Last time

Our 15 minutes of Unix focused on the different flavors of shells (text-based interfaces to the Unix operating system) and then a tour of file permissions (oh, glamour defined)

We then talked briefly about how certain shells not only let you type in commands but also assemble a series of commands into programs or shell scripts

We will not delve into much more detail about shell scripts except to say that such things exist; there is a fine tutorial at

http://steve-parker.org/sh/sh.s.shtml

A comment

As we look at Perl and particularly the interplay between Perl and Unix tools, you should start to think about programming choices

Our small shell script looped over files and executed some Unix commands; Perl can also get a list of files and execute the same commands

We will later see that R can do the same kinds of things; and what's worse, it's not just Unix-y sorts and cuts that become redundant, but even basic computations could be done in Perl or R or Java

So part of this class will be about deciding what tool to use; or at least to understand the tradeoffs when there's no obvious "right" answer

Last time

Understanding the functioning of a simple shell script also let us discuss shell variables (quantities like your $HOME and $PATH); we also had a look at how to make these and other programs executable

We then started on Perl; we covered some of the basics of the language (scalars and arrays) and even saw how our dear friends the regular expressions could be used
A comment

And yes, I suppose that means there is an aesthetics of computing and, in particular, statistical computing; it is useful to try to think about how we make computing choices

Are certain computations not possible in some languages? are they trivial in others?

What are the performance tradeoffs? Are some computations less efficient in a given language?

How “readable” or “naturally expressible” are some computations in the given language?

Today

We will start by discussing your assignments and use the answers to questions 2 and 3 to motivate where we're headed next

We will then do a little review of Perl and see how we can refine our assignment from Monday

Our assignment for the coming Monday will be to just get familiar with what we’ve learned so far, maybe work a bit more with the transcript data and update your Wiki’s for your projects

Next week we will start on the Dartmouth Wireless Data

Back to Judge Roberts...

A step backwards

Why these transcripts?

They are not what one might consider “regular” data in the sense that they are not nicely formatted

They let us control what we choose to extract; we can bring questions and suggest analyses of these (socially important) data, we are in charge of what is quantified

The data are just big enough that you have to rely on an exploratory style of computing to decide if you’ve got the data you really want

Recall the data...
SPECTER: Good afternoon, ladies and gentlemen.

We begin these hearings on the confirmation of Judge John Roberts to be chief justice of the United States with first the introduction by Judge Roberts of his beautiful family, and then a few administrative housekeeping details before we begin the opening statements, which will be 10 minutes in length, by each senator.

At the conclusion of the opening statements, we will then turn to the introductions by Judge Lugar, Judge Warner — actually, Senator Lugar, Senator Warner and Senator Bayh, and then the administration of the oath to Judge Roberts and his opening statement.

So, Judge Roberts, if you would at this time introduce your family we would appreciate it.

ROBERTS: (OFF-MIKE) Peggy Roberts and Barbara Burke. Barbara's husband Tim Burke is also here.
My uncle, Richard Podrasky (ph). Representing the cousins, my cousin, Jeannie Podrasky (ph).
My wife, Jane is right here, front and center, with our daughter, Josephine and our son, Jack. You'll see she has a very tight grasp on Jack.

(LAUGHTER)

SPECTER: Thank you very much, Judge Roberts.

Judge Roberts had expressed his appreciation to have the introductions early. He said the maximum time of the children's staying power was five minutes. And that is certainly understandable.

Thank you for doing that, Judge Roberts.

And now before beginning the opening statements, let me yield to my distinguished ranking member, Senator Leahy.

LEAHY: Well, Mr. Chairman, I want to thank you for all the consultations. I think we have had each other's home phones on speed dial, we've talked to each other so often. And I have every confidence our chairman will conduct a fair and thorough hearing. You know, less than a quarter of those of us currently serving in the Senate have exercised the Senate's advice-and-consent responsibility in connection with a nomination to be chief justice of the United States. I think only 23 senators have actually been involved in that.

Your assignment

1. Try the code snippets in this lecture and produce the files of words for each senator; write a shell script that will take each file and produce a sorted list of words (most to least frequently used). Turn in the top 15 and the bottom 15 words for senators Biden, Specter and Grassley and for Judge Roberts.

2. Comment on whether this program is actually capturing all the words used by these senators or something more; if we are not selecting just words, describe what needs to be added (not literal code, just concept).

