Perl

Scalar variables ($) are for single values.

Perl is a loosly typed language. You don't have to declare variable types. Languages like C/C++ are strongly typed.

Characters such as @, $, % are called metacharacters.

** = exponentiation

You can do stuff like: print "Two \$nums are ", $num *2, " and adding one to \$var maker ", $var++, "\n";

postIncrementing - this incremented value will not actually show up in this print statement because it is printed, then incremented. In order to achieve the desired effect here, we must preincrement instead: ++$var

"print" is a list operator, which is exemplified above.

The area between two bracer is called a block.

Subroutines are called with an ampersand, such as &print_results;

IF Formatting: if ($day eq "sunday") { block }

Perl Truths

1.) Any string is true except for "" and "0".
2.) Any number is true except for 0. This includes negative numbers.
3.) Any undefined variable is false. An undefined variable is one which doesn't have a value, i.e., has not been assigned to.

if($something) ← is a truth test.

Using single quotes with "print" does not allow for interpolation.
i.e., print '$x - $y is ', $x - $y, ", alright?\n";
- is -
$x - $y is 3, alright?

Comparison Operators:
== to compare (numbers) < <= !=
eq to compare (strings) lt le ne

$name = 'Mark';
$goodguy = 'Tony';
if ($name == $goodguy)
↑ will yield a "yes" result because the strings are being evaluated numerically (they are the same length).

if ($x = 10) will resolve as a truth test and will be a "yes", because you are assigning a value to $x right there. Gotta use the double equals if you want a comparison.

elsif

You can do:

```
print "Something\n" if $age > $max;
print "Something\n" unless $age < $max;
```

Perl has no case statement.

chop() will chop off the last character, and return it.
chomp() will chomp off the last character if it is a newline.

To get input, use <STDIN>, for example: $variable = <STDIN>;

Arrays:

@names = ("Emily", "Luke", "Sean", "Molly", "Jesse");
↑ Quotes are not required.

Call elements as $names[x]
Call the whole thing as @names
The whole thing is known as being in list context.

print @names will print the elements all in a row.
print "@names" will print the elements with a space between them.
↳ When a list is placed inside doublequotes, it is space delimited when interpolated.

Refering to more than one, but not all elements of an array is known as a slice.

It is possible to have both $variable and @variable
↳ These two variables are in different namespaces.

scalar(@names) returns the number of elements in the array.
For example: $total = scalar(@names);
print "$total";

So to get the index number of the last element in an array, you can do: $last = scalar(@names) - 1;
-or-
There is also this method: $#names

Now, to print the last element, you can do:
print "$names[$#names]";

-or-

print "@names[-1]";

Can call any number of elements in an array: @names[0,1]
...or a range: @names[0..2]
...or an individual element like this: $names[$x-1]
...or multiples like this: @names[$x-2,$x]
...or combos: @names[0..2,4]

".." is called the range operator.

for $x (0..$number) ← Gives $x the value of 0, then increments it by 1 until it is equal to $number.

```
foreach $person (@names)    ← Goes through (iterates) each element
{                             of @names, and assigns each element
    print "$person";          in turn to $person. (Can use just
}                             "for" for this.)
```

$_ is the default input and pattern searching variable. So you can do:

```
foreach (@names)
{
    print "$_";
}
```

...or not specifying anything means $_:

```
foreach (@names)
{
    print;
}
```

You can use "last" to exit a loop:

```
foreach $person (@names)
{
        print "$person\n";
        last if $person =~ /x/;
}
```

To add something to an array:

```
push (@names, $x);
push (@names, $x, 10, @cities[2..4]);
```

To remove something from an array:

```
pop (@names);
```

To add something to the beginning of an array:

```
unshift (@names, $x);
```

To remove something from the beginning of an array:

```
shift (@names);
```

Removing/Adding array elements:

```
splice (@names, 1, 2);
```

(@names: array; 1: offset in; 2: how many to remove)

```
splice (@names, 1, 0, @cities[1..3]);
```

(@names: array; 1: offset in; 0: how many to remove; @cities[1..3]: what to add in that spot)

"if defined $var" works.

