1.0.1.8.4

Introduction to Perl
Session 4

- hashes
- sorting

10 GOOD THING do this for clarity and conciseness	IDIOM Perlish construct	BAD USAGE unless you are a donkey, don’t do this

Recap

- array variables are prefixed by @ and are 0-indexed

```
@array = (1,2,3);
@array = (1..3);

$array[0];              # first element
$array[1];              # second element
$array[-1];             # last element
$array[-2];             # second-last element

$#array;                # index of last element

@newarray = @array;     # make a copy of array – list context

$length = @array;       # number of elements in array – scalar context

$array[$#array];        # last element
$array[@array-1];       # last element
```

IDIOM

- arrays are used when
  - you have an ordered set of values, or
  - you want to group values together, without caring about order

Recap

- we iterated over an array in two ways
  - iterate over elements
  - iterate over index

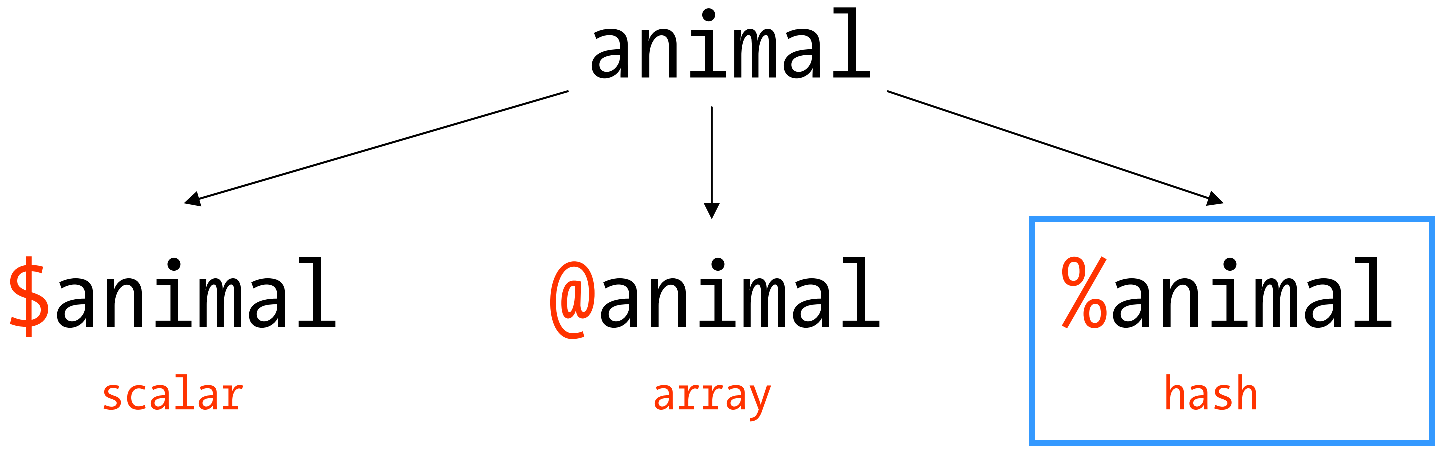
10 GOOD THING

```
@array = (1..10);

# iterate over elements
for $elem (@array) {
  print qq{element is $elem};
}

# iterate over index
for $i (0..@array-1) {
  print qq{index is $i element is $array[$i]};
}
```

- we saw that arrays grow and shrink as necessary
  - push/unshift were used to add elements to back/front of array
  - manipulating $#array directly changed the size of the array

Final Variable Type - Hash

- recall that Perl variables are preceded by a character that identifies the plurality of the variable



animal

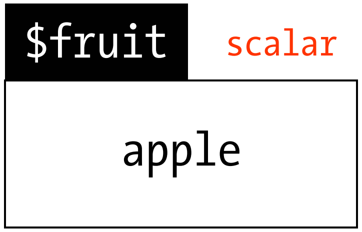
$animal
scalar

@animal
array

%animal
hash

- today we will explore the hash variable, prefixed by %
- an array is a set of elements indexed by a range of integers [0,1,2,...]
- a hash is a set of elements indexed by any set of distinct strings