3. Create a regular expression that describes commentary added by the transcriber and create a list of the unique additions.

A short shell script

Last time we created a Perl program that would direct different "words" (as defined by sequences of non-space characters separated by spaces) into separate files for each senator.

We also saw an example of a shell script that applied a chain of Unix commands to a number of files; here we change the shell script slightly so we can process the output from our Perl program.

```bash
#!/bin/sh
for i in `ls *.txt`
do
  sort $i | uniq -c | sort -rn > $i.cnt
done
```

The "backticks" tell the shell to expand the output of the command; here we use it to provide values for our variable i in the loop.
Warning!

Our approach to this problem (creating separate files and then using a shell script to process the files) is not the most efficient way to tackle our problem; ultimately, we want to compare word distributions for each senator and this approach is too complex.

We divided things up so you don’t have to know a lot about Perl and you get a chance to see a shell script in action.

Next time we will see a MUCH more efficient way to carry out the computation.

To be more precise...

Are all the entries words?

Do separate entries represent different words?

Do all the words say something about content?
### Dealing with case

In Kennedy's top 200 we see entries for “You” and “you”; both of these should be grouped into the same count.

Perl has built-in functions to deal with the case of characters:

```perl
use strict;
use warnings;

my $string = "Hello World!";

# uc() converts case
my $upper = uc($string);   # uppercase: "HELLO WORLD!
my $lower = lc($string);   # lowercase: "hello world!

# lc() converts case
my $lowercase = lc($string);  # lowercase: "hello world!
my $uppercase = uc($string);  # uppercase: "HELLO WORLD!
```

There are also functions to just alter the first character:

```perl
use strict;
use warnings;

my $string = "Hello World!";

# lcfirst() converts case
my $lowercased = lcfirst($string);  # "Hello World!
my $lowercased = lcfirst($string);  # "hello World!
```

### We can add a simple transformation to our Perl program

```perl
#!/usr/bin/perl

while($str = <>){
    chomp $str;
    @words = split " ",$str;
    if($words[0] =~ m/^[A-Z]+:/){
        print DF "$word
    }
    foreach $word (@words){
        shift @words;
        open(DF,">> $words[0].txt") or die "Cannot open
        print DF "$word
    }
    close(DF);
}
```

#### Kennedy's top 200 words

<table>
<thead>
<tr>
<th>14 You</th>
<th>9 make</th>
<th>7 still</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 American</td>
<td>9 great</td>
<td>7 some</td>
</tr>
<tr>
<td>13 which</td>
<td>9 go</td>
<td>7 society.</td>
</tr>
<tr>
<td>13 there</td>
<td>9 go</td>
<td>7 signed</td>
</tr>
<tr>
<td>13 most</td>
<td>9 extra-</td>
<td>7 racial</td>
</tr>
<tr>
<td>13 many</td>
<td>9 don't</td>
<td>7 quote.</td>
</tr>
<tr>
<td>9 could</td>
<td>9 constitutional</td>
<td>7 question.</td>
</tr>
<tr>
<td>9 basis</td>
<td>9 Justice</td>
<td>7 only</td>
</tr>
<tr>
<td>9 aisle</td>
<td>9 Justice</td>
<td>7 kind</td>
</tr>
<tr>
<td>9 Americans</td>
<td>8 year</td>
<td>7 included</td>
</tr>
<tr>
<td>8 year</td>
<td>8 passed</td>
<td>7 impact</td>
</tr>
<tr>
<td>7 year</td>
<td>7 opportunity</td>
<td>7 citizens.</td>
</tr>
<tr>
<td>6 year</td>
<td>6 opportunity</td>
<td>7 children</td>
</tr>
<tr>
<td>6 year</td>
<td>6 opportunity</td>
<td>7 Zimmer</td>
</tr>
<tr>
<td>5 year</td>
<td>5 opportunity</td>
<td>7 So</td>
</tr>
<tr>
<td>4 year</td>
<td>4 opportunity</td>
<td>7 Section</td>
</tr>
<tr>
<td>3 year</td>
<td>3 opportunity</td>
<td>7 Department</td>
</tr>
<tr>
<td>2 year</td>
<td>2 opportunity</td>
<td>7 Act.</td>
</tr>
<tr>
<td>1 year</td>
<td>1 opportunity</td>
<td>7 Act.</td>
</tr>
</tbody>
</table>

