

Hashes

- What are they?
- What's so special about hashes?
- When would I use one vs. an array?

What are hashes?

- Used to be called *associative arrays*, but that's too many syllables to say. So now we just say *hash*.
- The word hash is a colloquialism. It refers to how lookup tables like this are usually implemented, part of which involves calculating a quick "hash value" for each key.

What are hashes?

- Like an array: it's a collection of scalar data with individual elements selected by some index value.
- But, the index value doesn't have to be a number. Instead it can be an arbitrary scalar. These are called *keys*. Each key has a corresponding value
- Important concept: keys and values

Hashes vs Arrays

Array

```
@foo = qw(John Sue Jim Anne);
```

Key	Value
0	John
1	Sue
2	Jim
3	Anne

Hash

```
%foo =  
(Doug => 'Treder',  
Sue => 'Storm',  
2 => 'Two', foo => 'bar');
```

Key	Value
Doug	Treder
Sue	Storm
2	Two
foo	bar

Key => Value

- Hashes are designed for very fast lookups for a particular element. (ie: given a key, what is the corresponding value?)
- The elements of a hash do **not** have a particular order.
- Repeat : keys in a hash are *not* ordered

Hashes

- Hashes use the % sign, instead of \$ or @.
- Hashes are initialized using a list.

```
# create %users hash with 'dtreder' and  
# 'igor' as keys, and 'Doug Treder' and  
# 'Igor Stravinsky' as values
```

```
%users = ('dtreder', 'Doug Treder',  
          'igor', 'Igor Stravinsky');
```

Accessing a hash element

- Each element is accessed using {}, in a similar way to arrays.

```
@users = qw(dtreder billc gw bush);
$users[3] = 'monica';           # assignment

%users = ('dtreder', 'Doug Treder',
          'gw bush', 'George Bush');
$users{'al gore'} = 'Al Gore'; # assignment

print "$users{'dtreder'}\n"; # prints 'Doug Treder'
print "$users{'Doug Treder'}\n"; # prints what?
```

Coffee Shop



- Start with a %menu
- Take an \$order
- Lookup the \$price

```
print "Whaddya want? ";  
$order = <>; chomp $order;  
$price = $menu{ $order };  
print "That'll be $price\n";
```

Coffee Shop

- Start with a %menu
- Take an \$order - lookup the \$price

```
my %menu = ('latte' => 2.99,  
           'mocha' => 3.99,  
           'cappuccino' => 3.49 );  
while (1) {  
    print "Whaddya want?";  
    my $order = lc(<>); chomp $order;  
    print "That'll be $menu{$order}\n";  
}
```

Hash assignment

- Assigning hashes with lists: Perl will take every other element as a key, and every other element as a value:

```
%noises = qw(dog bark cat meow sheep baa);  
@foods = ('carrot', 'orange', 'lemon',  
          'yellow', 'orange', 'orange');  
%foods = @foods;
```

```
print $noises{'cat'}      . "\n";  
print $foods{'orange'}   . "\n";  
print $noises{'sheep'}   . "\n";  
print $foods[4]          . "\n";
```

=>

- The Fat Comma is *syntactic sugar*
- You should use the => syntax instead of a comma
- It *auto-quotes* the *bareword* before it:
- It doesn't work on multiple words - use quotes

```
# the Perl-ish way to create a hash:  
%users = (dtreder => 'Doug Treder',  
          gwbush => 'George Bush');
```

{sugar}



- The curly braces for a hash lookup *also* do the autoquoting on a single bareword
- Just like the fat comma
- Can help readability with a bit of DWIMery

```
my %menu = ('latte', 2.99, 'mocha', 3.99);  
printf "Price of latte \$.2f\n", $menu{'latte'};
```

{sugar}



- The curly braces for a hash lookup *also* do the autoquoting on a single bareword
- Just like the fat comma
- Can help readability with a bit of DWIMery

```
my %menu = (latte => 2.99, mocha => 3.99);  
printf "Price of latte \$.2f\n", $menu{latte};
```

New Items

- Adding items to the hash is easy
- Just assign to the hash key:
- Don't worry about making space
- Collisions *overwrite*:



```
$menu{macchiato} = 4.15;  
$menu{hot_chocolate} = 1.75;  
$menu{latte} = 4.99; # price increase!;
```