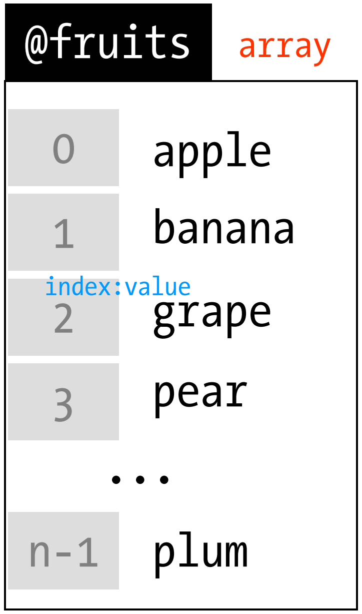
Scalars, Arrays and Hashes



`$fruit` scalar

apple

- scalar holds a single value
- “indexed” by variable name

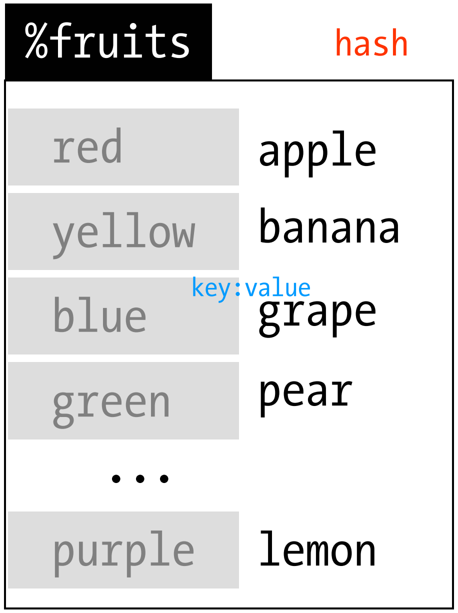


`@fruits` array

index:value

0	apple
1	banana
2	grape
3	pear
...	
n-1	plum

- array holds any number of values
- elements indexed by integers `0..n-1`, where `n` is the number of elements
- elements are stored in order, i.e. there is a sense of previous/next element



`%fruits` hash

key:value

red	apple
yellow	banana
blue	grape
green	pear
...	
purple	lemon

- hash holds any number of values
- values indexed by strings (keys), which must be unique
- values are not stored in order, there is no sense of previous/next element

Declaring and Initializing Hashes

- a hash is composed of a set of key/value pairs
- an element is accessed using $hash{key} syntax
  - c.f. $array[$index]
  - whereas [ ] were used for arrays, { } are used in hashes

```
@array  = ();     # empty array
%fruits = ();     # empty hash

$fruits{yellow} = “banana”;
$fruits{red}    = qq(apple);
$fruits{green}  = q(pear);

($fruits{purple},$fruits{orange}) = qw(plum mango);
```

Declaring and Initializing Hashes

- you can declare and initialize an entire hash at once
  - you do not need to quote single-word keys

```
%fruits = ( yellow => "banana",
            red    => "apple",
            green  => "pear" ) ;
```

- hash can be interpreted as an array with even number of elements with element 2i being the key and 2i+1 being the value

```
%fruits = ( yellow => "banana",
            red    => "apple",
            green  => "pear" );
```

notice the ( ) brackets here which are reminiscent of initializing an array

do not use { } brackets when initializing a hash

you'll get strange results which we will explore in Intermediate Perl

Accessing Hash Elements

- to fetch a hash value, use $hash{$key}

```
print qq(One red fruit is an $fruits{red});
print qq(One green fruit is a $fruits{green});
```

- if you have a list of the keys, you can iterate across the hash

```
@colors = qw(red green purple orange yellow);

for $color (@colors) {
  print qq($fruits{$color} is $color);
}
```

- most of the time you won’t have the list of keys and will need to get it from the hash directly – this is where keys comes in

Extracting Hash Keys with keys

- the keys function returns a list of the keys of the hash
  - the keys are returned in no particular (but reproducible if the hash is not altered) order

IDIOM

```
@colors = keys %fruits;
for $color (@colors) {
  print qq($fruits{$color} is $color);
}

# it’s better to avoid a temporary variable that holds the keys
for $color (keys %fruits) {
  print qq($fruits{$color} is $color);
}
```

An Example – OMG a real script!