---

### Dealing with case

We can add a simple transformation to our Perl program:

```bash
#!/usr/bin/perl

while($str = <>){
    chomp $str;
    @words = split " ",$str;
    if($words[0] =~ m/^[A-Z]+:/){
        close(DF);
        open(DF,">> $words[0].txt") or die "Cannot open
        print DF "$word
    }
    foreach $word (@words){
        shift @words;
        # removes first element |
        # of an array
    }
    close(DF);
}
```

#### Kennedy's top 200 words

<table>
<thead>
<tr>
<th>14 You</th>
<th>9 make</th>
<th>7 still</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 American</td>
<td>9 great</td>
<td>7 some</td>
</tr>
<tr>
<td>13 which</td>
<td>9 go</td>
<td>7 society</td>
</tr>
<tr>
<td>13 there</td>
<td>9 go</td>
<td>7 signed</td>
</tr>
<tr>
<td>13 most</td>
<td>9 extraordinary</td>
<td>7 racial</td>
</tr>
<tr>
<td>13 many</td>
<td>9 don't</td>
<td>7 quote</td>
</tr>
<tr>
<td>9 could</td>
<td>9 constitutional</td>
<td>7 question</td>
</tr>
<tr>
<td>9 basis</td>
<td>9 Justice</td>
<td>7 only</td>
</tr>
<tr>
<td>9 aisle</td>
<td>9 Justice</td>
<td>7 kind</td>
</tr>
<tr>
<td>9 Americans</td>
<td>8 year</td>
<td>7 included</td>
</tr>
<tr>
<td>8 year</td>
<td>8 passed</td>
<td>7 impact</td>
</tr>
<tr>
<td>7 year</td>
<td>7 opportunity</td>
<td>7 citizens</td>
</tr>
<tr>
<td>6 year</td>
<td>6 opportunity</td>
<td>7 children</td>
</tr>
<tr>
<td>5 year</td>
<td>5 opportunity</td>
<td>7 Zimmer</td>
</tr>
<tr>
<td>4 year</td>
<td>4 opportunity</td>
<td>7 So</td>
</tr>
<tr>
<td>3 year</td>
<td>3 opportunity</td>
<td>7 Section</td>
</tr>
<tr>
<td>2 year</td>
<td>2 opportunity</td>
<td>7 Department</td>
</tr>
<tr>
<td>1 year</td>
<td>1 opportunity</td>
<td>7 Act</td>
</tr>
<tr>
<td>1 year</td>
<td>1 opportunity</td>
<td>7 Act</td>
</tr>
</tbody>
</table>

---

### Dealing with case

Perl has built-in functions to deal with the case of characters:

```perl
use strict;
use warnings;

my $string = "Hello World!";

# uc() converts case
my $upper = uc($string);   # uppercase: "HELLO WORLD!
my $lower = lc($string);   # lowercase: "hello world!

# lc() converts case
my $lowercased = lcfirst($string);  # "hello World!
my $lowercased = lcfirst($string);  # "hello World!
```

There are also functions to just alter the first character:

```perl
use strict;
use warnings;

my $string = "Hello World!";

# lcfirst() converts case
my $lowercased = lcfirst($string);  # "hello World!
my $lowercased = lcfirst($string);  # "hello World!
```

### We can add a simple transformation to our Perl program:

```bash
#!/usr/bin/perl

while($str = <>){
    chomp $str;
    @words = split " ",$str;
    if($words[0] =~ m/^[A-Z]+:/){
        close(DF);
        open(DF,">> $words[0].txt") or die "Cannot open
        print DF "$word
    }
    foreach $word (@words){
        shift @words;
        # removes first element |
        # of an array
    }
    close(DF);
}
```

