CSCI 4152/6509
Natural Language Processing

Perl Tutorial

CSCI 4152/6509

Vlado Kešelj

About Perl

- created in 1987 by Larry Wall
- interpreted language, with just-in-time semi-compilation
- provides effective string manipulation, brief if needed
- convenient for system tasks
- syntax (and semantics) similar to:
  C, shell scripts, awk, sed, even Lisp, C++

Perl Strengths

- good prototyping language, expressive: It can express a lot in a few lines of code.
- can be used incrementally: useful even if you learn a small part of it. It becomes more useful when you know more; i.e., its learning curve is not steep.
- flexible; e.g, most tasks can be done in more than one way
- garbage collection: i.e., no worries about memory management
- free, open-source; portable, extensible
- powerful, string and data manipulation, regular expressions
- efficient, especially considering it is an interpreted language
- supports Object-Oriented style

Perl Weaknesses

- not as efficient as C/C++
- may not be very readable without prior knowledge
- OO features are an add-on, rather than built-in
- not a steep learning curve, but a long one
  (which is not necessarily a weakness)

Hello world

Choose your favourite editor and edit `hello.pl`:

```
print "Hello world!\n";
```

Type “`perl hello.pl`” to run the program, which should produce:

```
Hello world!
```

You can execute the Perl code by directly interacting with the Perl interpreter:

```
perl
print "Hello world!\n";
^D
```

(The last `^D` is actually Ctrl+D.)

This means that you can also do: `perl < hello.pl`

Another way to run a program

Let us edit again `hello.pl` into:

```
#!/usr/bin/perl
print "Hello world!\n";
```

Change permissions of the program and run it:

```
chmod u+x hello.pl
./hello.pl
```

Running ‘`perl -w hello.pl`’ may print useful warnings. The same effect is achieved by running:

```
#!/usr/bin/perl -w
print "Hello world!\n";
```

File Names

- extension '`.pl`' is common, but not mandatory
- extension '`.pm`' is used for Perl modules

Finding Help

- `man perl`, `man perlintro`, ...
- Web: `perl.com`, `CPAN.org`, `perlmonks.org`, ...
- books: the "Camel" book:
  "Learning Perl, 4th Edition" by Brian D. Foy; Tom Phoenix; Randal L. Schwartz (2005)
  Available on-line on Safari at Dalhousie
  `http://proquest.safaribooksonline.com/0596101058`

Syntactic Elements

- statements separated by semi-colon '`;`'
- white space does not matter except in strings
- line comments begin with '`#`'; e.g.
  `# a comment until the end of line`
- variable names start with $, @, or %:
  `$a` — a scalar variable
  `@a` — an array variable
  `%a` — an associative array (or hash)
  However: `$a[5]` is 5th element of an array, and
  `$a{5}` is a value associated with key 5 in hash `%a`
- the starting special symbol is followed either by a name (e.g., `$varname`) or a non-letter symbol (e.g., `$!`)
- user-defined subroutines are usually prefixed wiht &:
  `&a` — call the subroutine `a` (procedure, function)

Example Program 2

We can call this program `prog2.pl`:

```
#!/usr/bin/perl

print "What is your name? ";
$name = <>;
chomp $name;
print "Hello $name!\n";
```

`chomp` removes the trailing newline from `$name` if there is one. However, changing the special variable `$/` will change the behaviour of `chomp` too.

The declaration “`use strict;`” is useful to force more strict verification of the code. If it is used in the previous program, Perl will complain about variable `$name` not being declared, so you can declare it:

Examples 3 and 4

```
#!/usr/bin/perl
use strict;
my $name;
print "What is your name? ";
$name = <>;
chomp $name;
print "Hello $name!\n";
```

or

```
#!/usr/bin/perl
use strict;

print "What is your name? ";
my $name = <>;
chomp $name;
print "Hello $name!\n";
```

Example 5: Copy standard input to standard output

```
#!/usr/bin/perl
while ($line = <>) {
    print $line;
}
```

Special variable `$_` is the default variable for many commands, including `print` and expression `while (<>)`, so another version of the program would be:

```
#!/usr/bin/perl
while (<>) { print }
```

or even shorter

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

Variables

- no need to declare them unless “`use strict;`” is in place
- `use strict;` is a good practice for larger projects
- variable type is not declared (it is inferred from context)
- the main variable types:

  1. Scalars
     - numbers (integers and floating-point)
     - strings
     - references (pointers)
  2. Arrays of scalars
  3. Hashes (associative arrays) of scalars