- let’s create a script that performs the following
  - creates 1000 random 4bp sequences
  - stores and prints the number of times each sequence has been seen
  - returns sequences and counts of all sequences that contain aaa, ccc, ggg or ttt
  - returns the number of a, c, g and t characters across all sequences

```
@bp = qw(a t g c);
# explicitly initialize the list of sequences
@sequences = ();
for (1..1000) {
  # set the sequence to an empty string - not necessary
  $seq = “”;
  for (1..4) {
    # add a random base pair
    $seq = $seq . $bp[rand(@bp)];
  }
  push @sequences, $seq;
}

# in Intermediate Perl you will see how to take the code above and write instead
@sequences = map { join("", map { qw(a t g c)[rand(4)] } (1..4)) } (1..1000);
```

An Example

- we now have our 1,000 random sequences

```
@sequences ← qw(atgc aatg ggtc ... ggtc);
```

- let’s count how many times each sequence appears
  - we’re going to use a hash
  - the key is the sequence
  - the value is the number of times it is seen

```
%sequence_count = ();

for $seq (@sequences) {
  $sequence_count{$seq} = $sequence_count{$seq} + 1;
}
```

An Example

- to print the number of times each sequence has been seen, iterate through the hash of counts

```
for $seq (keys %sequence_count) {
  print qq(sequence $seq seen $sequence_count{$seq} times);
}

sequence acgc seen 3 times
sequence ggta seen 2 times
sequence aacg seen 3 times
sequence gatt seen 6 times
...
```

- how many unique sequences were seen?
  - this is the number of keys in the hash

```
my $unique_sequence_count = keys %sequence_count;
print qq(Saw $unique_sequence_count unique sequences);
```

An Example

- now let’s report on sequences that contain aaa, ttt, ggg or ccc
  - still iterating across the entire hash
  - applying regex to key – using alternation via | (i.e. aaa OR ttt OR ccc OR ggg)

```
for $seq (keys %sequence_count) {
  if ($seq =~ /aaa|ttt|ccc|ggg/) {
    print qq(3-homo polymer sequence $seq seen $sequence_count{$seq} times);
  }
}

3-homo polymer sequence aaag seen 3 times
3-homo polymer sequence gaaa seen 4 times
3-homo polymer sequence aaaa seen 9 times
3-homo polymer sequence accc seen 2 times
3-homo polymer sequence cccc seen 5 times
3-homo polymer sequence tttt seen 4 times
...
```

regex “|” is alternation
$str =~ /this|that/;

An Example

- finally, let’s count all the base pairs across all sequences

```
%bp_count = ();
# method 1 - iterate across sequences, split sequence into list of characters
for $seq (@sequences) {
  for $bp (split(“”,$seq)) {
    $bp_count{$bp} = $bp_count{$bp} + 1;
  }
}
# method 2 - iterate across hash, split key, increment by hash value
for $seq (keys %sequence_count) {
  for $bp (split(“”,$seq)) {
    $bp_count{$bp} = $bp_count{$bp} + $sequence_count{$seq};
  }
}

for $bp (keys %bp_count) {
  print qq(base pair $bp seen $bp_count{$bp} times);
}

base pair c seen 1053 times
base pair a seen 979 times
base pair g seen 997 times
base pair t seen 971 times
```

split(“”,$string) produces list of individual characters in $string

```
split(“”,”baby”)
→
qw(b a b y)
```

Iterating Across a Hash with values

- consider the task of determining the average number of times a sequence appears
  - we want the sequence counts, but not necessarily the sequences
  - we don’t care about the key
  - we care about the value

