1.0.1.8.8

Introduction to Perl Session 8

- recipes and idioms
- where to go from here

Setting a Default Value

- the op= operator is a useful shortcut
  - `a = a + b` → `a += b`
  - `a = a * b` → `a *= b`
  - `a = a || b` → `a ||= b`

```
# force default value if variable is false
$x ||= 5;

# set default values for input arguments
func($x,$y);

sub func {
  my $x = shift;
  # method A - shift or default
  my $y = shift || 5;
  # method B - shift, then default
  my $y = shift;
  $y || = 5;
}
```

- remember the difference between false and defined
  - zero is false, but defined

defined-or

- Perl 5.10 adds a new type of OR which uses `if defined` rather than `if`

```
# the defined-or                                       # the standard or
$c = $a // $b;                                         $c = $a || $b
# equivalent to                                        # equivalent to
if(defined $a) {                                       if($a) {
  $c = $a                                                $c = $a;
} else {                                               } else {
  $c = $b;                                               $c = $b;
}                                                      }

# $a=0 is a perfectly good value, which will be honoured
# $a ← 10 assignment will happen only when $a is undefined
$a //= 10;

# compare the above to ||=, for which 0 is not an acceptable value
# here, $a ← 10 assignment will happen when $a is false
$a ||= 10;
```

- use // when false (0) is an acceptable value

Swapping Values

- to swap values, Perl does not require a temporary variable

```
# initialize separately
$a = 5;
$b = 10;

# initialize together
($a,$b) = (5,10);

# swap simultaneously
# a ← 10  b ← 5
($a,$b) = ($b,$a);
```

Processing Strings One Character at a Time

- to split a string into component characters, use split with empty boundary

```
# initialize separately
$string = “wooly sheep”;
# split (//,$string) also works
# split (undef,$string) also works
@chars = split(“”,$string);

for $char (@chars) {
  print qq{give me an $char!};
}
```

- you can also use a while loop with global captured search

```
# initialize separately
$string = “wooly sheep”;
# split (//,$string) also works
while( $string =~ /(.)/g ) {
  print qq{give me an $1!};
}
```

Match with Confidence

- test whether a regex matches a string in scalar context
  - returns 0/1 if REGEX is found anywhere within the string

```
$found_match = $string =~ /REGEX/;
```

- pull out all matches using list context and /g
  - you must use /g or you will only get the first match

```
@matches = $sequence =~ /atgc/g;

# extract subpatterns with capture brackets
@matches = $sequence =~ /aaa(...)aaa/g;
```

counting characters in a string

- recall that =~ with /g returned all matches

```
$x = "aaaabbbccd";
@matches = $x =~ /a/g;    @matches ← qw(a a a a)

# to count the number of matches, force =~ to be evaluated in list context first,
# then evaluate in scalar context

$n = () = $x =~ /a/g;     $n ← 4

$n   = $x =~ /a/g;    does not work - =~ is evaluated in scalar context $n ← 1
($n) = $x =~ /a/g   does not return count – returns first match $n ← "a"
```

- use =~ tr/// to count

```
$x = "aaaabbbccd";
$n = $x =~ tr/a//;
```

Reversing Lists

- to reverse a list or string, don't forget the reverse operator
  - in scalar context
    - if passed a scalar, reverses the characters in the scalar – e.g, sheep → peehs
    - if passed a list, reverses the list and returns a concatenated list – e.g., qw(1 2 3) → "321"
  - in list context, reverses a list and returns it – e.g., qw(1 2 3) → qw(3 2 1)

```
@chars      = split(“”,"sheep");      → qw(s h e e p)

# scalar context, passed a scalar
$string_rev = reverse “sheep”;        → peehs
# list context, passed a list
@chars_rev  = reverse @chars;         → qw(p e e h s)
# scalar context, passed a list
$string_rev = reverse @chars;         → peehs

# challenge
print reverse “sheep”;                → sheep
print $y = reverse “sheep”;           → peehs
```