#### Kennedy's top 200 words

<table>
<thead>
<tr>
<th>14 You</th>
<th>9 make</th>
<th>7 still</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 American</td>
<td>9 great</td>
<td>7 some</td>
</tr>
<tr>
<td>13 which</td>
<td>9 go</td>
<td>7 society</td>
</tr>
<tr>
<td>13 there</td>
<td>9 go</td>
<td>7 signed</td>
</tr>
<tr>
<td>13 most</td>
<td>9 extraordinary</td>
<td>7 racial</td>
</tr>
<tr>
<td>13 many</td>
<td>9 don't</td>
<td>7 quote</td>
</tr>
<tr>
<td>9 could</td>
<td>9 constitutional</td>
<td>7 question</td>
</tr>
<tr>
<td>9 basis</td>
<td>9 Justice</td>
<td>7 only</td>
</tr>
<tr>
<td>9 aisle</td>
<td>9 Justice</td>
<td>7 kind</td>
</tr>
<tr>
<td>9 Americans</td>
<td>8 year</td>
<td>7 included</td>
</tr>
<tr>
<td>8 year</td>
<td>8 passed</td>
<td>7 impact</td>
</tr>
<tr>
<td>7 year</td>
<td>7 opportunity</td>
<td>7 citizens</td>
</tr>
<tr>
<td>6 year</td>
<td>6 opportunity</td>
<td>7 children</td>
</tr>
<tr>
<td>5 year</td>
<td>5 opportunity</td>
<td>7 Zimmer</td>
</tr>
<tr>
<td>4 year</td>
<td>4 opportunity</td>
<td>7 So</td>
</tr>
<tr>
<td>3 year</td>
<td>3 opportunity</td>
<td>7 Section</td>
</tr>
<tr>
<td>2 year</td>
<td>2 opportunity</td>
<td>7 Department</td>
</tr>
<tr>
<td>1 year</td>
<td>1 opportunity</td>
<td>7 Act</td>
</tr>
<tr>
<td>1 year</td>
<td>1 opportunity</td>
<td>7 Act</td>
</tr>
</tbody>
</table>
Words plus a little more

Notice that by splitting on spaces, we've picked out words and trailing punctuation

Before stripping this out, it's interesting to see who uses some of these expressions

Consider sentences starting with "Well,"; most were uttered by Roberts (does that make sense?)

To remove the punctuation we need to learn a little more Perl

% egrep 'Well,' *.cnt | sort -k2 -rn

ROBERTS:.txt.cnt: 294 Well,
KENNEDY:.txt.cnt: 16 Well,
CORNYN:.txt.cnt: 15 Well,
SPECTER:.txt.cnt: 13 Well,
HATCH:.txt.cnt: 12 Well,
BIDEN:.txt.cnt: 11 Well,
SESSIONS:.txt.cnt: 9 Well,
LEAHY:.txt.cnt: 8 Well,
GRAMA:.txt.cnt: 8 Well,
FEINSTEIN:.txt.cnt: 5 Well,
KYL:.txt.cnt: 4 Well,
GRASSLEY:.txt.cnt: 4 Well,
DURBIN:.txt.cnt: 4 Well,
BROWNBACK:.txt.cnt: 3 Well,
SCHUMER:.txt.cnt: 2 Well,
ROBERTS:.txt.cnt: 2 Well,
KOHl:.txt.cnt: 2 Well,
FEINGOLD:.txt.cnt: 1 Well,
DEWINE:.txt.cnt: 1 Well,

Finding and acting on patterns

Now we want to remove non-words and trailing punctuation; this will require us to look at how to specify the pattern we want and make some change

In our program, the =~ is called a binding operator; it tells Perl to run the regular expression over the indicated string; it is commonly used in three kinds of constructions

m/.../ allows us to test for the occurrence of a pattern within a string

s/.../.../ allows you to change your text when a match is found

tr/.../.../ makes translations from one set of characters to another

Substitutions

The binding operator returns the number of substitutions made; in addition to our usual regular expressions, Perl adds a few meta-characters to make things easier

$nominee =~ s/Roberts/Miers/; # "The Senate confirms Roberts" to # "The Senate confirms Miers"
$avoidance =~ s/\(\W\)\[Nn\]o(\W)/\1NOTE\2/; # "And Brennan (ph) said, No, no" to # "And Brennan (NOTE) said, No, no"
$protest =~ s/\(\W\)\[Nn\]o(\W)/\1NO\2/; # He said, No, no, no" to # "He said, NO, NO, NO"
$protest =~ s/\(\W\)\[Nn\]o(\W)/\1NO\2/g; # He said, No, no, no" to # He said, NO, NO, NO"
$num = $protest =~ s/\(\W\)(\W+)/\1\2/g; # $num = 3 substitutions

Matching

You can modify the kind of match you want by adding a suffix like m//i (for case-insensitive matching)

We did not mention these earlier, but Perl and egrep can use make use of special character classes

