Introduction to Perl: Third Lecture

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

January 2015
In this third session, we’ll go over more of Perl in detail, we’ll look at:

- arrays and lists
- hashes
- special variables (@ARGV, $_, %ENV)
- some more Perl one-liners, and
- regular expressions.

Aside: A better way to run Perl Programs

- We have seen that when we want to run a Perl program called eg1, we say: perl eg1.
- Wouldn’t it be better if we could just type eg1 to run our program?
- Then we could install our own Perl programs in a public place and let our friends run them - without them caring what language the programs are written in!
Well, on Unix we can:

- First, issue the Unix command:

  chmod +x eg1

  This makes the file executable.

- Second, edit `eg1` and add the following line at the top:

  `#!/usr/bin/perl`

  This is a special line interpreted by Unix when it executes a non-machine code program.

  Unix executes the named program (the Perl interpreter) with the script `eg1` as a command line argument.

  Perl starts up, reads `eg1` and proceeds to run it - and then ignores the first line because it’s a comment!

  This is why all Unix shells and most scripting languages use ‘#’ as their one-line comment character.

- Now, run `eg1` by `eg1` (if . is on your path), or `./eg1` if not.
Arrays and Lists

- An array is an ordered collection of scalars (strings or numbers), declared via `my @array`, the `@` being compulsory.

- An array such as `@fred` is *not the same as* `$fred`. Perl keeps the namespaces of arrays and scalars separate.

- Array indices start at 0.

- An array may be built up piece by piece:

  ```perl
  my @fred;
  $fred[0] = "hello";
  $fred[1] = 7.1+$a;
  $fred[2] = 17.3;
  $fred[3] = $c;
  ```

- Each element of the array is a scalar, which is why an individual element of `@fred` is accessed using `$fred[expr]` not `@fred[expr]`. This is admittedly confusing! Perl 6 will use `@fred[expr]`.

- Assigning to an element beyond the current end of the array (e.g. `$fred[10]=42`) extends the array. Intervening elements (here 4..9) become the undefined value, which looks just like 0.
A single item may be extracted from an array:

\[
\textit{\$sum += \$fred[\textit{i}];}
\]

The index expression will be truncated to an integer before the array is accessed.

Building an array piece by piece is painful: assign a bracketed comma-separated list of scalars straight into an array:

\[
\text{my @fred = ('hello', 7.1+$a, 17.3, $c );}
\]

Inside a list, the .. operator can be used as in @fred = ( 1..20 ) or @let = ( 'a'..'z' ).

If you have a list of single words, for example:

\[
\text{my @fred = ('hello', 'there', 'how', 'are', 'you');}
\]

Perl provides the quote words syntactic sugar:

\[
\text{my @fred = qw(hello there how are you);}
\]

You can iterate over an array by:

\[
\text{foreach my \$element (@fred)}
\{
\text{# now do something with \$element}
\}
\]
Arrays and Lists

Arrays as Tuples

- You can also break up an array into a list of variables:
  
  ```perl
  my( $a, $b, $c ) = @fred;
  ```

- This copies $fred[0] to $a, $fred[1] to $b and $fred[2] to $c. Any remaining elements in the array are ignored. If @fred has (say) only 2 elements then $c is set to the undefined value.

- An array can be used to soak up the remainder:
  
  ```perl
  my( $a, $b, @c ) = @fred;
  ```

- Can even put the remainder back in @fred:
  
  ```perl
  my( $a, $b ); or... ( my $a, my $b, @fred ) = @fred;
  ( $a, $b, @fred ) = @fred;
  ```

- Tupling gives you a very easy swap operation:
  
  ```perl
  ( $x, $y ) = ( $y, $x );
  ```

  which takes y and x, forms them into a two-element list, and assigns the first two elements of that list back into x and y.

- In summary, Perl arrays act as dynamic arrays, tuples, stacks and queues (as we’ll see later).
Some operators behave differently when placed in scalar context or in list context. List context is where a list is expected rather than a scalar eg. assigning to an array evaluates the RHS in list context. Also, arguments of print() are evaluated in list context.