Literally

Unlike scalar literals

```
123          "the fierce bunny"      undef
```

and list literals

```
(1,2,3,4)    qw(with sharp pointy teeth)
```

There's no hash literal. There's only copying an even-numbered list into a hash.

```
%menu = ( coffee => 2.99,  latte => 3.99 );
```

Literal Representation

- So, Perl unwinds the key/value pairs into a list.

```
%bar = (Doug => 'Treder', Al => 'Gore');
```

```
@foo = %bar; # foo might now contain  
# ('Doug', 'Treder', 'Al', 'Gore'); # or 'Al' first
```

```
%bar2 = @foo; # %bar2 now equals %bar
```

```
# this also works to copy a hash
```

```
%bar2 = %bar;
```

reverse swaps Key/Values

```
%names = (Doug => 'Treder',  
          George => 'Bush');
```

```
%names2 = reverse %names;
```

```
print $names2{'Bush'}, "\n";
```

```
# what's going on behind the scenes  
# here?
```

Hash Keys are Unique

```
%names = (John => 'Smith',  
          Susan => 'Smith');  
  
%names2 = reverse %names;  
  
print $names2{'Smith'}, "\n";  
  
# what does this print??
```

Uniqueness



Uniqueness

- How do you catch a unique rabbit?



Uniqueness

- How do you catch a unique rabbit?
- **Unique** up on it



Uniqueness

- How do you catch a unique rabbit?
- **Unique** up on it
- How do you catch a tame rabbit?



Uniqueness

- How do you catch a unique rabbit?
- **Unique** up on it
- How do you catch a tame rabbit?
- **Tame** way, **unique** up on it



Uniqueness

- How to filter out only the unique values

```
my @words = qw(one fish two fish red fish blue fish);
my %unique;

foreach my $word (@words) {
    $unique{$word} = 1;
}

print "Unique words: " . keys %unique;
```

keys

- There's no ordering on keys
- The order of keys could be different on different runs of your program
- Different on different computers or versions of perl
- But Perl will always keep a value with its key

Key Iterator



- Here's how to show all the keys
- In our example, the keys are also readable!

```
my %menu = (latte => 2.99, mocha => 3.99);  
printf "Price of latte \$.2f\n", $menu{latte};
```

Key Iterator



- Here's how to show all the keys
- In our example, the keys are also readable!

```
my %menu = (latte => 2.99, mocha => 3.99);  
foreach my $drink (keys %menu) {  
    printf "Price of $drink \$.2f\n", $menu{$drink};  
}
```

Key Iterator



- Here's how to show all the keys
- In our example, the keys are also readable!

Price of latte: \$2.99
Price of mocha: \$3.99

```
my %menu = (latte => 2.99, mocha => 3.99);  
foreach my $drink (keys %menu) {  
    printf "Price of $drink \$.2f\n", $menu{$drink};  
}
```

Key Iterator



- Sometimes your keys AREN'T readable.
- Printing them out makes no sense.
- Businesses love codes like this:

```
my %menu = (LAT486 => 2.99, MOCH27 => 3.99);  
foreach my $drink (keys %menu) {  
    printf "Price of $drink \$.2f\n", $menu{$drink};  
}
```

Key Iterator



- Sometimes your keys AREN'T readable.
- Printing them out makes no sense.
- Businesses love codes like this:

Price of LAT486: \$2.99
Price of MOCH27: \$3.99

```
my %menu = (LAT486 => 2.99, MOCH27 => 3.99);  
foreach my $drink (keys %menu) {  
    printf "Price of $drink \$.2f\n", $menu{$drink};  
}
```

Key Iterator



- Sometimes your keys AREN'T readable.
- Use another hash for the readable strings.

```
my %drinks = (LAT486 => 'latte',  
             MOCH27 => 'mocha');  
my %menu = (LAT486 => 2.99,  
           MOCH27 => 3.99);
```

```
foreach my $code (keys %menu) {  
    printf "Price of %s:\$%.2f\n",  
          $drinks{$code}, $menu{$code};  
}
```

Key Iterator



- Sometimes your keys AREN'T readable.
- Use another hash for the readable strings.

```
my %drinks = (LAT486 => 'latte',  
             MOCH27 => 'mocha');  
my %menu = (LAT486 => 2.99,  
           MOCH27 => 3.99);
```

```
foreach my $code (keys %menu) {  
    printf "Price of %s:\$%.2f\n",  
          $drinks{$code}, $menu{$code};  
}
```

