1. a. What is `gitlab.cse.unsw.edu.au` and why do I want use it?

   Its a server like < run by CSE to host repository of student programs for some CSE courses.
   Its required for COMP[29]041 assignments and labs (from now on),

   Its easy to use.

   Using it ensures you have a complete backup of all work on your program and can return to its state at any stage.

   It will also allow your tutor to check you are progressing on the lab as they can access your gitlab repository

   Similar servers (e.g. *github* or *bitbucket*) are heavily used in software development so you want the learn how to use them.

   b. What (basically) does `git add` *`file`* do?

   Tells git you want to keep track of versions of *file*

   c. What (basically) does `git commit –a` do?

   Tells git to save the current version of all the files it is tracking.
   You do this when development your program is at significant stage - a stage you might want to return to.

   d. What (basically) does `git push` do?

   It tells git to update another repository with all the commits in this repository.

2. What is the difference in value/type of the following Perl expressions:

   a. `"a"  vs  'a'`

   no difference - both are strings containing a single 'a' character

   b. `"A"  vs  A`

   no difference - both are strings containing a single 'A' character

   c. `"abc"  vs  'abc'`

   no difference - both are strings containing the three characters 'a','b','c'

   d. `"it\'s"  vs  'it\'s'`

   no difference - both are strings containing the four characters 'i','t',apostrophe,'s'

   e. `42  vs  "42"`

   no difference - both are strings containing two characters, '4' and '2'

   f. `3  vs  3.0`

   the first is a string containing one character and the second contains three characters, if evaluated in a numeric context they are the same: `3 ne 3.0` but `3 == 3.0`

   g. `"$2.50"  vs  '$2.50'`

   in the first case, the value of the Perl internal variable `$2` is interpolated into the string; since most likely this will be the empty string, the resulting string will be the three-character string ".50"
   in the second case, the `$` is not treated specially and so the result is the literal five-character string "$2.50"

3. Write a Perl program, `nargs.pl` which prints how many arguments it has been given. For example:

```
$ nargs.pl the quick brown fox
4
```

Sample Perl solution

```
#!/usr/bin/perl -w
print $#ARGV+1, "\n";
```

Sample Python solution

```
#!/usr/bin/env python
import sys
print(len(sys.argv))
```

4. Write a Perl program, `devowel.pl` which filters any vowels from its input. For example:

```
$ ./devowel.pl
The quick brown fox
jumped over the lazy dog.
Ctrl-d
Th qck brwn fx
jmpd vr th lzy dg.
```

Sample Perl solution - simple code

```
#!/usr/bin/perl -w
while ($line = <>) {
    $line =~ s/[aeiou]//gi;
    print $line;
}
```

Sample Perl solution - using $_

```
#!/usr/bin/perl -w
while (<>) {
    s/[aeiou]//gi;
    print;
}
```

Sample Perl solution - using $_ and -p switch

```
#!/usr/bin/perl -w -p
s/[aeiou]//gi;
```

Perl also makes it convenient to perform operations like this from the command line, for example:

```
$ perl -p -e 's/[aeiou]//gi'
```

Note that the `gi` above means **g**lobally (i.e. all occurrences) and case-**i**nsensitive.
Sample Python solution

```
#!/usr/bin/python
import fileinput, re
for line in fileinput.input():
    sys.out.write(re.sub(r'[aeiou]', '', line, flags=re.I))
```

5. Write a simple version of the `head` command in Perl, that accepts an optional command line argument in the form *-n*, where *n* is a number, and displays the first *n* lines from its standard input. If the *-n* option is not used, then the program simply displays the first ten lines from its standard input.
Examples of use:

```
$ perl head.pl <file2          # display first ten lines of file2
...
$ perl head.pl -10 <file2      # same as previous command
...
$ perl head.pl -5 <file2       # display first five lines of file2
...
```