<> is one such operator:

- In scalar context, eg $line = <$in>, it reads a single line.
- In list context, eg @line = <$in>, it reads the rest of the input, returning an array of lines - still with all the newlines.
- Fortunately, chomp @line chomps the newline from every line.

Similarly, array assignment:

- Assigning array to array, eg. @x = @y, copies the entire array.
- Assigning an array to a scalar, eg my $count = @y, means set $count to the number of elements in @y. i.e. the length of the array.
- Why? Because Larry Wall thought: what is the most commonly used scalar property of an array? and answered the length.

You can force a scalar context when you’re not sure what Perl would do by wrapping an expression in the function scalar().
• Declare a hash variable by my %fred, such a hash occupies a different namespace from $fred and @fred.

• A hash stores (*key*, *value*) pairs - for each string scalar (the *key*), it stores an arbitrary scalar (the *value*).

• Think: a two-column database table stored in memory, from unique keys to non-unique values, indexed on keys:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcw</td>
<td>225</td>
</tr>
<tr>
<td>ldk</td>
<td>225</td>
</tr>
<tr>
<td>sza</td>
<td>225</td>
</tr>
<tr>
<td>mjw03</td>
<td>228</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>

• Hashes have a highly efficient indexing system so you can look up a key’s associated value very quickly. Hashes are implemented as hash tables, hence the name.

• No equivalent mechanism of looking up which key(s) corresponds to a particular (non unique) value.

• If your values happen to be unique too: use two hashes, one mapping k→v and the other mapping v→k.
A hash literal can be written as a list of pairs with the key => value (*fat comma*) syntactic sugar:

```perl
my %roomno = (
  "dcw" => "225", "ldk" => "225",
  "sza" => "225", "mjw03" => "228"
);
```

The entire hash may be cleared by:

```perl
%roomno = ();
```

To add a single (key, value) pair into a hash, do:

```perl
$roomno{"susan"} = "566";
```

Perl allows you to omit the key quotes: `$roomno{susan} = "566";`

Our original hash literal example could be written as:

```perl
my %roomno = ();
$roomno{dcw} = "225"; $roomno{ldk} = "225";
$roomno{sza} = "225"; $roomno{mjw03} = "228";
```

To check whether a key is present in the hash, use `exists`, eg:

```perl
print "elvis has left the building\n" unless exists $roomno{elvis};
```

To retrieve a particular value from a hash, use:

```perl
my $room = $roomno{$person};
```

If the key `$person` is not present in the hash, the undefined value is returned.
To delete a single *(key, value)* pair from a hash:

```perl
delete $roomno{dcw};
```

To process an entire hash, you can use the **keys()** function:

```perl
foreach my $key (keys %roomno)
{
    my $value = $roomno{$key};
    print "$key in room $value\n";
}
```

`keys %roomno` builds a list containing all keys of `%roomno`. Could be huge!

Note: keys come out in an efficient *hash-table traversal* order - not alphabetical order! Hence, you often see:

```perl
foreach my $key (sort keys %roomno) # foreach sorted key in %roomno
{
    my $value = $roomno{$key};
    print "$key in room $value\n";
}
```

The idiomatic way to process both keys and values, in any order, is to use the **each()** function and a **while** loop:

```perl
while( my($key,$value) = each %roomno ) # foreach (key,value) pair in %roomno
{
    print "$key in room $value\n";
}
```

See **eg2** for a longer example of how to use hashes.
Perl has many special variables (see `perldoc perlvar` for a complete list). Here are a few of the most useful:

- In Unix, *environment variables are arbitrary* (name, value) *pairs*, created by `setenv NAME value` commands in the shell (by convention, uppercase names).
- To see the current set of environment variables, type `env` at the command line. A list of NAME=value pairs fly past.
- Once set, environment variables are passed around automatically to every Unix process in the current session. Perl makes these variables accessible via a single hash called `%ENV`.
- For example, an important environment variable is `HOME` (the pathname of your home directory). Get this by:

  ```perl
  my $home = $ENV{HOME} || die "no home?\n";
  ```

- Other platforms – such as Windows – also have environment variables, Perl on those platforms can access environment variables in the same way, but of course what environment variables exist and what they mean) are different.
When you invoke one of your Perl programs, you can place *arguments* on the command line, eg:

```
myprog first second third
```

When you do this, Perl makes the strings first, second and third available in a special array called `@ARGV`. Specifically:

```
$ARGV[0] = "first";
$ARGV[1] = "second";
$ARGV[2] = "third";
```

As usual, `@ARGV` evaluated in a scalar context gives the number of elements (in the example, 3).