Single-Quoted String Literals

```
  print ’hello\n’;           # produces ’hello\n’
  print ’It is 5 o\’clock!’; # ’ has to be escaped
  print q(another way of ’single-quoting’);
                     # no need to escape this time
  print q< and another way >;
  print q{ and another way };
  print q[ and another way ];
  print q- and another way with almost
           arbitrary character (e.g. not q)-;
  print ’A multi line
      string (embedded new-line characters)’;
  print <<’EOT’;
Some lines of text
  and more $a @b
EOT
```

Double-Quoted String Literals

```
print "Backslash combinations are interpreted in
       double-quoted strings.\n";
print "newline after this\n";
$a = ’are’;
print "variables $a interpolated in double-quoted
       strings\n";
# produces "variables are interpolated" etc.
@a = (’arrays’, ’too’);
print "and @a\n";
# produces "and arrays too" and a newline

print qq{Similarly to single-quoted, this is also
         a double-quoted string, (etc.)};
```

Scalar Variables

- name starts with `$` followed by:

  1. a letter and a sequence of letters, digits or underscores, or
  2. a special character such as punctuation or digit

- contains a single scalar value such as a number, string, or reference (a pointer)

- do not need to worry whether a number is actually a number or string representation of a number

```
$a = 5.5;
$b = " $a ";
print $a+$b;
```

(11)

Numerical Operators

- basic operations: `+ - * /`
- transparent conversion between int and float
- additional operators:
  `**` (exponentiation), `%` (modulo), `++` and `--` (post/pre inc/decrement, like in C/C++, Java)
- can be combined into assignment operators:
  `+= -= /= *= %= **=`

String Operators

- `.` is concatenation; e.g., `$a.$b`
- `x` is string repetition operator; e.g.,

```
print "This sentence goes on"." and on" x 4;
```

produces:

```
This sentence goes on and on and on and on and on
```

- assignment operators:
  `=   .=   x=`
- string find and extract functions: `index(str,substr[,offset])`, and `substr(str,offset[,len])`

Comparison operators

Operation	Numeric	String
less than	<	lt
less than or equal to	<=	le
greater than	>	gt
greater than or equal to	>=	ge
equal to	==	eq
not equal to	!=	ne
compare	<=>	cmp

Example:

```
print ">".(1==1)."<";  # produces: >1<
print ">".(1==0)."<";  # produces: ><
```

What is true and what is false — Beware

```
print ’’     ?’true’:’false’; #false
print 1      ?’true’:’false’; #true
print ’1’    ?’true’:’false’; #true
print 0      ?’true’:’false’; #false
print ’0’    ?’true’:’false’; #false
print ’ 0’   ?’true’:’false’; #true
print 0.0    ?’true’:’false’; #false
print "0.0"  ?’true’:’false’; #true
print ’true’ ?’true’:’false’; #true
print ’zero’ ?’true’:’false’; #true
```

The false values are: `0`, `’’`, `’0’`, or `undef`
True is anything else.

`<=>` and cmp

`$a <=> $b` and `$a cmp $b` return the sign of `$a - $b` in a sense:

-1 if `$a < $b` or `$a lt $b`,
0 if `$a == $b` or `$a eq $b`, and
1 if `$a > $b` or `$a gt $b`.

Useful with the `sort` command

```
@a = (’123’, ’19’, ’124’);
@a = sort @a;                print "@a\n"; # 123 124 19
@a = sort {$a<=>$b}   @a; print "@a\n"; # 19 123 124
@a = sort {$b<=>$a}   @a; print "@a\n"; # 124 123 19
@a = sort {$a cmp $b} @a; print "@a\n"; # 123 124 19
@a = sort {$b cmp $a} @a; print "@a\n"; # 19 124 123
```

Boolean Operators

```
Six operators:  &&  and
                ||  or
                !   not
```

Difference between && and ‘and’ operators is in precedence, and similarly for others.

Range Operators

```
..   - creates a list in list context,
       flip-flop otherwise
...  - same, except for flip-flop behaviour

@a = 1..10;   print "@a\n"; # out: 1 2 3...
@a = -5 .. 5; print "@a\n"; # out?
$a = 1; $b = 5; @c = ($a .. $b, -2 .. 2);
print "@c\n";               # ?
print map{$_.="\n"} (’aa’..’zz’);
```

Arrays

- an array is an ordered list of scalar values
- example

```
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";
```

Some Array Functions (Operators)

```
@a = (1,2,3);            # @a = (1, 2, 3)
push @a, 4;              # @a = (1, 2, 3, 4)
$b = pop @a;             # $b=4, $a = (1, 2, 3)
$b = shift @a;           # $b=1, $a = (2, 3)
unshift @a, 5;           # @a = (5, 2, 3)

$s = "This is a sentence.";
@a = split /[ .]+/, $s;
$s = join ’ <> ’, @a;
print $s, "\n";

print ’Print ’, ’is ’, ’also a list operator’, "\n";
print STDERR "print can use a filehandle\n";
```