Perl solution with while loop

```
#!/usr/bin/perl -w
$n_lines = 10;
if (@ARGV && $ARGV[0] =~ /-[0-9]+/) {
    $n_lines = $ARGV[0];
    $n_lines =~ s/-//;
    shift @ARGV;
}
$n = 1;
while (<STDIN>) {
    if ($n++ > $n_lines) {
        last;
    }
    print;
}
```

< Perl solution reading all input into an array

```
#!/usr/bin/perl -w
$n_lines = 10;
if (@ARGV && $ARGV[0] =~ /-[0-9]+/) {
    $n_lines = shift @ARGV;
    $n_lines =~ s/-//;
}
@lines = <STDIN>;
print @lines[0..$n_lines-1];
```

Cryptic One-line Perl solution

```
#!/usr/bin/perl -w
print ((<STDIN>)[0..(@ARGV&&-$ARGV[0]||10)-1]);
```

Python solution with while loop

```
#!/usr/bin/python

import re, sys

n_lines = 10

if len(sys.argv) > 1 and re.match(r'-[0-9]+', sys.argv[1]):
    arg = sys.argv[1]
    arg = arg[1:] # remove first character
    n_lines = int(arg)

n = 1
for line in sys.stdin:
    if n > n_lines:
        break
    sys.stdout.write(line)
    n += 1
```

Python solution reading all input into an array

```
#!/usr/bin/python

import re, sys


n_lines = 10
if len(sys.argv) > 1 and re.match(r'-[0-9]+', sys.argv[1]):
    n_lines = int(sys.argv.pop(1)[1:])

# inefficient - reads entire file
sys.stdout.write("".join(sys.stdin.readlines()[0:n_lines]))
```

Python solution using an iterator

```
#!/usr/bin/python

import re, sys, itertools

n_lines = 10
if len(sys.argv) > 1 and re.match(r'-[0-9]+', sys.argv[1]):
    n_lines = int(sys.argv.pop(1)[1:])

for line in itertools.islice(sys.stdin, n_lines):
    sys.stdout.write(line)
```

6. Modify the `head` program from the previous question so that, as well as handling an optional `-n` argument to specify how many lines, it also handles multiple files on the command line and displays the first `n` lines from each file, separating them by a line of the form `==> FileName <===` .
   Examples of use:

```
$ perl head.pl file1 file2 file3    # display first ten lines of each file
...
$ perl head.pl -3 file1 file2     # display first three lines of each file
...
```

   Sample Perl solution

```
#!/usr/bin/perl -w
if ($ARGV[0] =~ /-[0-9]+/) {
    $max = shift @ARGV;
    $max =~ s/-//;
} else {
    $max = 10;
}
# default is stdin if no files specified
$ARGV[0] = "-" if @ARGV == 0;
foreach $file (@ARGV) {
    open my $input, '<', $file or die "$file: can not open: $!\n";
    print "==> $file <==\n";
    @lines = <$input>;
    print @lines[0..$max-1];
    close $input;
}
```

Sample Python solution

```
#!/usr/bin/python

import re, sys, itertools

n_lines = 10

if len(sys.argv) > 1 and re.match(r'-[0-9]+', sys.argv[1]):
    n_lines = int(sys.argv.pop(1)[1:])

if len(sys.argv) == 1:
    sys.argv.append("-")

for filename in sys.argv[1:]:
    try:
        print("==> %s <==" % filename)
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)
        for line in itertools.islice(stream, n_lines):
            sys.stdout.write(line)
        if filename != "-":
            stream.close()
    except IOError as  e:
        (errno, strerror) = e.args
        print("%s: can not open: %s" % (filename, strerror))
```

7. Write a simple version of the `grep` command, that takes a regular expression as its first command line argument and then prints all lines in the standard input (or named files) that contain this pattern.
   Examples of use:

```
$ perl mygrep.pl 'a.*c' file1 file2 file3       # all lines containing a...c
...
$ perl mygrep.pl '[0-9]+' file1 file2 file3     # all lines containing numbers
...
$ perl mygrep.pl '^The' <file1                  # all lines starting with "The"
...
```