Price of latte: \$2.99
Price of mocha: \$3.99

each



- If you know you need all the keys and all the values,
- each is a more efficient way to get them all:

```
my %menu = (latte => 2.99, mocha => 3.99);  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \\\$.2f\\n", $price;  
}
```

each



- If you know you need all the keys and all the values,
- each is a more efficient way to get them all:

Price of latte: \$2.99
Price of mocha: \$3.99

```
my %menu = (latte => 2.99, mocha => 3.99);  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \\\$.2f\\n", $price;  
}
```

each



- If you know you need all the keys and all the values,
- each is a more efficient way to get them all:
- each gives you no control over the ordering, though

```
my %menu = (latte => 2.99, mocha => 3.99);  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \$.2f\n", $price;  
}
```

each



- If you know you need all the keys and all the values,
- each is a more efficient way to get them all:
- each gives you no control over the ordering, though

Price of mocha: \$3.99
Price of latte: \$2.99

```
my %menu = (latte => 2.99, mocha => 3.99);  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \$.2f\n", $price;  
}
```

each



- If you know you need all the keys and all the values,
- *and* you need control over the ordering:
- get the keys with `keys` and `munge`
- for example, sorting:

```
my %menu = (latte => 2.99, mocha => 3.99);  
foreach my $drink (sort keys %menu) {  
    printf "Price of $drink \$.2f\n", $menu{$drink};  
}
```

each



- If you know you need all the keys and all the values,
- *and* you need control over the ordering:
- get the keys with `keys` and `munge`
- for example, sorting:

Price of latte: \$2.99
Price of mocha: \$3.99

```
my %menu = (latte => 2.99, mocha => 3.99);  
foreach my $drink (sort keys %menu) {  
    printf "Price of $drink \$.2f\n", $menu{$drink};  
}
```

values

- values-- returns a list of all the values.
- (not used as much as keys)

```
my %foo = (Doug => 'Treder',  
          George => 'Clinton');
```

```
my @last_names = values %foo;  
print "My favorite people are: ";  
print join(', ', @lastnames);  
print "\n";
```

delete

- Taking a key-value pair out of the hash
- Should you use undef ?



```
# milk is just too expensive!  
undef $menu{latte};  
# no one can pronounce it!  
$menu{macchiato} = undef;
```

delete

- Taking a key-value pair out of the hash
- Should you use undef ?



```
# milk is just too expensive!  
undef $menu{latte};  
# no one can pronounce it!  
$menu{macchiato} = undef;
```

delete

- Using undef leaves the key in the hash with a value of undef



```
$menu{latte} = undef;
while (my ($drink, $price) = each %menu) {
    printf "Price of $drink \${%.2f}\n", $price;
}
```

delete

- Using undef leaves the key in the hash with a value of undef

Price of latte \$0.00
Price of mocha \$3.99



```
$menu{latte} = undef;  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \${%.2f}\n", $price;  
}
```

delete

- Use delete to remove a key and value.



```
delete $menu{latte};  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \$.2f\n", $price;  
}
```

delete

- Use delete to remove a key and value.

Price of mocha \$3.99



```
delete $menu{latte};  
while (my ($drink, $price) = each %menu) {  
    printf "Price of $drink \${%.2f}\n", $price;  
}
```

exists

- So, if you can have a key with value undef
- How can you check whether a key is in the hash?

```
my $found_pumpkin;
foreach my $item (keys %menu) {
    if ($item eq 'pumpkin_spice') {
        $found_pumpkin = 1;
    }
}
if ($found_pumpkin) {
    print "Yes, we have pumpkin spice!\n";
}
```

exists

- Can if you can have a loop with value and of

Doing linear scans over an associative array is like trying to club someone to death with a loaded Uzi.