The array function `shift()` can be used on `@ARGV`:

```
my $arg = shift @ARGV;
```

This sets `$arg` to element 0 of the array, and removes that element from the array, shifting the other elements down one.

Of course: it’s up to the program to decide what the strings *mean*!
If they are filenames to be opened and processed, the *open and process every line in every file* idiom is often used:

```perl
foreach my $arg (@ARGV)
{
    open( my $in, '<', $arg ) || next;
    while( my $line = <$in> )
    {
        chomp $line;
        # now process $line
    }
    close( $in );
}
```

This (processing several files, not caring where one ends and the next begins) is so common that Perl has a special shorthand:

```perl
while( my $line = <> )
{
    chomp $line;
    # now process $line
}
```

Exercise: generalise one of the earlier STDIN or single-file processing programs to take one or more command line arguments using either of these idioms.
You may find a puzzling shorthand, as in eg3:

```perl
while( <> )
{
    chomp;
    print "found '$_'\n" if /dun[ck]/i;
}
```

- Where are we storing the line we read with `<>`?
- What are we chomping?
- What are we case-insensitively matching `/dun[ck]/i` against?
- What’s that `$_` interpolated into the `print`?

- `$_` is the *implicit variable*: the *default argument* to many functions:
  - The default variable where `<>` stores its input line.
  - The default variable that `chomp` modifies.
  - The default variable to match a regex against.
  - The default value to print if none is given.
  - The default foreach variable, as in `foreach (@array)`.  
  - .. and many more cases.
The entire framework:

```perl
while( <> )
{
    chomp;
    # DO SOMETHING
}
```

may be wrapped around your Perl one-liner using perl’s `-nle` flags. So:

```perl
perl -nle 'print qq(found "$_") if /dun[ck]/i' list_of_files
```

is a poor man’s customised `grep`.

A useful feature to use with `-nle` oneliners is Perl’s `END { block }` syntax, which runs (you guessed it) after all input is exhausted. This allows you to do things like:

```perl
perl -nle '$sum += $_; END {print $sum}' list_of_files
```

Another useful Perl flag to use with `-nle` is `-a` - this autosplits every line on whitespace into an array of fields, allowing you (for example) to sum up column 5 of `ls -al`’s output:

```
ls -al | perl -lane '$sum += $F[4]; END {print $sum}'
```
We saw in the first session that we could write:

```
if( $name =~ /^Dun\[ck]/ )
```

This is an example of matching a string against a regular expression (or regex), as in the Unix filters `sed`, `grep` and `awk`.

A regular expression is a way of describing a class of similar strings in a very compact pattern notation. In the above example, the match will succeed if the current value of `$name` starts with:

- A capital ‘D’ [must be the very first character],
- The lower case letters ‘u’ and ‘n’ as the next two characters,
- then *either* a lower case ‘c’ or a ‘k’.

A whole regex is (usually) placed inside a pair of ‘/’ signs. Within the slashes, characters are interpreted pretty much like in a double-quoted string. In particular, variables are interpolated before pattern-matching occurs.

A regex is made up of single character patterns, grouping patterns, alternation patterns, anchoring patterns and bracketing patterns. We’ll look at each in turn.
• ‘.’ matches any single character.

• A single printable character matches itself (except meta-characters like ‘.’, ‘*’ etc, which may be preceded by a backslash when you really want to match the character itself!).

• [set] matches any single character in the set.
  For example, [aeiou] matches any single lower-case vowel.

• Also, the set may contain items of the form a-f, which is a shorthand for abcdef.
  For example, [a-z#%] matches any single lower-case letter, a hash-mark, or a percent sign.