Sample Perl solution using a while loop

```
#!/usr/bin/perl -w
$pattern = shift @ARGV;
while (<>) {
    print if /$pattern/;
}
```

Sample Perl solution using an array

```
#!/usr/bin/perl -w
# - grep(/pattern/,@array) returns array containing
#   just elements that match the pattern

$pattern = shift @ARGV;
print grep(/$pattern/, <>);

# behaves like ... cat f1 f2 f3 .. | grep pattern
# not the same as ... grep pattern f1 f2 f3 ..
```

Sample Python solution using a for loop

```
#!/usr/bin/python

import re, sys, fileinput

pattern = sys.argv.pop(1)
for line in fileinput.input():
    if re.search(pattern, line):
        sys.stdout.write(line)
```

More Python-ish (functional) solution

```
#!/usr/bin/python

import re, sys, fileinput

pattern = re.compile(sys.argv.pop(1))
sys.stdout.write("".join(filter(pattern.search, fileinput.input())))
```

8. Modify the `grep` command from the previous question so that accepts a `-v` command line option to reverse the sense of the test (i.e. display only lines that do *not* match the pattern). It should continue with its original behaviour if no `-v` is specified.

Sample Perl solution using a while loop

```
#!/usr/bin/perl -w
if ($ARGV[0] eq "-v") {
    $doOpposite = 1;
    shift @ARGV;
}
$pattern = shift @ARGV;
while (<>) {
    if ($doOpposite) {
        print if !/$pattern/;
    }
    else {
        print if /$pattern/;
    }
}
```

Sample Perl solution using an array and xor

```
#!/usr/bin/perl -w
if (@ARGV && $ARGV[0] eq "-v") {
    $doOpposite = 1;
    shift @ARGV;
}
$pattern = shift @ARGV;
print grep {$doOpposite^/$pattern/} <>;
```

Sample Python using a for loop

```
#!/usr/bin/python

import re, sys, fileinput

do_opposite = 0

if sys.argv[1] == "-v":
    do_opposite = 1;
    sys.argv.pop(1)

pattern = sys.argv.pop(1)

for line in fileinput.input():
    if re.search(pattern, line):
        if not do_opposite:
            sys.stdout.write(line)
    else:
        if do_opposite:
            sys.stdout.write(line)
```

Functional python solution

```
#!/usr/bin/python

import re, sys, fileinput

if sys.argv[1] == "-v":
    sys.argv.pop(1)
    p = re.compile(sys.argv.pop(1))
    f = lambda x: not p.search(x)
else:
    f = re.compile(sys.argv.pop(1))
sys.stdout.write(filter(f, fileinput.input()))
```

9. The following programs are all Perl versions of the `cat` program. Each of them either reads from standard input (if there are no command line arguments) or treats each command line argument as a file name, opens the file, and reads it. The final one shows just how concise Perl code can be. You may find the ideas in these programs useful in helping you solve the problems below.

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

# First Perl version of cat
# Verbose, but shows exactly what's happening

# if no args, read from stdin
if (@ARGV == 0) {
    while ($line = <STDIN>) {
        # note: line still has \n
        print $line;
    }
} else {
    foreach $file (@ARGV) {
        open my $input, '<', $file or die "Can not open $file: $!\n";
        while ($line = <$input>) {
                print $line;
        }
        close $input;
    }
}
```

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

# Second Perl version of cat
# More concise, by using special variable $_

if (@ARGV == 0) {
    while (<STDIN>) {
        print $line;
    }
} else {
    foreach $file (@ARGV) {
        open my $input, '<', $file or die "Can not open $file: $!\n";
        while (<$input>) {
                print;
        }
        close $input;
    }
}
```

