1.1.2.8.5

Intermediate Perl – Session 5

- substitution operator
- notes on split
- trapping errors
- I/O

Substitution Operator s/ / /

- the substitution operator is used to replace text
  - s/REGEX/replacement/

- select locations in the string to replace using the REGEX
  - behaviour of s/ / / can be modified using /g, /m, /s, /i like for regex
  - delimiters can be defined separately for the two parts
    - s{ }[ ]
    - s, ,{ }
    - as long as the delimiters are balanced

```
$x = “aabbbb”;

$x =~ s/a/c/;          cabbbb
$x =~ s/a/c/;          ccbbbb
$x =~ s/b+/d/;         ccd
$x =~ s/c+/d/;         dd
$x =~ s/d+/sheep/;     sheep
```

Global Substitution

- replacement of every instance of REGEX is achieved using /g
  - s/REGEX/replacement/g

```
# replace every a with c
$x = "aabbbbaa";
$x =~ s/a/c/g;          ccbbbbcc

# remove all digits (replace with nothing)
$y = "123abc456def";
$x =~ s/\d//g;          abcdef

# substitute all n-tuples with 1-tuple
$z = "aaabccdeffffff    123333"

$z =~ s/(.)\1*/$1/g;    abcdef 123
```

- recall the difference between \1 and $1
  - \1 is the current value of the 1st capture bracket during a match
  - $1 is the text captured by the 1st capture bracket after a successful match

Evaluated Substitutions with /e

- the second part of the substitution is not a regex, it is a replacement string
  - you can use references to captured text using $1, $2, $3... (not \1 \2 \3)
  - ask Perl to evaluate the replacement string by using /e
    - length($1) below is evaluated and the result is used for when replacement is done

```
# replace every a with c

$x = “aaaabbbccd”;
$x =~ s/((.)\2+)/length($1)/eg;      432d
```

- make sure you know what is being captured by your nested brackets!

- time gives seconds since epoch

```
$x = “meet you at _time_”;

$x =~ s/_time_/time/eg;              meet you at 1078434770
```

Iterated Evaluations with /ee /eee /eeee

- don't do it unless it's stupendously clear what is happening
- sprintf is frequently used with /e to reformat the input string

```
$x = "i'd like function sqrt applied to 2 please";
$x =~ s/function (\w+) applied to (.+?) please/sprintf("%s(%s)",$1,$2)/ee;
```

- let's break it down one /e at a time

```
    sprintf("%s(%s)","sqrt",2)
/e  sqrt(2)
/e  1.41…
```

Return Value of s///

- recall that m// returned meaningful things when called in scalar or list context
- s/// behaves very simply
  - returns number of substitutions in any context

```
$x = “aabbbbaa”;

$num = $x =~ s/a/c/g;          ccbbbbcc  num=4
@num = $x =~ s/c+/d/g;         dbbbbd    num=(2)
```

Substitution with Lookarounds – inserting text

- recall that `m//` may match text but can also be used to position the regex engine at a particular position

`m/abc/`

xxxabcxxx
^

cursor positioned after matching text

`m/(?=abc)/`

xxxabcxxx
^

cursor positioned at a location satisfying the lookaround (abc is in front of cursor)

- if `s/REGEX/replacement/` contains a REGEX which does not match any text but only positions the cursor, replacement will be inserted at that position
  - think of it as replacement of the matching empty string at a position

Substitution with Lookarounds – inserting text

- here I'm using a lookahead (?=) and lookbehind (?<=) to position the cursor after/before specific strings and inserting "x" at this location

```
$x = "aabbbbaa";
# each s/// is demonstrated on the original value of $x

$x =~ s/(?=bbbb)/x/;        aaxbbbbaa
$x =~ s/(?=bbb)/x/;         aaxbbbbaa
$x =~ s/(?=bbb)/x/g;        aaxbxbbbaa
$x =~ s/(?=bb)/x/g;         aaxbxbxbbaa
$x =~ s/(?=b)/x/g;          aaxbxbxbxbaa
$x =~ s/(?<=bbbb)/x/;       aabbbbxaa

$y = "aabbaacc11cc22";

$y =~ s/(?<=aa)(?=cc)/x/;   aabbaaxcc11cc22
```

```
# inserting a thousands separator
$x = "1234567";
$x =~ s/(?<=\d)(?=(\d{3})+$)/,/g;   # 1,234,567
```

- cursor position at least one digit behind cursor and 3n digits in front of cursor
- why is the $ anchor needed?

