1.1.2.8.4

Intermediate Perl – Session 4

- scope
  - strict pragma
  - my
  - our
- advanced regular expressions

variable scope

Perl is permissive – so don't get caught

- if you do not enable the strict pragma, newly mentioned variables spring into existence
  - just like hash keys (*autovivification*)

```
@big_numbers = ();
for (1..10) {
  push @bignumbers, $_**2;
}
```

- when variable names are misspelled, logical bugs arise that are very hard to squash

- luckily, there is a pragma (strict) that helps with this

- a pragma is a directive that alters the way the Perl interpreter/compiler behave

strict pragma

- when the strict pragma is enabled, all variables must be initially declared

```
use strict;
# $x is mentioned without being declared - raises error when strict in effect
$x = 10;

Global symbol "$x" requires explicit package name at ./myscript line 3.
Execution of ./myscript aborted due to compilation errors.
```

- variables are declared in perl using my

- the use of my registers the variable, permitting its use

```
use strict;
# now no problem
my $x = 10;
```

consequences of misspellings are mitigated with strict

- when you misspell a variable, if strict is used, your script will not run

```
use strict;
my @big_numbers = ();
for (1..10) {
  push @bignumbers, $_**2;
}

Global symbol "@bignumbers" requires explicit package name at ./myscript line 4.
Execution of ./myscript aborted due to compilation errors.
```

- you should always use strict
  - no matter what
  - no mattern when

- from now on, all code samples assume that use strict is in effect

my gives variables scope

- when you use my, you define its scope to the innermost outer block
  - the variable becomes a local variable

```
my @rands = ();
for (1..10) {
  # $x is visible only within the for{} block
  my $x = rand();
  push @rands, $x;
}
# this will produce an error because $x is out-of-scope (not visible outside for{})
print $x;

Global symbol "$x" requires explicit package name at ./myscript line 10.
```

- because blocks can be nested, so can scope
  - we'll see nested scope shortly

my allocates variables fresh each time

- everytime my is used, distinct instance of the variable is allocated

```
for (1..3) {
  # on each loop iteration, a fresh $x local variable is allocated
  my $x;
  # initial $x value is undef, which becomes 1 when incremented
  $x++;
  print $x;
}

1
1
1
```

my lexically scopes variables

- lexical means relating to vocabulary or words
- lexical scope means the visibility of a variable as related to the content of the code, not the way it runs
- my scopes variables lexically – the scope is determined at compile-time, not at runtime
- when a variable goes out of scope, its memory is deallocated and the garbage collector can go to work
  - perl uses reference-based garbage collection
  - a variable is garbaged if no visible references to it exist
  - problems can arise with circular references (read up on weaken)
    - see WeakRef module on CPAN
    - see 8.5.2 of Programming Perl 3rd ed

scope nests as blocks do

- in this example, different memory blocks are allocated for the two $x variables

```
my $x = 10;
{
  my $x = 20; # inside the blocks this $x obscures visibility of the outer block's $x
  print "inner x",$x;
}
print "outer x",$x;
inner x 20
outer x 10
```

- nesting scope is useful for nested for blocks

```
# if you don't need outer loop value in inner loop
for my $i (1..3) {
  print $i;
  for my $i (1..3) {
    print $i;
  }
}
1 1 2 3 2 1 2 3 3 1 2 3
```

```
# ... and if you do
for my $i (1..3) {
  for my $j (1..3) {
    # both $i and $j are visible here
  }
  # $j is not visible here
}
```

use my in subroutines

```
my $y = square(10);
# can't see $x here (that's good because $x is meant to be internal to square() )

sub square {
  # create local variable, use it
  my $x = shift;
  # upon return, $x is out of scope and is garbage collected
  return $x**2;
}
```

```
my $squares_ref = squares(10,11,12);
# @x is still alive, not visible but accessible through its reference

sub square {
  # create local variable, use it
  my @x = map { $_**2 } @_;
  # return a reference to it
  return \@x;
  # @x is out-of-scope, but is accessible through its reference
  # it is not garbage collected because we have a visible reference
}
```

use my in subroutines

- when the last reference to a variable goes out of scope, the variable is garbaged

```
{
  my $squares_ref = square(10,11,12);
  # @x is still alive, not visible but accessible through its reference
}
# now $squares_ref goes out of scope, no visible references exist to @x in square(),
# and both @x and $squares_ref are garbage collected

sub square {
  # create local variable, use it
  my @x = map { $_**2 } @_;
  # return a reference to it
  return \@x;
  # @x is out-of-scope, but is accessible through its reference
  # it is not garbage collected because we have a visible reference
}
```

my variables not seen to subroutines defined in outer blocks

- remember, lexically scoped variables are visible
  - in the block (and all inner blocks) where they were declared

```
my $x = 10;
my $y = square($x);

sub square {
  my $v = shift;
  # $x is visible here because the subroutine block is inner to declaration of $a
  print $x;
  return $v**2;
}
```