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

# Third Perl version of cat
# places input into an array

if (@ARGV == 0) {
    @lines = <STDIN>;
    print @lines;
} else {
    foreach $file (@ARGV) {
        open my $input, '<', $file or die "Can not open $file: $!\n";
        @lines = <$input>;
        print @lines;
        close $input;
    }
}
```

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

# Fourth Perl version of cat
# More concise, but makes filtering difficult

if (@ARGV == 0) {
    print <STDIN>;
} else {
    foreach $file (@ARGV) {
        open my $input, '<', $file or die "Can not open $file: $!\n";
        print <$input>;
        close $input;
    }
}
```

```
# Other versions of cat
# Make use of the fact that <> has a special meaning
# - if no command line arguments, read standard input
# - otherwise, open each argument as a file and read it
# Very concise, but ...
# - you'll need to put up with Perl's error messages
# - you treat all files as a single stream ... which means
#   - you can't distinguish which file each line comes from
#   - there is no scope for doing things at file boundaries
while (<>) { print; }
#or
@lines = <>;
print @lines;
#or
print <>;
```

Write a new version of `cat` so that it accepts a `-n` command line argument and then prints a line number at the start of each line in a field of width 6, followed by two spaces, followed by the text of the line. The numbers should constantly increase over all of the input files (i.e. don't start renumbering at the start of each file). The program always reads from its standard input.

Example of output:

```
$ perl cat -n myFile
     1  This is the first line of my file
     2  This is the second line of my file
     3  This is the third line of my file
          ...
  1000  This is the thousandth line of my file
```

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

if (@ARGV > 0 && $ARGV[0] eq "-n") {
    $doNumbering = 1;
    shift;
}
$nlines = 1;
while (<>) {
    printf "%6d  ",$nlines++ if $doNumbering;
    print;
}
```

Note that the `shift` is important. It removes the option from the argument list so that `<>` is left with the correct command-line arguments to process (i.e. just the file names).

10. Modify the `cat` program from the previous question so that it also accepts a `-v` command line option to display *all* characters in the file in printable form. In particular, end of lines should be shown by a `$` symbol (useful for finding trailing whitespace in lines) and all control characters (ascii code less than 32) should be shown as `^X` (where *X* is the printable character obtained by adding the code for 'A' to the control character code). So, for example, tabs (ascii code 9) should display as `^I`.
Hint: the `chr` and `ord` functions might be useful. Try

```
$ perldoc -f ord
```

for info about functions such as these.
Example of output:

```
$ perl cat -v < myFile
This file contains a tabbed list:$
^I- point 1$
^I- point 2$
^I- point 3$
And this line has trailing spaces   $
which would otherwise be invisible.$
```

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

if ($ARGV[0] eq "-n") {
    $doNumbering = 1;
    shift;
}

if ($ARGV[0] eq "-v") {
    $doVisible = 1;
    shift;
}

$nlines = 1;
while (<>) {
    printf "%6d  ",$nlines++ if $doNumbering;
    if (!$doVisible) {
        print;
    } else {
        chomp;
        foreach $c (split //) {
            if (ord($c) >= 32) {
                print "$c";
            } else {
                print "^".chr(ord($c)+64);
            }
        }
        print "\$\n";
    }
}
```

11. Write a version of the `tac` command in Perl, that accepts a list of filenames and displays the lines from each file in reverse order.

Sample Perl solution

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

if (@ARGV == 0) { # no args, read from stdin
    print reverse <STDIN>;
}
else {
    foreach $file (@ARGV) {
        open my $input, '<', $file or die "$file:can not open: $!\n";
        print reverse <$input>;
        close $input;
    }
}
```

The following might look attractive, but treats all files as a single input, and reverses that. It doesn't do it file by file like the real `tac`.

```
#!/usr/bin/perl -w
print reverse <>;
# same as ... cat f1 f2 f3 .. | tac
# not the same as ... tac f1 f2 f3
```

Sample Python solution

```
#!/usr/bin/python

import sys

if len(sys.argv) == 1:
    for line in reversed(list(sys.stdin)):
        sys.stdout.write(line)
else:
    for filename in sys.argv[1:]:
        with open(filename) as f:
            for line in reversed(list(f)):
                sys.stdout.write(line)
```