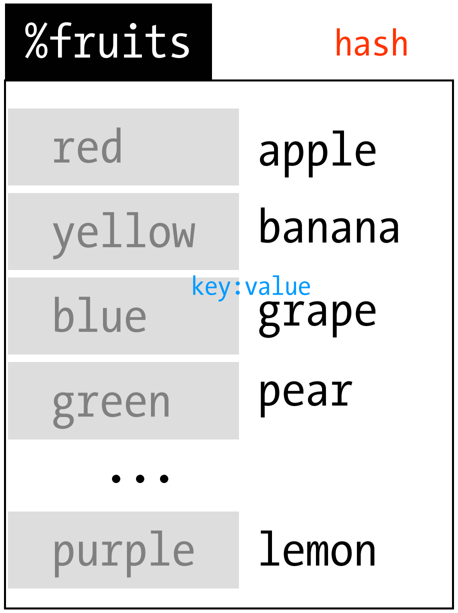
- we can accomplish this by verbosely iterating across with keys and fetching the counts via $sequence_count{$key}

```
$sum = 0;
for $seq (keys %sequence_count) {
  $sum = $sum + $sequence_count{$seq};
}
print “average sequence count is ”,$sum / keys %sequence_count;
```

- we can be more concise by using values

Iterating across a Hash with values

- recall that keys produced a list of a hash's keys
- values returns a list of a hash's values



%fruits hash

key:value

red	apple
yellow	banana
blue	grape
green	pear
...	
purple	lemon

```
keys %fruits → qw(red yellow blue green purple)

values %fruits → qw(apple banana grape pear lemon)
```

Iterating across a Hash with values

- we’re now in a position to determine the average count
  - if not, assume position

10 GOOD THING

```
$sum = 0;
for $count (values %sequence_count) {
  $sum = $sum + $count;
}
print “average sequence count is ”,$sum / keys %sequence_count;

averge sequence count is 4.01606425702811
```

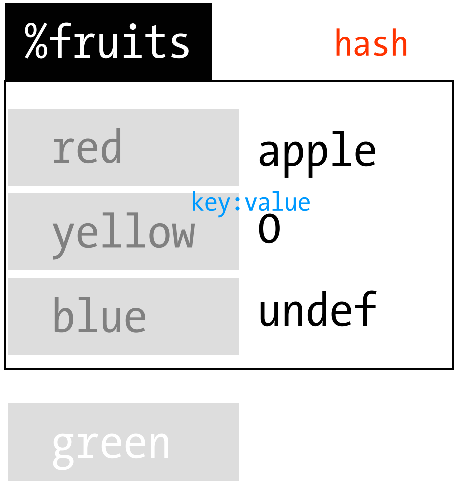
- remember that a hash has no inherent order
  - when you use keys, generally it is to use the list for iterating over the hash
  - when you use values, generally it is because you don’t need the keys

IDIOM

Checking for Existence

- given an array, you can easily determine whether a certain index is populated
  - fetch $#array
  - elements indexed by 0..$#array exist, though any of them may be undefined (undef)

- given a hash, it is frequently desirable to check whether a certain key exists
  - like with arrays, a key may exist but point to an undefined value (undef)



%fruits hash

key:value	
red	apple
yellow	0
blue	undef

green

```
if $fruits{red}             # value=apple, TRUE

if $fruits{yellow}          # value=0, FALSE
if defined $fruits{yellow} # value=0, 0 is defined → TRUE

if $fruits{blue}            # value=undef, FALSE
if defined $fruits{blue}   # value=undef, FALSE
if exists $fruits{blue}    # value=undef, key exists → TRUE

if exists $fruits{green}   # no such key, FALSE
```

Testing Values with defined vs exists

- exists is used on arrays/hashes to check whether an element/key has ever been initialized
  - an element is true only if it is defined
  - an element is defined only if it exists
  - both statements are not necessarily true in the converse
    - e.g., 0 is defined but is not true
    - e.g., undef exists, but it is not defined