Regex Bonus (??{CODE})

- the dynamic regex construct (??{CODE}) is available in perl 5.6
  - when (??{CODE}) construct is reached, the CODE is evaluated/executed and the result is inserted into the regular expression

- how do I match a number followed by exactly this many Xs?
  - e.g. 3XXX, 5XXXXX, 10XXXXXXXXXX

```
regex       /(\d)(??{ "X{$1}" })/

steps       /(3)(?{ "X{3}" })/
            /(3)X{3}/
```

- how do I match a number followed by its square? e.g. 24 39 416 525

```
regex       /(\d)(??{ $1**2 })/

39          /(3)(?{ 3**2 })/
            /(3)9/
```

splitting Up Isn't Hard to Do

- split splits strings along a character or regex match boundary
  - unlike m/REGEX/g, split returns the text between matches

```
$x = "sheep:are:fun";

# split along a string
@x = split(":",$x)          # (sheep,are,fun)

# split along a regex
@x = split(/\w:\w/,$x)      # (shee,r,un)

# split along characters
@x = split("",$x)           # (s,h,e,e,p,a,r,e,f,u,n)
@x = split(//,$x)           # (s,h,e,e,p,a,r,e,f,u,n)

# split along all space characters (special meaning of " " here)
$y = "   sheep    are fun    ";
@x = split(" ",$y)          # (sheep,are,fun)
```

split's Context

- split is always always used in a list context, since it returns a list
- split acts on $_ if no target string is supplied

```
$x = "1,20,300,15,500";

for my $num ( split(",",$x) ) {
  ...
}

@x = ( "1,2,3" , "4,5,6" )

for (@x) {
  for $num (split(",")) {
     ...
  }
}
```

Limit split Chunks

- split can take a third argument – the number of chunks to return

```
$x = "1,20,300,15,500";

split(",",$x,3)   # 1 20 300

split(",",$x,999)  # returns all chunks if <999 chunks in string
```

split Will Return Empty Chunks

- neighbouring chunk boundaries will result in the return of empty fields

```
$x = "1,20,300,,15,500";

split(",",$x)   # 1 20 300 "" 15 500
```

- however, trailing neighbouring chunk boundaries do not result in empty fields
  - … unless chunk limit operand is used (use large number like 999 or better still -1)
- leading neighbouring boundaries will cause empty fields

```
$x = "1,20,300,15,500,,";

split(",",$x)       # 1 20 300 15 500
split(",",$x,999)   # 1 20 300 15 500 "" ""
```

split with Capturing Parentheses

- capturing parentheses change split's behaviour
  - items captured by the parentheses are included in the output

```
$x = "aaa1bbb2ccc";

split(/\d/,$x)   # aaa bbb ccc
split(/(\d)/,$x) # aaa 1 bbb 2 ccc

$y = "aaa123bbb456ccc";

split(/(\d)\d(\d)/,$x)  # aaa 1 3 bbb 4 6 ccc
```

basic error trapping

die and warn

- to produce a warning message, use warn
  - script continues to run
  - message sent to STDERR, with line number if argument does not have trailing "\n"

```
for my $i (0..10) {
  warn "careful - counter is zero" if ! $i;
}
zero at ./tests line 8.
```