• If a set starts with a ‘~’ character (eg. [^a–z#%]), the set is negated - the pattern matches any character NOT in the set.

• Several useful character classes are predefined:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Digit</td>
<td>\d</td>
<td>[0–9]</td>
</tr>
<tr>
<td>Non-digit</td>
<td>\D</td>
<td>[^0–9]</td>
</tr>
<tr>
<td>Word</td>
<td>\w</td>
<td>[a-zA-Z0-9_]</td>
</tr>
<tr>
<td>Non-word</td>
<td>\W</td>
<td>[^a-zA-Z0-9_]</td>
</tr>
<tr>
<td>Whitespace</td>
<td>\s</td>
<td>space or tab</td>
</tr>
<tr>
<td>Non-whitespace</td>
<td>\S</td>
<td>not space or tab</td>
</tr>
</tbody>
</table>
- **Sequence of single-character patterns**: matches a corresponding sequence of characters. eg. `/[a-z]bc/` matches any lower case letter, followed immediately by a ‘b’, followed immediately by a ‘c’, anywhere in the string.

- **Optional**: ‘?’ makes the previous pattern optional - i.e. match zero or one times. eg. `/he?llo/` matches ‘hello’ or ‘hloo’.

- **Zero-or-more**: ‘*’ makes the previous pattern apply any number of times (from 0 upwards). eg. `/he*llo/` matches ‘hlo’, ‘hello’, ‘heello’ etc. It consumes the maximum number of ‘e’’s possible (it’s greedy).

- **One-or-more**: ‘+’ means match 1 or more times. eg. `/he+llo/` matches ‘hello’, ‘heello’, ‘heeeello’ etc but not ‘hloo’.

- If the greediness of ‘*’ and ‘+’ is ever a problem, use ‘*?’ or ‘+?’ to consume as few characters as possible.

- A regex can contain several of these operators: eg: `/h[uea]*1+o/` matches ‘hlo’, ‘hullo’, ‘hulllllo’, ‘heelo’, ‘heuauaeaeuellllllllo’ etc.
Placing ‘^’ at the start of a regex matches the *start* of the string. Similarly, ‘$’ at the end of a regex matches the *end* of the string.

‘\b’ constrains the regex to match only at a word boundary.

Without any anchoring, the regex can match anywhere.

There are two main ways of using regexes:

- **To check whether a string matches a regex.** We specify the string to match against using the =~ operator, or the *not match* operator !~:

  ```perl
  print "<$str> matches
" if $str =~ /h[aeiou]+l+o/;
  ```

  If a regex match is followed by i, as in /h[aeiou]+l+o/i, the matching is done case insensitively.

- **Secondly, a regex can be used to search and replace all occurrences of a regex within a string** (again, we specify the string to modify using the =~ operator):

  ```perl
  $str =~ s/[aeiou]+/a/g;
  ```

  The trailing g makes Perl replace ALL vowel sequences in $str with ‘a’. Without the g Perl would only replace the first match.
As a general way of testing regular expressions, I recommend a program like `eg4`:

```perl
#!/usr/bin/perl
#
# eg3: regex test harness..
#
print "Please enter a string: ";
my $str = <STDIN>;
chomp $str;
print "\nat start : <$str>\n";

# test search and replace:
$str =~ s/^\s+//;
print "\nafter s///: <$str>\n";

# test pattern match:
print "\n<$str> matches hello regex\n" if $str =~ /h[ea][au]\*l+o/;
```

This whole program exists in order to let you test search and replace and/or pattern matches using a string entered at the keyboard. By the way, `$s/^\s+//` is a useful regex - worth committing to memory - that removes any leading whitespace. Similarly, `$s/\s+$//` removes trailing whitespace.

I strongly recommend that you use this program to test lots of different regexes and their behaviour against various strings.
A regex of the form `/h[eu]a*llo|wo+tcha/` matches *either* `/h[eu]a*llo/` or `/wo+tcha/`. Note that `/a|b|c|g/` should be written as `/[abcg]/` instead for efficiency.

Brackets may be placed around any complete sub-pattern, as a way of enforcing a desired precedence. For example, in `/so+ng|bla+ckbird/` obviously `bird` is only part of `bla+ckbird`).