- be conscious of testing values (e.g. counts) which may be zero
  - are you testing for truth (excludes zero) or definition (includes zero)

```
if $sequence{atgc}          # TRUE only if atgc key exists and hash value is TRUE

if defined $sequence{atgc}  # TRUE if atgc key exists and hash value is defined (e.g. 0)

if exists $sequence{atgc}   # TRUE if atgc key exists (hash value may be undefined)
```

Quick Hash Recap

```
%fruits = ();
$fruits{red}    = “apple”;
$fruits{green}  = “pear”;
$fruits{yellow} = “lemon;

keys %fruits;   # qw(red green yellow), but in no particular order
values %fruits; # qw(apple pear lemon), but in no particular order
                #                      (but compatible with output of keys)

for $color (keys %fruits) {
 ... $fruits{$color} ...
}

for $fruit (values %fruits) {
 ... $fruit ...
}

print “no color purple” if ! exists $fruits{purple};
print “found color red” if exists $fruits{red};
print “found red fruit” if defined $fruits{red};
```

Sorting

- we've seen several Perl functions now, such as `print`, `split` and `join`
  - they each took one or more arguments
- Perl's `sort` is slightly different
  - it takes as arguments a function and a list
  - the list tells sort what to sort
  - the function tells sort how to sort
- what does sorting require?
  - a set of elements
  - for a given pair of elements, some method to determine which comes first
    - e.g. size (numbers) or alphabetical order (characters) or length (strings)

Sorting - Introduction

```
# by default sort will arrange things ASCIIbetically - good for strings
@sorted_sequences = sort @sequences;

for $seq (@sorted_sequences) {
  print $seq;
}

aaaa
aaaa
aaaa
aaaa
aaaa
aaaa
aaac
aaag
aaag
aaat
aaat
aaat
aaca
aaca
aaca
aacc
aacc
...
```

Sorting - Introduction

```
# remember - ASCIIbetically! - bad for numbers
for $num (sort (1..20)) {
  print $num;
}

1
10
11
12
13
14
15
16
17
18
19
2
20
3
4
5
6
7
8
9
```

Sorting – Specifying How

- to tell sort how to sort, the sort { CODE } LIST paradigm is used
  - CODE is Perl code that informs sort about the relative ordinality of two elements

```
@nums = (1..20);

# default sort - asciibetic - not what we want
@nums_sorted = sort @nums;

# numerical sort, ascending order
@nums_sorted = sort { $a <=> $b } @nums
```

- the <=> is the spaceship operator
  - returns relative ordinality of numbers

```
                    -1 if $a < $b
$a <=> $b    →       0 if $a == $b
                    +1 if $a > $b
```

Sort – Specifying How

- while <=> is the operator for relative ordinality of numbers, cmp is the corresponding operator for strings

```
# asciibetic sort, ascending order
@sequences_sorted = sort { $a cmp $b } @sequences;

# { $a cmp $b } is sort's default behaviour
# the above gives the same result as
@sequences_sorted = sort @sequences;
```

```
                       -1 if $a lt $b
$a cmp $b    →          0 if $a eq $b
                       +1 if $a gt $b
```

lt, eq, gt
string equivalents of
<, ==, > comparisons

Sort – Specifying Direction

- to specify the direction of sort, it is sufficient to exchange the position of the $a and $b variables

```
# numerical sort, ascending order
@nums_sorted = sort { $a <=> $b } @nums

# numerical sort, descending order
@nums_sorted = sort { $b <=> $a } @nums
```

Sorting in Place

- you can sort in place, without defining temporary variables

```
# sort in place
@sequences = sort @sequences;
```

- sort returns a list, so you can do anything with the output of sort that you can do with a list

```
# sort and concatenate in place
$big_sequence = join(“”, sort @sequences);
```

- what do you think these do?

```
$x   = sort @sequences;
($y) = sort @sequences;
```