A variable can be deleted by doing:

```
undef $var
```

Regex:

/ regex searches go between slashes /
putting an "i" after the end slash indicates case insensitivity
ex: /something/i

@names = qw(Luke Emily Linda Miguel);
↳ This means quote words - split the list by words.

/[KC]arl/
/[KC]arl[^sa]/ → ^ means to negate, and must be the first character in a [ ] to work.

/[A-K]arl/
/[A-KMp]arl/
/[\-K]arl/ → matches "Karl" or "-arl"
/a$/ → everything ending in "a"
/^a/ → everything starting with "a"
/(something)/; → When using parenthesis, the first match is put in $1, the second in $2, etc.
. → Means any character
* → Means zero or more of the previous character.

By default, .* in Perl causes "Greedy Matching". Use .*? to enable stingy matching.

- Greedy Matching means that when the initial match is found, the search jumps to the end of the string and works backwards from there to find a match.
- Example:

$_ = 'HTML munging time is here again !.';
/(.*)<\/i>/i; finds munging time is here again
/(.*?)<\/i>/i; finds munging

+ → Means one or more of the previous character.

You can use $1 in the same regex it was found in by calling it as \1.
/<(.*?)>(.*?)<\/\1>/i; ←
— Finds whatever is in whatever the first HTML tag is.

You can use a different delimiter besides / if you want, by doing this:
m#http://some.site.com/page/#;
↳ new delimiter

-or-

\Q escapes everything until \E or the delimiter.

Substitution is like "sed":

s/us/them/;	→ replaces first instance
s/us/them/g;	→ matches globally
\w	→ means "word", equivalent to a-zA-Z_0-9
\W	→ same as above, but negating instead (same as [^\w])
\b	→ looks for a word boundary
\d	→ means anything that is a digit
\D	→ negated version of \d

x repeats things, like if $input=3, $input x3 will be "333".

If you do $match=/some regex search/; and then do print "$match";, it will print the result as a truth test. If you want to actually print the result of the regex, you must do:
($match)=/some regex search/

(parenthesis) store stuff in a list context:

```
($word1, $word2) = /regex search/;
```

↑If you don't know how many matches there will be, you can catch them in an array!:

```
@words = /regex search/;
```

A regex returns "true" each time it matches, hence:

```
while (/<.*?>(.*?)<\/\1>/)
{
        print "Found the HTML tag $1, which has $2 inside.\n";
}
```

...will loop forever on the first tag it finds, while:

```
while (/<.*?>(.*?)<\/\1>/g)
{
        print "Found the HTML tag $1, which has $2 inside.\n";
}
```

...will end because /g forces the match to move on until it fails.

Parenthesis can also be used like this: /Pe(tra|ter|nny)/;

- although, if you don't need to store the result(s) in $1, $2, etc, then this creates unnecessary overhead. To avoid storing parenthisized results, use "?:":

```
/Pe(?:tra|ter|nny)/;
```

/z{5}/ → {} specifies how many of the preceding character to match.

/z{3,}/ → At least 3 of the previous character.

/z{1,3}/ → 1 to 3 "z".

/z{4,8}/ → 4 to 8 "z".

/z{5}?/ → ? forces a minimal match, i.e., /\w{5,}?/

vmware

↑
will give only the first five characters of the word.

$_ = "I am sleepy...snore...DING! Wake up!";
/snore/

$& → Prints the match: snore
$` → Prints the prematch: I am sleepy...
$' → Prints the postmatch: ...DING! Wake up!

- Note that using these slows down a program. To avoid this, try using parenthesis instead.

s/ / /e; → "e" means that the right hand side will be evaluated as an expression. You can do math in there and shit.

" /eeee; → For every additional "e" added, Perl will do another evaluation pass on the right hand side.

? → In regex, means that the preceding character/item is optional.