Parsing Out Substrings

- to extract parts of input strings, use regexs and capture brackets

```
($w,$h) = $message =~ /screen size is (\d+) by (\d+) pixels/;

# or verbosely

if( $message =~ /screen size is (\d+) by (\d+) pixels/ ) {
  ($w,$h) = ($1,$2);
}
```

- the first example works because `=~` is called in list context
  - returns all matching strings (optionally delineated by capture brackets)
- the second example works because pattern buffers `$1,$2` are set after a successful match

Trimming Strings

- chomp is used to safely remove a newline from the end of a string
- other leading/trailing characters are commonly discarded
  - spaces
  - zeroes
  - non-word characters

```
# remove leading spaces
$x =~ s/^\s*//;
# remove trailing spaces
$x =~ s/\s*$//;
# remove both leading and trailing spaces
$x =~ s/^\s*(.*?)\s*$/$1/;

# challenge - why not the following regex?
$x =~ s/^\s*(.*)\s*$/$1/;     why is the ? important?

# remove leading zeroes
$x =~ s/^0*//;

# remove a variety of leading characters
$x =~ s/^[0\s;]*//;
```

Creating Integer Ranges

- use the range operator .. to create ranges of integers, or even characters

```
@range     = (10..20);
@range_rev = reverse (10..20);

for (10..20) {
  print;
}

# range of characters
for (a..z) {
  $alphabet .= $_;
}

$alphabet = join("",(a..z));
```