\w # a word character [a-zA-Z0-9-]
\W # a non-word character
\d # a digit [0-9]
\D # a nondigit
\s # whitespace character [ \t\n\r\f]
\S # a non-whitespace character
Substitutions

\(s///i\) allows for case-insensitive matching

Similarly, \(s///\) will just substitute the first pattern it finds; if you want to make a change each time it finds a pattern, you would add "g" to the end

In addition to the extra meta-characters, Perl also gives you the same ability to make back-references; the special characters \1, \2 and so on still represent portions of matched text and now can be used in the substitution

So to drop the commas from our words we might do

\[
\begin{align*}
\text{foreach } \$\text{word} (@\text{words}){ & \\
\text{\$word =~ s/\$///; } & & & \\
\text{or for the periods } & \\
\text{\$word =~ s/\.$//; } & & & \\
\text{or for any non-word character } & \\
\text{\$word =~ s/\W$//; } & & & \\
\end{align*}
\]

Does that do what we want?
In particular, consider what happens when someone is quoting another source; how would we find these in the text and what action should we take?

Translations

\texttt{tr///} translates one set of characters into another

\[
\begin{align*}
\text{\$str =~ tr/A-Z/a-z/; } & & \text{# to lower case } & \\
\text{\$str =~ /\{A,E,I,O,U\}/[1,2,3,4,5]/ } & & \text{# not sure what to call this } & \\
\end{align*}
\]

Also...

We can use a simple substitution to make the program a little nicer; for example, our filenames are KENNEDY:.txt

It's pretty easy to strip off the "." and have a filename of the form KENNEDY.txt

Or, perhaps more simply, we could just do \texttt{chop(\$name)} instead of \texttt{\$name=-s://$//} (why?)

Also...

We can use a simple substitution to make the program a little nicer; for example, our filenames are KENNEDY:.txt

It's pretty easy to strip off the "." and have a filename of the form KENNEDY.txt

Or, perhaps more simply, we could just do \texttt{chop(\$name)} instead of \texttt{\$name=-s://$//} (why?)
3. Can we find a regular expression to capture the transcribers additions?

Based on a casual browsing of the first few lines of the transcripts, we might consider a regular expression that looks for strings of upper-case letters enclosed in parentheses:

```
% cat *.txt | egrep '\([A-Z]+\)' | sort | uniq
(APPLAUSE)
(CROSSTALK)
(LAUGHTER)
(RECESS)
```

We can then use the “-v” option in `grep` or `egrep` to ignore lines that have this pattern; the following pipe pulls out those lines that include an open parenthesis and leaves out all those that match the pattern we identified above.

```
% cat *.txt | egrep '\(' | egrep -v '\([A-Za-z\-]+\)' | sort | uniq
```

What's left over?

```
% cat *.txt | egrep '\\([A-Z]+\)' | sort | uniq
(APPLAUSE)
(CROSSTALK)
(LAUGHTER)
(RECESS)
```

Since some of the added material involves both lower-case letters and a hyphen, we can extend our definition of the pattern slightly and see what's left over; again, we are using the first `egrep` to identify lines with an open parenthesis and the second to ignore lines that have the pattern we're working on:

```
% cat *.txt | egrep '\(' | egrep -v '\([A-Za-z\-]+\)' | sort | uniq
```

Here we do the same thing, but this time we consider strings that have a right parenthesis; these two lists are the same except for two exceptions.

Notice that we've been working on our processed files; the data we are passing to `egrep` comes has been divided based on spaces already thanks to our Perl program. Let's look back in the original transcript to find `SPEAKING`:

```
% egrep SPEAKING transcript*.txt
transcript3.txt:And I'm always impressed with somebody with that facility. There is a Latin phrase. And this is not a (inaudible). I'll translate it: (SPEAKING IN LATIN). He who acts through another acts for himself. And that's not the case in Herrera?
```

Regrouping

If we want to delete the transcriber, we might consider a global substitution that would remove the parenthetical comments as we described them:

```
% sed 's/\([A-Z]+\)//g'
```

At this point, we've accomplished what we wanted pedagogically; you see how to match patterns in Perl and how to act on those patterns (in truth, there's a lot we can do to totally clean up these data -- consider quoted expressions like 'unamerican', for example)

We will return to cleaning up the transcripts shortly; but for now let's revisit some of the basics of Perl as a language.
Scalars and lists