$ → In regex, means end of line.

End of Regex.

\t → tab
\f → formfeed
\r → carriage return
\n → new line
\s → any whitespace (also available in negated form: \S)

split /point/, source, number of splits.

Example:

```
$_ = "Piper:PA-28:Archer:00-ROB:Antwerp";

@details = split /:/, $_;

foreach (@details)
{
    print "$_\n";
}
```

join 'glue operator', things to join;

Example:

```
$cool = join ' ', $var1, var2, var3;
```

last will take you out of a loop. You can also use it to break out to a label.

srand initializes the random number generator.

rand generates a random number between 0 and 1, or 0 and a given number.

. can be used as a concatenator

Examples:

```
$new = $x.$y;
$new.= $z;
```

Files

Reading: open STUFF, $stuff
↳ filehandle ↳ file location

You could just do:

```
$STUFF = "location of file";
open STUFF
```

It's good to do this:

```
open STUFF, $stuff or die "Can't open $stuff for read: $!";
```

↳ Perl's error (which may or may not exist/be helpful).

die obviously kills your program.

< > is a line input operator. It reads from the beginning of the file up until and including the first newline (\n). The read data goes into $_. That's how your <STDIN> works as well :)

If you put it in a loop, it'll keep reading lines until there is no more data:

```
open STUFF;
while (<STUFF>)
{
    print "Line $. is: $_";
}
```

$. is a special variable which holds the current line number.

A shorter version of the above while loop:

```
print "Line $. is $_" while (<STUFF>);
```

Writing: open STUFF, ">$stuff";
print STUFF "whatever";

Appending: open STUFF, ">>$stuff";

Closing: close STUFF;

Filehandles do not have to be capitalized, but it is sort of convention to do so, as they won't be confused with variables, or double-declared with a variable, since Perl is case sensitive.

scalar(localtime) has the local date, and time, and date...

Print out a bunch of files: print <>;
(and just input the files at the command line)

$^I is a special variable, the Inplace Edit variable. When it has a value:

1.) The name of the file to be in-place edited is taken from the first element of @ARGV and renamed to its existing name plus the value of $^I
2.) The file is read by the diamond operator (<>), placing a line at a time in $_.
3.) A new filehandle, ARGVOUT, is opened on the original file.
4.) "print" prints automatically to ARGVOUT, not the normal STDOUT

- If no backup is desired, assign a null string to $^I.

tr - Transliteration operator. It is more efficient than regex for changing single characters.
Ex: tr/A-Z/a-z/; #All uppercase become lowercase.

$/ - A special variable called the Default Input Record Seperator. <> will read up until whatever $/ is set to. By default, it is set to newline.

You can print multiple lines/things in a row like this:

```
print <<PRT;
Hey, how are you?
I'm fine

Well dog-gone, this is sorta like <pre> in HTML!
PRT
```

↑ (It doesn't have to be "PRT", it can be anything.)
- The tutorial calls this a "HERE document".

Change Perl's working directory: chdir location;

-T - Tests for text file(s).

Open a directory: opendir HANDLE, $source

You can get the first command line parameter by shifting it out of the @ARGV array: $file = shift

readdir - reads a directory; requires a filehandle.

next can be used to omit things in loops.
Ex:

```
while ($file = readdir DIR)
{
        next if $file =~ /^\./;  # to skip reporting "." and ".." in
                                 # the dir listing. Or any .files
        print "Found a file: $file\n";
}
```

closedir to close a directory: closedir HANDLE;

Associative Arrays (Hashes)

They are like arrays, but are key-based instead of number-based:
(Index-based)

