1.1.2.8.3

Intermediate Perl – Session 3

- `map`
  - transforming data
- `sort`
  - ranking data
- `grep`
  - extracting data
- use the man pages
  - `perldoc -f sort`
  - `perldoc -f grep`, etc

The Holy Triad of Data Munging

- Perl is a potent data munging language

- what is data munging?
  - search through data
  - transforming data
  - representing data
  - ranking data
  - fetching and dumping data

- *data* can be anything, but you should always think about the representation as independent of interpretation
  - instead of a list of sequences, think of a list of string
  - instead of a list of sequence lengths, think of a vector of numbers
  - different data with the same representation can be munged with the same tools

Cycle of Data Analysis

- you prepare data by
  - reading data from an external source (e.g. file, web, keyboard, etc)
  - creating data from a simulated process (e.g. list of random numbers)

- you analyze the data by
  - sorting the data to rank elements according to some feature
    - sort your random numbers numerically by their value
  - you select certain data elements
    - select your random numbers > 0.5
  - you transform data elements
    - square your random numbers

- you dump the data by
  - writing to external source (e.g. file, web, screen, process)

Brief Example

```
$N = 100;

# create a list of N random numbers in the range [0,1)
# URD - uniform random deviate
@urds = map { rand() } (1..$N); # is (0..$N-1) better here?

# extract those random numbers > 0.5
@big_urds = grep( $_ > 0.5, @urds);

# square the big urds
@big_square_urds = map { $_**2 } @big_urds;

# sort the big square urds
@big_square_sorted_urds = sort { $a <=> $b } @big_square_urds;
```

Episode I

map

Transforming data with map

- `map` is used to **transform data** by applying the same code to each element of a list
  - *x* → *f(x)*

- there are two ways to use map
  - `map EXPR, LIST`
    - apply an operator to each list element
    - `map int, @float`
    - `map sqrt, @naturals`
    - `map length, @strings`
    - `map scalar reverse, @strings`

  - `map BLOCK LIST`
    - apply a block of code to each list element, available as `$_` (**alias**)
    - `map { $_*$_ } @numbers`
    - `map { $lookup{$_} } @lookup_keys`

Ways to map and Ways Not to map

I’m a C programmer

```
for($i=0;$i<$N;$i++) {
  $urds[$i] = rand();
}
```

I’m a C/Perl programmer

```
for $idx (0..$N-1)
  push @urds, rand();
}
```

I’m a Perl programmer

```
my @urds = map rand(), (1..$N);
```

Ways to map and Ways Not to map

- do not use map for side effects unless you are certain of the consequences
  - you will regret it anyway
  - exceptions on next slide

```
@a    = ();
@urds = map { $a[$_]++ ; rand() } (1..$N);
```

- do not stuff too much into a single map block

Common Uses of map

- initialize arrays and hashes

```
@urds  = map rand(), (1..$N);
@caps  = map { uc($_) . “ “ . length($_) } @strings;
@funky = map { my_transformation($_) } (1..$N);
%hash  = map { $_ => my_transformation($_) } @strings;
```

- in-place array and hash transformation

```
map { $fruit_sizes{$_} ++ } keys %fruit_sizes;
map { $_++ } @numbers;
```

- map flattens lists – it executes the block in a list context

```
# a a a b b c
map { split(//,$_) } qw(aaa bb c)
# 1 1 2 1 4 3 1 4 9 4 1 4 9 16 5 1 4 9 16 25
map { $_ , map { $_ * $_ } (1..$_) } (1..5);
```

Generating Complex Structures With map

- use it to create lists of complex data structures

```
my @strings = qw(kitten puppy vulture);
my @complex = map { [ $_, length($_) ] } @strings;
my %complex = map { $_  => [ uc $_, length($_) ] } @strings;
```

@complex

```
[
  'kitten',
  6
],
[
  'puppy',
  5
],
[
  'vulture',
  7
]
```

%complex

```
  'puppy' => [
             'PUPPY',
             5
           ],
'vulture' => [
               'VULTURE',
               7
             ],
'kitten' => [
              'KITTEN',
              6
            ]
```

Distilling Data Structures with map

- extract parts of complex data structures with map

```
my @strings = qw(kitten puppy vulture);
my %complex = map { $_ => [ uc $_, length($_) ] } @strings;

# extract 2nd element from each list
my @lengths1 = map { $complex{$_}[1] } keys %complex;
my @lengths2 = map { $_->[1] } values %complex;
```

```
     %complex

  'puppy'  => [
               'PUPPY',
               5
             ],
'vulture'  => [
                 'VULTURE',
                 7
               ],
'kitten'  => [
               'KITTEN',
               6
             ]
```

