Faculty of Computer Science, Dalhousie University *29/31-Jan-2020*

CSCI 4152/6509 — Natural Language Processing

Lab 3: Perl Tutorial 3

Lab Instructor: Dijana Kosmajac, Tukai Pain
Location: rm1102, Mona Campbell building
Notes copyright: Vlado Keselj and Magdalena Jankowska

Perl Tutorial 3

Lab Overview

- We will continue with the Perl Tutorial
- In this lab you will learn more about
  - IO
  - arrays
  - hashes
  - references

Files to be submitted:

1. `array-examples.pl`
2. `test-hash.pl`
3. `letter_counter_blanks.pl`
4. `out_letters.txt`
5. `word_counter.pl`
6. `out_word_counter.txt`
7. `word_counter2.pl`

Step 1. Logging in to server bluenose

1-a) Login to the sever bluenose
As in the previous labs, login to your account on the server bluenose.
1-b) Change directory to `csci4152` or `csci6509`
Change your directory to `csci4152` or `csci6509`, whichever is your registered course. This directory should have been already created in your previous lab.

Create the directory `lab3` and change your current directory to be that directory:

1-c) `mkdir lab3`
1-d) `cd lab3`

This is the directory where you should keep files from this lab.

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
Perl arrays are dynamic in a sense of extensibility: we can easily make them larger or shorter without taking care of memory management. This is why they are also called lists.

– Some examples:

```
my @a = (); # creating 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
```

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]; }
```

The last example of the for-loop uses C-like form, similar to Java and some other languages. Namely, in the code:

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

part `my $i=0` declares a local variable `$i` and sets it initially to 0. The loop is repeated while `$i` is less or equal `$#a`. The variable `$#a` is a special variable which denotes the last index of the array `@a`. Since array indexing starts generally from 0, it means that `$#a` is equal to $n - 1$, where n is the number of elements of array `@a`. The part `$i++$` means that the index `$i` is incremented by 1 after each iteration of the loop.

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)
```

Array Functions: shift, unshift, sort, split

- **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)
```

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

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

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

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

Array Functions: join, print

- **join** *string1*`,` *string2*`;` joins array elements into a string

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

- **print** takes a list of arguments as well

```
print 'Print ', ' a', ' list', "\n";
print STDERR "print can use a filehandle\n";
```

Step 2: Example with Arrays

- Type and test the following program in a file named '`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: Submit the program `array-examples.pl` using the `submit-nlp` command.

Associative Arrays (Hashes)

– Similar to array; associates keys with values
– Example

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

– **keys** returns an array of keys (in no specific order)
– **values** returns an array of values (in no specific order)
– Examples

```
@a = keys %p;   # or keys(%p), no order
@b = values %p; # or values(%p), no order
```

Iterating over a Hash

– Example

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

'Barewords' in Keys

– For more convenience, so-called barewords are allowed without quotes as keys in hashes; e.g.:

```
%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`'

Step 3: Example with Associative Array

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

```
#!/usr/bin/perl
# File: 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" }
```

Submit: Submit the program `test-hash.pl` using the `submit-nlp` command.

Step 4: `letter_counter_blanks.pl`

4-a) Copy the following files to your `lab4` directory:
`~prof6509/public/TomSawyer.txt`
`~prof6509/public/letter_counter_blanks.pl`

If you forgot the copy command on bluenose (a Linux system), remember that you can use the following commands:

```
cp ~prof6509/public/TomSawyer.txt .
cp ~prof6509/public/letter_counter_blanks.pl .
```

The first file is the novel “The Adventures of Tom Sawyer” by Mark Twain that we saw before, and the second file is a sample unfinished program.

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

The program counts the frequencies of letters (case insensitive) in an input text from standard input or from files specified in the command line. It is supposed to print the letters with their frequencies in the decreasing order of their frequency.

4-c) Run the program on the file `TomSawyer.txt` and save the output to the file `out_letters.txt` (the following is a one-line command):

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

4-d) Submit `letter_counter_blanks.pl` and `out_letters.txt`

Submit: Submit the program `letter_counter_blanks.pl` and the file `out_letters.txt` using the `submit-nlp` command.

Step 5: `word_counter.pl`

Write a Perl program `word_counter.pl` that counts words (**case insensitive**) in an input text from the standard input or from files specified in the command line. We will define a word here as a maximum sequence of Perl “word characters”: letters, digits or the underscore, that is as a string captured by the regular expression `\w+`

You may want to copy your program `letter_counter_blanks.pl` and modify it, so that it counts words instead of letters.

The program must not print all extracted words. Instead, it should provide answers to the following two questions:
1. What are 10 most common words in the input text? and
2. How many words appear only once in the input text?

A word that appears only once in a text is called *hapax legomenon,* or in plural *hapax legomena*.

The program must produce output in the following format:

```
10 most common words are:
word1 word2 word3 word4 word5 word6 word7 word8 word9 word10
The number of hapax legomena is ???
```

where `word1` `word2` etc. are the ten most common words sorted from the most frequent to the least frequent. If some words among these ten most frequent words have the same frequency, then their exact order does not matter. If input does not have 10 unique words, then print as many as there are, still sorted by frequency from the most frequent down. The string `???` above should be replaced with the number of hapax legomena.

Run the program on the file `TomSawyer.txt`. Save the output to the file `out_word_counter.txt`:

```
./word_counter.pl TomSawyer.txt > out_word_counter.txt
```

Submit: Submit the files: `word_counter.pl` and `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
```

Using References (1)

Method 1: If your reference is a simple scalar, 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)

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)

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 (3)

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: `word_counter2.pl`

Copy the previous program `word_counter.pl` to `word_counter2.pl`. You should now edit the program `word_counter2.pl` and add a new subroutine to it. The subroutine named `f` should take two parameters: a word and a reference to the hash that stores the frequencies of words. It should return the frequency of the input word (0, if it is not present in the hash).

Test the program `word_counter2.pl` on the file `TomSawyer.txt` to find the frequencies of the words: ‘tom’, ‘sawyer’, and ‘huck’.

Submit: Submit the program `word_counter2.pl`

This is the end of Lab 3.