Scalars are “single” things, like a number or a string

```perl
$value = 1.0;
$string = "Hi Mom";
```

Lists are groups of things; they are stored in arrays

```perl
@family = ("father","mother","child");
$family[3] = "grandfather";
@prices = ("cheap", "modest", "expensive");
```

Scalar data: Numbers

A *literal* is the way a value is represented in the source code of a Perl program

We can refer to the following kinds of numbers (*numeric literals*) in a Perl program

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.45</td>
</tr>
<tr>
<td>256.000</td>
</tr>
<tr>
<td>7.3e45</td>
</tr>
<tr>
<td>-1.2E-3</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>2004</td>
</tr>
<tr>
<td>-200</td>
</tr>
</tbody>
</table>

The website below describes the internal representation of numbers in Perl 5.8; for the most part Perl will convert data to the format it wants behind the scenes

http://search.cpan.org/~jhi/perl-5.8.0/pod/perlnumber.pod#Numeric_operators_and_numeric_conversions

Scalar data: Strings

You can include strings in your Perl code in a couple of ways using either single or double quotes

Double-quoted strings

Any number of characters enclosed in double quotes.

The \ is now endowed with the power to specify certain control characters; \n is a newline, \t is a tab. As before, you escape \n and " with a \n
```perl
"The war\n is a waste."
"miers’ age is 60"
"Roberts\tMiers"
"On 9\11"
"Did you blame \"Bush\"?"
```

Double quoted strings allow for something called interpolation that we will describe shortly

Single-quoted strings

Any number of characters enclosed in single quotes.

Any character but a ’ and a \ stands for itself; to get these escape them with a \n
```perl
‘The war is a waste.’
‘miers’ age is 60’
‘On 9\11’
```
Scalar data: Arithmetic operators

Most of these do exactly what you would expect

\$x + \$y   # addition
\$x * \$y   # multiplication
\$a % \$b   # modulus (remainder of \$a divided by \$b)
\$a ** \$b  # exponentiation
\$a += 3   # shorthand for \$a = \$a + 3
\$a *= 4   # shorthand for \$a = \$a * 3
\$b = \$a++;    # assign \$a to \$b then increment \$a
\$b = ++\$a;    # assign incremented value to \$b

Scalar data: String operators

Most of these do more or less what you would expect

\$x = "hi";
\$y = "ho";
\$x . \$y       # concatenation "hiho"
\$x x 3        # repetition "hihihi"

\$a = "unqualified";
"Miers is \$a."     # interpolation
'Miers is '. \$a    # another way to do that

$who = "Miers";
$who . "'ll lose.\n"    # hmm
"${who}'ll lose.\n" # Using {}'s to clear things up

Scalar data: String operators

Some more useful operators for string scalars; these identify the position of text within a string and to subset or replace the text at a specific position

\$big = "Kerry and Bush are in a dead heat";
\$where = index($big,"Bush");        # returns 10
\$where = index($big,"e",2);         # returns 17
\$where = rindex($big,"e");         # last occurrence
\$where = rindex($big,"e");         # returns 30

\$winner = substr($big, 10,4)        # Bush

\$new_string = substr($big,0,5,"Nader");
substr($big,0,5) = "Nader"     # Nader and Bush...

Lists: Array operators

Several operators help manipulate lists of items

push, pop, shift, sort, reverse

@array = 1 .. 10;
$digit = pop(@array); # removes 10
pop @array;           # removes 9
push (@array, 11);    # adds 11 to the end
push @array, 5 .. 20;

@prices = ("cheap", "modest", "expensive");
@prices = sort @prices; # now cheap, expensive, modest
@swapped = reverse @prices; # expensive, modest, cheap
Operator precedence

For arithmetic operations, multiplication before addition, exponentiation before multiplication, etc.

But things can get more interesting when we mix different kinds of operators; see the full list below

http://www.perldoc.com/perl5.8.4/pod/perlop.html#Operator-Precedence-and-Associativity

Operator context

Various operators expect certain kinds of values as parameters; we have seen that some operators act on arrays while others act on scalars.

We say that these operators “provide” a context (scalar, string, numeric, Boolean)

Perl will automatically convert the data into the form required by the current context (gulp)

    $counting = "123";
    print ($counting +1) . "\n";  # gives 124

    35.0 eq 35   # true

    $number = 5 + @prices;   # gives 5+3 = 8

And here’s where Perl gets a bad rap...