@myarray

Index No.	Value
0	The Netherlands
1	Belgium
2	Germany
3	Monaco
4	Spain

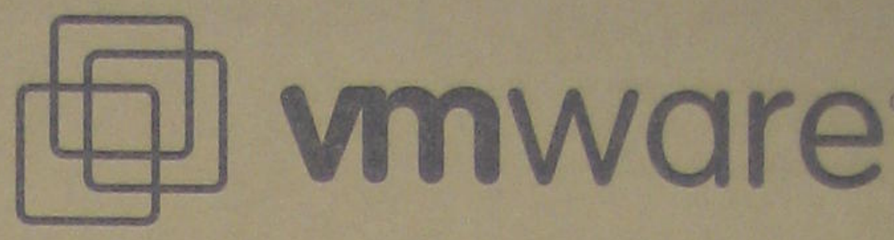
%myhash

Key	Value
NL	The Netherlands
BE	Belgium
DE	Germany
MC	Monaco
ES	Spain

```
print "$myarray[1]";
```

```
print "$myhash{'BE'}";
```

- They are good if you want to be able to look things up by keyword.
- They are generally faster than indexed arrays.
- Their elements do not stay in order. Perl reorders them for quick access.
- If you use a key twice, the second second value overwrites the first.
- Duplicate values are not a problem.



Assigning: $countries{PT}='Portugal';
Deleting: delete $countries{NL};

Show all keys: print keys %countries;
Show all values: print values %countries;
A slice of hash: print @countries{'NL' 'BE'};
Show number of elements: print scalar(keys %countries);
Check if a key exists: print "It's there!\n" if exists $countries{'NL'};

Two ways to iterate hashes:

```
foreach (keys %countries)
{
        print "The key $_ contains $countries{$_}\n";
}

while (($code, $name)= each %countries)
{
        print "The key $code contains $name\n";
}
```

(The second one is faster)

Sorting
So simple:

```
foreach (sort keys %countries)
{
        print "The key $_ contains $countries{$_}\n";
}
```

Reverse sorting: just add 'reverse' before 'sort'!

Sometimes 'reverse' can be used on its own, i.e., print reverse @letters;

You can feed Perl list operators to each other, and do stuff like:

```
print "Enter string to be reversed: ";
print join "", reverse split //, <STDIN>;
```

↑ Emphasis on "You can", for purposes of readability, you probably should not.

The way the sort function works is that it compares two variables, $a and $b (specifically those), giving one of three results:

- 1 if $a is greater than $b.
- -1 if $b is greater than $a.
- 0 if $a and $b are equal.

- This means that you can create your own sort routines, and feed them to sort. For example:

```
foreach (sort supersort keys %countries)
{
    print "$_ is the country code for $countries{$_}\n";
}
```

```
sub supersort
{
    if ($a > $b)
    {
        return 1;
    }
    elsif ($a < $b)
```

```
    {
        return -1;
    }
    else
    {
        return 0;
    }
}
```

<=> - The "spaceship" operator. This does exactly what the above supersort subroutine does (returns 1, -1, or 0 depending on the comparison of two given values.

```
foreach (sort { $a <=> $b } keys %countries)
{
    print "$_ is the country code for $countries{$_}\
    n";
}
```

↑ The part in {} braces up here is defining the contents therein as the subroutine for sort to use.

cmp - The same as spaceship, but for strings.

Good rule of thumb: When comparing numbers, your comparison operator should not contain alphas, and when comparing strings, it should. "Don't talk to strangers."

You can sort multiple lists by feeding them into a new array:

```
@sorted= sort values %countries, @nations;
```

↳ (sorts multiple lists into one big new list.)

You can foreach over more than one list value:

```
foreach (@nations, values %countries)
{
        print "$ \n";
}
```

Grep & Map

format: @results = grep /ing/, @activities;

$matches = " " ;

↳ doing it into a scalar gives the number of matches found.

@results = grep s/ing//, @activities;

↳ can use substitution, but this will also modify the original list.

You can use a block...

@results = grep { s/ing// if /^[gsp]/ } @activities;

↳ ... in which case you no longer need the comma, because it's obvious where the expression ends, as it's inside a block delimited with {}.

map is the same as grep except it returns the results (success, or not?) of what it does, aka true results as "1".

ord is a function that changes letters to their ASCII equivalent codes.
chr does the opposite of ord.