- don’t forget that values returns all values in a hash

- use values instead of pulling values out by iterating over all keys
  - unless you need the actual key for something

Episode II

sort

Sorting Elements with sort

- sorting with sort is one of the many pleasures of using Perl
  - powerful and simple to use

- sort takes a list and a code reference (or block)

- the sort function returns -1, 0 or 1 depending how $a and $b are related
  - $a and $b are the internal representations of the elements being sorted
  - returns -1 if $a < $b
  - returns 0 if $a == $b
  - returns 1 if $a > $b

<=> and cmp for sorting numerically or ascibetically

- for most sorts the spaceship <=> operator and cmp will suffice
  - if not, create your own sort function

```
# sort numerically using spaceship
my @sorted = sort {$a <=> $b} (5,2,3,1,4);

# sort ascibetically using cmp
my @sorted = sort {$a cmp $b} qw(vulture kitten puppy);

# create a reference to sort function
my $by_num = sub { $a <=> $b };

# now use the reference as argument to sort
@sorted = sort $by_num (5,2,3,1,4);
```

Adjust sort order by exchanging $a and $b

- sort order is adjusted by changing the placement of $a and $b in the function
    - ascending if $a is left of $b
    - descending if $b is left of $a

```
# ascending
sort {$a <=> $b} @nums;
# descending
sort {$b <=> $a} @nums;
```

- sorting can be done by a transformed value of $a and $b
    - sort strings by their length

```
sort { length($a) <=> length($b) } @strings;
```

- sort strings by their reverse

```
sort { scalar(reverse $a) cmp scalar(reverse $b) } @strings;
```

Shuffling

- what happens if the sorting function does not return a deterministic value?
  - e.g. ordinality of $a and $b are random

```
# shuffle completely
sort { rand() <=> rand() } @nums;
```

- you can shuffle a little, or a lot, by peppering a little randomness into the sort routine

```
# shuffle to a degree
sort { $a+$k*rand() <=> $b+$k*rand() } (1..10);
```

```
k=2    1  2  3  4  5  7  6  8  9  10
k=3    2  1  3  6  5  4  8  7  9  10
k=5    1  3  2  7  4  6  5  8  9  10
k=10   1  2  5  8  4  7  6  3  9  10
```

Sorting by Multiple Values

- sometimes you want to sort using multiple fields

```
m ica qk bud d ipqi nehj t yq dcdl e vphx kz bhc pvfu
```

- sort strings by their length, and then asciibetically

```
sort { ( length($a) <=> length($b) ) || ( $a cmp $b ) } @strings;
```

```
d e m t kz qk yq bhc bud ica dcdl ipqi nehj pvfu vphx
```

- ascending by length, but descending asciibetically

```
sort { ( length($a) <=> length($b) ) || ( $b cmp $a ) } @strings;
```

```
t m e d yq qk kz ica bud bhc vphx pvfu nehj ipqi dcdl
```

Sorting Complex Data Structures

- sometimes you want to sort a data structure based on one, or more, of its elements
  - $a and $b will usually be references to objects within your data structure
  - sort the hash values

```
# sort using first element in value
# $a,$b are list references here
@sorted_values = sort { $a->[0] cmp $b->[0] } values %complex;
```

- sort the keys based on values

```
@sorted_keys = sort { $complex{$a}[0]
                      cmp
                      $complex{$b}[0] } keys %complex;
```

```
             %complex

  'puppy'   => [
               'PUPPY',
               5
             ],
'vulture'   => [
                 'VULTURE',
                 7
               ],
'kitten'  => [
               'KITTEN',
               6
             ]
```

Multiple Sorting of Complex Data Structures

- %hash here is a hash of lists (e.g. $hash{KEY} is a list reference)
  - ascending sort by length of key followed by descending sort of first value in list
  - we get a list of sorted keys – %hash is unchanged

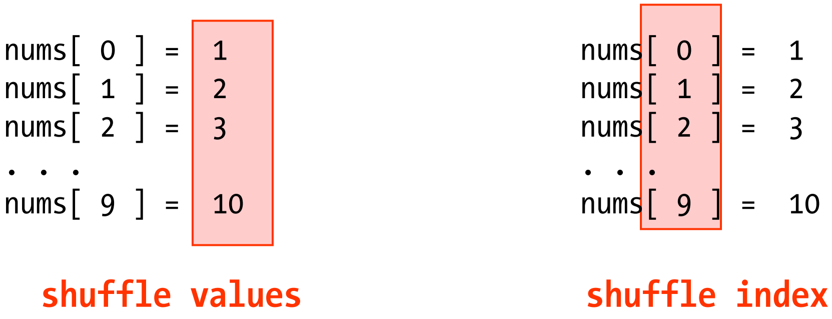
```
@sorted_keys = sort { ( length($a) <=> length($b) )
                      ||
                      ( $hash{$b}[0] cmp $hash{$a}[0] )
                    } keys %hash;

for $key (@sorted_keys) {
   $value = $hash{$key};
   ...
}
```

Slices and Sorting – Perl Factor 5, Captain!