- Larry Wall

}

exists

- So, if you can have a key with value undef
- How can you check whether a key is in the hash?
- If the value is 0 or undef, a simple `if` will be false!

```
if ($menu{pumpkin_spice}) {  
    # doesn't work if price is 0 or undef!  
    print "Yes, we have pumpkin spice!\n";  
}
```

exists

- So, if you can have a key with value undef
- How can you check whether a key is in the hash?
- If the value is 0 or undef, a simple `if` will be false!
- `exists` checks whether the key is in the hash

```
if (exists $menu{pumpkin_spice}) {  
    print "Yes, we have pumpkin spice!\n";  
    if ($menu{pumpkin_spice} <= 0) {  
        print "And it's free today!\n";  
    }  
}
```

exists vs defined

```
%foo = (Doug => 'Treder',  
        John => 0,  
        Fred => undef);  
  
foreach $key (keys %foo) {  
    print "$key: ";  
    print "True\n"    if ($foo{$key});  
    print "Exists\n"  if (exists $foo{$key});  
    print "Defined\n" if (defined $foo{$key});  
}  
  
# what does this print?
```

Using Hashes

- Hashes are very powerful, and one of the Perl's greatest strengths.
- Thinking with hashes is part of Thinking in Perl.
- Hashes can sometimes replace complex loops or algorithms.
- Use a hash whenever you want to represent a set, a relation, a table, a structure, or a record.
- When faced with a new problem, it's a good idea to consider a hash first.
- Perl hashes are *cheap fast and powerful*.

real world uses

Cheating at Rock, Paper, Scissors



Cheating at Rock, Paper, Scissors

```
while (1) {  
    print "Your move? ";  
    chomp (my $user_move = <STDIN>);  
    print "Our move: ".our_move($user_move)."\n";  
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
```

```
while (1) {  
    print "Your move? ";  
    chomp (my $user_move = <STDIN>);  
    print "Our move: ".our_move($user_move)."\n";  
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock => 'paper',

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper    => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper    => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper    => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper    => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}

sub our_move {
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper    => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}

sub our_move {
    my $user_move = shift;
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}

sub our_move {
    my $user_move = shift;
    return $what_beats{$user_move};
}
```

Cheating at Rock, Paper, Scissors

```
use strict;
my %what_beats = ( rock    => 'paper',
                  paper   => 'scissors',
                  scissors => 'rock' );

while (1) {
    print "Your move? ";
    chomp (my $user_move = <STDIN>);
    print "Our move: ".our_move($user_move)."\n";
}

sub our_move {
    my $user_move = shift;
    return $what_beats{$user_move};
}
```

Counting

- We want to know who is hitting our website.
- Find the browser most people use
- Find out which users hit us the most
- Print out all users, in order of how long they spend on the website.

Counting -

- We want to know who is hitting our website.
- Find the browser most people use
- Find out which users hit us the most
- Print out all users, in order of how long they spend on the website.

weblogs

- Let's assume a very simple set of access logs

```
# IP Address - Date - Request - UserAgent  
127.0.0.1 - [10/Oct/2000:13:55:36 -0700] -  
"GET /apache_pb.gif HTTP/1.0" - "Mozilla/4.0  
(compatible; MSIE 7.0; Windows NT 5.1)"
```

Each section is separated by " - "

parsing weblogs



parsing weblogs

```
use strict;
```

parsing weblogs

```
use strict;  
my $filename = shift @ARGV;
```

parsing weblogs

```
use strict;  
my $filename = shift @ARGV;
```

parsing weblogs

```
use strict;  
my $filename = shift @ARGV;  
  
open my $fh, "<", $filename  
    or die "Failed reading $filename : $!";
```

parsing weblogs

```
use strict;
my $filename = shift @ARGV;

open my $fh, "<", $filename
  or die "Failed reading $filename : $!";
while (my $line = <$fh>) {
  chomp $line;
  my ($ip, $date, $request, $browser) =
    split(" - ", $line);

}
```

parsing weblogs

```
use strict;
my $filename = shift @ARGV;

open my $fh, "<", $filename
  or die "Failed reading $filename : $!";
while (my $line = <$fh>) {
  chomp $line;
  my ($ip, $date, $request, $browser) =
    split(" - ", $line);

}
}
```

parsing weblogs

```
use strict;
my $filename = shift @ARGV;

open my $fh, "<", $filename
  or die "Failed reading $filename : $!";
while (my $line = <$fh>) {
  chomp $line;
  my ($ip, $date, $request, $browser) =
    split(" - ", $line);

}
}
close $fh or die "Failed closing : $!";
```

parsing weblogs

```
use strict;
my $filename = shift @ARGV;
my (%ip_count, %browser_count);
open my $fh, "<", $filename
    or die "Failed reading $filename : $!";
while (my $line = <$fh>) {
    chomp $line;
    my ($ip, $date, $request, $browser) =
        split(" - ", $line);

}
}
close $fh or die "Failed closing : $!";
```

parsing weblogs

```
use strict;
my $filename = shift @ARGV;
my (%ip_count, %browser_count);
open my $fh, "<", $filename
    or die "Failed reading $filename : $!";
while (my $line = <$fh>) {
    chomp $line;
    my ($ip, $date, $request, $browser) =
        split(" - ", $line);
    $ip_count{ $ip } ++;
    $browser_count { $browser } ++;
}
}
close $fh or die "Failed closing : $!";
```

parsing weblogs

```
foreach my $ip ( keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