- to exit fatally, use die
  - script stops
  - message sent to STDERR, with line number if argument does not have trailing "\n"

```
for my $i (0..10) {
  die "can't - counter is zero" if ! $i;
}
zero at ./tests line 8.
```

eval

- to catch a fatal error in code, and recover, use eval { };
  - if an error is encountered, $@ is set with error string

```
for my $i (0..1) {
  print 1/($i-1);
}
-1
Illegal division by zero at ./tests line 8.
```

```
eval {
  for my $i (0..1) {
    print 1/($i-1);
  }
};
if($@) {
  # catch and fix
  print "error caught - message from eval is $@";
}
-1
error caught - message from eval is Illegal division by zero at ./tests line 9.
```

eval + die

- if die is called and $@ is set, you get a propagated message

```
eval {
  for my $i (0..1) {
    print 1/($i-1);
  }
};
die if $@;
-1
Illegal division by zero at ./tests line 9.
        ...propagated at ./tests line 12.
```

```
eval {
  die "I want to exit";
};
die if $@;

I want to exit at ./tests line 8.
        ...propagated at ./tests line 10.
```

Carp

- the Carp module extends functionality of `die` and `warn`
  - adds additional stacktrace output

- `carp` is like `warn` but gives trace

```
f();
print "next";

sub f {
 g();
}

sub g {
 carp "hi from carp";
}

hi from carp at ./tests line 16
        main::g() called at ./tests line 12
        main::f() called at ./tests line 8
next
```

Carp

- croak is like die, but gives trace

```
f();
print "next";

sub f {
 g();
}

sub g {
 croak "hi from croak";
}

hi from croak at ./tests line 16
        main::g() called at ./tests line 12
        main::f() called at ./tests line 8
```

Carp

- the `shortmess()` function returns the trace that would have been produced by `carp` and `croak`

```
f();
print "next";

sub f {
 g();
}

sub g {
 my $msg = Carp::shortmess("just a message");
 print $msg;
}

just a message at ./tests line 16
        main::g() called at ./tests line 12
        main::f() called at ./tests line 8
next
```

trap croak

- you can trap croak, just like you can trap die

```
eval {
  f();
};
die "died with $@" if $@;

sub f {
 g();
}

sub g {
 croak "croaked";
}

died with [croak at ./tests line 18
        main::g() called at ./tests line 14
        main::f() called at ./tests line 9
        eval {...} called at ./tests line 8
] at ./tests line 11.
```

I/O

Writing to Files

- specify the mode in which the file will be opened using
  - "`>filename`" for writing
  - "`>>filename`" for appending
  - "`<filename`" for reading (default)

```
my $infile  = "~/data.txt";
my $outfile = "~/lengths.txt";

open(IN,$infile)      || die "cannot open file [$infile]"
open(OUT,">$outfile") || die "cannot write to file [$outfile]";
while(<FILE>) {
  chomp;
  print OUT $.,length,"\n";  # print line number and length to handle OUT
}
close(IN);
close(OUT)
```

File Tests

- test whether you can read from a file, write to a file before doing anything

```
my $infile  = "~/data.txt";
my $outfile = "~/lengths.txt";

die "file does not exist [$infile]" unless -e $infile
die "file is not a text file [$infile]" unless -T $infile
die "cannot read from file [$infile]" unless -r $infile
die "cannot write to file [$outfile]" unless -w $infile
```

Common File Tests

- testing a file requires IO operation which may be slow if disk is slow
  - test the same file using _
  - if -e filename && -r _

```
-r  File is readable by effective uid/gid.
-w  File is writable by effective uid/gid.
-x  File is executable by effective uid/gid.
-o  File is owned by effective uid.

-e  File exists.
-z  File has zero size.
-s  File has nonzero size (returns size).

-f  File is a plain file.
-d  File is a directory.
-l  File is a symbolic link.

-T  File is a text file.
-B  File is a binary file (opposite of -T).

-M  Age of file in days when script started.
-A  Same for access time.
-C  Same for inode change time.
```

