1.2.2.1.1

Perl's sort/grep/map

- 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

- the “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
  - then think of what operations you can apply to your representation
  - 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

```
use strict;

my $N = 100;

# create a list of N random numbers in the range [0,1)
my @urds = map {rand()} (1..$N);

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

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

# sort the big square urds
my @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
  - think of f(x) and f(g(x)) – the latter applies f() to the output of g(x)
  - x :-> g(x), g(x) :-> f(g(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; list element is available as $_ **(alias)**, return value of block is used to create a new list
    - map { $_*$_ } @numbers
    - map { $lookup{$_} } @lookup_keys

Ways to **map** and Ways Not to **map**

I'm a C programmer


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

I'm a C/Perl programmer

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

I'm trying to forget C

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

I'm a Perl programmer


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

Map Acts on Array Element Reference

- the $_ in map's block is a reference of an array element
  - it can be therefore changed in place
  - this is a side effect that you may not want to experiment with

```
my @a = qw(1 2 3);
my @c = map { $_++ } @a;  # a is now (2,3,4)
```

- in the second call to map, elements of @a are altered
  - $_++ is incrementing a reference, $_, and therefore an element in @a

- challenge – what are the values of @a, @b and @c below?

```
my @a = qw(1 2 3);

my @b = map { $_++ } @a;
# what are the values of @a,@b now?
my @c = map { ++$_ } @a;
# what are the values of @a,@b,@c now?
```

Challenge Answer

```
my @a = qw(1 2 3);

my @b = map { $_++ } @a;

# @a = (2 3 4)
# @b = (1 2 3)

my @c = map { ++$_ } @a;

# @a = (3 4 5)
# @c = (3 4 5)
```

- remember that $_++ is a post-increment operator
  - returns $_ and then increments $_
- while ++$_ is a pre-increment operator
  - increments $_ and then returns new value ($_+1)

Common Uses of **map**

- initialize arrays and hashes

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

- array and hash transformation
  - using map's side effects is good usage, when called in void context

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

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

```
@a = map { split//,$_) } qw(aaa bb c) # returns qw(a a a b b c)
@b = map { $_ , map { $_ * $_ } (1..$_) } (1..5);
```

Nested Map

- what would this return?

```
@a = map { $_ , map { $_ * $_ } (1..$_) } (1..5);
```

- inner map returns the first N squares
- outer map acts as a loop from 1..5
  - 1 : inner map returns (1)
  - 2 : inner map returns (1,4)
  - 3 : inner map returns (1,4,9)
  - 4 : inner map returns (1,4,9,16)
  - 5 : inner map returns (1,4,9,16,25)

- final result is a flattened list

```
@a = (1,1,4,1,4,9,1,4,9,16,1,4,9,16,25);
```

Generating Complex Structures With **map**

- since map generates lists, 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;

my @lengths1 = map { $complex{$_}[1] } keys %complex;
my @lengths2 = map { $_->[1] } values %complex;
```

- 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

%complex

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

More Applications of Map

- you can use map to iterate over application of any operator, or function

- read the first 10 lines from filehandle FILE

```
my @lines = map {scalar <FILE>} (1..10);
```

- challenge: why scalar <F> ?
  - inside the block of map, the context is an array context
  - thus, <FILE> is called in an array context
  - when <FILE> is thus called it returns ALL lines from FILE, as a list
  - when <FILE> is called in a scalar context, it calls the next line

```
# this is a subtle bug - <FILE> used up after first call
my @lines = map {<FILE>} (1..10);
# same as
my @lines = <FILE>;
```

map with regex

- recall that inside map's block, the context is array

```
@a = split(//,"aaaabbbccd");

@b = map { /a/ } @a;
# @b = (1 1 1 1)

@b = map { /(a)/ } @a;
# @b = (a a a a)

@c = map { /a/g } @a;
# @c = (a a a a)
```

```
@a = split(//,"aaaabbbccd");

@b = map { s/a/A/ } @a;
# @b = (1 1 1 1)
# @a = (A A A A b b b c c d)
```

Episode II

sort

Sorting Elements with **sort**

- sorting with **sort** is one of the many pleasures of using Perl
  - powerful and simple to use
- we talked about sort in the last lecture
- 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 $b are the internal representations of the elements being sorted
  - they are not lexically scoped (don't need **my**)
  - they are package globals, but no need for **use vars qw($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);

# define how to sort - pedantically
my $by_num1 = sub { if ($a < $b) {
                      return -1;
                    } elsif ($a == $b) {
                      return 0;
                    } else {
                      return 1;
                    }
                  };

# same thing as $by_num1
my $by_num2 = sub { $a ⇔ $b };

@sorted = sort $by_num1 (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, $b
  - sort strings by their length

```
sort {length($a) ⇔ length($b)} @strings;
```

  - sort strings by their reverse

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

Sort Can Accept Subroutine Names

- sort SUBNAME LIST
  - define your sort routines separately, then call them

```
sub ascending {
  $a <=> $b
}

sort ascending @a;
```

- store your functions in a hash

```
my %f = ( ascending=>sub{$a<=>$b},
          descending=>sub{$b<=>$a},
          random=>sub{rand()<=>rand()} );

sort { &{$f{descending}} } @a
```

Shuffling

- what happens if the sorting function does not return a deterministic value?
  - e.g., sometimes 2<1, sometimes 2=1, sometimes 2>1

```
# shuffle
sort { rand() ⇔ rand() } @nums;
```

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

```
# shuffle
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
  - $a,$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
my @sorted_values = sort { $a->[0]
                           cmp
                           $b->[0]
                         } values %complex;
```

  - sort the keys using object they point to

```
my @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
  - ascending sort by length of key followed by descending lexical sort of first value in list
  - we get a list of sorted keys – **%hash is unchanged**

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

foreach my $key (@sorted_keys) {
   my $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

- you sort the array (hash) index (key)
  - cool, but sometimes tricky to wrap your head around


```
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() } (0..@nums-1) ] = @nums;
```

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

shuffle values

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

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 (sorted by animal name), 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 we want the sorted table to look like this
    - thus a points to 2, which is the rank of the animal that comes second in @sorted_data

```
my %table = (a=>3, b=>1, c=>2);
```

Application of Slice Sorting – cont'd

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

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

@table{ b c a } = (1,2,3)
$table{b} = 1
$table{c} = 2
$table{a} = 3
```

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

construct a hash slice with keys as . . .

first field from . . .

sort by 2nd field of . . .

@data

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;
```

  - which is OK, but 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**

```
map {$_->[0]}
sort { $a->[1] ⇔ $b->[1] }
map { [ $_, length($_) ] }
@strings;
```

- another way to mitigate expense of sort routine is the **Orcish manoeuvre** (|| + cache)
  - use a lookup table for previously computed values of the sort routine (**left as Google exercise**)

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
  - 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 grepped

- 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 Grepping

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

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

# there is more than one way to do it
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
}
```

1.1.2.8.2

Introduction to Perl – Session 3

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