Subroutines return the value of the last expression evaluated.

You can feed grep & map subroutines:

@result = map { &visit } @array;

External Commands

exec will terminate your perl script if it is successful.

system("command");
→ If it's successful, it returns "0".

backticks (` `) return the output of the backticked process.
→ Use them only when this is particularly what you need.

Open a Process - You can open a process into a filehandle!

open TRIN, "ps ax |" or die "Can't get these processes :$!";
(the data is piped in from the process)

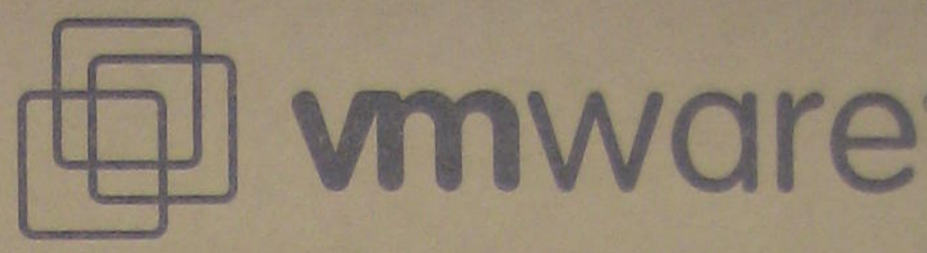
- You can also pipe data to a process by using | as the first character.

All environment variables can be accessed and set via Perl. They are in the %ENV hash.

-M returns the age in days.

printf is like 'print', but it formats the string before printing, and it works much like the C version of the command.

Ex: printf "%4.4d", $x;
→ number of digits to display
→ minimum column width



- only having one value will default to column width control.
  i.e.: printf "%4d", $x;

- adding a dash before the number will left justify the output. i.e.: printf "%-20s rules!", $name;

sprintf is like printf, except it doesn't print. You just use it to format strings. i.e.: $new = sprintf "%03s", $old;

foreach() and for() do the same thing.

Quote Execute (qx( )) ← Anything within it is executed, variable interpolated.

Ex: $result = qx(ls /home/$user);

system outputs the result of the command to the screen, qx doesn't.

A oneliner is the act of feeding Perl code directly on the command line. It is done by using the -e flag:

perl -e "while (<>) {print if /^[bv]/i}" shop.txt

(remember that (<>) opens whatever is in @ARGV.)

Adding the -n flag will automatically put a while (<>) { } around whatever commands are given.

perl -ne "print if /^[bv]/i" shop.txt

Double quotes must be escaped in one liners.

Add the -i flag for in-place file editing. -i is equivelent to $^I.

perl -i.bak -pe "s/^>+ ?//" email.txt

The -p flag adds a print statement.

You can use oneliners for math!: perl -e "print 50/200+2"

-or, Jozified-

perl -e "print 50/200+2, \"\n\""

Here's a oneliner for ASCII number lookup:

perl -e "for $i (50..90) { print chr($i), \"\n\" }"

Wrapping something in int() will return a whole number.

Passing parameters to subroutines: &howfar($height,$distance);

Parameters end up in the subroutine as the @_ array, where you can assign them as you please: ($ht,$ds)=@_;

(or: my ($height,$dist)=@_)

You can put a block by itself anywhere:

```
print "hi.\n";
{
    print "hello!\n";
}
print "hey..\n";
```

A block can have its own namespace. Variables declared with my are visible inside the block only. So, you can have variables of the same name as elsewhere in the program, but not related to them.

Once a variable is declared with "my", it is only visible within that block, and is used by its regular name.

The technical term for when a variable is visible only within a block is lexical scoping.

To return a result from a subroutine, assign a variable to the subroutine:

```
$distance = &howfar($height, $length);
```

You can "my" multiple variables by putting them in parenthesis:

```
my ($distance, $time);
```

Subroutines return a list.

As in other places, from a subroutine Perl returns the result of the last expression evaluated.