- within a function, you can use variables scoped in outer blocks

my variables not seen to subroutines defined in outer blocks

- variables are not visible within subroutines whose blocks are parallel or outside of the variable's block

```
# $x not visible
{
  # $x visible
  my $x = 10;
  my $y = root($x);
}
# $x not visible

sub square {
  my $v = shift;
  # $x is not visible because it was scoped in a parallel (not outer) block
  print $x;
  return $v**2;
}

Global symbol "$x" requires explicit package name at ./myscript
```

our vs my

- if you want global variables, use our

```
my $x = "outside";
print $x;             outside
fn();                 inside
print $x;             outside

sub fn {
  my $x = "inside";
  print $x;
}
```

```
our $x = "outside";
print $x;             outside
fn();                 inside
print $x;             inside

sub fn {
  $x = "inside";
  print $x;
}
```

- the difference between my and our requires the introduction packages
  - packages are Perl's namespaces
  - our creates package variables with simple names
  - my creates lexically scoped variables (not in package) with simple names

packages

- a namespace defines the boundary of variables' scope
  - multiple namespaces allow variables with the same name to be used independently

- in Perl, namespaces are called packages

- if the namespace is not specified, the default `main` namespace is assumed

- so far, we've been always working in the main namespace

- there are two kinds of variables
  - package variables
    - associated with the package
    - can be refered to with package name (e.g. `$PACKAGE::VARIABLE`)
  - lexically scoped variables
    - not associated with the package at all
    - cannot be refered to with a package name

packages

- each package PACKAGE has a symbol table, which is a hash %PACKAGE::

- when you write a script the default main:: package comes with a variety of prefab special variables

```
print join("\n",keys %main::);
_                                $_ default input and pattern matching space
/                                $/ input record separator
"                                $" element separator for double quoted arrays
@                                $@ syntax message from last eval
$                                $$ process ID of script
0                                $0 program name
ARGV                             command line parameters
.... and more
```

- these special variables can be referenced using the package name
  - $main::$$
  - or within the main package, $$ (since $main:: is assumed)

advanced regular expressions

Capturing Parentheses

- matches inside () are stored for later use in lexically scoped $1, $2, $3...
  - $1, $2 only set if match was successful
  - $+ copy of highest numbered $1, $2,...
  - $^N copy of most recently closed $1, $2, ...

```
$x = “abc123456”

$x =~ /^(.)(..)/;
print $1,$2;             a bc
```

- order of capture determined by position of first opening parenthesis

```
$x =~ /(((.).).)/      $1=abc   $2=ab   $3=a
                       $^N              $+
```

Non-capturing Parentheses

- (?:) does not populate the pattern buffers $1, $2, $3...

```
$x = “abc123456”;

$x =~ /(((.).).)/        $1=abc  $2=ab  $3=a
$x =~ /((?:(.).).)/      $1=abc  $2=a
```

- non-capturing (?:) permits grouping without capturing

```
$x = “abc123456”;

$x =~ /a(bc|cb)(1)/       $1=bc  $2=1
$x =~ /a(?:bc|cb)(1)/     $1=1
```

Backreference

- \1 \2 \3 refer to what is matched by capturing parentheses while the match is proceeding
  - values in \1 \2 \3 are available even if the match is not successful
  - backreferences used to match “more of the same”

```
$x = “aaabbb”;

@m = $x =~ /(.)\1\1(.)\2\2/;      @m = (a,b)

$x =~ /^(.)\1\1ccc/;              \1 is “a” while regex engine is running
```

- do not use $1 within the match unless you want it to be the $1 set by the last successful match

```
# this is likely not what you intend
@m = $x =~ /(.)$1$1(.)$2$2/;
```

Current Match Location and pos()