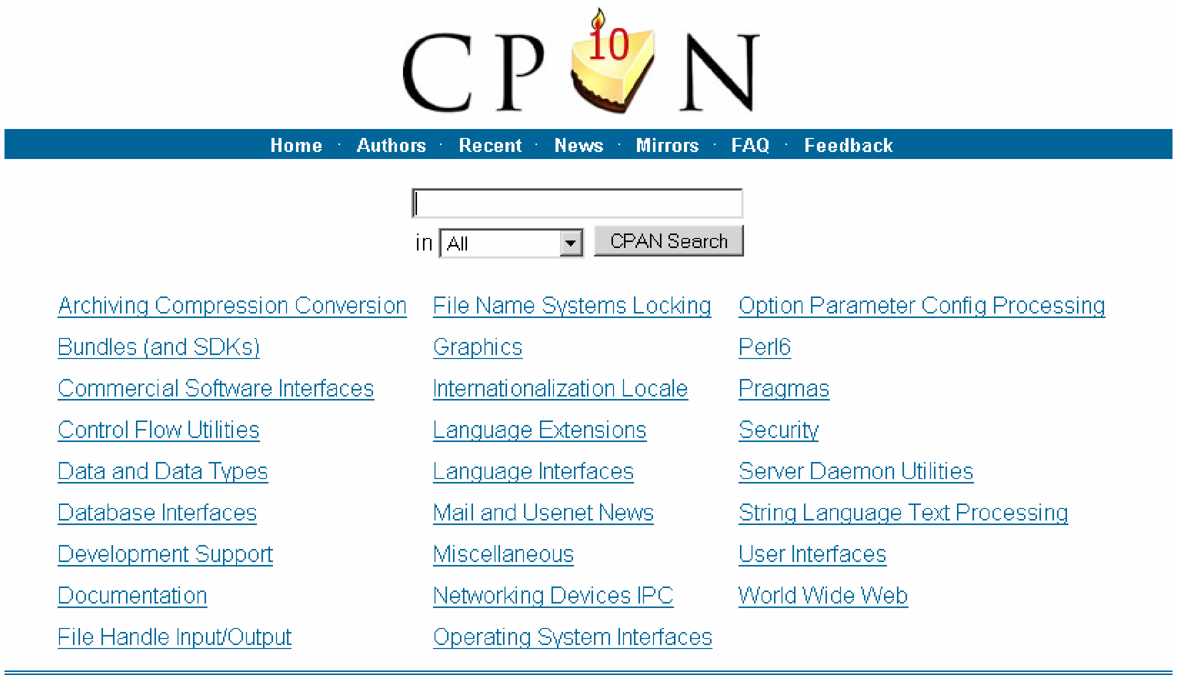
Using Array Slices

- an array slice is a list of several array elements
- you specify a set, or range, of indeces and obtain a list of corresponding elements
- syntax is a little wonky, but makes sense if you think about it

```
@list = (0..9);

$list[0]                first element
$list[1]                second element
($list[0],$list[1])     first, second elements
@list[0,1]              first, second elements
@list[0..2]             first three elements
@list[0..@list-1]       all elements

$list[0]                element, scalar context
@list[0]                slice, list context - same as ($list[0])

# array in original order
@list[0..@list-1]
# two ways to reverse an array - reverse elements or indexes!
@newlist = reverse @list;
@newlist = @list[ reverse(0..@list-1) ];
```

Using Modules

- modules are collections of Perl code written by other users that perform specific tasks
- modules can be downloaded from CPAN – Comprehensive Perl Archive Network
  - search.cpan.org



Archiving Compression Conversion	File Name Systems Locking	Option Parameter Config Processing
Bundles (and SDKs)	Graphics	Perl6
Commercial Software Interfaces	Internationalization Locale	Pragmas
Control Flow Utilities	Language Extensions	Security
Data and Data Types	Language Interfaces	Server Daemon Utilities
Database Interfaces	Mail and Usenet News	String Language Text Processing
Development Support	Miscellaneous	User Interfaces
Documentation	Networking Devices IPC	World Wide Web
File Handle Input/Output	Operating System Interfaces	

Math::VecStat

- a simple module is Math::VecStat
  - provides statistics about a list: min, max, average, sum, and so on
- import the module by use
- some module require that you specify which functions you wish to import into your namespace
- CPAN provides documentation about each module
  - man Math::VecStat

```
use Math::VecStat qw(average sum);

# both functions have been imported into current namespace
$avg = average(@list);
$sum = sum(@list);

# we didn't import this function, so must call it explicitly
$min = Math::VecStat::min(@list);
```

Fetching Current Date

- the main date function is localtime
  - list context returns
    - $sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst
    - month is 0-indexed !!!
    - add 1900 to year !!!
  - scalar context returns formatted date

```
$date = localtime;
print $date;

Tue May 30 14:11:56 2006

@list = localtime;
printf("day %d month %d year %d",$list[3],$list[4],$list[5]);
day 8 month 6 year 108

printf("day %d month %d year %d",(localtime)[3,4,5]);
```

Getting Epoch Value

- the UNIX epoch value is seconds since epoch
  - turn of epoch is Thu Jan 1 1970 (UTC)
- use `timelocal` from `Time::Local` module
- use `localtime(EPOCH)` to convert back to date values

```
@list = localtime;
# fetch the current day, month and year via array slice
($s,$min,$h,$d,$mm,$y) = @list[0..5];

# determine turn of epoch right now
$epoch = timelocal($s,$min,$h,$d,$mm,$y);
1215543818

# timelocal is the reverse of localtime – turns S,M,H,D,M,Y into epoch time
$epoch = timelocal( (localtime)[0..5] );

# epoch midnight tonight
print timelocal( 0,0,0, (localtime)[3..5] );
1215500400
```

Changing Array Size

- you grow an array by allocating new values

```
@list = ();
$list[99] = 1;
# you now have a 100 element array

$list[99] = undef;
# you still have a 100 element array – you cannot shrink array by setting
# elements to ‘undef’ since ‘undef’ is a perfectly good element value

$#list = 9;
# you now have a 10 element array – explicitly set the index of last element
```

- recall that `@list` in scalar context gives the size of list (number of elements)
- `$#list` is the index of the last element
  - `$#list ← @list-1`