To get multiple things, or whatever you want back from a subroutine, use return();

```
return ($distance, $time);
```

A scalar takes on the last value of the list. So if you are returning from a subroutine, but assigning to a scalar, the scalar will contain the last value returned. You can change the return-catching variable into an element of a list (by putting it in parenthesis), which will cause the subroutine's return

to come back "in order" (because it's going list-to-list).

```
($distance) = &howfar($height, $length);
```

↓

You might consider always assigning the results of subroutines this way.

$, is a special variable that defines what Perl should print in between lists it is given. By default, it is nothing.

Global variables like $, can't be lexically scoped. They must instead be manipulated with local:

```
sub change
{
    local $, = "*";
    print "Words: ", @words, "\n";
}
```

local assigns temporary values to global variables, whereas my creates new variables that have the same name. my variables are faster, while local variables are visible to called subroutines.

Perl only returns a single list. If you want to return arrays, you'll have to return them as references:

```
return (\@array1, \@array2);
```

...which you assign back to scalars:

```
($a1, $a2) = &subroutine("Data");
```

...now, if you print those scalars normally, they will print the reference location of the array:

```
print "$a1 and $a2";
```

⟶ ARRAY(0x5235c0) and ARRAY(0x5235e0)

...if you want to print them dereferenced (the actual contents of the array, you have to add "@":

```
print "@$a1 and @$a2";
```

Modules

How to call:

```
use Module::Name;
```

...that'll load the module and now you can access all of its powers/functionality/functions.

```
print "Found it: $File::Find::dir
```

↑ Accesses the $dir variable in the File::Find module.

File::Find's find() function accepts two basic parameters: the name of a subroutine which defines what you want to do with the list of files being returned, and a list of directories to be searched. The subroutine is passed as a reference:

```
find(\&subroutine, $dir1, $dir2);
```

lc - The lowercase function. Guess what it does.

```
$name = lc $name;
```

When creating a Perl module, always put 1; at the bottom of it. (Perl requires that all modules return true.)

Name your Perl module with the file extension .pm

Run a script with -w to show all warnings Perl gives while running the script.

EMC²
where information lives

Make a Perl script self executable via shebang

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

...of course you can throw in any Perl command line arguments in there with it.

You can use the module "strict" to restrict unsafe constructs. If you use it, Perl becomes strict about variables, references, and subroutines (vars, refs, and subs).

```
use strict;
```

Referring to a variable by its fully qualified name:

```
$MAIN::name = "value";    (when it's in the main block)
```

Perl has a debugger, which is pretty cool. It's available by using the -d flag. Some basic useful features/commands:

(All followed by Enter... except "Enter":)

s → Execute single step.
n → Executes main program step-by-step, but "just does" subroutines.
/xx/ → Searches through the program for xx.
p → Prints. For example: p $variable p @array
Enter → Repeats the last n or s command.

You can also insert Perl code into the debugger to modify the program in real-time. i.e.:

```
$variable=~s/\s//gi  #The debugger doesn't
                     #seem to like spaces.
```

EMC²
where information lives

Precedence

|| has more precedence than "or." For example:

```
$file = "not-there.txt";
open FILE, $file || print "1: Can't open file: $!\n";
open FILE, $file or print "2: Can't open file: $!\n";
```

↳ Only the second one prints. The boxes indicate what is first evaluated in each scenario. The double-pipes (||) are like a magnet. Secondly evaluated after the boxed areas are the right-hand sides.

&& has more precedence than "and". Same rules as above apply.

&& evaluates the left-hand expression first, and if it is false, doesn't bother with the right-hand side. &, however, evaluates the other side of the expression regardless of the result of the left-hand side.

! has more precedence than not.

xor is the exclusive or. How it works is:

- If one expression is true, xor returns true.
- If both expressions are false, xor returns false.
- If both expressions are true, xor returns false. ← (the crucial difference from or)

-=END OF NOTES FROM ROBERT'S PERL TUTORIAL=-