More Complex Sorting

- the CODE passed to sort can be anything you want
  - remember, it is expected to return -1, 0 or 1 based on the relative ordinality
  - it can use other information to sort your elements

```
# recall length() returns the length of a string
@strings = sort { length($a) <=> length($b) } @strings
```

- applying a function to $a and $b during sort is common
  - sort based on transformed values

IDIOM

```
# for some function f()
sort { f($a) <=> f($b) } @array
```

Shuffling

- you can short circuit the sort algorithm by feeding it random results

```
sort { rand() <=> rand() } @array
```

- here relative ordinality is not based on the value of sorted elements, but determined based on two random numbers
- since `CODE` should return `-1, 0, 1` all you need is to return one of these values, at random
  - rand(3) returns a random float in the range [0,3)
  - int(rand(3)) truncates the decimal, resulting in random value from [0,1,2]
  - int(rand(3))-1 therefore maps randomly onto [-1,0,1]

```
sort { int( rand(3) ) - 1 } @array
```

Sorting Based on Hash Values

- frequently you want to iterate through an array or hash in an ordered fashion based on array or hash contents
- we iterated through the hash using keys, but remember that this was done in no order in particular (hashes aren't ordered data structures)
- recall the %sequence_counts hash
  - how do we iterate across it from most to least frequently seen sequence?

```
# this iteration is in no particular order
for $seq (keys %sequence_count) {
  print qq(sequence $seq seen $sequence_count{$seq} times);
}
```

- we want the keys to be sorted based on their associated values
  - first key points to largest value
  - second key points to second-largest value, etc

Sorting Based on Hash Values

```
# this iteration is from most to least common sequence
for $seq (sort { $sequence_count{$b} <=> $sequence_count{$a} } keys %sequence_count) {
  print qq(sequence $seq seen $sequence_count{$seq} times);
}

sequence tcca seen 11 times
sequence ctcg seen 11 times
sequence aagc seen 10 times
sequence tatc seen 10 times
sequence cgcg seen 10 times
sequence ggga seen 8 times
sequence cccg seen 8 times
sequence tata seen 8 times
sequence gagc seen 8 times
sequence ccga seen 8 times
sequence cttt seen 8 times
sequence gtga seen 7 times
sequence tgct seen 7 times
...
```

Sorting Based on Array Values

- consider an array of 10 random numbers

```
for (1..10) { push @random_numbers, rand() }

# iterate across the index of the array in the order it was created
for $i ( 0..@random_numbers-1 ) {
  print qq(index $i value $random_numbers[$i]);
}

# sort across the index based on array values - ascending, numerical order
for $i ( sort { $random_numbers[$a] <=> $random_numbers[$b] } (0..@random_numbers-1) ) {
  print qq(index $i value $random_numbers[$i]);
}
```

(A) marks the first loop; (B) marks the second loop.

(A)

```
index 0 value 0.735566278709605
index 1 value 0.247935712860926
index 2 value 0.381146766836238
index 3 value 0.0509500776300023
index 4 value 0.00419793862081264
index 5 value 0.973254105396197
index 6 value 0.390908373233685
index 7 value 0.438150045622688
index 8 value 0.605247161178035
index 9 value 0.141159687585446
```

(B)

```
index 4 value 0.00419793862081264
index 3 value 0.0509500776300023
index 9 value 0.141159687585446
index 1 value 0.247935712860926
index 2 value 0.381146766836238
index 6 value 0.390908373233685
index 7 value 0.438150045622688
index 8 value 0.605247161178035
index 0 value 0.735566278709605
index 5 value 0.973254105396197
```

1.0.8.1.4

Introduction to Perl
Session 4

- you now know
  - all about hashes
  - declaring and initializing a hash
  - iterating across keys and values of a hash
  - checking for existence of a key
  - checking for definition of a value
  - numerical and asciibetical sorting
  - changing sort order
  - random shuffling
  - sorting based on complex conditions

  (and that's a lot!)