Be wary of $_

- the current iterator value is $_
- $_ is an alias
- whatever $_ points to, can be altered in place

BAD USAGE

```
for (@list) {
  # read-only access to elements of @list - good
  print $_;
}

for (@list) {
  # you are altering $_ - since $_ is an alias, you are altering @list
  $_++;
}
```

Adding/Removing Elements from a List

- you cannot have a list of lists, unless you use references
- if you combine two lists, you will get a single, flattened list

```
# all these are valid ways to extend a list

push @list, $value;
push @list, @otherlist;
@list = (@onelist,@anotherlist);
@list = ($value,@anotherlist);
```

- remove elements with `shift` (from the front) or `pop` (from the back)

```
# ($x,@list) = ($list[0],@list[1..@list-1])
$x = shift @list

# (@list,$x) = (@list[0..@list-2],$list[-1]);
$x = pop @list;
```

Randomizing a List

- randomize a list by using a random sort routine

```
# ascending numerical sort
@list = sort { $a <=> $b } @list;

# random sort - shuffle
# pair-wise comparison independent of actual values - returns -1,0,-1 randomly
@randlist = sort { rand() <=> rand() } @list;

# shuffle the list by shuffling indices, not elements
@randlist = @list[ sort { rand() <=> rand() } (0..@list-1) ];
```

Using Hashes Effectively

- use a hash when storing relationships between data
  - fruit and color
  - base pair and frequency

```
# e.g., @clones contains a list of clones, e.g, qw(A0001A01, A0001B01, etc)
for (@clones) {
  $count{$_}++;
}
# use hashes to store pair-wise relationships
for $i (0..@clones-1) {
  for $j ($i+1..@clones-1) {
    ($ci,$cj) = @clones[$i,$j];
    if(clones_overlap($ci,$cj)) {
      $overlap{$ci} .= $cj;  # e.g., $overlap{A0001A01} = "A0012F01A0018G03A0024B03"
      $overlap{$cj} .= $ci;
    }
  }
}
# now extract names of all clones that overlap $clonename
@overlap_clones = $overlap{$clonename} =~ /.{8}/g;
```

- this example is artificial – you'll see better ways to do this when see references

Deleting from a Hash

- the only way to remove a key from a hash is to use delete

```
$hash{sheep} = “wooly”;

$hash{sheep} = undef;
# key sheep still exists, points to ‘undef’ value
if(exists $hash{sheep}) {
  # yup - key exists and this code runs
}

delete $hash{sheep};
if(exists $hash{sheep}) {
  # nope - key does not exist and this code does not run
}
```

Copy and Substitute in a Single Step

- copying a string and modifying it is a very common pair of steps

```
$y = $x;              # copy
$y =~ s/sheep/pig/g;  # substitute
```

- you can do both in one shot
  - you must use the brackets, or precedence *will* kill you

```
($y = $x) =~ s/sheep/pig/g;
```

- challenge – what is assigned to $y?

```
$x = “aaa”;

$y = $x =~ s/a/b/;  # what is $x and $y ?
$y = $x =~ s/a/b/g; # what is $x and $y ?
```

Morals

- print evaluates its arguments in list context – watch out!
- undef is a perfectly good value for a list or hash element
  - shrink lists by adjusting #$list
  - delete keys by using delete
  - distinguish between testing for truth (zero not ok) or definition (zero ok)
- $_ is an alias, not a copy of a value
  - do not adjust the value of $_ unless you are sure-footed
- character class [abc] matches only one character, not three
- for and foreach are synonymous
- qq{} interpolates but q{} does not
- use (m..n) range operator where possible (m≤n)
- keys/values return elements in no particular (but compatible) order
- replace strings with s/// rather than substr
  - s/REGEX/REPLACEMENT/ - the second argument is not a regex

1.0.8.1.8

Introduction to Perl
Session 8

- congratulations!