a}[0]; #using a reference

$value=$h{'one'};         #using a hash
$value=${$ref_h}{'one'}; #using a reference
```

While this is optional for simple scalars (i.e., you can use Method 1), this is necessary otherwise — for example when you store references to arrays in a hash `%hash_of_ref_to_arrays`

```
$value=${$hash_of_ref_to_arrays{'one'}}[0];
```

Using References (3): Arrow Operator ->

Method 3: Accessing elements of arrays or hashes using references directly and using the arrow operator ->

Instead of:

```
$$ref_a[0]
$$ref_h{'one'}
```

one can use:

```
$ref_a->[0]
$ref_h->{'one'}
```

Using References (4): Omitting Arrow Operator

If the arrow -> is between bracketed indexes of arrays or hashes, e.g.,

```
$ref_a->[0]->[10] #$ref_a is a reference to an array
                   #storing references to arrays
$ref_a->[0]->{'k'} #$ref_a is a reference to an array
                    #storing references to hashes
$ref_h->{'one'}->{'k'} #$ref_h is a reference to a hash
                       #storing references to hashes
```

then the arrow between bracketed indexes can be omitted

```
$ref_a->[0][10]
$ref_a->[0]{'k'}
$ref_h->{'one'}{'k'}
```

Using References to Pass Arrays or Hashes to a Subroutine

Arrays and hashes can be passed to a subroutine via references:

```
sub print_array {
    my $ref_a=shift; #takes a reference to an array
                    #as a parameter
    foreach my $element (@$ref_a) {
        print "Element: $element\n"
    }
}
sub add_element {
    my ($ref_a, $element) = @_;
    push(@$ref_a, $element);
}
my @a=(’Mon’,’Tue’,’Wed’); #array
add_element(\@a,’Thu’);
print_array(\@a); # array is changed
```

Passing Arrays or Hashes to Subroutine Directly

We can also pass arrays or hashes directly as list of arguments:

```
sub print_array {
  foreach my $e (@_) { print "Element: $e\n" }
}

sub print_hash {
  my %p = @_;
  foreach my $k (keys %p) { print "$k => $p{$k}\n" }
}

print_array(1, 2, 3, ’four’);
print_hash( one=>first, two=>second,
  ’any key’ => ’some value’ );
```

Step 6: `lab3-word_counter2.pl`

- Copy your previous program `lab3-word_counter.pl` to `lab3-word_counter2.pl`
- Add a subroutine `f` to the program `lab3-word_counter2.pl` that takes two parameters: a word and a reference to the hash that stores the frequencies of the words, and returns frequency of the input word, or 0 if it is not present.
- Test the program on `TomSawyer.txt` to find frequencies of the words ‘Tom’, ‘Sawyer’, and ‘Huck’
- Submit the program `lab3-word_counter2.pl`

Using Perl Modules and an Ngrams Module

- Perl module: `Text::Ngrams`
- Files available in: `~prof6509/public`

Step 7: Copy `Ngrams.pm` and `ngrams.pl`

- Use commands

```
cp ˜prof6509/public/ngrams.pl .
cp ˜prof6509/public/Ngrams.pm .
```

```
mkdir Text
cp Ngrams.pm Text
```

Step 8: Checking Modified `ngrams.pl`

```
#!/usr/bin/perl -w

use strict;
use vars qw($VERSION);
#<? read_starfish_conf(); echo "\$VERSION = $ModuleVersion;"; !>#+
$VERSION = 2.007;#-

use lib ’.’;

use Text::Ngrams;
use Getopt::Long;
...
```

Test ngrams.pl

- You can try the command:

```
./ngrams.pl
```

then typing some input, and pressing ‘`C-d`’; i.e., Control-D combination of keyboard keys. For example, if you type input:

```
natural language processing
```

you should get the output:

```
BEGIN OUTPUT BY Text::Ngrams version 2.007

1-GRAMS (total count: 28)
FIRST N-GRAM: N
 LAST N-GRAM: _
-------------------------
_ 3
A 4
```

Step 9: Test that `ngrams.pl` is using the local version of Ngrams module

- Insert temporarily a ‘`die`’ command in `Ngrams.pm`
- Try running `ngrams.pl`, and confirm that it reports an error
- Remove the ‘`die`’ command from `Ngrams.pm`

Step 10: Using the Ngram module

- Use the Ngram module on the `TomSawyer.txt` file, as specified in the notes
- Copy the file `~prof6509/public/TomSawyer.txt` to your `lab3` directory
- Run `ngrams.pl` and store output in `lab3-ngram-output.txt`
- Compress the output to `lab3-ngram-output.txt.gz` file
- Submit `lab3-ngram-output.txt.gz` file using `nlp-submit`

This is the end of Lab 3.