```
foreach my $browser ( keys %browser_count) {  
    print "$browser : $browser_count{$browser}\n";  
}
```

parsing weblogs

```
# sorting?
```

```
foreach my $ip (sort keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

```
foreach my $browser (sort keys %browser_count) {  
    print "$browser : $browser_count{$browser}\n";  
}
```

```
# what order will IPs be printed?
```

```
# what order will browsers be printed?
```

parsing weblogs

```
foreach my $ip (sort
    { $ip_count{$b} <=> $ip_count{$a} }
    keys %ip_count)
{
    print "$ip : $ip_count{$ip}\n";
}

foreach my $browser (sort
    { $browser_count{$b} <=> $browser_count{$a} }
    keys %browser_count) {
    print "$browser : $browser_count{$browser}\n";
}
```

sort

- We've used sort to sort a list like the list of keys from a hash
- Sometimes you need to sort some other way than alphabetical
- sort has an alternate syntax to let you control the sort

sort

- alphabetical sort

```
my @sorted = sort keys %hash;
```

```
# same thing, with the block syntax
```

```
my @sorted = sort { $a cmp $b } keys %hash;
```

sort with block

- The block syntax is

```
sort { BLOCK } @list
```

- The block should use the \$a and \$b globals to compare pieces of the list for sorting
- You tell perl how to sort any two elements (\$a and \$b)
- Perl takes care of the rest

sort with block

- The sort BLOCK has to return -1, 0, or 1
- -1 or less: \$a comes first
- 1 or more : \$b comes first
- 0 : \$a and \$b should stay in the same order
- \$a cmp \$b does this for alphabetical order
- \$a <=> \$b does this for numbers

sorting order

- Sorting in descending order can be done two ways:

```
foreach my $num (sort { $a <=> $b } @numbers) {  
    print "$num\n";  
}  
# descending order  
foreach my $num (reverse sort { $a <=> $b } @numbers) {  
    print "$num\n";  
}  
# same thing - slightly more efficient  
foreach my $num (sort { $b <=> $a } @numbers) {  
    print "$num\n";  
}
```

Sorting the Hash



Sorting the Hash

- Back to our hash.



Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.



Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.



Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

```
# how to sort on the count of hits to the IP ?
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

```
# how to sort on the count of hits to the IP ?  
# where's the number of hits per IP stored?
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

```
# how to sort on the count of hits to the IP ?  
# where's the number of hits per IP stored?
```

Sorting the Hash

- Back to our hash.
- We don't want to sort the IPs, like this.

```
foreach my $ip (sort { $a <=> $b } keys %ip_count) {  
    print "$ip : $ip_count{$ip}\n";  
}
```

```
# how to sort on the count of hits to the IP ?  
# where's the number of hits per IP stored?
```

```
# in $ip_count{ $a } or $ip_count{ $b }
```

Sorting the Hash



Sorting the Hash

- Use the number of hits per IP to sort



Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:



Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:



Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort  
    { $ip_count{$b} <=> $ip_count{$a} }  
    keys %ip_count)
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort  
    { $ip_count{$b} <=> $ip_count{$a} }  
    keys %ip_count)
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort
    { $ip_count{$b} <=> $ip_count{$a} }
    keys %ip_count) {
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort  
    { $ip_count{$b} <=> $ip_count{$a} }  
    keys %ip_count) {
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort
    { $ip_count{$b} <=> $ip_count{$a} }
    keys %ip_count) {
    print "$ip : $ip_count{$ip}\n";
}
```

Sorting the Hash

- Use the number of hits per IP to sort
- Reverse the ordering to get most hits to least hits:

```
foreach my $ip (sort
    { $ip_count{$b} <=> $ip_count{$a} }
    keys %ip_count) {
    print "$ip : $ip_count{$ip}\n";
}
```