- sort can be used very effectively with hash/array slices to transform data structures in place
  - rearrange list elements by explicitly adjusting index values
  - e.g. $a[$newi]=$a[$i]
  - or, @a[@newi] = @a

```
my @nums = (1..10);
my @nums_shuffle_2;
# shuffle the numbers - explicity shuffle values
my @nums_shuffle_1 = sort { rand() <=> rand() } @nums;
# shuffle indices in the slice
@nums_shuffle_2[ sort { rand() <=> rand() } @nums ] = @nums;
```



```
nums[ 0 ] =  1                nums[ 0 ] =  1
nums[ 1 ] =  2                nums[ 1 ] =  2
nums[ 2 ] =  3                nums[ 2 ] =  3
. . .                         . . .
nums[ 9 ] =  10               nums[ 9 ] =  10
```

shuffle values   shuffle index

Application of Slice Sorting

- suppose you have a lookup table and some data
  - `%table = (a=>1, b=>2, c=>3, … )`
  - `@data = ( [“a”,”vulture”],[“b”,”kitten”],[“c”,”puppy”],…)`

- you now want to recompute the lookup table so that key 1 points to the first element in sorted `@data`, key 2 points to the second, and so on. Let’s use lexical sorting.
  - the sorted data will be

```
# sorted by animal name
my @data_sorted = ([“b”,”kitten”],[“c”,”puppy”],[“a”,”vulture”]);
```

  - and the sorted table

```
# key 1 points to 1st element in list of first animal
my %table = (b=>1, c=>2, a=>3);
```

Application of Slice Sorting – cont’d

```
%table = (b=>1, c=>2, a=>3)
@data = ( [“b”,”kitten”],[“c”,”puppy”],[“a”,”vulture”])
```

```
@table { map { $_->[0] } sort { $a->[1] cmp $b->[2] } @data } = (1..@data)
```

hash slice with keys b,c,a

extract first letter of list (b, c, a)

sort data based on animal string

Schwartzian Transform

- used to sort by a temporary value derived from elements in your data structure
  - we sorted strings by their size like this

```
sort { length($a) <=> length($b) } @strings;
```

- if length() is expensive, we may wind up calling it a lot
  - the Schwartzian transform uses a map/sort/map idiom
    - **create a temporary data structure with map**
    - apply **sort**
    - **extract** your original elements with **map**

```
extract       sort by temporary data       create temporary structure
map {$_->[0]} sort { $a->[1] <=> $b->[1] } map { [ $_, length($_) ] } @strings;
```

- mitigate expense of sort routine is the **Orcish manoeuvre** (|| + cache)
  - use a lookup table for previously computed values of the sort routine

Episode III

grep

grep is used to extract data

- test elements of a list with an expression, usually a regex
- grep returns elements which pass the test
  - use it like a filter

```
@nums_big = grep( $_ > 10, @nums);
```

- please never use grep for side effects
  - you'll regret it

```
# increment all nums > 10 in @nums
grep( $_ > 10 && $_++, @nums);
```

Hash keys can be greped

- iterate through pertinent values in a hash

```
my @useful_keys_1 = grep( $_ =~ /seq/, keys %hash);

my @useful_keys_2 = grep /seq/, keys %hash;

my @useful_keys_3 = grep $hash{$_} =~ /aaaa/, keys %hash;

my @useful_values = grep /aaaa/, values %hash;
```

- follow grep up with a map to transform/extract grepped values

```
map { lc $hash{$_} } grep /seq/, keys %hash;
```

More greping

- extract all strings longer than 5 characters
  - grep after map

```
# argument to length (when missing) is assumed to be $_
grep length > 5, @strings;

# there is more than one way to do it - but this is the very long way
map { $_->[0] } grep( $_->[1] > 5, map { [ $_, length($_) ] } ) @strings
```

- looking through lists

```
if( grep($_ eq “vulture”, @animals)) {
  # beware - there is a vulture here
} else {
  # run freely my sheep, no vulture here
}
```

1.1.2.8.3

Introduction to Perl – Session 3

- `grep`
- `sort`
- `map`
- Schwartzian transform
- sort slices