If you meant "'/so+ng|bla+ck/ followed by /bird/"`, then write that as `/ (so+ng|bla+ck)bird/`.

If you want a repetition of *anything* longer than a single character pattern, you need brackets, as in `/(hello)*/`. Without brackets, `/hello*/` means `/hell/` followed by `/o*/` of course!

Brackets have another useful side effect: they tell Perl’s regex engine to *remember* or *capture* the text fragment that matched the inner pattern for later reporting or reuse. eg:

```perl
my $str = "I’m a melodious little soooongbird, hear me sing";
print "found <$1>\n" if $str =~ /(so+ng|bla+ck)bird/;
```

After the match succeeds, the capture buffer variable `$1` contains `soooong` - the part of `$str` matching the bracketed regex.
Aside: to turn capturing off, while retaining the grouping behaviour: use (?:inner), eg. /(?+:so+ng|bla+ck)bird/.

Use up to nine bracketed sub-patterns in a single pattern match - capture variables $1$ to $9$ - available for use as soon as the pattern match has succeeded.

Capture buffers can be used in a search and replace operation:

```
$str =~ s/\s*(\w+)\s+(\w+)/$2 $1/;
```

which swaps the first two space separated words in the string (if there are two space separated words at the start of the string).

Another example: /first(.* )second/ matches exactly the same strings as /first.* second/, but remembers the particular sequence of characters found between first and second as $1$.

If the string contains several occurrences of first and second, greediness causes the regex to match the leftmost first and the rightmost second:

```
.....first...first...second...first...second........
^^^^^!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^
```
We can also reuse a capture buffer (under the syntax \1) to enforce the same literal text is found twice in a pattern match:

```
/first(.*)second\1/
```

This will only match strings like:
```
............firstXYZsecondXYZ............
```

but not strings like:
```
.................firstABCsecondXYZ.........
```

Test **eg4** out with a variety of inputs and regexes and check you understand how they work.

If your pattern contains lots of ‘/’ characters - while you can write each as ‘\’’ - it’s easier to change the regex quote character:

```
$str =~ m%^/\([^/\]*\)%
$str =~ s!//[^/]*$!!
```

Here, the character immediately following ‘m’ (for match) or ‘s’ (for search and replace) is used as the regex quote character.

That’s a basic overview of Perl regexes; there are loads more features (more are added every year). [perldoc perlre](http://perldoc.perl.org/perlre.html) for more details.
Another Useful operator: \texttt{tr}

\begin{verbatim}
$str =~ tr/firstcharlist/secondcharlist/[cds]
\end{verbatim}

- \texttt{tr} is the character transliterator. It works very like the Unix filter \texttt{tr} - turning each occurrence of a character from the first character list into the corresponding character from the second character list.
- eg: \texttt{tr/aB/Ab/} uppercases every ‘a’ and lowerscases every ‘B’.
- \texttt{tr} is rather like a series of regexes that only use character classes - the above example is equivalent to \texttt{s/a/A/g} followed by \texttt{s/B/b/g}. But \texttt{tr} is much more efficient.
- \texttt{tr//} is bound to a variable using the =~ syntax (like regexes).
- Like \texttt{s///}, \texttt{tr//} also returns a scalar value - a count of how many characters were modified/deleted.

Let’s give some examples:

\begin{verbatim}
$str =~ tr/A-Z/a-z/
$str =~ tr/xyz/ZYX/
$str =~ tr/A-Z//d
$str =~ tr/A-Z//cd
$str =~ tr/aeiou/V/
$str =~ tr/aeiou/V/s
$count = ($str =~ tr/a-z/a-z/)
\end{verbatim}

- \texttt{lowercase every character in $str}.
- \texttt{turn every occurrence of x into z, y into Y and z into X}.
- \texttt{delete all upper case letters}.
- \texttt{delete all characters except upper case letters}.
- \texttt{replace any lower case vowel with a ‘V’}.
- \texttt{replace each sequence of vowels with a single ‘V’}.
- \texttt{Set $count to the number of lower case letters found in $str (without changing $str)}.