IO::File

- IO::File abstracts I/O
  - a benefit is that you get a scalar file handle

```
use IO::File;

my $fh = IO::File->new("data.txt");
# you can now pass $fh to subroutines, just like any scalar
while(my $line = $fh->getline) {
  # $fh->getline is more readable and always returns one line, regardless of context
  print $line;
}
$fh->close();
```

- to read about handle's methods, see IO::Handle

Creating Temporary Files and Directories

- to make temporary files, use tempfile
  - file will be created in system temporary directory (tmpdir() from File::Spec)

```
use File::Temp qw(tempfile);

# create a temporary file
my ($fh,$filename) = tempfile    # GLOB(0x81912d0) /tmp/M8idOGppBX
# create a file with a template name in a particular directory
my ($fh,$filename) = tempfile("sheepfileXXXX",DIR=>"/home/martink/tmp");
# delete the file after script is done
my ($fh,$filename) = tempfile("sheepfileXXXX",DIR=>"/home/martink/tmp", unlink=>1);
```

- $fh = tempfile()
  - scalar context, file automatically deleted, you don't know its name (anonymous)
- ($fh,$filename) = tempfile()
  - list context, file not automatically deleted
- (undef,$filename) = tempfile()
  - file not created, you get a random filename though

Creating Temporary Files and Directories

- to make temporary directories use tempdir

```
use File::Temp qw(tempdir);

# create a temporary directory within DIR
my $dir = tempdir(DIR=>"/home/martink/tmp");   #/home/martink/tmp/WJ6gBPOiJv

# specify a particular directory name template - trailing X's randomized
my $dir = tempdir("sheepXXXX", DIR=>"/home/martink/tmp");  # /home/martink/tmp/sheep5AC

# delete directory (and any files in it) after end of script
my $dir = tempdir("sheepXXXX", DIR=>"/home/martink/tmp", CLEANUP=>1);
```

STDOUT and STDERR

- standard output (STDOUT) is buffered, and standard error (STDERR) is not buffered
  - lines sent to these two outputs may appear out of order

- STDOUT and STDERR can be redirected independently

```
>cat simple.pl
#!/usr/local/bin/perl
print “message\n”;
print STDERR “error\n”;

% simple.pl > stdout.txt 2> stderr.txt

% simple.pl > stdout.txt 2> /dev/null

% simple.pl &> both.txt
```

- typically, STDERR is for error messages or debugging and STDOUT for output

Changing Default Filehandles

- when you print in perl, the output goes to STDOUT by default
  - unless redirected, STDOUT is the terminal

- to redirect print statements to another handle (e.g., that of a file) use select
  - select always returns the current handle
  - if supplied with a handle, it sets it as the current default output handle

```
print “hello”; # STDOUT default

my $old = select($fh);

print “hello”; # to handle $fh

select($old);

print “hello”; # back to STDOUT
```

Reading from Processes

- open a pipe to a process to read the output of another program
  - add a trailing pipe to the filename

```
# save STDOUT to OLDOUT
open(PROC,"/usr/local/bin/analyzethis |");
while(<PROC>) {
  print "process says $_";
}
```

Reading Directories

- to open a directory use `opendir` then use `readdir` to get directory listing
  - `$item` will be a file name relative to `$dir`

```
my $dir = "/home/martink";
die "[$dir] not a directory" unless -d $dir;
opendir(DIR,$dir);
while(my $item = readdir(DIR)) {
  next if $item eq "." || $item eq ".."; # you get . and .. too!
  print "$item";
  print "hark! a directory $item\n" if -d "$dir/$item";
}
```

- consider using `IO::Dir`

- to create directories, use `File:Path` module (`mkpath` and `rmpath`)
  - will create directory tree, as needed

1.1.2.8.5

Introduction to Perl – Session 5

- substitution operator s///
- die, warn and Carp
- I/O
  - IO::File
  - STDERR/STDOUT
  - file tests, -r -e -s