- remember that the regex engine bumps-along the string as it looks for matches
- during matching with /g the engine position is not reset to the start of the string
- use pos() to get/set the engine position

```
$x = “12345”;

while ($x =~ /(..)/g) {
  print “$1 at”,pos($x);                    12 at 2, 34 at 4
}

while ($x =~ /(..)/g) {
  print “$1 at”,pos($x);                    12 at 2, 23 at 3, 34 at 4, 45 at 5
  # backup the engine
  pos($x)--;
}
```

pos() – Extracting Random Subtext

- shuffle engine position with pos() and rand() to randomly sample a string

```
use String::Random qw(random_string);
my $long_string = random_string(“c” x 1000);

# this loop never finishes
while ($long_string =~ /(.{10})/g) {
  print “$1 at”,pos($long_string);
  pos($long_string) = int rand(990);
}
```

```
gjltblecjr 10
mjekdgzrax 273
dshagdtdbb 77
woqoksgguw 619
lpvdoaccfk 510
kexnedksty 644
jdvjgsgeqn 721
yvduoqoahm 67
bncgqlysip 897
urwuvbbfzo 467
dbvjbpwpdl 19
ptuwodgsbu 669
wuvbbfzofu 469
epkwehnllz 366
lnsxyubonu 241
```

\G – anchor of last match

- recall that ^ and $ are anchors – they match a position within the string, not specifically a character

- \G refers to the position of the last match ended
  - use \G to preventing bump-along
  - optionally set start position with pos()

```
$x = “abc123def456”;

while($x =~ /\G([a-z])/g) {
  print $1;                   a b c - never gets to d
}

# put cursor * at abc*123def456
pos($x) = 3;
while($x =~ /\G(\d)/g) {
  print $1;                   1 2 3
}
```

/gc – A Lexer Example

- /g does not reset the cursor position after a successful match, but it does after a failed match
- /gc does not reset cursor after a failed match

- a lexer parses a string into a series of known tokens

```
my $x = "abcd1efgh234ij5k";

my $atend;
do {
    if($x =~ /\G([a-z])/gc) {
      print "in letter block $1";
    } elsif ($x =~ /([a-z])/gc) {
      print "start of letter block $1";
    } else {
      $atend = 1;
    }
} while ! $atend;
```

```
in letter block a
in letter block b
in letter block c
in letter block d
start of letter block e
in letter block f
in letter block g
in letter block h
start of letter block i
in letter block j
start of letter block k
```

Greedy Quantifiers

- quantifiers like `* + ? {n}` are greedy
  - they attempt to match as much as possible
  - they give up some of their match if it is required for an overall match to be successful

```
$x = “abc123def456”;

$x =~ /(.*)/;       $1 is the whole string

$x =~ /(.*)6/;      $1 is abc123def45

$x = “aaaabab”;

$x =~ /(.*)ab/;     $1 is aaaab
```

- when the engine is making the match, greedy matches are always tried
  - if a match fails the engine backtracks and takes some of the match away from the greedy quantifier

Lazy Quantifiers

- lazy quantifiers *? +? ?? {n}? prefer not to match
- regex engine skips over lazy quantifiers, unless the match cannot be made

```
$x = “aabbbb”;

$x =~ /(.*?)(.*)/;   $1 = “”  $2 = aabbbb

$x =~ /(.*?)a(.*)/;  $1 = “”  $2 = abbbb

$x =~ /(.*?)b(.*)/;  $1 = aa  $2 = bbb
```

- the optional ? is greedy but ?? is lazy

```
$x = “aabbbb”;

$x =~ /(.?)(.*)/;    $1 = “a”  $2 = abbbb

$x =~ /(.??)(.*)/;   $1 = “”   $2 = aabbbb
```

Lookaround – the lookahead

- lookaround patterns do not consume any matching text
  - they do not advance the regex engine position
  - they limit the neighbourhood of were a match starts or ends
- lookahead (?=regex)
  - match begins only before certain regex is seen in front of the engine's current position

```
$x =~ /[a-z]+(?=\d)/   match words ending with a digit
                       abc abc5 abc123
```

- negative lookahead (?!regex)
  - same as lookahead, but negated

```
$x =~ /[a-z]+(?!\d)/   match words not ending with a digit
                       abc abc5 5abc abc!
```

Lookaround – the lookbehind

- lookbehind (?<=regex) is similar to lookahead except it forces engine to see regex behind where a match starts

```
$x =~ /(?<=\d)[a-z]+/   match words starting with a digit
                        5abc 51abc abc abc5
```

- a negative lookbehind (?<!regex) is the negated version of the lookbehind

\s \m – Confusing Modes

- \s is the single line mode
  - treat a multi-line string as a single string
  - it means that . matches everything, including a new line

- \m is the multi-line mode
  - treat a single string as a multi-line string
  - ^ will match after a newline, not just at the start of a string
  - $ will match before a newline, not just at the end of a string
  - \A plays the role of the start-of-string anchor
  - \Z plays the role of end-of-string anchor

1.1.2.8.4

Introduction to Perl – Session 4

- scope
  - strict pragma
  - my
  - our
- advanced regular expressions
  - \G end of last match anchor
  - greedy vs lazy
  - lookahead
  - lookbehind
  - multi- and single-line modes