Hashes (Associative arrays)

- a structure, associates keys with values
- example

```
%p = (’one’ => ’first’, ’two’ => ’second’);
$p{’three’} = ’third’;
$p{’four’} = ’fourth’;
@a = keys %p;   # or keys(%p), no order
@b = values %p; # or values(%p), no order
```

Control Structures

- `if-elsif-else` and `unless`
- while loop
- for loop
- foreach loop

If-elsif-else

```
if (EXPRESSION) {
  STATEMENTS;
} elsif {           # optional
  STATEMENTS;
} elsif {      # optional additional elsif’s
  STATEMENTS;
} else {
  STATEMENTS;  # optional else
}
```

Other equivalent forms, e.g.:

```
if ($x > $y) { $a = $x }
$a = $x if $x > $y;
$a = $x unless $x <= $y;
unless ($x <= $y) { $a = $x }
```

While Loop

```
while (EXPRESSION) {
  STATEMENTS;
}
```

- `last` is used to break the loop (like `break` in C/C++/Java)
- `next` is used to start next iteration (like `continue`)
- `redo` is similar to next, except that the loop condition is not evaluated
- labels are used to break from non-innermost loop, e.g.:

```
L:
while (EXPRESSION) {
   ... while (E1) { ...
         last L;
}  }
```

next vs. redo

```
#!/usr/bin/perl

$i=0;
while (++$i < 5) {
    print "($i) "; ++$i;
    next if $i==2;
    print "$i ";
} # output: (1) (3) 4

$i=0;
while (++$i < 5) {
    print "($i) "; ++$i;
    redo if $i==2;
    print "$i ";
} # output: (1) (2) 3 (4) 5
```

For Loop

```
for ( INIT_EXPR; COND_EXPR; LOOP_EXPR ) {
   STATEMENTS;
}
```

Example:

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

Foreach Loop

Examples:

```
@a = ( ’lion’, ’zebra’, ’giraffe’ );
foreach $a (@a) { print "$a is an animal\n" }

# or use default variable
foreach (@a) { print "$_ is an animal\n" }

# more examples
foreach my $a (@a, ’horse’) { print "$a is animal\n"}

foreach (1..50) { print "$_, " }
```

`for` can be used instead of `foreach\` as a synonym.

Basic I/O

```
# read STDIN and print, or from file specified
# in the command line
while ($line = <>) { print $line }

# or
while (<>) { print } # using default variable $_

$line = <>;  # reads one line
@lines = <>; # reads all lines,
             # (context-dependent behaviour)

print "a line\n"; # output, or
printf "%10s %10d %12.4f\n", $s, $n, $fl;
     # formatted output
```

Subroutines

```
sub say_hi { print "Hello\n"; }
&say_hi();   # call
&say_hi;     # call, another way since we have no params
say_hi;      # works as well (no variable sign =
             #  sub, i.e., &)

sub add2 {
    my $a = shift; my $b = shift;
    return $a + $b;
}
print &add2(2,5);  # produces 7

# alternative definition
sub add2 { return $_[0] + $_[1] }
# @_ is array of parameters
# shift with no arguments takes @_ by default
# (or @ARGV outside of a subroutine)
```

Subroutines (2)

```
sub add {
  my $ret = 0;
  while (@_) { $ret += shift }
  return $ret;
}
print &add(1..10); # produces 55
```

Regular Expressions

A simple way to test a regular expression:

```
while (<>) {
   print if /book/;
}
```

i.e., print lines that contain substring ‘`book`’

`/chee[sp]eca[rk]e/` would match: cheesecare, cheepecare, cheesecake, cheepecake

option `/i` matches case variants; e.g., `/book/i` would match Book, BOOK, bOoK, etc., as well

RegEx: Character Class

/200[012345]/	match one of the chars
/200[0-9]/	character range
/From[^:]/	match any character but
/[^a-zA-Z]the[^a-zA-Z]/	multiple ranges
/[]]/	to match]
/[]-]/	to match] or -
/[$^]/	to match $ or ^
[0-9ABCDEFa-f]	to match one-digit hexadecimal number
.	(period) any character but new-line
\d	any digit; i.e., same as [0-9]
\D	any character but digit
\s	any white-space character, including new-line
\S	any character but white-space, i.e., printable
\w	any word character (letter, digit, or underscore)
\W	any non-word character